FuzzChick: Coverage-Guided, Specification-Based Testing

Leonidas Lampropoulos
Michael Hicks
Benjamin C. Pierce

HCSS
April 29, 2019

Q: Is it a good idea to combine specification-based testing a la QuickCheck with fuzzing?

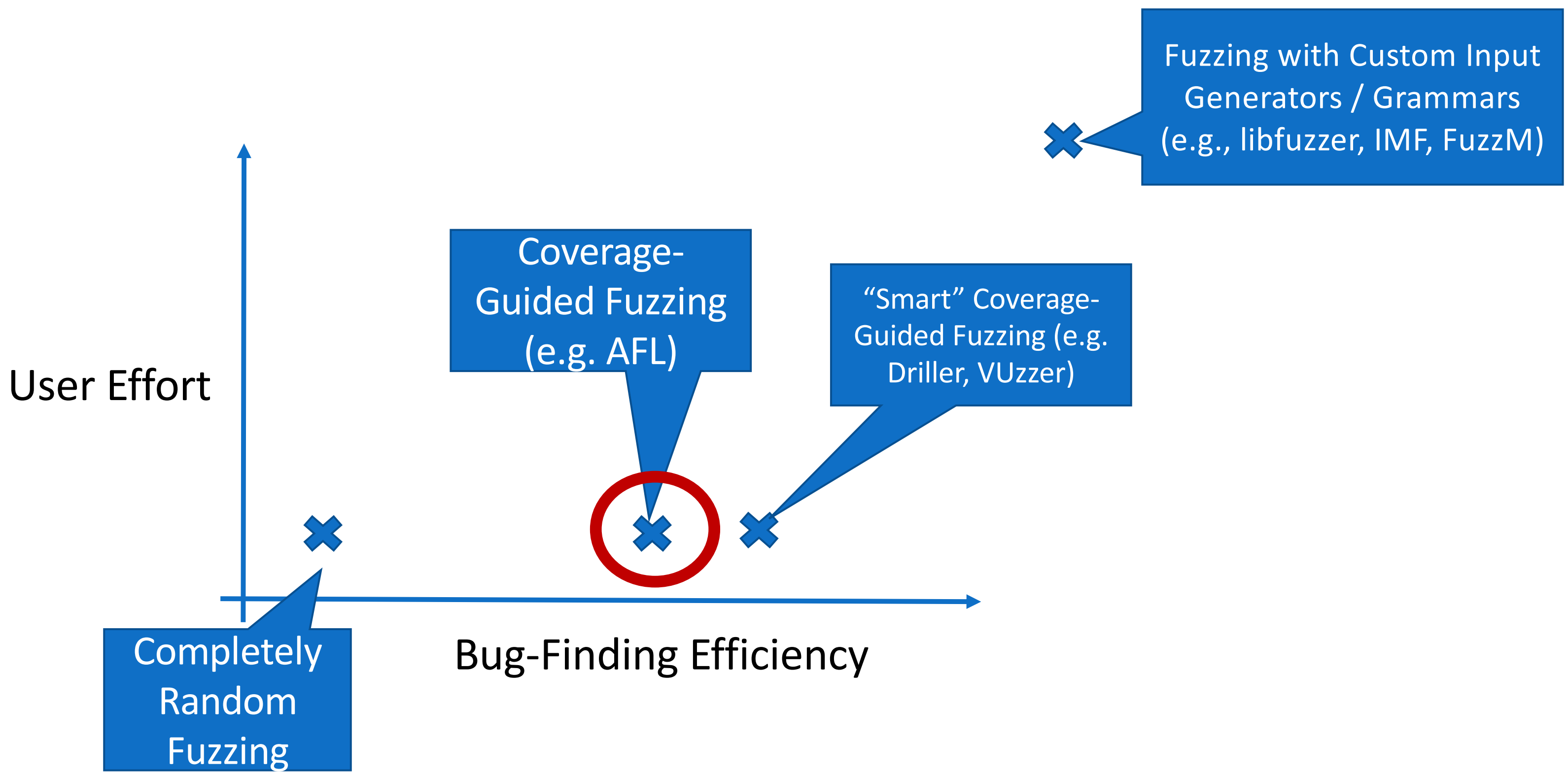
A: Yes

Fuzz Testing

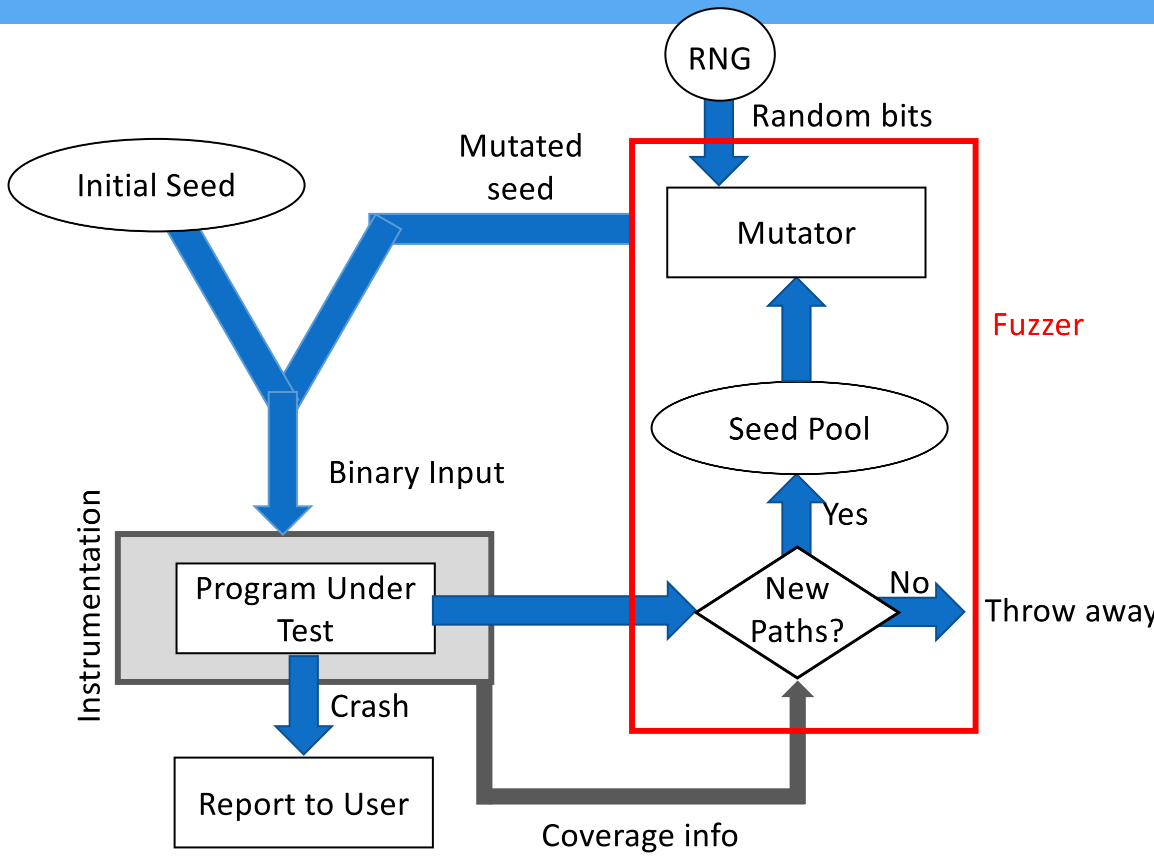
Basic Idea

- Start with a sample input to a System-Under-Test
- Use *bit-level mutations* to generate lots of similar inputs
- See if any of them lead to crashes

Some Flavors of Fuzzing



Coverage-Guided Fuzzing



Random Specification-Based Testing

Basic Idea

- Programmer writes a *formal specification* of software system or component as a function from sample inputs to Booleans
  - Executable "property" of S-U-T
- Tool generates many random inputs and applies the function to each one
  - If a counterexample is found, a greedy *shrinking* process is used to find a minimal one
- Attractive midpoint between unit tests and full-scale formal verification
- Famously embodied in Haskell QuickCheck

