Coverage Guided, Property Based Testing

LEONIDAS LAMPROPOULOS, University of Maryland, USA and University of Pennsylvania, USA
MICHAEL HICKS, University of Maryland, USA
BENJAMIN C. PIERCE, University of Pennsylvania, USA

Property-based random testing, exemplified by frameworks such as Haskell's QuickCheck, works by testing an executable predicate (a *property*) on a stream of randomly generated inputs. Property testing works very well in many cases, but not always. Some properties are conditioned on the input satisfying demanding semantic invariants that are not consequences of its syntactic structure—e.g., that an input list must be sorted or have no duplicates. Most randomly generated inputs fail to satisfy properties with such *sparse preconditions*, and so are simply discarded. As a result, much of the target system may go untested.

We address this issue with a novel technique called *coverage guided, property based testing (CGPT)*. Our approach is inspired by the related area of coverage guided fuzzing, exemplified by tools like AFL. Rather than just generating a fresh random input at each iteration, CGPT can also produce new inputs by mutating previous ones using type-aware, generic mutation operators. The target program is instrumented to track which control flow branches are executed during a run and inputs whose runs expand control-flow coverage are retained for future mutations. This means that, when sparse conditions in the target are satisfied and new coverage is observed, the input that triggered them will be retained and used as a springboard to go further.

We have implemented CGPT as an extension to the QuickChick property testing tool for Coq programs; we call our implementation FuzzChick. We evaluate FuzzChick on two Coq developments for abstract machines that aim to enforce flavors of noninterference, which has a (very) sparse precondition. We systematically inject bugs in the machines' checking rules and use FuzzChick to look for counterexamples to the claim that they satisfy a standard noninterference property. We find that vanilla QuickChick almost always fails to find any bugs after a long period of time, as does an earlier proposal for combining property testing and fuzzing. In contrast, FuzzChick often finds them within seconds to minutes. Moreover, FuzzChick is almost fully automatic; although highly tuned, hand-written generators can find the bugs faster than FuzzChick, they require substantial amounts of insight and manual effort.

Additional Key Words and Phrases: random testing, property-based testing, fuzz testing, coverage, QuickChick, AFL, FuzzChick

1 INTRODUCTION

Random testing methods probe a system's behavior using randomly generated inputs. In *property-based* random testing, illustrated in Figure 1a, the system's expected behavior is specified as a collection of *properties*—executable boolean predicates over inputs. For example, in QuickChick [Dénès et al. 2014; Lampropoulos and Pierce 2018; Paraskevopoulou et al. 2015a,b], a modern property-based random tester for the Coq proof assistant, the following property states that the `sort` function should always produce sorted lists:

```
Definition prop_sort_correct (l : list nat) : bool := is_sorted (sort l).
```

QuickChick will repeatedly generate random lists and apply `prop_sort_correct` to each one, continuing until either the property returns `false` or the process times out. QuickChick produces random values by invoking a *generator* function of the appropriate type. While generators can be written by hand, doing so requires nontrivial time and expertise; therefore, most property-based

Authors' addresses: Leonidas Lampropoulos, University of Maryland, USA , University of Pennsylvania, USA, llamp@seas.upenn.edu; Michael Hicks, University of Maryland, USA, mwh@cs.umd.edu; Benjamin C. Pierce, University of Pennsylvania, USA, bcpierce@cs.upenn.edu.

2019. 2475-1421/2019/1-ART1 $15.00
https://doi.org/

testing tools provide mechanisms for synthesizing them automatically [Bulwahn 2012a; Claessen and Hughes 2000; Papadakis and Sagonas 2011]. For example, QuickChick can use Coq's typeclass mechanism to synthesize an appropriately typed generator using a predefined library of generator combinators [Lampropoulos et al. 2018].

A weakness of synthesized generators is that they are less effective than hand-crafted ones at finding bugs when properties have *sparse preconditions*. Here is an example of such a property:

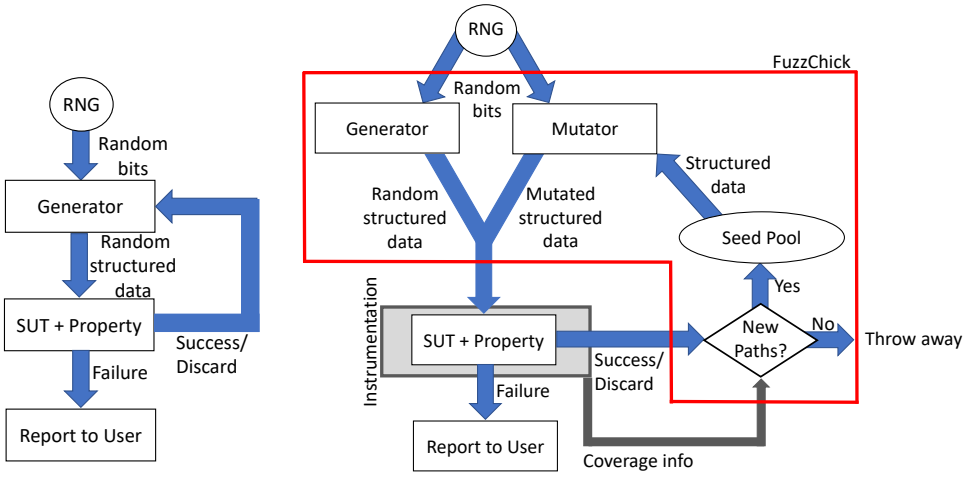
```
Definition prop_insert_correct (x : nat) (l : list nat) : bool :=
  is_sorted l ==> is_sorted (insert x l).
```

This property says that if a list `l` is sorted, then so should the list produced by `insert x l`. The `==>` operator denotes a property with a precondition: if its left-hand side is `true`, then the right-hand predicate is tested; otherwise the property holds vacuously. For this example, QuickChick will generate many random lists, but `insert` will only be tested on those lists that happen to be sorted. The vast majority of such lists will be very small, since the probability that a random list is sorted decreases quickly with its length. Accordingly, most of the generated tests will succeed vacuously, and the right-hand side of the implication will rarely be checked.

To make property testing with synthesized generators more effective in such cases we take inspiration from research on *fuzz testing* (or *fuzzing*). This technique, first conceived by Miller et al. [1990], works by repeatedly feeding random bytes to a target program and seeing whether it crashes (with an OS-level fault). If the target program imposes structural constraints on its input then the core program logic will only be tested when those constraints are satisfied. If the constraints are *sparse*, then input bytes generated in a purely random fashion are unlikely to satisfy them, limiting the power of random tests to find bugs. Sound familiar?

The fuzzing community has addressed this problem by developing a technique called *coverage guided fuzzing* (CGF), exemplified in the popular fuzzer AFL [2019]. Rather than generate each new input from scratch, an input is obtained by *mutating* a prior test's input. The process starts with a *seed* input provided by the test engineer. The target program is instrumented (e.g., during compilation) to efficiently track which control-flow edges are traversed when executing a given test. If this input causes the program to traverse a previously unvisited edge, then it is deemed "interesting" and retained as an object of future mutations. Doing so allows the fuzzer to gradually explore many of the program's control paths, especially ones that are reached by relatively few inputs. In particular, if the program carries out a test that is satisfied by few inputs, a CGF tool will remember such an input when it arises and use it as a springboard to cover more paths. CGF easily outperforms completely random input generation, and it has even been shown to automatically "reverse engineer" fairly complicated input formats [lcamtuf 2019b].

This paper introduces *coverage guided, property based testing* (CGPT), a novel combination of property testing and CGF. We have implemented this technique in FuzzChick, a redesign of QuickChick. FuzzChick is depicted in Figure 1b. As with ordinary fuzzing, the property and the code it tests are instrumented to gather control-flow coverage information. This information is used to determine whether the prior input is interesting, in which case it will be repeatedly mutated to produce subsequent inputs. FuzzChick synthesizes a standard random generator, which it uses to produce the initial seed input (the test engineer does not have to provide one). It also synthesizes a collection of *mutators*, each inspired by the kinds of mutators used in traditional fuzzing. FuzzChick mutators are type-aware: instead of mutating binary streams, they mutate inputs at the algebraic datatype level while preserving their type, which allows the mutations to be more targeted and effective. They are applied during the testing process on seeds that were considered interesting to obtain new ones, which are then in turn checked for new paths and mutated. If mutation-based



(a) QuickChick (b) FuzzChick

Fig. 1. QuickChick vs FuzzChick workflows. QuickChick (left) uses a simple testing loop: a generator produces random structured data from a standard source of pseudo-random bits, and this data is used to test the desired property. On success, repeat; on failure, report the counterexample. FuzzChick (right) instead instruments the property being tested (including the system under test (SUT), but not including the testing framework itself). Coverage information is used to determine whether the input (a data structure) is interesting or not. In addition, the tool decides between mutating an existing interesting input (if one exists) or generating new inputs randomly to reset the process. Both the mutator and the generator operate at the level of algebraic data types.

input generation covers no new paths for a long time, FuzzChick will just call the normal generator. This serves to "reset" the state space exploration process, escaping from local minima.

Revisiting the sorted list example, suppose the random generator has produced the list `[1;3;2;4]`. In a pure random setting, this test would be immediately discarded because it is not sorted. In FuzzChick, however, it can be mutated to yield a few "similar" lists. For instance, it might be changed to either `[1;3;0;4]` or `[1;3;5;4]` by mutating its third element. In the first case, testing with the mutated list would cover no new paths, and the mutated test would be thrown away. In the latter case, we would be able to cover more branches of the `is_sorted` predicate, since the list contains a larger sorted prefix and FuzzChick treats visiting a branch more times as an increase in coverage. This new list could in turn be reused, bringing us one step closer to a sorted input!

We evaluated FuzzChick by using it to test two formalized developments of *secure machines* (§ 4). The most complicated machine spans more than 10k lines of Coq code, including definitions and specifications (∼ 2000 LoC), testing infrastructure (∼ 2000 LoC), and proofs (∼ 6000 LoC). Both machines aim to enforce flavors of noninterference. Informally, noninterference states that an external observer without access to secret information cannot distinguish between two runs of a program that differ only in such secret information. As such, noninterference is a perfect example of a very sparse property: it requires randomly generating pairs of states that are identical in all of their "public" locations. Both machines have been proved correct [Azevedo de Amorim et al. 2014], so in their original state they do not have any bugs left to find. Therefore, we systematically inject bugs in the machines' security checks and then see whether random testing can produce

programs that are evidence of a noninterference violation. We measure effectiveness as *mean time to failure*—i.e., how long does it take to produce an example violation?

