MUTAGEN: Reliable Coverage-Guided, Property-Based Testing using Exhaustive Mutations

(accepted preprint)

Agustín Mista
Chalmers University of Technology
Gothenburg, Sweden
mista@chalmers.se

Alejandro Russo
Chalmers University of Technology
Gothenburg, Sweden
russo@chalmers.se

***Abstract*—Automatically-synthesized random data generators are an appealing option when using property-based testing. There exists a variety of techniques that extract static information from the codebase to produce random test cases. Unfortunately, such techniques cannot enforce the complex invariants often needed to test properties with sparse preconditions.**

Coverage-guided, property-based testing (CGPT) tackles this limitation by enhancing synthesized generators with structure-preserving mutations guided by execution traces. Albeit effective, CGPT relies largely on randomness and exhibits poor scheduling, which can prevent bugs from being found.

We present MUTAGEN, a CGPT framework that tackles such limitations by generating mutants *exhaustively*. Our tool incorporates heuristics that help to minimize scalability issues as well as cover the search space in a principled manner. Our evaluation shows that MUTAGEN not only outperforms existing CGPT tools but also finds previously unknown bugs in real-world software.

***Index Terms*—random testing, mutations, heuristics**

I. INTRODUCTION

Random Property-Based Testing (RPBT) is a popular technique for finding bugs using executable testing properties [10, 25, 40, 6, 12]. A practical limitation of RPBT is the need for random data generators used to instantiate the testing properties, and writing highly-tuned generators can take several thousand person-hours of trial and error [29]. Luckily, there exist several approaches that automatically synthesize random data generators by extracting static information from the codebase, e.g., data type definitions and application public interfaces (APIs). [17, 34, 33, 14, 28, 3]. These approaches, however, are unable to synthesize generators capable of producing data satisfying complex invariants not easily derivable from the codebase. Generating random valid programs to test compilers is a clear example of this limitation [8], where developers are forced to write specialized generators by hand [39, 41, 47].

Coverage-Guided, Property-Based Testing (CGPT) [29] is a technique that borrows ideas from the fuzzing community to generate highly-structured values while still using automatically derived generators. CGPT keeps queues of *interesting* previously executed test cases that can be transformed using structure-preserving mutations to produce new ones. Intuitively, mutating an existing interesting test case is more likely to produce a new interesting test case than generating a new one from scratch. Moreover, unlike the generic bit-level mutators often used by the fuzzing community [7, 37], structure-preserving mutations specified at the data type level can effectively produce only syntactically valid mutants. Such an approach has shown to be effective when fuzzing systems accepting structurally complex inputs [22, 46, 20]. Notably, CGPT uses the data type information of the inputs to the testing properties to derive specialized structure-preserving mutators directly and without the need for external grammars — making strongly-typed programming languages an ideal match for this technique. In addition, CGPT relies on execution traces to distinguish interesting test cases — a technique popularized by *coverage-guided fuzzers* like *AFL* [32]. Here, test cases are interesting (and therefore worth mutating) only when they exercise new parts of the code in the system under test.

In this work, we establish several aspects of the seminal CGPT approach by Lampropoulos et al. that leave room for improvement (see §II). In particular: 1) if not done carefully, automatically derived structure-preserving mutators can become "shallow", unlikely to transform deep test cases more than superficially; 2) the queuing mechanism can cause delays if interesting test cases are enqueued frequently and there is no way to prioritize them; and 3) the heuristic used to assign a "mutation budget" to each interesting test case (often referred to as a *power schedule*) requires fine tuning and can be hard to generalize. Overcoming these obstacles is important to make CGPT more suitable for testing real-world software.

To tackle these limitations, we introduce MUTAGEN, a CGPT framework that applies mutations *exhaustively* (see §III). That is, given an interesting test case, our tool forces every structure-preserving mutation that can be applied to it to be evaluated exactly once. This has two main advantages. Firstly, every subexpression of the input test case is mutated on the same basis, ensuring that deep transformations are not omitted due to randomness. Moreover, computing mutations exhaustively eliminates the need for a heuristic power schedule.

Internally, MUTAGEN distinguishes two kinds of mutations. On one hand, *deterministic (pure) mutations* encode transformations that yield a single mutated test case obtained by swapping data constructors around, as well as rearranging or returning subexpressions. On the other hand, *non-deterministic*

(random) mutations are used to represent transformations over large enumeration types. This mechanism let us selectively escape the scalability issues of exhaustiveness by yielding a random generator that replaces a specific subexpression of an input test case with a randomly generated one. This generator is later sampled a *relatively small* number of times. This way MUTAGEN avoids, for instance, mutating every number inside a test case into every other number of its range.

MUTAGEN's testing loop incorporates two novel heuristics that help finding bugs more reliably (§IV). In the first place, our tool uses last-in-first-out (LIFO) scheduling with priority when enqueueing interesting test cases for mutation. This way, interesting test cases that discover larger parts of untested code are given a higher priority. Moreover, LIFO scheduling allows the testing loop to jump back and forth between enqueued test cases as soon as new more interesting ones become available, eliminating potential delays when the mutation queues grow more often than they shrink.

The second heuristic controls the number of test cases sampled from random mutations by monitoring how often we generate interesting test cases. Whenever this frequency stalls, MUTAGEN resets the testing loop and increases the effort put into sampling random mutations. This way, our tool automatically adjusts this parameter on the fly.

We validated our ideas in two different ways. We first compared MUTAGEN against *FuzzChick*, the reference CGPT implementation by Lampropoulos et al., on all the existing cases studies described in their original work. These case studies focus on finding counterexamples for buggy variations of two Information-Flow Control (IFC) machines of different complexity. Our results (§VI) indicate that: when bugs are relatively easy to find, MUTAGEN can reliably find them faster than *FuzzChick*. On the other hand, when bugs are harder to find, our tool outperforms *FuzzChick* in terms of failure rate at the cost of (possibly) needing more time to find them. Notably, MUTAGEN is capable of finding bugs that *FuzzChick* was not able to find in our evaluation nor in its original one.

Additionally, we compared MUTAGEN against *QuickCheck* [10], the most widely used RPBT tool in Haskell, on an existing WebAssembly engine implementation of industrial strength. There, MUTAGEN is capable of reliably finding 15 planted bugs in the validator and interpreter, as well as 3 previously unknown ones. Moreover, this case study lets us evaluate the performance versus the overhead of our tool (and its custom code instrumentation mechanism). All in all, our evaluation indicates that *testing mutants exhaustively together with our heuristics to escape scalability issues* can be an appealing technique for finding bugs reliably without sacrificing speed.