Koen Claessen

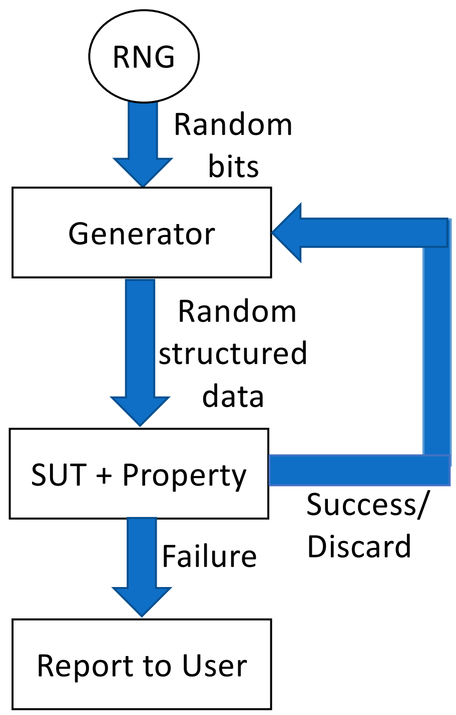
John Hughes

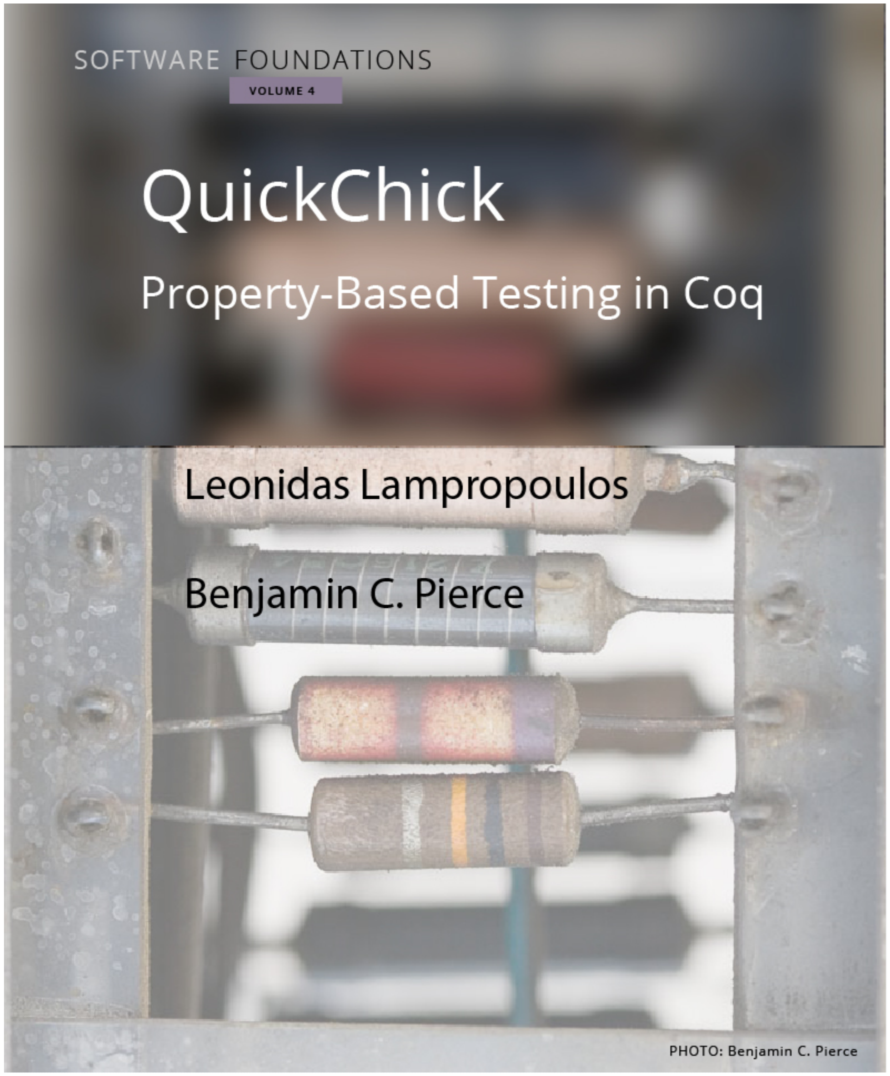
An Example Property

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

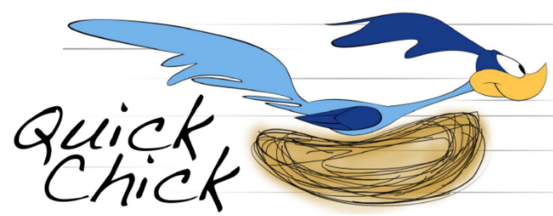
QuickCheck uses the *type* of this function to automatically generate random inputs of the appropriate form (lists of numbers)

Random Specification-Based Testing





- A variant of Haskell's QuickCheck tool…
- ported to the Coq proof assistant…
- and fed on steroids
  - e.g., a mechanically verified coretness proof for the testing framework itself