We compare the performance of FuzzChick against two alternative, fully automated property-based random testers: QuickChick and an adaptation of Crowbar [2017], a previously proposed integration of the AFL fuzzer and a property testing framework. Crowbar works by using AFL to fuzz the source of randomness used by a property tester's generator, rather than the inputs directly as FuzzChick does. (Crowbar is described in detail in § 2.2.) As Crowbar was originally developed for use with its own simplistic OCaml-based property tester, we could easily adapt it to work with QuickChick. Both these two and FuzzChick use automatically synthesized generators, while FuzzChick also uses automatically synthesized mutators.

On these challenging applications, FuzzChick significantly outperforms both QuickChick and Crowbar. For the simpler of the two machines, FuzzChick discovers most bugs within seconds, while the other two systems mostly time out after an hour. For the more complicated and realistic one, FuzzChick requires a few minutes for most bugs, while the other systems didn't find most of the bugs within eight hours. We also compared the performance of FuzzChick against QuickChick when using a collection of highly tuned, *hand-written* generators [Hriţcu et al. 2013b, 2016]. With them, QuickChick finds most bugs extremely quickly, in less than one second. However, the human cost of such generators is high: they took most of a person-year to develop and comprise almost 1000 LOC. In short, FuzzChick represents a strong advance in automated input generation, but there is still opportunity to improve.

In summary, we offer the following contributions:

- We present coverage guided, property based testing (CGPT), a novel combination of property testing and coverage guided fuzzing. We have implemented CGPT in FuzzChick, an extension of the QuickChick property-based random testing tool for Coq (§ 3). FuzzChick synthesizes type-specific mutation operators, drawing inspiration from and generalizing AFL's bit-level operators. It also synthesizes traditional input generators, which it uses both to produce an initial seed and to restart when testing gets stuck in a local minimum.
- We evaluate FuzzChick by using it to test the formal development of two abstract machines, one simple and one more realistic, which aim to enforce a non-interference property (§ 4). The evaluation systematically modifies these machines so that they omit various security checks, and measures how quickly randomly testing the noninterference property can uncover them. Compared against QuickChick and Crowbar, FuzzChick is far more effective: it discovers most injected bugs orders of magnitude faster. FuzzChick is still equally far away from the efficiency of expertly hand-written generators, but incurs far less manual work (§ 5).

We discuss related work in § 6 and conclude in § 7.

2 BACKGROUND

We begin by describing QuickChick, a property-based random tester for Coq that serves as our starting point. We also describe Crowbar [2017], a prior approach to combining property testing and fuzzing, and our reimplementation of it, which we use as a point of comparison in the evaluation section. Our main contribution—the idea of coverage guided, property-based testing (CGPT) implemented in the tool FuzzChick—builds on these foundations; it is presented in § 3.

2.1 QuickChick

Coq is a widely used proof assistant that has been used to prove the correctness of numerous complex software systems including the CompCert optimizing C compiler [Leroy 2009] and the CertiKOS hypervisor [Gu et al. 2016]). QuickChick [Lampropoulos 2018; Lampropoulos and Pierce

2018; Paraskevopoulou et al. 2015a] is a tool that integrates random testing into Coq; its goal is to help proof engineers iron out most of the bugs in both their programs and their specifications before spending the (often significant) energy required to prove correctness formally. It works by attempting to automatically generate counterexamples to hypothesized theorems. If no such counterexample can be found, the proof engineer can proceed with greater confidence.

QuickChick originally started as a straightforward clone of Haskell's QuickCheck tool [Claessen and Hughes 2000] and still shares the same overall architecture, shown in Figure 1a. To test a piece of software, QuickChick requires four ingredients:

- an *executable property* that is repeatedly tested against inputs of some type t,
- a *generator* for producing random inputs of type t,
- a *printer* of type $t \longrightarrow$ `string` for reporting any counterexamples that are found, and
- a *shrinker* of type $t \longrightarrow$ `list` t to "minimize" counterexamples before reporting them.

In Figure 1a, we group printers and shrinkers together under "Report to user," as they are not the focus of this paper and are identical across the different variants that we study.

The main loop of QuickChick, shown in Algorithm 1, is extremely simple: given a property P, a generator *gen* and an upper limit for the number of tests, QuickChick keeps generating random inputs until one fails or the limit is reached.

Algorithm 1 QuickChick Testing Loop

```
function testLoop(P, gen, maxTests)
    i ← 0
    while i < maxTests do                          ▷ Loop until test limit
        x ← gen                                    ▷ Generate an input
        result ← P (x)                             ▷ Run the property over the input
        if !result then return Bug x               ▷ Bug Found
        end if
    end while
    return NoBug
end function
```

As an example of how QuickChick might be used, suppose we want to use Coq to formalize and ultimately prove a simple property involving binary trees (with payloads at both nodes and leaves—this choice allows us to showcase several aspects of our mutation operators later). The `Tree` type definition and a `mirror` function, which mirrors a tree by swapping its children recursively, can be written as follows. [1]

```
Inductive Tree A :=
| Leaf : A -> Tree A
| Node : A -> Tree A -> Tree A -> Tree A.

Fixpoint mirror {A : Type} (t : Tree A) : Tree A :=
  match t with
    | Leaf x => Leaf x
    | Node x l r => Node x (mirror r) (mirror l)
```

[1] In Coq, putting curly braces around a function parameter declaration, like `{A}` here, is a request to the typechecker to infer this parameter, allowing it to be omitted at use sites. The keyword `Inductive` introduces an algebraic datatype declaration: in this case, the datatype has constructors `Leaf` and `Node`, each of which takes parameters of appropriate type and yields a tree. The `Fixpoint` keyword introduces a recursive function.

```
  end.
```

For our property, suppose we want to show that `mirror` is an involution: i.e., mirroring a tree twice yields the original tree.

```
Definition mirror_twice {A : Type} (t : Tree A) :=
  mirror (mirror t) == t.
```

Before trying to prove this property, we can use QuickChick to search for possible counterexamples. To do so, we need to define a generator, printer, and shrinker for elements of type `Tree A`. We can do this by invoking the command

```
Derive (GenSized, Show, Shrink) for Tree.
```

The `Derive` command uses the representation of the `Tree` type to automatically derive the three needed components. QuickChick can then use these components (implicitly—they will be supplied by Coq's typeclass inference mechanism, e.g., in the following command) when invoking the `QuickChick` command at the top level:

```
QuickChick mirror_twice.
```

This command implements the testing loop described above; it runs 10000 tests (by default) and reports that all of them succeeded. It also reports that none were discarded because of precondition failure (as expected, since `mirror_twice` doesn't have a precondition).

```
QuickChecking mirror_twice...
+++ Passed 10000 tests (0 discards)
```

If we had made a mistake in our definition of `mirror`, say by mirroring the left subtree twice in the `Node` branch...

```
Fixpoint mirror {A : Type} (t : Tree A) : Tree A :=
  match t with
    | Leaf x => Leaf x
    | Node x l r => Node x (mirror l) (mirror l)
  end.
```

...QuickChick would complain, outputting a (minimal) counterexample.

```
QuickChecking mirror_prop
Node 0 (Leaf 1) (Leaf 0)
*** Failed after 4 tests and 5 shrinks. (0 discards)
```

Let's now take a closer look at QuickChick's implementation, and in particular, at its automatically derived generators. Coq includes a built-in functional programming language, which we've used above to write our program and property. To execute programs in this language efficiently, and to get access to "non-logical" run-time facilities like IO and randomness, we extract Coq programs to OCaml and link them with a run-time library that provides these features. For the rest of this paper, we keep our use of Coq-specific features to a minimum, so that a reader unfamiliar with Coq can think of QuickChick and the target program as if they were implemented in OCaml.

In essence,[2] a QuickChick generator that produces inputs of some type `A` is just a function from a random seed type (intuitively, an integer) to the output type. Its type `G A` is defined as follows:

```
Definition G A := RandomSeed -> A.
```

[2]To streamline this discussion, we are eliding an additional parameter to real QuickChick generators that controls the size of generated structures.

The most basic block by which we build generators is `ret` (short for return). Given an element `x` of type `A`, the expression `ret x` represents a singleton probability distribution: no matter what random seed `r` it is given, `(ret x) r` will always produce `x`.

```
Definition ret {A : Type} (x : A) : G A :=
  (fun r => x).
```

(The `fun` keyword introduces an anonymous function.) We also need to compose generators: given a generator for elements of type `A` and a function that, given an `A` produces a generator for `B`s, we can `bind` them together to form a generator for `B`s.[3]

```
Definition bind {A B : Type} (g : G A) (k : A -> G B) : G B :=
  (fun r =>
    let (r1,r2) := randomSplit r in
    let a := g r1 in (k a) r2).
```

To compose generators, we need to first run `g` to obtain an element `a` of type `A`, and then run `(k a)` to obtain a `B`. One subtlety is that these two generators should be given independent random seeds to avoid introducing bias in the resulting distribution. We use `randomSplit` for this: given an input seed `r` it produces a pair of statistically independent seeds `r1` and `r2` [Claessen and Pałka 2013]. To make programs easier to read, we use the notation `x <- e1 ;; e2` to mean `bind e1 (fun x => e2)`.

QuickChick provides a useful library of generator combinators, inspired by the ones in Haskell's QuickCheck. These combinators can be used if desired to write generators by hand; more importantly for present purposes, they are also used by the `Derive` command when synthesizing generators for arbitrary algebraic data types. The most common and expressive one is `freq` (short for frequency):

```
Definition freq {A : Type} : list (nat * G A) -> G A.
```

As its type says, `freq` takes a list of generators, each associated with a natural number that is interpreted as a weight. It picks one of these generators at random, based on the discrete distribution induced by the weights.

As mentioned earlier, QuickChick can automatically derive a generator for a type based on the structure of its definition. We need not dig into the details of the derivation algorithm here, but let's look at generator it produces for `Tree A` (modulo a bit of prettification). The derived generator produces `Tree`s up to some `size` limit. Since `Tree`s are parametric it also needs an argument, `arbitrary`, that can produce random instances of the inner type.[4]

```
Fixpoint genSizedTree {A : Type} (arbitrary : G A) (size : nat) : G (Tree A) :=
  match size with
  | O =>
     x <- arbitrary;;
     ret (Leaf x)
  | S size' =>
     freq [ (1, x <- arbitrary;;
                ret (Leaf x))
          ; (size, x <- arbitrary;;
                   l <- genSizedTree size';;
                   r <- genSizedTree size';;
```

[3]Haskell-inclined readers will recognize that `G` is similar to the Reader monad.