We additonally present threats to validity in §VII and discuss related work in §VIII, to finally conclude in §IX.

II. BACKGROUND

This section briefly introduces the motivation, ideas and limitations behind CGPT [29]. To illustrate it, we focus on a simple property defined over binary trees. Such a data structure can be defined in Haskell with 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 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`. If we assume existence of a function `balanced` of type `Tree a -> Bool` that asserts that a tree satisfies some notion of balancedness, we can write properties to validate that the operations defined over binary trees preserve this invariant. For instance, to validate the implementation of an `insert` function, we assert that, given an element `x` and a balanced tree `t` as input, inserting `x` into `t` will produce a balanced tree as output:

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

(The definitions of `balanced` and `insert` are not important here.) The arrow operator (`==>`) indicates that `balanced t` is a precondition of this property, so test cases where the input tree is unbalanced will get *discarded* prematurely.

The only missing piece is a random generator of trees. For this, we can define a naïve generator for trees of integers as:

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

This definition (simplified to make it more accessible) follows a common type-directed approach used by some existing generator synthesizer tools. At each step, `genTree` picks a `Tree` data constructor with uniform probability, and calls itself to generate recursive subexpressions, carefully reducing the input size limit `size` by a unit at a time. This ensures termination by generating only leaves when the size reaches zero (case `size == 0`). Integers payloads are generated by calling an external random generator (`genInt`) defined elsewhere.

Readers familiar with RPBT will notice that `genTree` is not suitable for testing `prop_insert` with *QuickCheck*, as this generator produces mostly unbalanced trees which do not satisfy the property's precondition, thus leaving its postcondition (`balanced (insert x t)`) largely untested.

Coverage-Guided Property-Based Testing: CGPT alleviates the problem of testing properties with non-trivial preconditions while using automatically derived generators by enhancing the testing process with: 1) *target code instrumentation*, to capture execution information from each test case; and 2) *high-level, structure-preserving mutations*, to produce syntactically valid test cases from existing ones.

Using code instrumentation in tandem with mutations is a well-known technique in the fuzzing community [32, 45, 1, 13, 26]. Notably, CGPT can additionally use the result of the testing properties' preconditions to distinguish semantically valid test cases from invalid ones. This is useful to favor mutating valid test cases over discarded ones.

The CGPT testing loop uses two queues to store valid and discarded previously executed test cases along with a mutation budget that controls how many times they can be mutated before being finally thrown away. This budget is calculated using a heuristic derived from AFL's power schedule, i.e., more budget to test cases that lead to shorter executions, or that discover more parts of the code. On each iteration, the testing loop selects the next test case by mutating the first value on the queue of valid test cases. If such queue is empty, it mutates the first test case from the queue of discarded test cases. If both queues are empty, CGPT generates a new random value from scratch. The loop then runs this test case and evaluates whether it was interesting. If the test case was interesting, it gets enqueued into its corresponding queue (either valid or discarded), This process alternates between random generation and mutation until a bug is found or the test limit is reached.

Limitations of CGPT: Lampropoulos et al. compared the mean-time-to-failure (MTTF) of CGPT against random testing using both automatically derived generators and manually-written ones, where their results show that CGPT lies in between these two approaches. While MTTF is a useful global metric, we argue that a meticulous evaluation ought to consider failure rate, i.e., the ability to find a bug in a given run as an important metric when comparing PBT tools. After repeating each original experiment 10 times, we observed that *FuzzChick* was only able to find 7 (out of 20) and 18 (out of 33) bugs with 100% failure rate in the two IFC machine case studies. Notably, *FuzzChick* was unable to find any counterexample for 3 of the planted bugs. With this observation in mind, we consider three aspects to tackle CGPT's reliability issues:

- *Mutators distribution:* for simplicity, the mutators proposed by Lampropoulos et al. are derived to follow a top-down approach: mutations can happen at the top level or be recursively applied to an immediate subexpression of the input test case with approximately the same probability. This makes deep recursive mutations very unlikely, as their probability decreases multiplicatively with each recursive call. Hence, these mutators can only effectively transform shallow test cases, excluding scenarios involving deeply nested data structures. *Ideally, mutations should happen on every subexpression of the input test case on a reasonable basis*.
- *Scheduling:* CGPT uses single-ended queues to store valid and discarded interesting test cases, where new test cases are placed *at the end* of their corresponding queue. If a test case discovered a whole new portion of the target code, it will not be mutated until the rest of the queue ahead of it gets processed, limiting the effectiveness of the testing loop whenever queues grow more often than they shrink. In an extreme case, interesting test cases might not get processed at all within the testing budget. *Thus, we should prioritize mutating novel test cases right away*.
- *Power schedule:* it is unclear how well this heuristic assigns a budget to each interesting test case. On one hand, assigning too much budget to not-so-interesting wastes precious testing time. On the other hand, assigning too little budget to interesting test cases might prevent bugs from being discovered at all! Finding a balanced heuristic can be quite challenging in the general case. *Ideally, the scheduling mechanism should be as unbiased as possible.*

III. MUTAGEN

This section describes the main ideas behind MUTAGEN, our revised CGPT tool written in Haskell. [1]

MUTAGEN works by mutating test cases in an exhaustive and precise manner, where 1) each subexpression of a test case is associated with a set of structure-preserving mutations, and 2) each one of these mutations is scheduled *exactly once*. We realized that, by using an exhaustive mutation approach, we avoid needing a heuristic power schedule to assign a budget to each interesting test case. Moreover, computing mutants exhaustively ensures that interesting mutations are not omitted or overly exercised due to randomness. This approach is inspired by exhaustive bounded testing tools like *SmallCheck* [43] or *Korat* [5] — refer to §VIII for a detailed discussion.

A. Exhaustive Mutations

In MUTAGEN, mutators are defined as the set of mutants that can be obtained by transforming the input test case *at the top-level* (the root data constructor). For a given type `a`, we represent a mutator of `a`'s with a function from `a`'s to a list of mutants. In Haskell, we introduce the type synonym:

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

As mentioned earlier, concrete mutants can be obtained either from a pure or a random mutation, which we define as follows:

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

We first focus on pure mutants, which encode deterministic transformations over the *outermost* data constructor of the input — recursive mutations will be introduced soon.