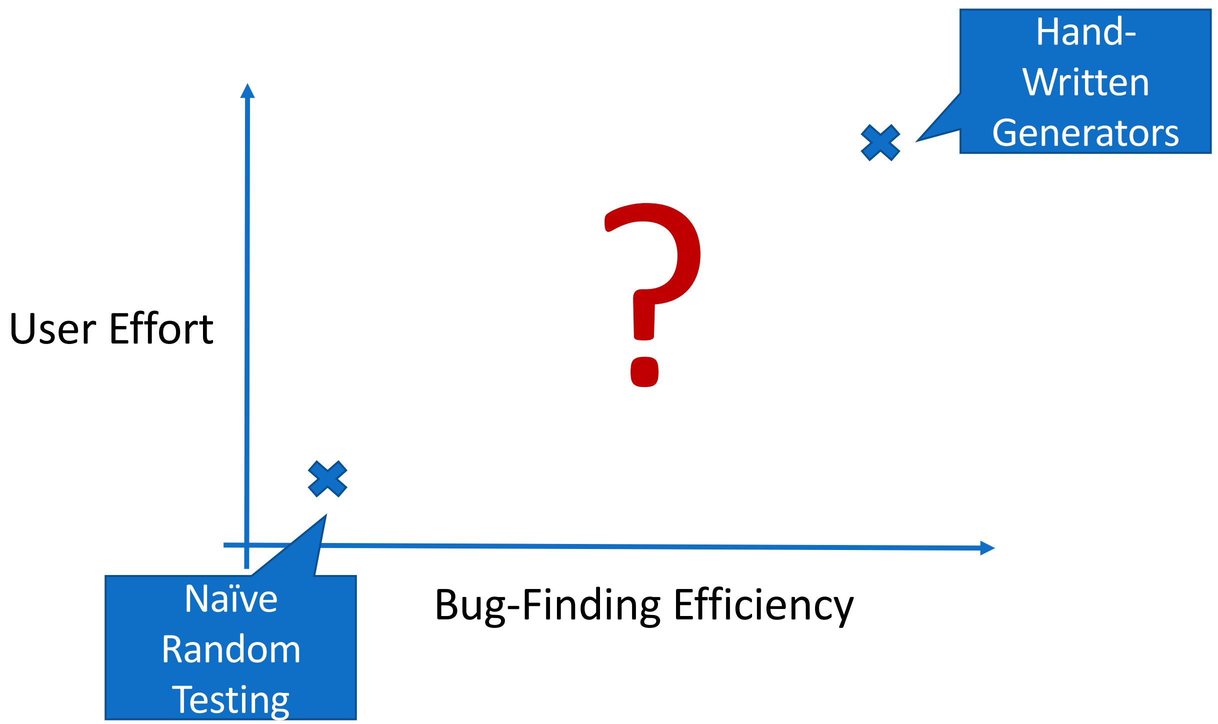
A Harder Property

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

QuickChick’s default behavior:

- Generate many random input lists
- Evaluate is_sorted on each one
  - Discard the ones for which is_sorted returns false
- Evaluate is_sorted (insert x l) on those that are left

Flavors of Random Specification-Based Testing



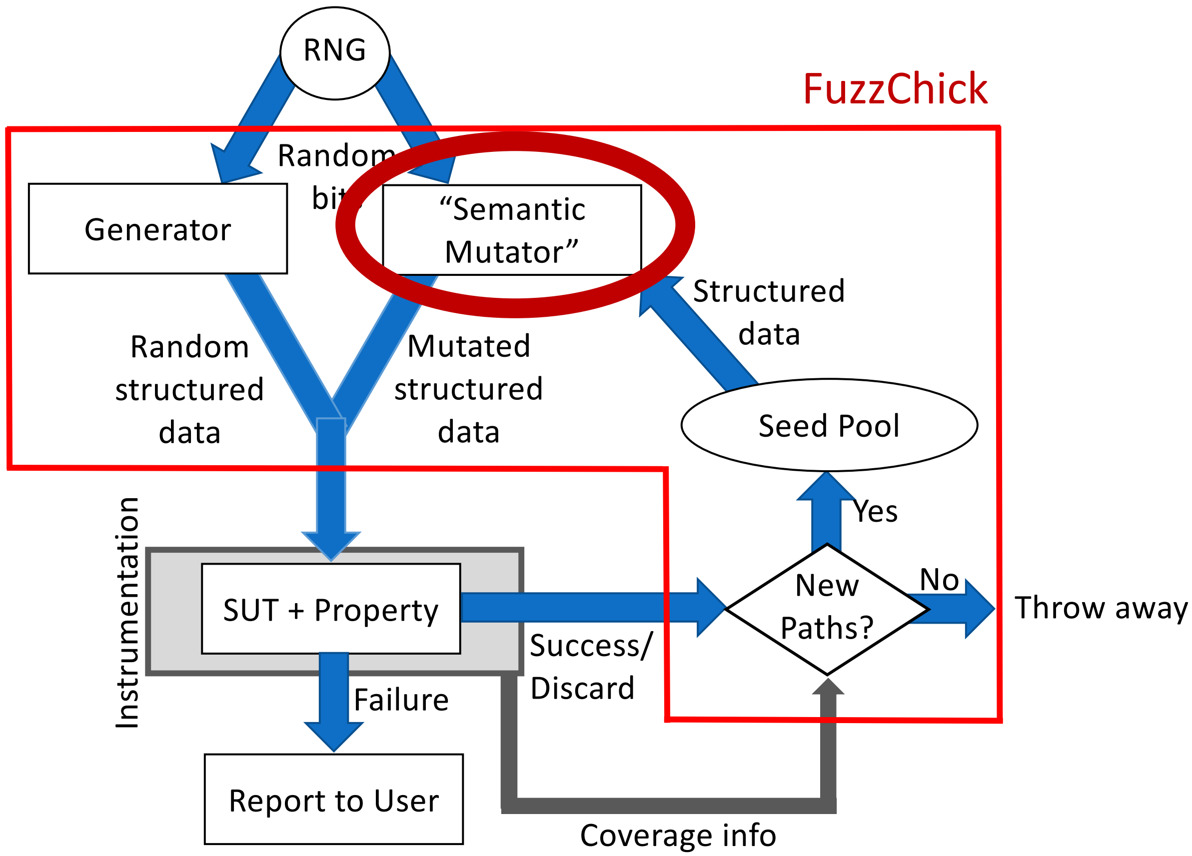
Key Insight

Use coverage information to guide the mutation of complex structured data just like AFL uses it to mutate bit strings!