[4]In the real implementation of QuickChick, the `arbitrary` parameter is provided automatically by Coq's typeclass mechanism. We've converted it to an ordinary parameter here to make the presentation more accessible.

```
                    ret (Node x l r))
          ]
  end.
```

This generator begins by matching on its `size` parameter: if `size` is zero, then it can only create a `Leaf`; if non-zero, it has a choice. To produce a `Leaf` it needs to also create an element of type `A` to use as the payload. This is done using the `arbitrary` function. The `O` branch of the match generates an `arbitrary` value for `x` and `ret`urns it wrapped in a `Leaf` constructor. To produce a `Node`, we need to generate an element `x` of type `A` using `arbitrary`, and we need to generate its left and right subtrees. We do this by recursively calling `genSizedTree`, but with a smaller size parameter. In the `S` branch of the match, we can either produce a `Leaf` or a `Node`. We make the choice using `freq`, skewing the distribution more heaving towards `Node`s when the `size` parameter is large: The weights given to `freq` will result in a `Leaf` $\frac{1}{\texttt{size}+1}$ of the time and a `Node` the remaining $\frac{\texttt{size}}{\texttt{size}+1}$ of the time.

2.2 Crowbar and QcCrowbar

While automatically derived generators are useful, they do not work well when testing properties with sparse preconditions. Consider for example binary search trees, which satisfy a property similar to `prop_insert_correct` from the introduction: inserting an element in a binary search tree yields a binary search tree.

```
Definition bst_insert_correct (x : nat) (t : Tree nat) :=
  is_bst t ==> is_bst (bst_insert x t).
```

If we attempt to test that property with the derived generator, QuickChick gives up:

```
QuickChecking bst_insert_correct...
*** Gave up! Passed only 1281 tests
Discarded: 20000
```

Only roughly 6% out of all trees randomly generated by `genSizedTree` turn out to be valid according to `is_bst`, so the `is_bst (bst_insert x t)` portion of the property is rarely tested.

Fuzz testing tools face a similar problem when testing programs operating on structured inputs: since valid inputs are rarely produced randomly, most randomly generated inputs will be quickly discarded by the target program, and little of its code will be tested. Modern tools like AFL [2019] address this problem by using *coverage-guided fuzzing* (CGF). The idea is to track which portions of the target program's code are executed ("covered") during a test, and to retain inputs that cover new parts of the code. Future inputs are generated by mutating these retained, "interesting" inputs. Can we employ a similar idea to improve the efficacy of testing properties with sparse preconditions?

A very clever recent attempt at doing so appeared in Crowbar [2017], which performs property testing on OCaml programs. It is depicted in Figure 2. At a high level, Crowbar uses AFL to fuzz the stream of random bits that controls the generators used by property-based testing. To achieve this effect, Crowbar simply replaces the random number generator by a stream of bytes that it reads from a file. The testing framework raises an exception when the property under test is invalidated, so the whole program can be fuzzed using an out-of-the box fuzzer such as AFL. Crowbar uses an OCaml compilation switch that adds AFL instrumentation, and gives this program to AFL.

The testing loop for Crowbar is exactly that of AFL—a sketch is shown in Algorithm 2. Given a property P and an input seed corpus S, Crowbar keeps picking a seed from S, deciding how many times to mutate it (in the fuzzing literature, this amount is often called the *energy* of a seed and different ways of calculating this energy are called *power schedules*), mutating it and then either

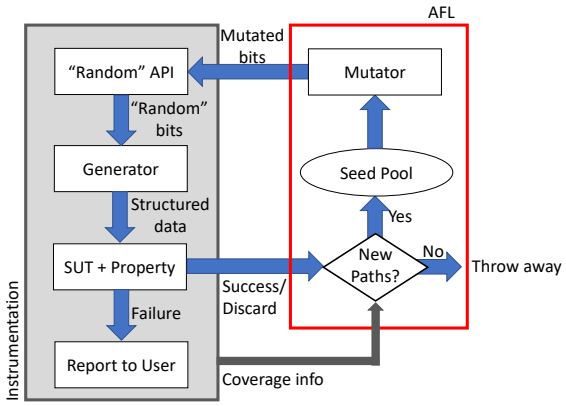


Fig. 2. QcCrowbar Workflow

reporting a crash (violations of the property),[5] or updating the seed corpus with new seeds that were deemed interesting (because they explored new branches). This process is repeated until a predefined time limit is reached.

Algorithm 2 Crowbar Fuzzing Loop

```
function TestLoop(P, S)
    repeat
        s ← nextSeed(S)                          ▷ Pick the next seed to mutate
        p ← calcEnergy(s)                        ▷ Power Schedule
        for i = 1 to p do
            s′ ← mutate(s)                       ▷ Mutate the input
            if !P(s′) then return Bug s′         ▷ Bug Found
            else if isInteresting (s′) then
                S ← S ∪ {s′}                     ▷ Add to queue
            end if
        end for
    until Timeout
    return NoBug
end function
```

As QuickChick also targets OCaml via extraction from Coq, it is not too difficult to adapt the Crowbar approach for QuickChick; we call the result QcCrowbar. We retargeted the extraction of `RandomSeed` to type `rnd` and implemented a few wrappers for the rest of the randomness primitives that QuickChick uses. The entire executable produced is then a valid target for AFL.

As we will see in § 5, QcCrowbar is roughly as effective as plain QuickChick with completely random test case generation—i.e., not very effective, when the properties involved become more

[5]The actual AFL implementation, rather than returning the first counter-example, will continue fuzzing to try to uncover more bugs until a given time limit is reached. Here, for ease of comparison with the other systems, we focus on the generation aspect of the AFL fuzzing loop.

complex. The reason is that the binary input that is used as a source of randomness can only be modified by AFL *at the bit level*; such bit-level manipulations of the input don't have any obvious correspondence to changes in the distribution of the structured data that is eventually fed into the target program. AFL's cleverly crafted mutations normally fiddle with the bits of a data structure; in the Crowbar approach, they fiddle with the randomness used to *generate* a data structure. Therefore, the changes that it makes can have much larger and less predictable effects on the structure that gets generated.

Crowbar has other drawbacks, too. The way the testing infrastructure is set up, whenever we want to test a command using QcCrowbar, a single OCaml file is produced that contains, in addition to the property under test, all of the operational logic of QuickChick, including generation, printing and shrinking. For concreteness, a standard size of an extracted file is roughly 4000-6000 lines. Ideally, we would prefer to instrument only the property itself: we want to cover the precondition, in order to guide generation towards non-discarded inputs, and the property itself, to guide generation towards non-explored paths—we do not care if the printing code or other parts of QuickChick itself are covered. This means that a large part of the coverage information provided is unnecessary, while incurring significant instrumentation overhead. This, together with the expensive decision-making logic and the repeated memory accesses for seed manipulation, result in a significant overhead compared to pure random execution. On our more complex case study (§ 4), QuickChick using hand-written random generators variants is able to execute roughly 40× more tests per second!

These somewhat disappointing observations are not criticisms of AFL, which was built and optimized for a different use case. (Indeed, the fact that AFL works at all in this unusual setup is a testament to its amazing engineering!) Still, they leave open the question of whether the general techniques developed for advanced fuzzing tools can be applied to effective property-based testing.

We answer this question in the affirmative. To do so, the main idea that we need is that, rather than relying on bit-level manipulations to mutate the source of randomness coming *into* the generator, we should mutate the high-level outputs coming *out* of it. Enter CGPT.

3 COVERAGE GUIDED, PROPERTY-BASED TESTING IN FUZZCHICK

In this section we introduce coverage guided, property-based testing (CGPT), an integration of property testing with coverage guided fuzzing (CGF). We do so via FuzzChick, an extension of QuickChick that uses CGPT. We discuss FuzzChick's testing loop first (§ 3.1), and then discuss our approach to automatically deriving the mutation operations that it uses (§ 3.2). We will keep using the binary trees of the previous section as our running example.

3.1 CGPT Testing Loop

The main idea behind CGPT is that testing inputs should only be rarely produced by generators; more often they should be produced by mutators operating on seeds chosen by a coverage guided fuzzing process. This is shown in Figure 1b. In FuzzChick, both generators and mutators are synthesized automatically based on the input type. The generator synthesis is standard; we discuss our algorithm for synthesizing mutators in the next subsection.

The CGPT testing loop is given in Algorithm 3. We generate an input x using (mostly) mutation operators, run the property, and check the result. If the property fails, we report the counterexample to the user. If not, we check whether the input is interesting or not (i.e., whether it exercises new paths based on coverage information), assign it some energy, and add it to the appropriate queue. At a high level, this loop is fairly similar to the one in AFL and the one shown for Crowbar in the previous section (Algorithm 2).

The devil, of course, is in the details. The two main differences from Crowbar are the search strategy (how CBPT picks the next seed to mutate) and the generation of inputs (whether we use

Algorithm 3 CGPT Testing Loop

```
function testLoop(P, gen, mut, maxTests)
    QSucc, QDisc ← ∅                                   ▷ Initialize queues
    while i < maxTests do                              ▷ Loop until test limit
        g ← pick(QSucc, QDisc, gen, mut)               ▷ Pick how to generate next input
        x ← g                                          ▷ Run the generator/mutator
        result ← P (x)                                 ▷ Run the property over the input
        if !result then return Bug x                   ▷ Bug Found
        else if isInteresting (x) then
            e ← calcEnergy (x)                         ▷ Power Schedule
            if !discarded result then
                enqueue(QSucc, (x, e))                 ▷ If not discarded and interesting add to QSucc
            else enqueue(QDisc, (x, e))                ▷ If discarded and interesting add to QDisc
            end if
        end if
    end while
    return NoBug
end function

function pick(QSucc, QDisc, gen, mut)
    if !isEmpty(QSucc) then
        (x, e) ←dequeue(QSucc)                         ▷ Return top, decrease energy by one
        if e > 0 then return mut(x)                    ▷ If there is energy left, mutate
        else return pick(QSucc, QDisc, gen, mut)       ▷ Otherwise, look for another seed
        end if
    else if !isEmpty(QDisc) then
        (x, e) ←dequeue(QDisc)                         ▷ Return top, decrease energy by one
        if e > 0 then return mut(x)                    ▷ If there is energy left, mutate
        else return gen                                ▷ Otherwise generate randomly
        end if
    else return gen                                    ▷ If no seeds exists, generate randomly
    end if
end function
```

mutators or generators). Instead of having a single seed corpus like Crowbar, we take advantage of the fact that we can differentiate between successful and discarded runs. We maintain two queues: one for interesting seeds that succeed in satisfying the precondition (*QSucc*) and one for seeds that were discarded (*QDisc*). While we prioritize seeds that pass the precondition, as nearby inputs will also tend to satisfy it, we do remember discarded inputs that revealed new paths. The idea is that, just like in the sorted example of the introduction, non-discarded inputs can sometimes be discovered by following a sequence of inputs that, though they are discarded themselves, move closer to satisfying the precondition.