These transformations can either 1) return an immediate subexpression of the same type as the input, or 2) swap the outermost data constructor with a (possibly) different one of the same type, reusing the immediate subexpressions of the input in any combination that produces a well-typed value.

```
mutate t = case t of
  Leaf x -> [
    PURE (Branch def x def)
  ]
  Branch l x r -> [
    PURE l, PURE r,
    PURE (Leaf x),
    PURE (Branch l x l),
    PURE (Branch r x r),
    PURE (Branch r x l)
  ]
```

Fig. 1. MUTAGEN `Tree` mutator.

Fig. 1 illustrates a mutator for the `Tree` data type. This definition simply enumerates mutants that transform the outermost data constructor. Moreover, notice how a default value `def` used to fill the subtrees when "growing" a leaf into a branch. In practice, `def` corresponds to the simplest expression we can construct for the mutant to be type-correct. In our example, the default `Tree` value is a leaf containing the smallest value of the payload type, e.g., `Leaf 0` is the default value of `Tree Int`. Using a small default value, as opposed to a randomly

[1] Although we make use of Haskell's powerful type system, our ideas should apply to other statically typed languages with minor effort.

generated one (as done by the original CGPT) is also inspired by exhaustive bounded testing tools, and avoids introducing unnecessary randomness when growing data constructors.

Formally, for a type T defined in terms of the data constructors C_i, each one with fields of (possibly different) types t_j^i:

$$T \;:=\; C_1\ t_1^1\ t_2^1\ ...\ |\ C_2\ t_1^2\ t_2^2\ ...\ |\ ...$$

MUTAGEN synthesizes the `mutate` function so it pattern matches on the root data constructor of the input as follows:

$$\texttt{mutate}(C_i\ x_1\ x_2\ ...) = \texttt{mut}_\texttt{r}(C_i\ x_1\ x_2\ ...) \cup \texttt{mut}_\texttt{s}(C_i\ x_1\ x_2\ ...)$$

Firstly, $\texttt{mut}_\texttt{r}$ computes the set of possible mutations that return an immediate subexpression of the same type as the input:

$$\texttt{mut}_\texttt{r}(C_i\ x_1\ x_2\ ...) = \{\texttt{PURE}\ x_k \mid x_k \in \texttt{filter}(T, \{x_1,\ x_2,\ ...\})\}$$

Then, $\texttt{mut}_\texttt{s}$ builds every mutation that swaps the root data constructor with a (possibly) different one, reusing (or defaulting to) compatible subexpressions whenever possible:

$$\texttt{mut}_\texttt{s}(C_i\ x_1\ x_2\ ...) = \left\{ \texttt{PURE}(C_j\ x_1'\ x_2'\ ...) \;\middle|\; \begin{array}{l} C_j \in \{C_1, C_2, ...\} \\ x_k' \in \overline{\texttt{filter}}(t_k^j, \{x_1,\ x_2,\ ...\}) \\ C_j\ x_1'\ x_2'\ ...\ \neq\ C_i\ x_1\ x_2\ ... \end{array} \right\}$$

The helper `filter` simply returns the subset of the input values X that match the type t, whereas $\overline{\texttt{filter}}$ returns the default value of the type t ($\texttt{def}_t$) if the result of `filter` is empty:

$$\texttt{filter}(t, X) = \{x \mid x \in X, \texttt{typeof}(x) = t\}$$

$$\overline{\texttt{filter}}(t, X) = \begin{cases} \texttt{filter}(t, X) & \texttt{if filter}(t, X) \neq \emptyset \\ \{\texttt{def}_t\} & \texttt{otherwise} \end{cases}$$

This ensures that a small constructor can always be grown into a larger one by inserting default subexpressions whenever needed. (Recalling the `Tree` mutator from Fig. 1, we show this for the case of mutating a `Leaf` into a `Branch`.)

We can finally focus on random mutants, which let us *selectively avoid exhaustiveness* when mutating values of large enumeration types (e.g. numbers). Instead of creating a `PURE` mutant for every numerical subexpression exhaustively, we condense them into a generator that can be sampled to produce new random values. This way, a mutator for integers becomes:

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

This approach allows MUTAGEN to control the amount of effort put into mutating any subexpression of an input test case associated with a random mutation. This can avoid dedicating unnecessary effort to mutating data payloads when the execution of the testing property or the system under test is independent of their values (see §IV).

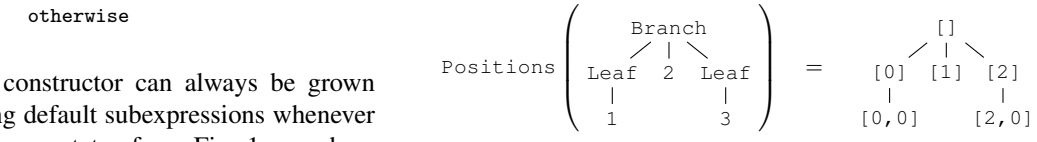
Algorithm 1: MUTAGEN Testing Loop

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

Algorithm 2: Mutation Batch Initialization

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

Mapping top-level mutations everywhere: so far we have defined mutations that transform only the root node of the input. To apply these mutations to every subexpression we use two utility functions. Firstly, a function `Positions` traverses the input and builds a Rose tree [31] of *mutable positions*, i.e., lists of indices encoding the path from the root to every mutable subexpression. For instance, the mutable positions of the value `Branch (Leaf 1) 2 (Leaf 3)` are:



Then, we define a function `MutateInside` that takes a desired position within an input test case and mutates its corresponding subexpression, returning a list of mutants. This function traverses the desired position, calling itself recursively until it reaches the desired subexpression, where a mutation encoded by `mutate` can be applied directly. The definition of these functions consists of boilerplate code that our tool synthesizes automatically, thus we omit them to preserve space.

B. Testing loop

We now introduce the base testing loop of MUTAGEN, outlined in Algorithm 1. Like in CGPT, we use two queues, QValid and QDiscarded to store valid and discarded interesting test cases, respectively. Our tool precomputes all the mutations of a given test case before enqueueing them. These mutations

Algorithm 3: MUTAGEN Seed Selection

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

are put together into lists we call *mutation batches* — one for each mutated test case. To initialize a mutation batch (outlined in Algorithm 2), we first flatten all the mutable positions of the input test case in level order. Then, we iterate over all these positions, retrieving all the mutants associated to each corresponding subexpression. For each one of these: 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, 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.