“Semantic Mutation”

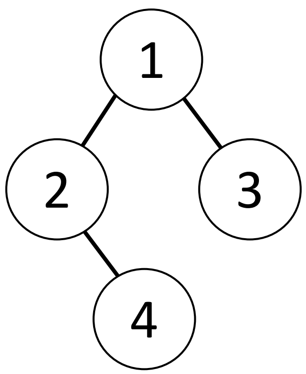
Coverage-Guided, Specification-Based Testing

FuzzChick



FuzzChick
RNG
Random
bits
Generator
"Semantic Mutator"
Random structured data
Mutated structured data
Structured data
Seed Pool
Yes
New Paths?
No
Throw away
Instrumentation
SUT + Property
Success/ Discard
Failure
Report to User
Coverage info

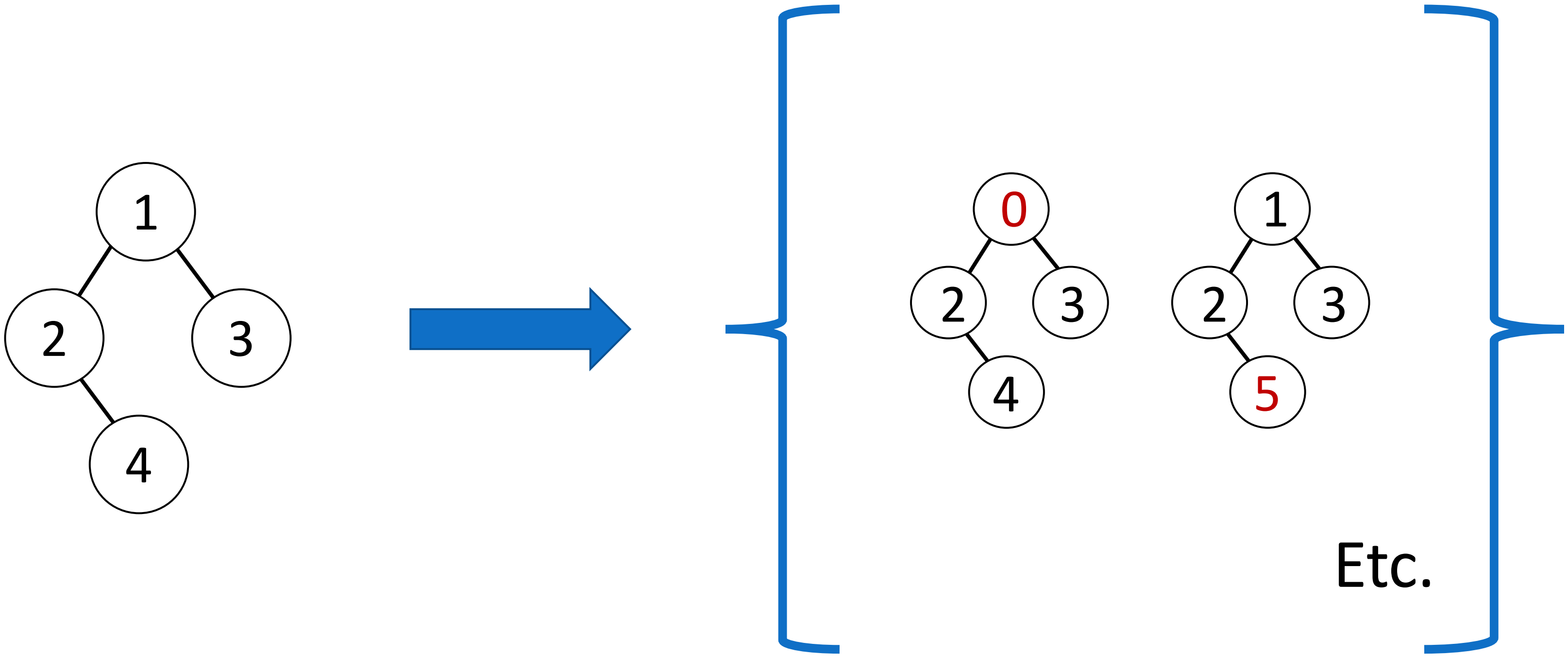
Semantic Mutators



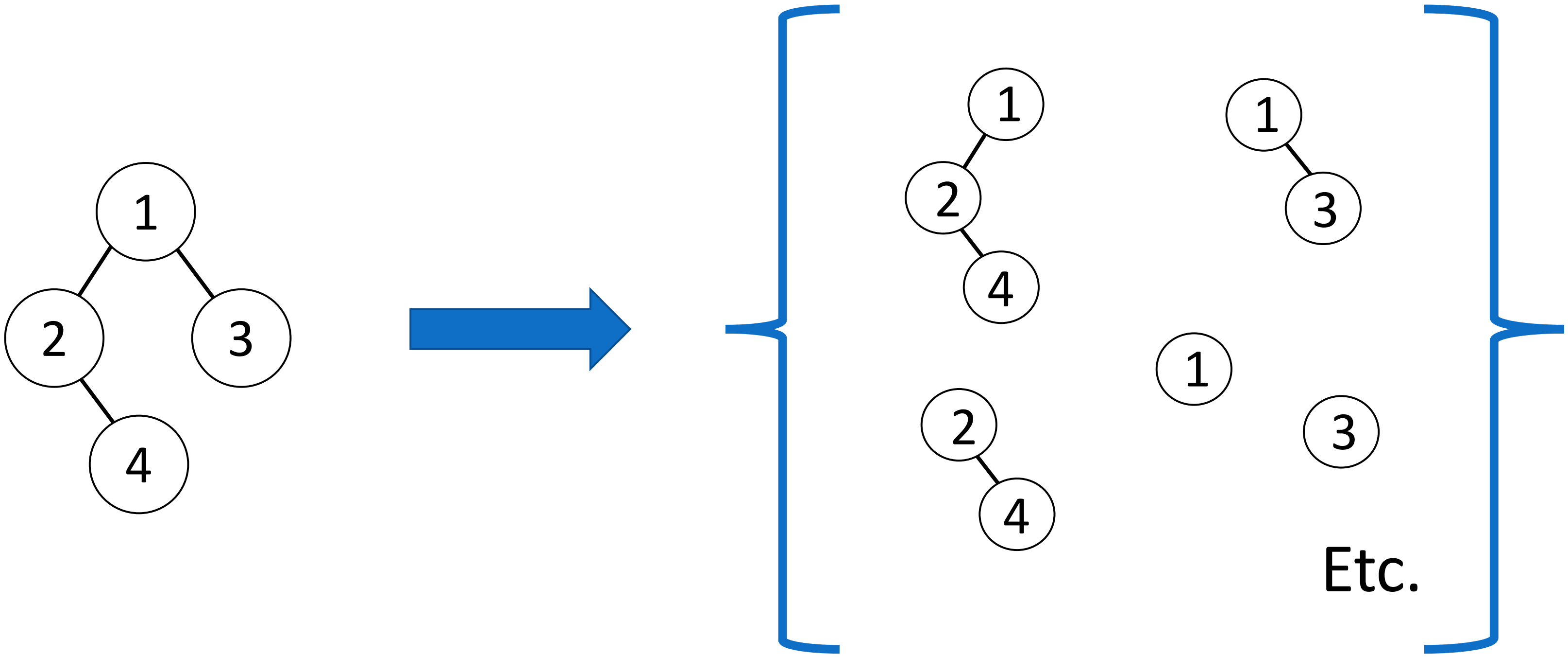
$\left\{ \right.$ All "stepwise variants"* $\left. \right\}$