The other difference is how the next input will be generated. Notice that CGPT mutations operate at a higher level of abstraction than their lower-level counterparts in CGF tools, or Crowbar. Rather than simply flipping bits in a semantically blind fashion, CGPT mutators operate at a higher level, directly manipulating and constructing values of the desired type. This means there is a more direct relationship between the original value and the mutated one, which should assist the search

process. In addition, inputs need not *only* be produced by mutation, as in CGF. Instead, CGPT can switch between mutation and generation. If there is a seed in the interesting queue that still has energy left, we mutate that one. If not, we alternate between mutating a seed in the discarded queue (if available) and generating completely random inputs. Initially, since the queues are empty, we obtain seeds randomly. As we explore more paths, the queues get populated by seeds and we switch to mutation-based generation. If, at any point, fuzzing appears to become stuck, we fall back to random generation. Thus, the test engineer need not provide an initial seed for the fuzzing process, which is good, as Klees et al. [2018] have shown that a poor choice of initial seed (which can often be hard to judge) can harm overall performance.

Finally, there is a more minor difference in terms of the power schedule: how much we mutate each seed. For the most part, to keep the comparison fair, the energy assigned to each seed is similar to the one that AFL would assign it (that is, more energy for seeds that lead to short executions, or for seeds that exercise a lot of new paths). However, using the same rationale as above, we assign discards proportionally less energy than seeds that lead to successful runs.

3.2 Deriving Mutators

A key element of CGPT is the use of mutation operators for producing new inputs. In FuzzChick, a mutator has type `T -> G T`: given an input seed of some type `T`, a mutator is a random generator for other values of type `T`. Rather than require the programmer to write mutators by hand, we have developed an algorithm for deriving them automatically. We target simple algebraic data types, possibly with type parameters, such as might be found in any functional language (omitting the fancier data types made possible by Coq's dependent type system). Formally, we consider several ways of automatically deriving mutators for a datatype `T` with constructors C_i, each of type $\overline{T_i} \to$ `T`. The type `Tree`, for example, has constructors C_0 = `Leaf`, of type `A` $\to$ `Tree A`, and C_1 = `Node`, of type `A` $\to$ `Tree A` $\to$ `Tree A` $\to$ `Tree A`.

The first way of deriving mutators is *recursive mutation*. That is, we obtain a mutator for an element of `T` by mutating one of its subterms using an appropriate mutator $\texttt{mutate}^{\texttt{T}_k}$, where $\texttt{T}_k$ is the type of the subterm. Given an element of `T` beginning with constructor C_i, we can recursively mutate any one of its arguments and keep the rest unchanged. We call this mutation operator $\texttt{mutate}_r$.[6]

$$\texttt{mutate}_r^{\texttt{T}}\ (C_i\ x_1\ \ldots\ x_n) \;=\; \{C_i\ x_1'\ \ldots\ x_n' \mid k \in \{1 \ldots n\},\ \ x_k' \in \texttt{mutate}^{\texttt{T}_k}\ x_k,\ \ x_j' = x_j \text{ for } j \neq k\}$$

For the binary tree example, both constructors contain subexpressions that can be fuzzed. Since trees are parametric, we need to assume that its parameter type `A` can also be mutated using some function $\texttt{mutate}^{\texttt{A}}$. Concretely, applying $\texttt{mutate}_r^{\texttt{Tree A}}$ to a `Leaf x` yields the following set of mutants:

$$\texttt{mutate}_r^{\texttt{Tree A}}\ (\texttt{Leaf a}) \;=\; \left\{\texttt{Leaf a'} \mid \texttt{a'} \in \texttt{mutate}^{\texttt{A}}\ \texttt{a}\right\}$$

Similarly, applying $\texttt{mutate}_r$ to `Node a l r` yields

$$\begin{aligned}\texttt{mutate}_r^{\texttt{Tree A}}\ (\texttt{Node a l r}) \;=\;& \{\texttt{Node a' l r} \mid \texttt{a'} \in \texttt{mutate}^{\texttt{A}}\ \texttt{a}\} \\ \cup\;& \{\texttt{Node a l' r} \mid \texttt{l'} \in \texttt{mutate}^{\texttt{Tree A}}\ \texttt{l'}\} \\ \cup\;& \{\texttt{Node a l r'} \mid \texttt{r'} \in \texttt{mutate}^{\texttt{Tree A}}\ \texttt{r'}\},\end{aligned}$$

where $\texttt{mutate}^{\texttt{Tree A}}$ is the mutator for `Tree A` (a combination of $\texttt{mutate}_r^{\texttt{Tree A}}$ with the other mutators discussed below).

[6] To lighten the notation, the mathematical definitions of the mutators in this section are slightly imprecise: Strictly speaking, in order to produce results of type `G A`, they should all take a random seed as a second argument. We consistently elide this argument, in effect changing the type of mutators from `A -> G A` to `A -> Set A`.

The second kind of mutation operators correspond to bit-level mutations in AFL that result in the deletion of bits. At the algebraic datatype level, we can mutate a term to a subterm with the same type. We call this operator $\texttt{mutate}_s$.[7]

$$\texttt{mutate}_s^{\texttt{T}}\ (C_i\ x_1\ \ldots\ x_n)\ =\ \{x_k \mid k \in \{1 \ldots n\},\ \ \texttt{T}_k = \texttt{T}\}$$

In the particular case of binary trees, for example, this operator can only be applied to the `Node` constructor, since a `Leaf` term has no subterms that are trees.

$$\texttt{mutate}_s^{\texttt{Tree A}}\ (\texttt{Node a l r})\ =\ \{\texttt{l}, \texttt{r}\}$$

Another way to mutate a term that also roughly translates to deleting bits is to mutate it to one with a different constructor, subject to the requirement that that constructor's type is a subsequence of the one we're trying to mutate. We call this operator $\texttt{mutate}_d$. That is, given two constructors, C_i and $C_{i'}$, of types $\overline{T_i} \to \texttt{T}$ and $\overline{T'_i} \to \texttt{T}$, where C_i expects n arguments and $C_{i'}$ expects n', we require a mapping $\pi : \{1 \ldots n'\} \to \{1 \ldots n\}$, such that the type of each argument of $C_{i'}$ can be mapped to the type of some argument in the C_i constructor: $\forall k \in \{1 \ldots n'\}, T_k = T_{\pi(k)}$. Then we can mutate a term by switching the top-level constructor from C_i to $C_{i'}$ and dropping some of the arguments:

$$\begin{aligned}&\texttt{mutate}_d^{\texttt{T}}\ (C_i\ x_1\ \ldots\ x_n)\\ &\quad =\ \{C'_i\ x_{\pi(1)}\ \ldots\ x'_{\pi(n')} \mid i' \neq i,\ \pi : \{1 \ldots n'\} \to \{1 \ldots n\},\ \forall k \in \{1 \ldots n'\}.\ \texttt{T}'_k = \texttt{T}_{\pi(k)}\}\end{aligned}$$

In the binary tree example, the `Leaf` constructor expects a single argument of type `A`. Since a `Node` also includes such an argument, we can mutate `Node a l r` to `Leaf a`.

$$\texttt{mutate}_d^{\texttt{Tree A}}\ (\texttt{Node a l r})\ =\ \{\texttt{Leaf a}\}$$

Note that while the intuition behind $\texttt{mutate}_d$ was that it results in "deleting bits", it can also be applied to change between constructors with identical signatures: for example mutating a `true` boolean to `false`.

More generally, we can change from one constructor $C_{i'}$ of `T` to *any* other constructor (or indeed the same constructor) $C_{i'}$ by applying *any* function π from argument positions of one to argument positions of the other. Missing arguments to $C_{i'}$ can be filled in simply by generating a random element of the appropriate type (using `arbitrary`).

$$\begin{aligned}\texttt{mutate}_g^{\texttt{T}}\ (C_i\ x_1\ \ldots\ x_n)\ =\ \{C'_i\ y_1\ \ldots\ y_{n'} \mid\ & \pi : \{1 \ldots n\} \to \{1 \ldots n'\} \cup \{\bot\},\\ & \forall k \in \{1 \ldots n'\}.\\ & \qquad (\pi(k) = j \wedge \texttt{T}'_j = \texttt{T}_k \wedge \texttt{y}_j = \texttt{x}_k)\}\\ \vee\ & \quad (\pi(k) = \bot \wedge \texttt{y}_j = \texttt{arbitrary}^{\texttt{T}'_j})\end{aligned}$$

For example, we can mutate a `Leaf x` to `Node x l r`, where the left and right subtrees are generated arbitrarily.

$$\texttt{mutate}_g^{\texttt{Tree A}}\ (\texttt{Leaf a})\ =\ \{\texttt{Node a l r} \mid \texttt{l}, \texttt{r} \in \texttt{arbitrary}^{\texttt{Tree A}}\}$$

Interestingly, this does not have an exact equivalent at the AFL level; AFL mutations cannot generate random bitstrings out of thin air.

Putting everything together, where the different mutation operators constitute different `freq` choices, yields the derived mutator for trees shown in Figure 3—now written out in actual Coq

[7] Readers familiar with QuickCheck may notice a striking similarity to its shrinking operations.

```
Fixpoint mutate_tree {A : Type}
          (mutate : A -> G A) (arbitrary : G A) (t : Tree A)
         : G (Tree A) :=
  match t with
  | Leaf x =>
    freq [ (* Recursively mutate subterms *)
           (1, x' <- mutate x ;; ret (Leaf x'))
           (* Mutate to a larger constructor by generating subterms *)
         ; (1, l <- arbitrary ;; r <- arbitrary ;; ret (Node x l r))
         ]
  | Node x l r =>
    freq [ (* Recursively mutate subterms *)
           (1, x' <- mutate x;; ret (Node x' l r))
         ; (1, l' <- mutate_tree l;; ret (Node x l' r))
         ; (1, r' <- mutate_tree r;; ret (Node x l r'))
           (* Mutate to a subterm of the same type *)
         ; (1, ret l)
         ; (1, ret r)
           (* Mutate to a smaller constructor, dropping subterms *)
         ; (1, ret (Leaf x))
         ]
  end.
```

Fig. 3. A binary tree mutator, in Coq

notation.[8] The `arbitrary` function provides a way of generating elements of type `A`, just like in the previous section. Similarly, the `mutate` function provides a way of mutating values of type `A`. (Both of these functions are implicitly parameterized on the type that they should return, using Coq's typeclass mechanism. The details of how this works are not too important; the effect is that Coq's type inference mechanism fills in appropriate type superscripts everywhere.)