Then, the seed selection algorithm (Algorithm 3) picks the next test case using the same criteria as CGPT, prioritizing valid test cases over discarded ones, falling back to random generation when necessary. For this, we simply pick the next mutated test case from the current precomputed batch, jumping to the next batch in line when the current one becomes empty.

Having selected the next test case, the testing loop proceeds to execute it, capturing both the result (valid, discarded, or failed) and its execution trace. If the test case fails, it is reported as a bug. If not, the algorithm evaluates if it was interesting based on its trace information and the one from previously executed test cases (represented by TLog). If the test case was interesting, its mutants are precomputed and enqueued on its corresponding queue. This process is repeated until finding a bug or reaching the test limit N.

A notable difference with CGPT's testing loop is the criterion for enqueuing discarded tests. We found that, especially for large data types, the queue of discarded candidates tends to grow disproportionately large, making it hardly usable while consuming large amounts of memory. To improve this, we resort to mutating 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) was valid. Then, we enqueue discarded test cases only if they descend from a valid parent (see Algorithm 1). This way we fill the discarded queue with lesser but more interesting test cases.

Algorithm 4: Priority LIFO Heuristic

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

IV. MUTAGEN HEURISTICS

In this section we introduce two heuristics implemented on top of the base testing loop of MUTAGEN described in §III.

A. Priority LIFO Scheduling

This heuristic tackles the issue of enqueuing new interesting test cases at the end of (possibly) long queues of not-so-interesting ones. For this, MUTAGEN captures the execution trace of each test case and computes its novelty relative to previously executed ones with respect to their *edge coverage*, i.e, test cases that discover new edges in the system under test are considered interesting, and their priorities are proportional to the number of edges they discovered.

Using this mechanism, we can modify MUTAGEN's base testing loop replacing each mutation queue with a priority queue indexed by the novelty of their test cases. These changes are illustrated in Algorithm 4. Statements in red indicate important changes to the base algorithm, whereas ellipses denote parts of the code that remain unchanged.

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

B. Tuning Random Mutations Parameter

As introduced in §III, our tool is parameterized by the number of times it samples the random generators associated with random mutations (R). But, how many test cases should we sample? Answering this question precisely can be challenging, so this second heuristic aims to alleviate the problem.

We found that the smaller the number of times we sample from random mutations, the easier it is for the trace log that records executions to get saturated, i.e., when interesting test cases stop getting discovered or are discovered very seldomly.

Algorithm 5: Adaptive Random Mutations Heuristic

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

We realized that we can use this information to dynamically adapt the number of times we sample from random mutations. This idea is described in Algorithm 5. The process is as follows: 1) we start the testing loop with the R parameter set to one, and 2) each time we find that a test is not interesting (i.e. boring), we increment a counter. Then, 3) if we have not produced any interesting test case after a certain number of tests (1000 tests seems to be a reasonable threshold in practice), we duplicate the number of random mutations and the threshold. Additionally, we reset the trace log so interesting test cases found on a previous iteration can be found and enqueued for mutation again with a higher effort dedicated to sampling from random mutations.

Notably, this heuristic can be useful when the execution of the system under test depends on invariants over numeric data (e.g, the number of pixels declared by the header of an image matching the size of its actual payload). There, starting with a single random mutation will quickly saturate the trace log with discarded (invalid) tests, and this heuristic will continuously increase the effort put into sampling from random mutations until some randomly generated value satisfies the required invariant, making the overall test case valid.

V. Case Studies

We evaluated the performance of MUTAGEN using three case studies. The first two are IFC abstract machines that enforce *noninterference* [16, 44] using runtime checks. While similar in spirit, these abstract machines have a completely different complexity. The first one follows a relatively simple stack-based execution model, with a limited number of instructions. The second one is substantially more featureful, including registers, dynamic memory allocation and a larger instruction set, among others. Notably, both machines were originally proven correct by Azevedo de Amorim et al. [2] in Coq, and later degraded by systematically introducing bugs in their IFC policy enforcing mechanism. Lampropoulos et al. borrowed these case studies from existing literature [23, 24] to compare *FuzzChick* against RPBT using automatically derived and hand-tuned random generators. Here, we reproduce all their experiments and compare them against our tool. Worth mentioning, we mechanically translated these case studies to Haskell in order to run MUTAGEN on their test suites.

The third case study evaluates MUTAGEN in a realistic scenario, and targets *haskell-wasm* [42], an existing WebAssembly engine of industrial strength. Unfortunately, the current state of *FuzzChick*'s development does not allow to easily port new case studies into its framework, so comparing MUTAGEN with *FuzzChick* on this case study has been out of the scope of this work. Instead, we compare MUTAGEN against *QuickCheck*, evaluating its effectiveness versus the relative overhead of our custom code instrumentation.

A. IFC Stack and Register Machines

These abstract machines enforce noninterference, a hyperproperty based on the notion of *indistinguishability*. Intuitively, two machine states are indistinguishable if they only differ on secret data. Using this notion, the variant of noninterference we are interested in is called *single-step noninterference* [23] (SSNI). Given two indistinguishable machine states, SSNI asserts that running a single instruction on both machines brings them to resulting states that are also indistinguishable. To achieve this, every runtime value handled by these abstract machines is labeled with a security level, i.e., L (for "low" or public) or H (for "high" or secret). Security labels are then propagated throughout the execution of the program whenever the machines execute an instruction. For this, both machines use a different rule table to specify their IFC policy. These tables store the dynamic check that each machine needs to perform before running each instruction, along with the resulting security labels corresponding to the program counter and the instruction result. For instance, to execute the `Store` instruction (which stores a value in a memory pointer), the IFC Stack machine checks that both the labels of the program counter and the pointer together can flow to the label of the destination memory cell. If this condition is not met, this machine immediately halts its execution. After this check, the machine overwrites the value at the destination cell and updates its label with the maximum sensibility of the involved labels. In the rule table, this looks as follows:

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

Where l_{pc}, l_p, l_v, and $l_{v'}$ represent the labels of: the program counter, the memory pointer, and the old and new values stored in that memory cell. The symbol $\vee$ simply denotes the join of two labels, i.e., the *maximum* of their sensibilities. To preserve space, we encourage the reader to refer to the work of Hriţcu et al. [23, 24] and Lampropoulos et al. [29] for further details about the implementation and semantics of these case studies.