* Actually, a *probability distribution* over all stepwise variants…

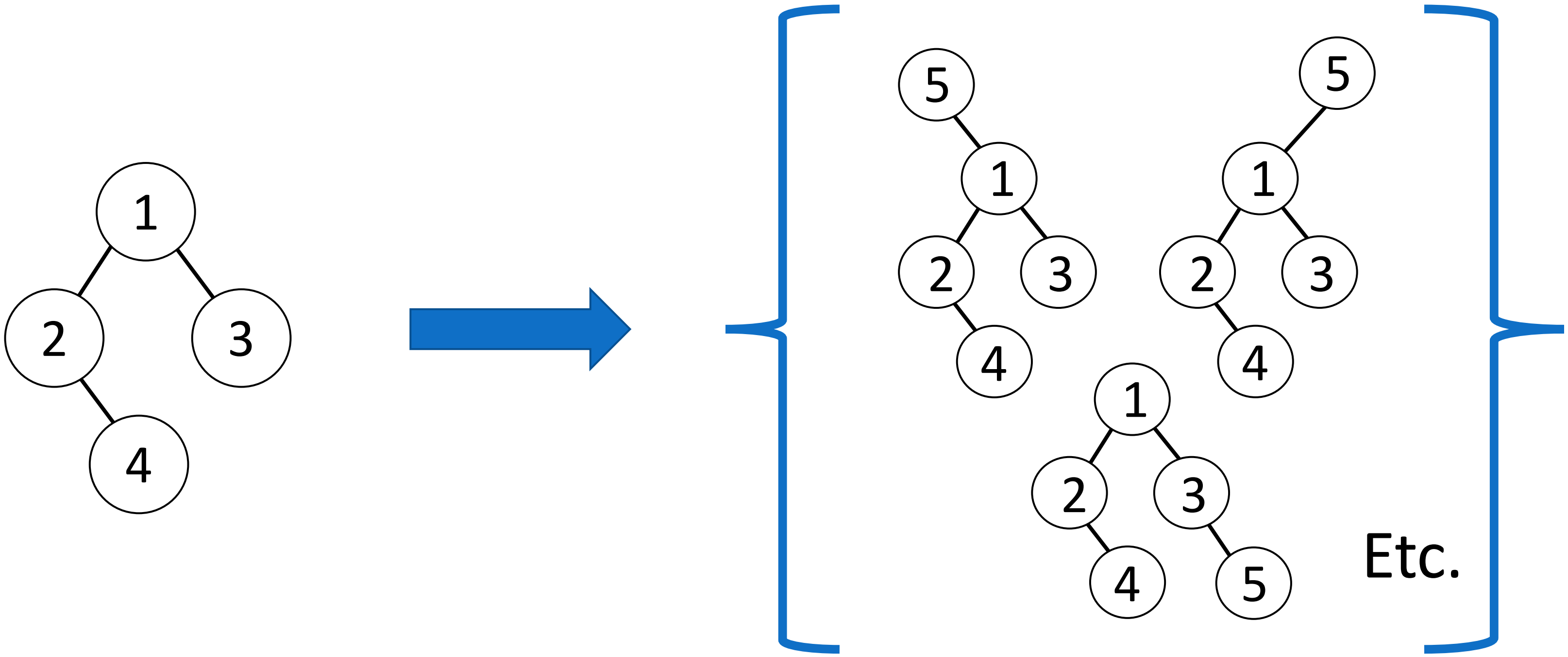
Semantic Mutators: Modification



Semantic Mutators: Deletion



Semantic Mutators: Addition

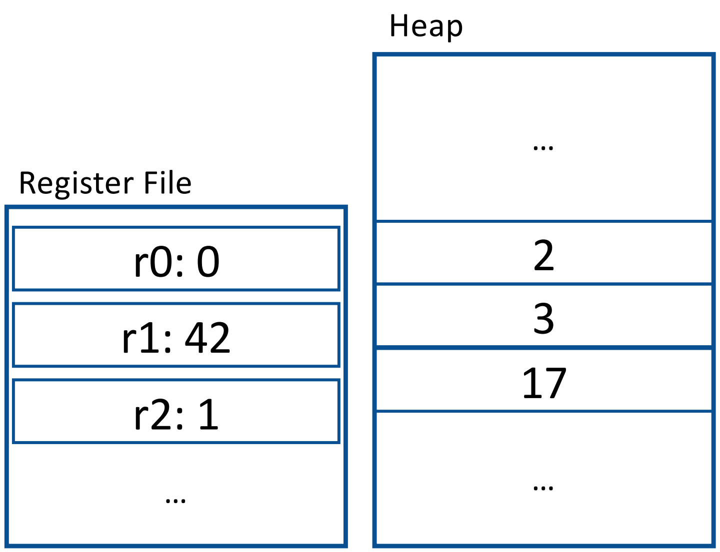


Evaluation

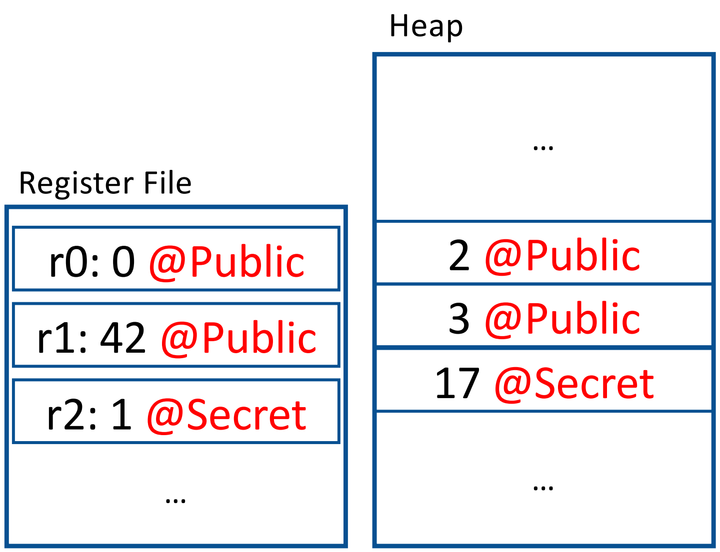
Case Study: Dynamic IFC

- System under test:
  - Simple machine with built-in dynamic information-flow monitor
  - Sensitive data is tagged “Secret”
  - Monitor detects illicit flows from Secret inputs to Public outputs
    - i.e. violations of *noninteference*
- Evaluation setup:
  - Manually create many buggy “variants” of correct monitor
  - See how long it takes to find a counterexample for each bug, under various testing regimes
    - Purely random
    - FuzzChick
    - Hand-crafted test input generators