Our chosen type `A -> G A` for mutation operators precludes a particular category of AFL mutations: splicing using multiple seeds. That is, we can't other existing seeds (or parts of other existing seeds) in our mutators. This was a deliberate design decision to keep mutators lightweight and avoid the overhead of keeping and traversing seeds during the mutation process; however, splicing may well be a useful operator for FuzzChick, as it is for AFL. We leave exploring this bit of the design space for future work.

4 CASE STUDIES

To evaluate FuzzChick, we measured its bug-finding performance on two existing Coq developments, both formalizing abstract machines that aim to enforce *noninterference* [Goguen and Meseguer 1982; Sabelfeld and Myers 2003] using run-time checks. Informally, noninterference states that an external observer without access to secret information cannot distinguish between two runs

[8] The code in the figure follows our current implementation in omitting some of the mutations generated by the $\texttt{mutate}_g$ operator: we always choose mutants with a *different* constructor and such that the arguments to the mutant are either a *subset* or a *superset* of the arguments to the original (not a mixture of the two). The fully general mutator will sometimes produce many more mutants of a given value, and we have not experimented with it enough yet to understand whether or not this yields more effective testing overall.

of a program that differ only in such secret information. Noninterference is a good property on which to evaluate CGPT, since it has a sparse precondition, as we will see. We describe the two machines in this section and present our measurements and comparisons in § 5. This section serves two purposes: to provide more background on noninterference, explaining the particularities of this setting that make random testing hard; and to explain in detail the systematic fault injection mechanism.

The first Coq development (§ 4.1) is a simple, stack-based machine with a handful of instructions that employ simple data labeling policies. The second (§ 4.2) is more featureful—including, among other things, registers, a richer label lattice, dynamic memory allocation, and a larger instruction set. Both machines are borrowed from published papers [Hriţcu et al. 2013b, 2016], and they have already been proved correct in Coq [Paraskevopoulou et al. 2015b]. Thus, by using explicit fault injection, we can maintain perfect control over the ground truth of our experiments, unlike prior fuzzing evaluations [Klees et al. 2018]. In particular, the dynamic IFC checks performed by each of the machines are isolated into a separate "rule table," which makes it straightforward to systematically inject bugs and see whether testing can detect them. Finally, we also have access to highly tuned, hand-written generators for the inputs to these machines; these serve as a point of comparison in the evaluation in § 5.

4.1 The IFC Stack Machine

The stack machine consists of a program counter, a stack, and separate data and instruction memories. At the core of dynamic IFC enforcement lies the notion of *labels* [Montagu et al. 2013]. Each runtime value is associated with such a label, representing a security level, which is propagated throughout the execution of the program to represent the secrecy of each piece of data. The basic values in the stack machine are labeled integers, called *atoms*, where each label can be either L ("low," denoting public information) or H ("high," denoting secret information). These labels form a trivial 2-element lattice where $L \sqsubseteq H$; we write $\ell_1 \vee \ell_2$ for the *join* (least upper bound) of ℓ_1 and ℓ_2 in this lattice. Memories are just lists of such atoms, while stacks can contain either atoms or specially marked return addresses, which capture the program counter at the time a call was made. Finally, the stack machine has a minimal set of instructions:

$$\mathit{Instr} \;::=\; \mathit{Push}\ n \mid \mathit{Load} \mid \mathit{Store} \mid \mathit{Add} \mid \mathit{Noop} \mid \mathit{Call}\ n \mid \mathit{Return} \mid \mathit{Halt}$$

The argument to *Push* is an integer that gets pushed on the stack with label L, while the argument to *Call* is the number of stack elements that are treated as arguments to the call.

Putting all this together, a machine state S is formally a 4-tuple, written $\boxed{pc \mid s \mid m \mid i}$, consisting of a program counter pc (an integer), a stack s (atoms or return addresses), a memory m (a list of atoms), and an instruction memory i (a list of instructions).

4.1.1 Noninterference. The security property our machine enforces, noninterference, is based on a notion of "indistinguishability." Intuitively, two machine states are indistinguishable if they only differ in secret data. We build up the notion formally from the machine parts.

Definition 1.

- Two atoms $n_1@\ell_1$ and $n_2@\ell_2$ are *indistinguishable*, written $n_1@\ell_1 \approx n_2@\ell_2$, if either $\ell_1 = \ell_2 = H$ or else $n_1 = n_2$ and $\ell_1 = \ell_2 = L$.
- Two instructions i_1 and i_2 are indistinguishable if they are the same.
- Two return addresses $R(n_1@\ell_1)$ and $R(n_2@\ell_2)$ are indistinguishable if either $\ell_1 = \ell_2 = H$ or else $n_1 = n_2$ and $\ell_1 = \ell_2 = L$.
- Two lists (memories, stacks, or instruction memories) xs and ys are indistinguishable if they have the same length and their elements are pairwise indistinguishable.

Definition 2. Machine states $S_1 = \boxed{pc_1 \mid s_1 \mid m_1 \mid i_1}$ and $S_2 = \boxed{pc_2 \mid s_2 \mid m_2 \mid i_2}$ are indistinguishable if their corresponding components are indistinguishable: $pc_1 \approx pc_2$, $s_1 \approx s_2$, $m_1 \approx m_2$, and $i_1 \approx i_2$.

The particular variant of noninterference we are interested in is *termination-insensitive noninterference* [Sabelfeld and Myers 2003]. We write $S \Downarrow S'$ to denote that if we repeatedly step the machine S we will reach the halting state S' (its *pc* points to a *Halt* instruction).

Definition 3. A machine semantics is *end-to-end noninterfering* if, for any indistinguishable states $S_1 \approx S_2$ that execute to completion successfully, $S_1 \Downarrow S_1'$ and $S_2 \Downarrow S_2'$, we have $S_1' \approx S_2'$.

As Hriţcu et al. [2013b] realized, this end-to-end property, while intuitively simple, is quite hard to falsify through random testing. Indeed, to discover a counterexample a testing tool needs to (a) generate indistinguishable starting states, with programs that (b) run for long enough to reach an interesting configuration where the bug might occur, and then (c) return to a low-*pc* state and terminate successfully.

Instead, Hriţcu et al. found that testing a stronger property, the inductive one that is needed to actually prove the correctness of the design, is much easier. This property, called *single-step noninterference* (SSNI), comprises three cases, often called *unwinding conditions* [Goguen and Meseguer 1982] in the literature:

Definition 4. A machine semantics is *single-step noninterfering* if:

(1) for all $S_1, S_2 \in Low$, if $S_1 \approx S_2$, $S_1 \Rightarrow S_1'$, and $S_2 \Rightarrow S_2'$, then $S_1' \approx S_2'$;
(2) for all $S \notin Low$ if $S \Rightarrow S'$ and $S' \notin Low$, then $S \approx S'$;
(3) for all $S_1, S_2 \notin Low$, if $S_1 \approx S_2$, $S_1 \Rightarrow S_1'$, $S_2 \Rightarrow S_2'$, and $S_1', S_2' \in Low$, then $S_1' \approx S_2'$.

Single-step noninterference as a property to test imposes a particularly challenging, sparse precondition: the pair of states generated, S_1 and S_2, must be low-equivalent *and* the machines need to be able to execute (the same) single instruction successfully. A naive generator for states (like QuickChick's derived one) will have great difficulty satisfying this precondition.

4.1.2 IFC Rules and Systematic Mutations. Noninterference is enforced by propagating the labels of values within the machine as it executes instructions and checking whether information ever "leaks" from secret locations to public ones. For example, the rule for *Store*, which stores a pointer in the stack, first checks that the join of the *pc* label and the pointer's label is allowed to "flow" to the label of the memory cell (this is often called the "no-sensitive-upgrades" check [Austin and Flanagan 2009; Zdancewic 2002]). Then, it overwrites the cell's contents, updating the label of the element stored with the two labels. Such mechanisms are usually baked into the semantics:

$$\frac{\begin{array}{c} i(pc) = Store \quad m(p) = n'@\ell_n' \\ \ell_p \vee \ell_{pc} \sqsubseteq \ell_n' \quad m' = m[p := n@(\ell_n \vee \ell_p \vee \ell_{pc})] \end{array}}{\boxed{pc@\ell_{pc} \mid p@\ell_p : n@\ell_n : s \mid m \mid i} \Rightarrow \boxed{(pc{+}1)@\ell_{pc} \mid s \mid m' \mid i}} \quad \text{(Store)}$$

One abstraction that proves very handy in our experiments is to extract the IFC content of such rules into a separate rule table that the operational semantics consult. For instance, the Store rule now looks as follows:

$$\frac{\begin{array}{c} i(pc) = Store \quad m(p) = n'@\ell_n' \\ (\ell_{pc}', \ell_{res}) = lookup(STORE, \ell_p, \ell_n, \ell_n', \ell_{pc}) \\ m' = m[p := n@\ell_{res}] \end{array}}{\boxed{pc@\ell_{pc} \mid p@\ell_p : n@\ell_n : s \mid m \mid i} \Rightarrow \boxed{(pc{+}1)@\ell_{pc}' \mid s \mid m' \mid i}} \quad \text{(Store')}$$

The rule table contains the following entry associated with *Store*, where ℓ_p,ℓ_n,ℓ'_n and ℓ_{pc} are the corresponding arguments to lookup.

Check	*Final pc Label*	*Result Label*
$\ell_p \vee \ell_{pc} \sqsubseteq \ell'_n$	ℓ_{pc}	$\ell_n \vee \ell_p \vee \ell_{pc}$

Each rule contains a *Check* portion, which causes the machine to halt if executing the current instruction would cause an IFC violation. It also gives the new *pc* label (which, for non-control-flow instructions, is equal to the old one), as well as the label of the result of whatever operation we are performing (here, the label of the element to be stored in the memory cell).