Bugs are systematically injected in the IFC enforcing mechanism of both machines by weakening the checks stored in their corresponding IFC rule table. For instance, the following are the buggy rule variations (in red) for the `Store` instruction of the IFC Stack machine:

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

This way, there are 20 different buggy ways the IFC Stack machine can be tampered with to violate its IFC policy and invalidate SSNI. Likewise, 33 different IFC-violating bugs can be inserted in the IFC Register machine.

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

B. WebAssembly Engine

WebAssembly [19] is a language designed for executing low-level code on the web. WebAssembly programs are first validated and later executed in a sandboxed environment. The language is relatively simple, in essence 1) it contains only four base types, representing both integers and IEEE754 floating-point numbers of either 32 or 64 bits; 2) values of these types are manipulated by functions written using sequences of stack instructions; 3) functions are organized in modules and must be explicitly imported/exported; 4) memory blocks can be imported, exported, and grown dynamically; among others. WebAssembly semantics are fully specified, and programs must be consistently interpreted across engines — despite some subtle details we will address soon. For this, the WebAssembly standard provides a reference implementation with all the functionality expected from a compliant engine.

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

In this work, we apply 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.

Id	Subsystem	Category	Description
1	Validator	Bug	Invalid memory alignment validation
2	Validator	Discrepancy	Validator accepts returning multi-value blocks
3	Interpreter	Bug	Function invoker ignores arity mismatch
4	Interpreter	Bug	Allowed out-of-bounds memory access
5	Interpreter	Discrepancy	Non-standard NaN reinterpretation

TABLE I
ISSUES FOUND BY MUTAGEN IN *haskell-wasm*.

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

TABLE II
BUGS INJECTED INTO *haskell-wasm*.

For this, we took advantage of the reference implementation to find discrepancies (that could potentially lead to bugs) via differential testing [30]. Our testing properties assert that any result produced by *haskell-wasm* matches that of the reference implementation. Notably, MUTAGEN discovered three latent bugs that the existing test suite was unable to reveal. Moreover, MUTAGEN exposed two other discrepancies between *haskell-wasm* and the reference implementation. These discrepancies trigger parts of the specification that were either not yet supported by the reference implementation (multi-value blocks), or that produce a well-known non-deterministic undefined behavior (NaN reinterpretation) [41]. All these findings (see Table I) were confirmed by the authors of *haskell-wasm*.

Having sorted these issues out, we mechanically injected 10 new bugs in the validator as well as 5 new bugs in the interpreter of this engine (see Table II). These bugs either 1) remove an existing integrity check (to weaken the WebAssembly type-system/validator); or 2) simulate a copy-paste bug [9], replacing the implementation of an instruction with a compatible one (e.g., `i32.add` by `i32.sub`).[2]

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

```
prop_validator m = isValidHaskellWasm m ==> isValidRef m
```

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

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

Testing the WebAssembly Interpreter: Testing the WebAssembly interpreter is substantially more complex than testing the validator since it requires running and comparing the

[2]These bugs were inspired by the real bug #1 we found prior to this step.

output of test case programs. To achieve this, the generated test cases need to comply with a common interface that can be invoked both by *haskell-wasm* and the reference implementation. For simplicity, we write a helper function `mkModule` to build a stub WebAssembly module that initializes one memory block and exports a single function. This helper is parameterized by the definition of the module's single function, along with its type signature and name. Then, we can use `mkModule` to define a testing property parameterized by a function type, along with its definition and invocation arguments:

```
prop_interpreter ty fun args =
  do let m = mkModule fun ty "f"
     resHaskellWasm <- invokeHaskellWasm m "f" args
     resRef         <- invokeRef         m "f" args
     return (resHaskellWasm === resRef)
```

This testing property instantiates a module stub `m` with the input WebAssembly function (`fun`) and its type signature (`ty`). Then, it invokes the function `f` of the module `m` both on *haskell-wasm* and the reference interpreter with the provided arguments (`args`). Finally, the property asserts whether their results are equivalent.[3] Interestingly, equivalence 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 I), and our equivalence relation needs to take that into account.

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

```
prop_interpreter_i32 f x y =
  prop_interpreter (Type { args=[I32, F32], res=[I32] })
                   f [VI32 x, VF32 y]
```

This property lets us generate functions of a fixed type and invoke them with randomly generated inputs of the expected types. We use this property to find all the bugs injected into *haskell-wasm*'s interpreter in the next section.

VI. Evaluation

We repeated each experiment 10 times in a workstation with 64GB of RAM and an Intel Core i7-8700 CPU. In all cases, we used a one-hour timeout to stop the execution if an experiment had not yet found a counterexample. Moreover, we followed the approach taken by Lampropoulos et al. and collected the mean-time-to-failure (MTTF) of each bug, i.e., how quickly a bug can be found in wall clock time. In addition, we collected the failure rate (FR) observed for each bug, i.e. the proportion of times each tool finds each bug within the one-hour testing budget. Unlike Lampropoulos et al., *we only aggregate the MTTF of successful runs*, i.e, when a bug was found, since

[3]We also set a short timeout to discard potentially diverging programs.

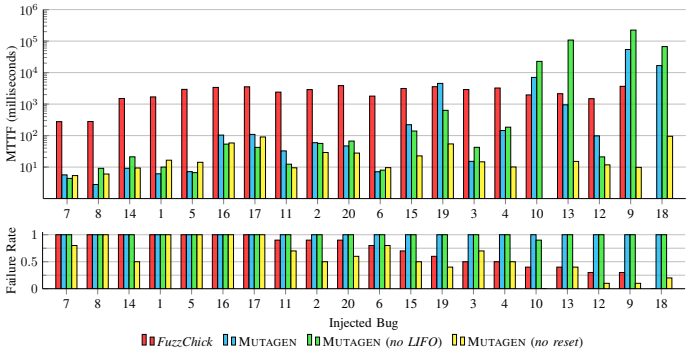


Fig. 2. *FuzzChick* versus MUTAGEN in the IFC stack machine.

doing so for all runs heavily inflates results when the failure rate is below 100%. In all case studies, we additionally show the effect of the heuristics described in §IV by individually disabling them. We call these variants *no LIFO* and *no reset*. For *no reset*, the number of times we sample random mutants (R) is no longer controlled by MUTAGEN, so we arbitrarily fixed it to R=25 throughout the experiment. [4]

A. IFC Stack and Register Machines

The results of these case studies are shown in Fig. 2 and Fig. 3, respectively. In both cases, the injected bugs are ordered by the failure rate achieved by *FuzzChick* in decreasing order. Moreover, notice the logarithmic scale used on the MTTF.

Firstly, we observed a statistically significant improvement in terms of the overall failure rate: MUTAGEN achieved 100% and 90.9% failure rate (versus 71% and 74.2% for *FuzzChick*) in the IFC Stack and Register cases studies, respectively.

Moreover, if we observe the MTTF achieved by each tool, we recognize in both cases that MUTAGEN is significantly faster than *FuzzChick* when finding "easy" bugs, i.e., those which both tools can reliably find on each run. However, the results are not as intuitive for the "harder" bugs, i.e, where the failure rate of some tool drops below 100%. To better understand the tradeoffs between these two metrics, we grouped bugs into four categories based on the statistical evidence[5] we observed over the corresponding MTTF achieved by each tool: 1) when *FuzzChick* is faster than MUTAGEN, 2) when MUTAGEN is faster than *FuzzChick*, 3) when results are inconclusive, i.e., no statistical evidence in favor of either tool, and 4) when *FuzzChick* always fails to find a bug. We avoid considering the case where MUTAGEN always fails to find a bug as this scenario did not occur in our experiments. In each case, we additionally computed the mean failure rate across bugs for each tool. These curated results are shown in Tables III and IV. In both cases, we can observe that our tool is faster than *FuzzChick* for a significant number of bugs, while there are only two bugs in the IFC Register Machine case study where *FuzzChick* consistently outperforms MUTAGEN. The inconclusive cases reveal that MUTAGEN achieves a considerably larger failure rate without being significantly slower than *FuzzChick*.

[4]A replication package [35] is available for reviewing purposes.

[5]Based on each tail of a Mann-Whitney U-Test with threshold $p < 0.05$.

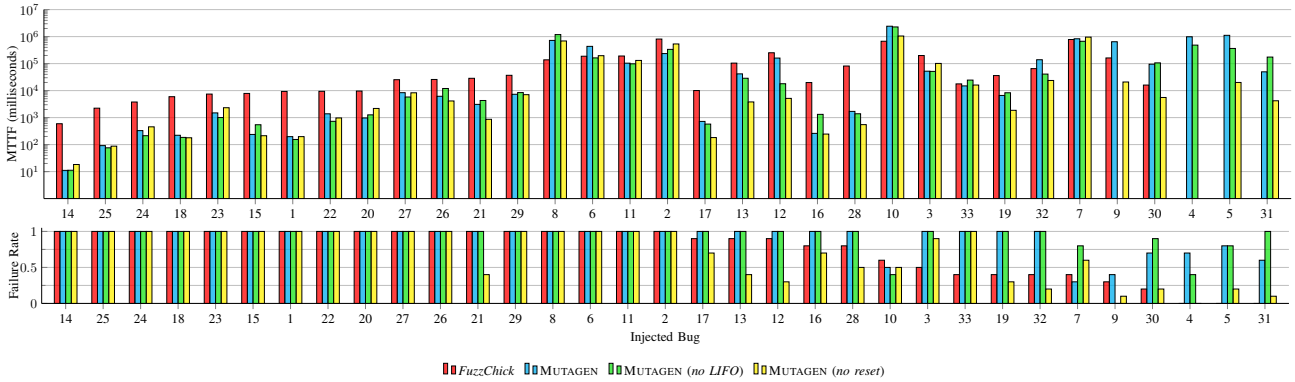


Fig. 3. *FuzzChick* versus MUTAGEN in the IFC register machine.

In terms of the MUTAGEN heuristics, we observed that disabling our LIFO scheduling (case *no LIFO*) does not show a large impact on the results. We noticed that, for these case studies, when a new interesting test case gets enqueued, all its mutants (and their descendants) are quickly processed before new ones start piling up, keeping the mutation queues empty most of the time (generation mode). On the other hand, dynamically tunning of the random mutation parameter seems critital to find the harder bugs, as disabling it (case *no reset*) heavily affects MUTAGEN's failure rate in such cases.

B. WebAssembly Engine

The results of this case study are shown in Fig. 4, ordered by the MTTF achieved by MUTAGEN. We first focus on the bugs injected in the validator (Fig. 4 left). There, we quickly conclude that *QuickCheck* is not well suited to find most of the bugs — it merely finds the easier bugs #5 and #3 in just 1 out of 10 runs. The reason behind this is simple: an automatically derived generator is virtually unable to produce valid WebAssembly modules other than the trivial empty one. Using the same random generator, however, MUTAGEN consistently finds every bug in less than 4 seconds. Moreover, disabling the heuristics does not affect the failure rate but tends to add some time overhead to the MTTF, where the *no LIFO* and *no reset* variants are 2.1x and 1.4x slower than the baseline on average.

Bugs where	Count	Mean Failure Rate	
		FuzzChick	MUTAGEN
FuzzChick is faster	0	-	-
Neither tool is faster	2	0.35	1
MUTAGEN is faster	17	0.79	1
FuzzChick always times out	1	0	1

TABLE III
CURATED RESULTS FOR THE IFC STACK MACHINE CASE STUDY.

Bugs where	Count	Mean Failure Rate	
		FuzzChick	MUTAGEN
FuzzChick is faster	2	0.8	0.75
Neither tool is faster	8	0.52	0.8
MUTAGEN is faster	20	0.93	1
FuzzChick always times out	3	0	0.7

TABLE IV
CURATED RESULTS FOR THE IFC REGISTER MACHINE CASE STUDY.

If we now focus on the bugs injected into the interpreter (Fig. 4 right), we notice that finding bugs now requires minutes instead of seconds, as both interpreters need to validate and run the inputs before producing a result to compare. We also observe a significant improvement in the performance of *QuickCheck* in terms of failure rate. This is of no surprise: we deliberately reduced the search problem to generating functions bodies instead of complete WebAssembly modules. Notably, *QuickCheck* finds counterexamples for the bug #14 almost instantly. This is because this bug can be found using a very small counterexample, and *QuickCheck* prefers sampling small test cases at the beginning of the testing loop. While MUTAGEN uses this approach when in generation mode, our scheduler does not take the size of an interesting test case into account when computing its priority — future work will investigate this possibility. Nonetheless, MUTAGEN still outperforms *QuickCheck* on the remaining bugs in terms of MTTF. Moreover, the *no reset* variant resulted in a 2.9x average slowdown with respect to the baseline, whereas the *no LIFO* variant shows a subtle speedup at the cost of no longer finding the bug #15 with 100% failure rate.