This factorization of IFC rules permits a systematic approach to evaluating our generators. Every check performed and every join between labels when constructing the results is there to enforce a particular noninterference policy. This means that if we *remove* any such check or taint we are guaranteed to introduce a bug (or perhaps reveal that our IFC system is too restrictive, which would also be a useful outcome of testing). Thus, we can systematically construct all possible variations of a candidate rule in the rule table, using the lattice structure of the labels. For example, each row of the following table represents a distinct mutant of the Store' rule's check:

Check	*Final pc Label*	*Result Label*
$\ell_{pc} \sqsubseteq \ell'_n$	ℓ_{pc}	$\ell_n \vee \ell_p \vee \ell_{pc}$
$\ell_p \sqsubseteq \ell'_n$	ℓ_{pc}	$\ell_n \vee \ell_p \vee \ell_{pc}$
$\ell_p \vee \ell_{pc} \sqsubseteq \ell'_n$	$\bot$	$\ell_n \vee \ell_p \vee \ell_{pc}$
$\ell_p \vee \ell_{pc} \sqsubseteq \ell'_n$	ℓ_{pc}	$\ell_p \vee \ell_{pc}$
$\ell_p \vee \ell_{pc} \sqsubseteq \ell'_n$	ℓ_{pc}	$\ell_n \vee \ell_{pc}$
$\ell_p \vee \ell_{pc} \sqsubseteq \ell'_n$	ℓ_{pc}	$\ell_n \vee \ell_p$

4.2 The IFC Register Machine

The register machine is a more realistic, scaled up version of the stack machine. It was first presented in Hriţcu et al. [2016], where it was also proved in Coq to be noninterfering. The register machine contains a plethora of features, originally intended to model an experimental processor architecture [Chiricescu et al. 2013], that make generating well-distributed inputs much harder and executing tests much slower. We briefly describe the delta between this machine and the stack machine; for the interested reader, the machine is defined formally in Hriţcu et al.

The first difference that makes this machine more realistic is that uses registers instead of just a stack. All instructions take registers as arguments and targets for their results. The instructions are:

$$\begin{aligned} \mathit{Instr} ::= {} & \mathit{Put}\ n\ r_d \mid \mathit{Mov}\ r_s\ r_d \mid \mathit{Load}\ r_p\ r_d \mid \mathit{Store}\ r_p\ r_s \mid \mathit{Add}\ r_1\ r_2\ r_d \mid \mathit{Mult}\ r_1\ r_2\ r_d \mid \mathit{Eq}\ r_1\ r_2\ r_d \mid \\ & \mathit{Noop} \mid \mathit{Halt} \mid \mathit{Jump}\ r \mid \mathit{BranchNZ}\ n\ r \mid \mathit{Call}\ r_1\ r_2\ r_3 \mid \mathit{Return} \mid \mathit{Alloc}\ r_1\ r_2\ r_3 \end{aligned}$$

The IFC infrastructure for this machine is also beefed up by generalizing the label lattice. The hard-coded 2-element lattice of the stack machine is replaced by an arbitrary lattice, which we instantiate in our experiments with a sets-of-principles lattice [Stefan et al. 2012] or a four-element diamond lattice. In addition, we encode *observable, first-class labels*, a feature used in a number of modern IFC systems [Giffin et al. 2012; Hriţcu et al. 2013a; Stefan et al. 2011]. That is, the "attacker" observing the system can detect discrepancies between labels, and the machine itself contains instructions for comparing labels, joining them, etc.

$$\mathit{Instr} ::= \mathit{Lab}\ r_s\ r_d \mid \mathit{PcLab}\ r_d \mid \mathit{PutLab}\ l\ r_d \mid \mathit{Join}\ r_1\ r_2\ r_d \mid \mathit{FlowsTo}\ r_1\ r_2\ r_d$$

In addition, integer data, labels, data pointers, and instruction pointers are strongly typed in this machine: they belong to different syntactic categories.

Finally, this machine has a labeled memory model and dynamic memory allocation. The memory is separated into 4 distinct regions, one for each label. When performing an *Alloc* r_1 r_2 r_3 instruction, we specify the size of the memory block to be allocated (in r_1), as well as the label of the block (in r_2). The machine then allocates a fresh block of memory at the region associated with the label in r_2 and returns a pointer to it (in r_3). This pointer is *stamped* with the label of the context at the point of allocation (program counter plus the labels of r_1 and r_2). A key invariant of the machine that turned out to be necessary for the noninterference proof is that when we are accessing a block of memory stamped with some label l, then we are accessing it through a sequence of pointers whose labels are at least as secure as l. This invariant is an additional precondition that our generators must satisfy: if we generate a pair of machine states that don't have this property, then we can trigger noninterference violations that would actually be false negatives!

All these features make the indistinguishability precondition for noninterference much harder to satisfy. While it was (occasionally) possible to generate two equal instructions for the stack machine, a naive generator has virtually zero chance of generating indistinguishable register machine instructions, let alone whole machine states, as each instruction is chosen from a pool of 20, with each including one or more references to registers. Moreover, the added features make it hard to generate instructions that can actually be executed: for that, most instructions' arguments need to be correctly typed (e.g., one can't execute a load with a register that is not a pointer). Moreover, having any sequence of pointers that can reach a memory block that is insufficiently label-protected violates the reachability precondition. As a result, smart manual generators are necessary for successful random testing in QuickChick. These generators comprise roughly 650 lines of Coq code, capturing which registers contain what type of data, computing meets of reachable pointer labels to produce valid stamps, and ensuring that the pair of machines are low-equivalent.

5 EXPERIMENTAL EVALUATION

In this section we describe the experiments we carried out to evaluate the effectiveness of FuzzChick. The performance metric we are interested in is *mean time to failure* (MTTF), which measures how quickly a bug can be found (in wall-clock time). To measure it, we first systematically inject all possible rule-table variants, such as the ones shown for the *Store* rule in the previous section, one at a time. For the stack machine, this results in 20 different ways of getting the security enforcement policy wrong; for the register machine, 33. For each of these variants, we test the (broken) artifact independently and log the time until failure (or timeout) across multiple (5) runs. Despite the relatively small number of runs, our results show that FuzzChick's performance lies in the middle of the other approaches with statistical significance ($p = 0.05$), mostly because of the order of magnitude difference. The stack machine experiments were run in a machine with eight 2.30GHz Intel i7 CPUS and 16GB of ram, while the register machine experiments in a twenty-four 2.4GHz CPUs and 110GB RAM running Red Hat Enterprise Linux Server 7.4.

Comparing the MTTF of FuzzChick against QuickChick and QcCrowbar with automatically synthesized generators and against QuickChick with hand-written generators, we find that FuzzChick's bug-finding performance falls in the (large) middle ground between the previously available fully automatic approaches and the fully manual one—much better than prior automatic approaches, but not as good as highly tuned manual testing. In particular, despite requiring very little manual setup, FuzzChick is able to find most of the injected bugs within seconds or minutes, even for the more challenging register machine, while the plain QuickChick and QcCrowbar are unable to find most bugs even for the simpler stack machine after an hour or more of testing.

Injected Fault #	QuickChick	QcCrowbar	FuzzChick	QuickChick (hand-written)
1	66.3	1h (t/o)	2.481	0.000
2	273.2	1h (t/o)	3.039	0.001
3	492.1	1h (t/o)	3.037	0.001
4	1h (t/o)	1h (t/o)	149.9	0.003
5	1h (t/o)	1h (t/o)	3.793	0.001
6	2452.7	1h (t/o)	2.030	0.001
7	1.115	3380.5	0.687	0.001
8	3.292	717.0	0.490	0.000
9	1h (t/o)	1h (t/o)	2283.0	0.003
10	1h (t/o)	1h (t/o)	1h (t/o)	0.003
11	1h (t/o)	1h (t/o)	153.4	0.000
12	351.4	1h (t/o)	388.5	0.002
13	1h (t/o)	1h (t/o)	414.3	0.001
14	349.4	1h (t/o)	1.341	0.000
15	1h (t/o)	1h (t/o)	382.8	0.002
16	1h (t/o)	1h (t/o)	2.900	0.002
17	1h (t/o)	1h (t/o)	3.201	0.003
18	1h (t/o)	1h (t/o)	1055.6	0.010
19	1h (t/o)	1h (t/o)	19.2	0.003
20	1h (t/o)	1h (t/o)	2.537	0.001

Fig. 4. MTTF (in seconds) across different bugs for the stack machine

QuickChick	QcCrowbar	FuzzChick	QuickChick (hand-written)
81905.3	16510.5	25192.7	69634.2

Fig. 5. Number of tests executed per second across different methods

5.1 Stack machine

The results for the stack machine can be found in Figure 4. The first two columns are our baseline: plain QuickChick and QcCrowbar with automatically derived generators. The third column shows the performance of FuzzChick with a mostly automatic generator described below. The last column shows QuickChick with the hand-written "smart generators" of Hriţcu et al. [2013b, 2016].

Using derived generators alone (column QuickChick) fails to uncover most bugs within an hour of testing. The reason quickly becomes apparent if we gather some simple statistics from the runs: out of 200 million tests (for each bug), only about 5000 on average are *not* rejected! To see why the rest are rejected, recall the property we're testing, noninterference. To check noninterference, first, the input machines must be indistinguishable and, second, both machines need to take a step without crashing. Either of these checks cause many generated inputs to be discarded. For one thing, generating pairs of machines randomly and then checking for indistinguishability will very rarely succeed. But the default derived generators for pairs of machines does exactly that:

```
Definition gen_pair_state : G (state * state) :=
  st1 <- arbitrary ;;
  st2 <- arbitrary ;;
```

```
    ret (st1, st2).
```

Worse, there are many configurations that can cause the machine to crash—for example, attempting to execute an *Add* instruction while the stack is empty, or *Return* when there is no return stack frame available. But the automatically generated instruction generator does not take that into account either; it simply generates instructions uniformly at random:

```
Definition gen_instruction : G instruction :=
  freq [ (1, ret Nop)
       ; (1, z <- arbitrary;; ret (Push z))
       ; (1, z <- arbitrary;; ret (BCall z))
       ; (1, ret BRet)
       ; (1, ret Add)
       ; (1, ret Load)
       ; (1, ret Store)].
```

The QcCrowbar implementation (column QcCrowbar) fares even worse. It does manage to discover two out of the twenty IFC violations in some of the runs, but the AFL backend is unable to overcome the deficiencies of the generator, in part because its instrumentation overhead leads to executing many fewer tests (Figure 5). With naive random testing we were executing roughly 82,000 tests per second. QcCrowbar executes tests almost five times slower ($\sim 16,500$ tests/sec).