Finally, this case study allows us to analyze the overhead introduced by the custom code instrumentation and internal processing used in MUTAGEN versus the black-box approach used by *QuickCheck*. Table V compares the total number of executed and passed tests per second achieved by each tool.

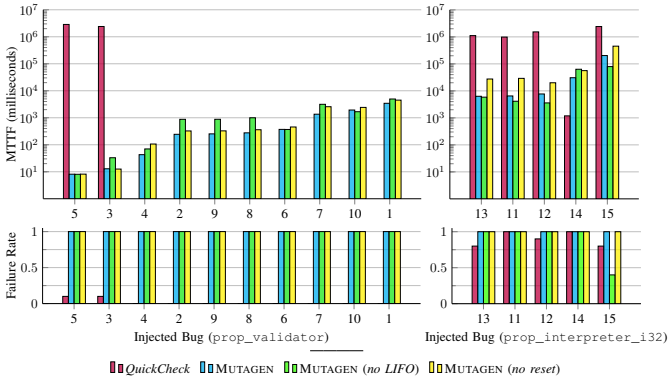


Fig. 4. *QuickCheck* versus MUTAGEN in the WebAssembly case study.

Property	*QuickCheck*		MUTAGEN	
	Total	Passed	Total	Passed
`prop_validator`	31882.78	0.0003	3505.68	756.52
`prop_interpreter_i32`	106619.31	18.02	2142.14	500.67

TABLE V
TESTS PER SECOND ON THE WEBASSEMBLY CASE STUDY ACROSS TOOLS.

Although MUTAGEN is considerably slower than *QuickCheck* at executing tests (roughly 9x and 49x slower when testing `prop_validator` and `prop_interpreter_i32` respectively), it runs substantially more valid tests that pass the sparse preconditions in the same amount of time, *which can ultimately lead us to find bugs faster*.

VII. THREATS TO VALIDITY

We evaluated MUTAGEN in three different scenarios that require generating highly-structured inputs, where it was able to find several planted and real previously-unknown bugs. In particular, we compared our tool against all the existing case studies previously considered by Lampropoulos et al.. However, we cannot generalize that our tool will be effective at finding bugs in other scenarios. To compensate, MUTAGEN is a fully-automated tool that synthesizes all the needed boilerplate, making it an appealing alternative whenever *QuickCheck* is unable to penetrate properties with sparse preconditions.

As mentioned in §V, our evaluation required us to translate both cases studies used by Lampropoulos et al. from Coq to Haskell. Although this translation was performed mechanically, we do not have formal guarantees that our Haskell version of the code behaves exactly as the original one.

VIII. RELATED WORK

Automated Random Data Generation: DRAGEN [34] is a meta-programming tool that synthesizes random generators from data types definitions, using stochastic models to predict and optimize their distribution toward user-defined target ones. DRAGEN2 [33] extends this idea with support for extracting library APIs and function input patterns from the codebase. *QuickFuzz* [17, 18] is a fuzzer that exploits these ideas to synthesize random generators from existing Haskell libraries, which are combined with off-the-shelf low-level fuzzers to find bugs in heavily used programs.

Automatically deriving random generators is substantially more complicated when the generated data must satisfy sparse preconditions. Claessen et al. [11] developed an algorithm for generating inputs constrained by boolean preconditions with almost-uniform distribution. Lampropoulos et al. [27] extended this approach by adding a limited form of constraint solving controllable by the user. Recently, Lampropoulos et al. [28] proposed a mechanism to obtain constrained generators automatically from inductively defined relations in Coq.

All these generational approaches are somewhat orthogonal to the ideas behind MUTAGEN, and while our tool is tailored to improve the performance of poor automatically derived generators, it can benefit from using better generators to find initial (valid) interesting seeds faster.

Coverage-Guided Fuzzing: *AFL* [32] is the reference tool when it comes to coverage-guided fuzzing. *AFLFast* [4] extends AFL using Markov chain models to tune the power scheduler toward testing low-frequency paths. MUTAGEN's scheduler is deliberately simple and does not account for path frequency — future work should investigate this possibility. *CollAFL* [15] is a variant of AFL that uses path- instead of edge-based coverage to distinguish executions more precisely by reducing path collisions. We tested this approach in MUTAGEN and found that some bugs can be found faster and more reliably using a prefix-tree-based prioritization of interesting test cases. However, storing the trace of every executed test case a in prefix tree consumes large amounts of memory and the lookup performance heavily degrades over time. Future work should investigate the tradeoffs of this approach in depth.

Havrikov and Zeller [21] have proposed an algorithm that uses input grammars to systematically cover the input space in a bounded fashion, which closely relates to our approach given the similarities between input grammars and values described by algebraic data types. However, MUTAGEN uses the execution trace feedback to decide when to grow recursive grammar nodes one step at a time, whereas the approach by Havrikov and Zeller unfolds these steps into exhaustively testing k-grams pairs of grammar constructions.

BeDivFuzz [36] is a fuzzing approach that separates mutations into "structure-changing" and "structure-preserving", which closely relate to MUTAGEN's pure and random mutant kinds, respectively. BeDivFuzz uses this distinction to search for diverse input structures (via structure-changing mutations) and then apply structure-preserving mutations to them to produce structure-equivalent variants. In turn, MUTAGEN uses this distinction to avoid testing *every* structure-equivalent variant exhaustively.

Zest [38] and Crowbar [13] are two fuzzing tools that mutate the bits representing the pseudo-random choices taken by the input generators instead of relying on specialized structure-preserving mutators. While our approach carries the burden of synthesizing such mutators, it allows us to implement future high-level optimizations, e.g, leveraging lazy evaluation to avoid mutating unevaluated parts of an interesting test case.