The generator used for FuzzChick has a slight manual twist: we write one small manual generator that uses the automatically synthesized generator for states to return a pair of *identical* states.

```
Definition gen_pair_state : G (state * state) :=
  st <- arbitrary ;;
  ret (st, st).
```

This generator could obviously reveal no bugs if we used it with QuickChick or QcCrowbar, since the initial machines are always identical. However, the FuzzChick mutators, by employing coverage information, are able to massage the results of this overly rigid generator towards counterexamples. The result is shown in the FuzzChick column. We can see that it can find 12 bugs within a few seconds and 7 more within minutes (MTTF shown across 5 runs). One particularly hard bug is not found within the one-hour time limit.

On the other end of the spectrum we have the smart manual generators of Hriţcu et al.. These give excellent performance: all bugs are found within 10 milliseconds of testing. However, they are significantly more expensive to write: these generators themselves are 271 lines (even for this simple stack machine), and they constituted a publishable research contribution on their own. By contrast, the manual effort needed to obtain the generator used for FuzzChick above is literally writing the three lines of code above plus ten lines of `Derive` commands for the different datatypes.

5.2 The Register Machine

The results for the register machine are similarly organized and appear in Figure 6. The basic story here is the same as with the stack machine. The optimized hand-written generators are still blazingly fast at finding bugs (in under a second), while using derived generators with QuickChick or QcCrowbar leads to multi-hour timeouts almost across all bugs. The FuzzChick column offers once again an attractive middle ground: it finds most of the bugs within minutes and five more within hours, while timing out on three.

The number of tests run per second for each of the four approaches is shown in Figure 7. Interestingly, FuzzChick executes slightly more tests per second than the naive random generators. Apparently, when machine states are complex enough, mutating existing pairs of machines is faster

Injected Fault #	QuickChick	QcCrowbar	FuzzChick	QuickChick (hand-written)
1	4h (t/o)	8h (t/o)	36.8	0.056
2	4h (t/o)	8h (t/o)	351.6	0.292
3	4h (t/o)	8h (t/o)	5969.6	0.154
4	4h (t/o)	8h (t/o)	8h (t/o)	0.452
5	4h (t/o)	8h (t/o)	17361.6	0.017
6	4h (t/o)	8h (t/o)	407.7	0.015
7	4h (t/o)	8h (t/o)	939.1	0.052
8	4h (t/o)	8h (t/o)	653.1	0.057
9	4h (t/o)	8h (t/o)	8h (t/o)	0.073
10	4h (t/o)	8h (t/o)	2320.2	0.083
11	4h (t/o)	8h (t/o)	137.6	0.009
12	4h (t/o)	8h (t/o)	112.9	0.613
13	4h (t/o)	8h (t/o)	482.4	0.658
14	208.0	8h (t/o)	9.501	0.157
15	3h (t/o)	8h (t/o)	30.8	0.326
16	4h (t/o)	8h (t/o)	1174.4	0.080
17	4h (t/o)	8h (t/o)	115.3	0.141
18	4h (t/o)	8h (t/o)	31.1	0.312
19	4h (t/o)	8h (t/o)	141.1	0.073
20	4h (t/o)	8h (t/o)	46.5	0.047
21	4h (t/o)	8h (t/o)	84.7	0.087
22	12781.9	8h (t/o)	40.4	0.072
23	12902.6	8h (t/o)	41.5	0.137
24	12822.7	8h (t/o)	29.0	0.021
25	4275.6	8h (t/o)	29.3	0.006
26	4h (t/o)	8h (t/o)	50.3	0.002
27	4h (t/o)	8h (t/o)	61.6	0.002
28	4h (t/o)	8h (t/o)	9976.1	0.094
29	4h (t/o)	8h (t/o)	53.1	0.223
30	4h (t/o)	8h (t/o)	8h (t/o)	0.159
31	4h (t/o)	8h (t/o)	8h (t/o)	0.214
32	4h (t/o)	8h (t/o)	8497.9	0.365
33	4h (t/o)	8h (t/o)	30749.3	0.129

Fig. 6. Register machine results

QuickChick	QcCrowbar	FuzzChick	QuickChick (hand-written)
3906.6	96.0	4991.2	7660.2

Fig. 7. Number of tests executed per second across different methods

than generating everything from scratch! Also interestingly (and, to us, somewhat puzzlingly), the performance of QcCrowbar, when testing the register machine compared to the stack machine, seems to deteriorate significantly more than the performance of the randomized testers does.

These results show that CGPT has promise. In particular, FuzzChick, while not a panacea, is far better than QuickChick or QcCrowbar with automatic generators. Further exploring the remaining gap between FuzzChick and handwritten generators offers an exciting avenue for future work.

6 RELATED WORK

The random testing literature is vast; we concentrate here on two main threads: synthesizing property-based random generators from preconditions, and innovations in fuzz testing that aim to satisfy (sparse) constraints.

6.1 Property-based generators

The "precondition problem" was identified almost at the same time as property-based random testing was invented [Claessen and Hughes 2000]. In the years since then, many attempts have been made to automatically produce generators for constrained inputs. Gligoric et al. [2010] develop UDITA, a Java-based probabilistic language for generating linked structures efficiently. Bulwahn [2012b] introduces the notion of smart enumerators in Isabelle's QuickCheck, which only enumerate inputs satisfying some precondition. On the random testing side, Claessen et al. [2014] develop an algorithm for generating constrained inputs with a uniform (or near-uniform) distribution. Fetscher et al. [2015] build upon this work to implement a similar approach in PLT Redex and demonstrate excellent results in practice. Lampropoulos et al. [2017] further extend this approach by adding a restricted form of constraint solving under user control. Finally, Lampropoulos et al. [2018] synthesize correct-by-construction QuickChick generators, given preconditions expressed in the form of (restricted) inductive relations in Coq. All of these automatic approaches yield generators that are slower by an order of magnitude than their hand-written counterparts. They also may be complementary to CGPT: a better generator could be used by CGPT directly (and produce more interesting seeds faster), while the same techniques that produce generators for constrained inputs could be adapted to produce mutators instead.

6.2 Coverage-guided Fuzzing

Coverage-guided fuzzing (CGF) is the de facto standard, with AFL [2019], honggfuzz [2019], and libFuzzer [2019] as prominent examples. We discuss several threads of research aimed at improving the effectiveness of fuzzing in the presence of hard-to-satisfy constraints in the target program.

Improving Code Coverage. Much research in this area aims to improve on the basic idea of maximizing code coverage. AFLFast [Böhme et al. 2016] updates AFL's scheduling algorithm to mutate seeds it deems more likely to lead to interesting new paths. CollAFL [Gan et al. 2018] gathers path-based, rather than edge-based, coverage information. Fairfuzz [Lemieux and Sen 2018] and Vuzzer [Rawat et al. 2017] use static analysis and instrumentation to prioritize rarely covered parts of the program, and they try to mutate parts of an input that drive a program further down rare paths. Angora [Chen and Chen 2018] and T-Fuzz [Peng et al. 2018] likewise use a variety of techniques to reach and move past paths guarded by sparse constraints. Driller [Stephens et al. 2016] and QSym [Yun et al. 2018] combine ideas from fuzzing and symbolic execution (exemplified by KLEE [Cadar et al. 2008] and Sage [Godefroid et al. 2008b]) to more reliably explore new code paths. Many of these ideas could be adapted to the main CGPT fuzzing loop, and we would expect to see corresponding benefits.

Zest [Padhye et al. 2018] is a recent system that combines generator-based testing (similar to QuickCheck-style property-based random testers) and AFL-style fuzzing for Java programs. Like Crowbar [2017], Zest fuzzes the set of random choices made by the generator. Like CGPT, it distinguishes "discarded" runs that terminate due to the inputs being invalid from those that

succeed or fail. It then may prefer valid inputs as targets for subsequent mutations. On a variety of benchmarks, Zest achieves far greater coverage than AFL fuzzing or property testing alone. Since Zest does not integrate input generation with mutation (the latter is indirect, as in Crowbar), its fuzzing loop is different in important ways from that of CGPT, and it may suffer the drawbacks mentioned in § 2.2. It would be interesting to modify Crowbar to see if its performance improves when using Zest's scheduling algorithm.

Grammar-based fuzzing. Property-based testing tools work by synthesizing well-formed inputs directly, and CGPT is no different. Researchers in the fuzzing community have also examined the benefits of producing inputs that are well-formed, or nearly so. Skyfire [Wang et al. 2017] and Orthrus [Shastry et al. 2017] do this by generating well-formed initial seeds, according to a probabilistic context-sensitive grammar inferred from real-world examples. QuickFuzz [Grieco et al. 2016, 2017] allows seed generation through the use of grammars that specify the structure of valid, or interesting, inputs (mainly file formats). DIFUZE [Corina et al. 2017] performs an up-front static analysis to identify the structure of inputs to device drivers prior to fuzzing.

Other fuzzers generate grammar-compliant inputs during the fuzzing campaign. Grammar-based whitebox fuzzing [Godefroid et al. 2008a] biases the scheduling of a whitebox fuzzer toward grammar-compliant inputs. Glade [Bastani et al. 2017] synthesizes inputs from a learned grammar; Learn&Fuzz [Godefroid et al. 2017] does likewise but mutates the example afterward. Godefroid et al. suggest that well-formed inputs should be preferred two-to-one over malformed ones to optimize bug finding.

Structure-guided mutation. Some fuzzing work has considered application-specific mutation strategies. AFL can be configured to use a *dictionary* [lcamtuf 2019a] of useful bit patterns (e.g., file format "magic numbers") to be inserted, duplicated, or removed. Type-based mutation [Jain et al. 2018] works by inferring the type of bytes—e.g., as ranges, offsets, or magic numbers—in an input based on how the program uses them. Mutators are selected that may break assumed but unenforced invariants (e.g., overflowing an integer or buffer) based on the type.

Smart Greybox Fuzzing [Pham et al. 2018] (SGF) develops structural mutation operators according to a *virtual file structure* that it infers from valid (file-based) inputs. Like FuzzChick's synthesized mutators, these are based on high-level ideas of *deletion*, *addition*, and *splicing*. However, the input format is learned imperfectly from examples, so the mutations are heuristic. SGF also employs a power schedule that assigns more energy to seeds with a higher degree of grammar compliance. Grammar-aware Greybox fuzzing [Wang et al. 2018] employs similar mutations, deriving them from a provided grammar rather than a learned one.