Exhaustive Bounded Testing: A different category of property-based testing tools does not rely on randomness. Instead, test cases are exhaustively enumerated and tested from smaller to larger up to a certain size bound. *Feat* [14] 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. This bijection can be traversed exhaustively or, more interestingly, randomly accessed. This allows the user to easily generate random test cases uniformly simply by sampling natural numbers. However, test cases are enumerated based only on their type definition, so this technique is not suitable for testing properties with sparse preconditions expressed elsewhere. *SmallCheck* [43] is a Haskell tool that also follows this approach. It progressively executes the testing properties against all possible input test cases of up to

a certain depth. Similarly, *Korat* [5] is a Java tool that uses method specification predicates to automatically generate all non-isomorphic test cases up to a given small size.

These approaches rely on pruning mechanisms to avoid generating too many equivalent test cases before their exhaustiveness becomes impractical. *LazySmallCheck* is a variant of *SmallCheck* that uses lazy evaluation to automatically prune the search space by detecting unevaluated subexpressions. In *Korat*, pruning is done by instrumenting method precondition predicates and analyzing which parts of the execution trace correspond to each evaluated subexpression.

Our tool uses exhaustiveness as a way to reliably enforce that all possible mutants of an interesting test case are scheduled. In contrast to fully-exhaustive tools, MUTAGEN relies on randomly generated test cases as a shortcut to find initial interesting test cases without enumerating them exhaustively. MUTAGEN additionally supports lazy pruning, i.e., it can detect unevaluated subexpressions and avoid producing mutations over their corresponding positions. This can improve the overall performance when testing non-strict properties. In our case studies, however, the precondition of the testing properties fully evaluate their inputs before executing the postconditions, thus we avoided including this optimization in our evaluation. Our future work will investigate the effect of MUTAGEN's lazy pruning against non-strict testing properties.

IX. CONCLUSIONS

We presented MUTAGEN, a coverage-guided, property-based testing tool that extends the original CGPT approach with an exhaustive mutation mechanism that generates every possible mutant for each interesting test case, scheduling them to be tested exactly once. Our experimental results indicate that MUTAGEN outperforms the simpler CGPT approach implemented in *FuzzChick* in terms of both failure rate and tests until first failure. Moreover, we show how our tool can be applied in a real-world testing scenario, where it quickly discovers 15 planted and 3 previously unknown bugs.

REFERENCES

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

[2] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hriţcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach. A verified information-flow architecture. *SIGPLAN Not.*, 49(1):165–178, January 2014. ISSN 0362-1340. doi: 10.1145/2578855.2535839.

[3] Maciej Bendkowski, Katarzyna Grygiel, and Paul Tarau. Boltzmann samplers for closed simply-typed lambda terms. In *In Proceedings of International Symposium on Practical Aspects of Declarative Languages*. ACM, 2017.

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

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

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

[7] CACA Labs. zzuf - multi-purpose fuzzer. http://caca.zoy.org/wiki/zzuf, 2010.

[8] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A survey of compiler testing. *ACM Computing Surveys*, 53(1), 2020. ISSN 0360-0300. doi: 10.1145/3363562.

[9] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, page 73–88, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 1581133898. doi: 10.1145/502034.502042.

[10] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2000.

[11] Koen Claessen, Jonas Duregård, and Michał H. Palka. Generating constrained random data with uniform distribution. In *Proceedings of the Functional and Logic Programming FLOPS*, 2014.

[12] Maxime Dénès, Cătălin Hriţcu, Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. Quickchick: Property-based testing for coq. In *The Coq Workshop*, 2014.

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

[14] Jonas Duregård, Patrik Jansson, and Meng Wang. Feat: Functional enumeration of algebraic types. In *Proceedings of the ACM SIGPLAN International Symposium on Haskell*, 2012.

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

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

[17] Gustavo Grieco, Martín Ceresa, and Pablo Buiras. QuickFuzz: An automatic random fuzzer for common file formats. In *Proceedings of the ACM SIGPLAN International Symposium on Haskell*, 2016.

[18] Gustavo Grieco, Martín Ceresa, Agustín Mista, and Pablo Buiras. QuickFuzz testing for fun and profit. *Journal of Systems and Software*, 134, 2017.

[19] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th*

ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 185–200, 2017.

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

[21] Nikolas Havrikov and Andreas Zeller. Systematically covering input structure. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, page 189–199. IEEE Press, 2019. ISBN 9781728125084. doi: 10.1109/ASE.2019.00027.

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

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

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

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

[26] Rody Kersten, Kasper Luckow, and Corina S Păsăreanu. Poster: AFL-based fuzzing for java with kelinci. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2511–2513, 2017.

[27] Leonidas Lampropoulos, Diane Gallois-Wong, Cătălin Hriţcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. Beginner's luck: a language for property-based generators. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages, POPL*, 2017.

[28] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. Generating good generators for inductive relations. *In Proceedings ACM on Programming Languages*, 2(POPL), 2017.

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

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

[31] Lambert Meertens. First steps towards the theory of rose trees. *CWI, Amsterdam*, 1988.

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

[33] Agustín Mista and Alejandro Russo. Generating random structurally rich algebraic data type values. In *Proceedings of the 14th International Workshop on Automation of Software Test*, 2019.

[34] Agustín Mista, Alejandro Russo, and John Hughes. Branching processes for quickcheck generators. In *Proceedings of the ACM SIGPLAN International Symposium on Haskell*, 2018.

[35] Agustín Mista and Alejandro Russo. MUTAGEN: Reliable Coverage-Guided, Property-Based Testing using Exhaustive Structure-Preserving Mutations (Replication Package), 2022. URL https://doi.org/10.5281/zenodo.7197927.

[36] Hoang Lam Nguyen and Lars Grunske. Bedivfuzz: Integrating behavioral diversity into generator-based fuzzing. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 249–261, 2022. doi: 10.1145/3510003.3510182.

[37] Oulu University Secure Programming Group. A Crash Course to Radamsa. https://github.com/aoh/radamsa, 2010.

[38] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 329–340, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362245. doi: 10.1145/3293882.3330576.

[39] Michał Pałka, Koen Claessen, Alejandro Russo, and John Hughes. Testing an optimising compiler by generating random lambda terms. In *The IEEE/ACM International Workshop on Automation of Software Test (AST)*, 2011.

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

[41] Árpád Perényi and Jan Midtgaard. Stack-driven program generation of webassembly. In *Asian Symposium on Programming Languages and Systems*, pages 209–230. Springer, 2020.

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

[43] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *ACM SIGPLAN Notices*, volume 44, pages 37–48. ACM, 2008.

[44] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003. doi: 10.1109/JSAC.2002.806121.

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

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

[47] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Pro-*

ceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, pages 283–294, 2011.