7 CONCLUSIONS

We have presented coverage guided, property based testing (CGPT), adapting key ideas from coverage guided fuzzing (CGF), exemplified by tools like AFL, to the setting of property testing. In particular, rather than always generating a distinct input from scratch for each test, CGPT can mutate prior inputs, favoring those that result in more code coverage during testing. Indeed, its scheduling algorithm is able to switch between mutation and generation to better optimize increases in coverage. Unlike CGF, mutations in CGPT are at the level of high-level input types, rather low-level bit-twiddling operations. Our FuzzChick implementation of CGPT, which extends the QuickCheck property tester for the Coq proof assistant, synthesizes mutators for a type `T` automatically, based on `T`'s algebraic structure. We evaluated the performance of FuzzChick against both QuickChick and Crowbar, a previous (and more direct) attempt to combine CGF and property testing. We found that FuzzChick gives orders of magnitude better performance using simple, mostly automatically derived generators, though it is still significantly slower than expertly hand-written

ones. Future work should consider how to close this gap further; ideas for next steps could consider smarter generator or mutator synthesis, better-tuned scheduling, and machine learning.

ACKNOWLEDGMENTS

We thank Pei-jo Yang for contributions to this work, and Yishuai Li and Nicolas Koh for comments on earlier drafts.

REFERENCES

AFL 2019. American Fuzzing Lop (AFL). http://lcamtuf.coredump.cx/afl/.

Thomas H. Austin and Cormac Flanagan. 2009. Efficient purely-dynamic information flow analysis. In *Workshop on Programming Languages and Analysis for Security (PLAS) (PLAS)*. ACM, 113–124. http://slang.soe.ucsc.edu/cormac/papers/plas09.pdf

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. In *Proceedings of the 41st Symposium on Principles of Programming Languages (POPL) (POPL)*. ACM, 165–178. http://www.crash-safe.org/node/29

Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing Program Input Grammars. In *PLDI*.

Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing As Markov Chain. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

Lukas Bulwahn. 2012a. The New Quickcheck for Isabelle - Random, Exhaustive and Symbolic Testing under One Roof. In *2nd International Conference on Certified Programs and Proofs (CPP) (Lecture Notes in Computer Science)*, Vol. 7679. Springer, 92–108. https://www.irisa.fr/celtique/genet/ACF/BiblioIsabelle/quickcheckNew.pdf

Lukas Bulwahn. 2012b. Smart Testing of Functional Programs in Isabelle. In *18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR) (Lecture Notes in Computer Science)*, Vol. 7180. Springer, 153–167. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.229.1307&rep=rep1&type=pdf

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *OSDI*.

Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *IEEE Symposium on Security and Privacy (S&P)*.

Silviu Chiricescu, André DeHon, Delphine Demange, Suraj Iyer, Aleksey Kliger, Greg Morrisett, Benjamin C. Pierce, Howard Reubenstein, Jonathan M. Smith, Gregory T. Sullivan, Arun Thomas, Jesse Tov, Christopher M. White, and David Wittenberg. 2013. SAFE: A Clean-Slate Architecture for Secure Systems. In *Proceedings of the IEEE International Conference on Technologies for Homeland Security*.

Koen Claessen, Jonas Duregård, and Michał H. Pałka. 2014. Generating Constrained Random Data with Uniform Distribution. In *Functional and Logic Programming (Lecture Notes in Computer Science)*, Vol. 8475. Springer, 18–34. https://doi.org/10.1007/978-3-319-07151-0_2

Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 268–279. http://www.eecs.northwestern.edu/~robby/courses/395-495-2009-fall/quick.pdf

Koen Claessen and Michał Pałka. 2013. Splittable pseudorandom number generators using cryptographic hashing. In *ACM SIGPLAN Symposium on Haskell*. ACM, 47–58. http://publications.lib.chalmers.se/records/fulltext/183348/local_183348.pdf

Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

Crowbar 2017. Crowbar. https://github.com/stedolan/crowbar.

Maxime Dénès, Cătălin Hriţcu, Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2014. QuickChick: Property-based testing for Coq. The Coq Workshop. http://prosecco.gforge.inria.fr/personal/hritcu/talks/coq6_submission_4.pdf

Burke Fetscher, Koen Claessen, Michal H. Palka, John Hughes, and Robert Bruce Findler. 2015. Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System. In *24th European Symposium on Programming (Lecture Notes in Computer Science)*, Vol. 9032. Springer, 383–405. http://users.eecs.northwestern.edu/~baf111/random-judgments/

Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *S&P*.

Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John Mitchell, and Alejandro Russo. 2012. Hails: Protecting Data Privacy in Untrusted Web Applications. In *10th Symposium on Operating Systems Design and Implementation*

(OSDI). USENIX, 47–60. http://www.scs.stanford.edu/~deian/pubs//giffin:2012:hails.pdf

Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. 2010. Test generation through programming in UDITA. In *32nd ACM/IEEE International Conference on Software Engineering*. ACM, 225–234. https://doi.org/10.1145/1806799.1806835

Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008a. Grammar-based Whitebox Fuzzing. In *PLDI*.

Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008b. Automated Whitebox Fuzz Testing. In *NDSS*.

Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: Machine Learning for Input Fuzzing. In *ASE*.

Joseph A. Goguen and José Meseguer. 1982. Security Policies and Security Models. In *S&P*.

Gustavo Grieco, Martín Ceresa, and Pablo Buiras. 2016. QuickFuzz: an automatic random fuzzer for common file formats. In *International Symposium on Haskell*.

Gustavo Grieco, Martn Ceresa, Agustn Mista, and Pablo Buiras. 2017. QuickFuzz Testing for Fun and Profit. *J. Syst. Softw.* (2017).

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. 653–669. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

honggfuzz 2019. honggfuzz. http://honggfuzz.com/.

Cătălin Hriţcu, Michael Greenberg, Ben Karel, Benjamin C. Pierce, and Greg Morrisett. 2013a. All Your IFCException Are Belong To Us. In *34th IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 3–17. http://www.crash-safe.org/node/23

Cătălin Hriţcu, John Hughes, Benjamin C. Pierce, Antal Spector-Zabusky, Dimitrios Vytiniotis, Arthur Azevedo de Amorim, and Leonidas Lampropoulos. 2013b. Testing Noninterference, Quickly. In *18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 455–468. http://prosecco.gforge.inria.fr/personal/hritcu/publications/testing-noninterference-icfp2013.pdf

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 (JFP); Special issue for ICFP 2013* 26 (April 2016), e4 (62 pages). https://doi.org/10.1017/S0956796816000058 Technical Report available as arXiv:1409.0393.

Vivek Jain, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. 2018. TIFF: Using Input Type Inference To Improve Fuzzing. In *ACSAC*.

George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. 2123–2138. https://doi.org/10.1145/3243734.3243804

Leonidas Lampropoulos. 2018. *Random Testing for Language Design*. Ph.D. Dissertation. University of Pennsylvania.

Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. 2017. Beginner's Luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 114–129. http://dl.acm.org/citation.cfm?id=3009868

Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2018. Generating good generators for inductive relations. *PACMPL* 2, POPL (2018), 45:1–45:30. https://doi.org/10.1145/3158133

Leonidas Lampropoulos and Benjamin C. Pierce. 2018. *QuickCHick: Property-Based Testing In Coq*. Electronic textbook. http://www.cis.upenn.edu/~bcpierce/sf

lcamtuf. 2019a. AFL dictionary. https://lcamtuf.blogspot.com.au/2015/01/.

lcamtuf. 2019b. afl-generated, minimized image test sets (partial). http://lcamtuf.coredump.cx/afl/demo/.

Caroline Lemieux and Koushik Sen. 2018. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. *IEEE/ACM International Conference on Automated Software Engineering*.

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

libFuzzer 2019. libFuzzer. https://llvm.org/docs/LibFuzzer.html.

Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44.

Benoît Montagu, Benjamin C. Pierce, and Randy Pollack. 2013. A Theory of Information-Flow Labels. In *26th IEEE Computer Security Foundations Symposium (CSF)*. IEEE, 3–17. http://www.crash-safe.org/node/25

Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2018. Zest: Validity Fuzzing and Parametric Generators for Effective Random Testing. *CoRR* abs/1812.00078 (2018).

Manolis Papadakis and Konstantinos F. Sagonas. 2011. A PropEr integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang, Tokyo, Japan, September 23, 2011*.

39–50. https://doi.org/10.1145/2034654.2034663

Zoe Paraskevopoulou, Cătălin Hriţcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. 2015a. A Coq Framework For Verified Property-Based Testing. Workshop on Coq for PL. https://coqpl.cs.washington.edu/wp-content/uploads/2014/12/quickchick.pdf

Zoe Paraskevopoulou, Cătălin Hriţcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. 2015b. Foundational Property-Based Testing. In *6th International Conference on Interactive Theorem Proving (ITP) (Lecture Notes in Computer Science)*, Christian Urban and Xingyuan Zhang (Eds.), Vol. 9236. Springer, 325–343. http://prosecco.gforge.inria.fr/personal/hritcu/publications/foundational-pbt.pdf

Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: fuzzing by program transformation. In *IEEE Symposium on Security and Privacy (S&P)*.

Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. 2018. Smart Greybox Fuzzing. *CoRR* abs/1811.09447 (2018). arXiv:1811.09447 http://arxiv.org/abs/1811.09447

Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*.

A. Sabelfeld and A.C. Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21, 1 (Jan. 2003), 5–19. https://doi.org/10.1109/JSAC.2002.806121

Bhargava Shastry, Markus Leutner, Tobias Fiebig, Kashyap Thimmaraju, Fabian Yamaguchi, Konrad Rieck, Stefan Schmid, Jean-Pierre Seifert, and Anja Feldmann. 2017. Static Program Analysis as a Fuzzing Aid. In *Research in Attacks, Intrusions, and Defenses (RAID)*.

Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. 2012. Disjunction Category Labels. In *NordSec*.

Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. 2011. Flexible dynamic information flow control in Haskell. In *4th Symposium on Haskell*. ACM, 95–106. http://www.scs.stanford.edu/~deian/pubs//stefan:2011:flexible-ext.pdf

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 *Network and Distributed System Security Symposium (NDSS)*.

Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*.

Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2018. Superion: Grammar-Aware Greybox Fuzzing. *CoRR* abs/1812.01197 (2018).

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)*. USENIX Association, Baltimore, MD, 745–761. https://www.usenix.org/conference/usenixsecurity18/presentation/yun

Stephan A. Zdancewic. 2002. *Programming Languages for Information Security*. Ph.D. Dissertation. Cornell University. http://www.cis.upenn.edu/~stevez/papers/Zda02.pdf