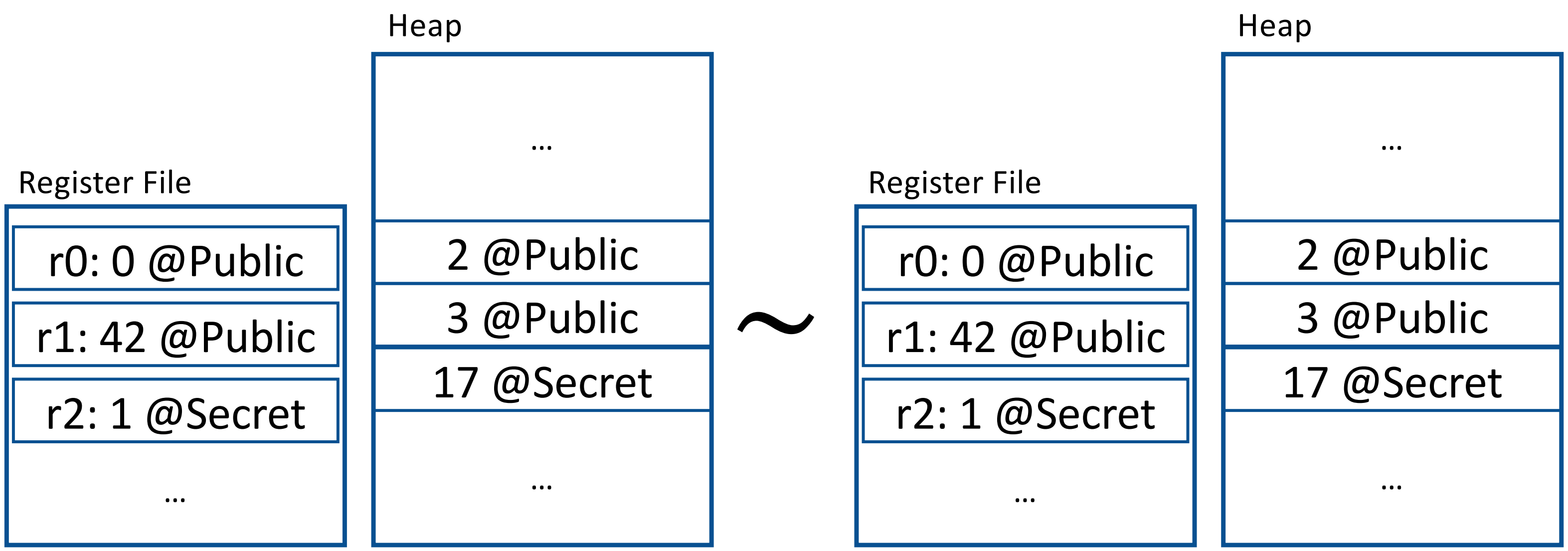
Noninterference – Abstract Machines



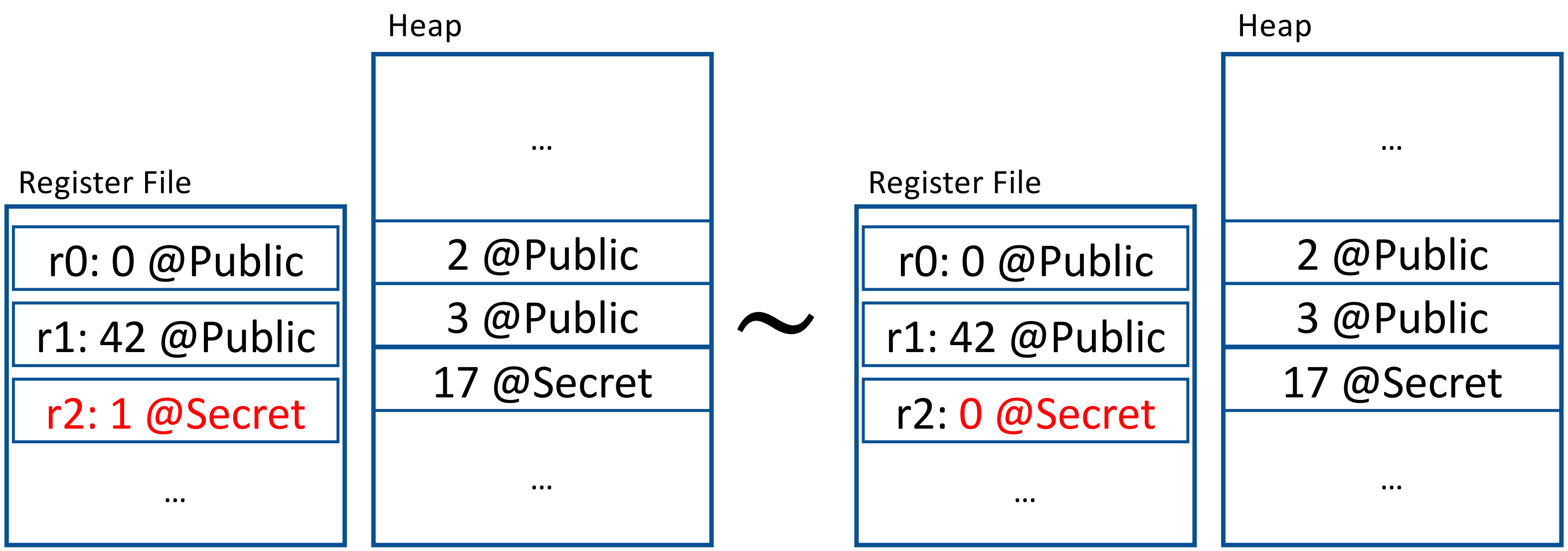
Noninterference – Security Labels



Noninterference – Indistinguishability



Noninterference – Indistinguishability



Noninterference – Property

```
Definition prop_noninterference (m1 m2 : machine) : bool :=
    indistinguishable m1 m2 ==>
    indistinguishable (step m1) (step m2).
```

- Generate many random input machines
  - Register file, heap, *and program*
- Evaluate indistinguishable on each one
  - Discard the ones for which indistinguishable returns false
- Step the machines
- Evaluate indistinguishable on the result

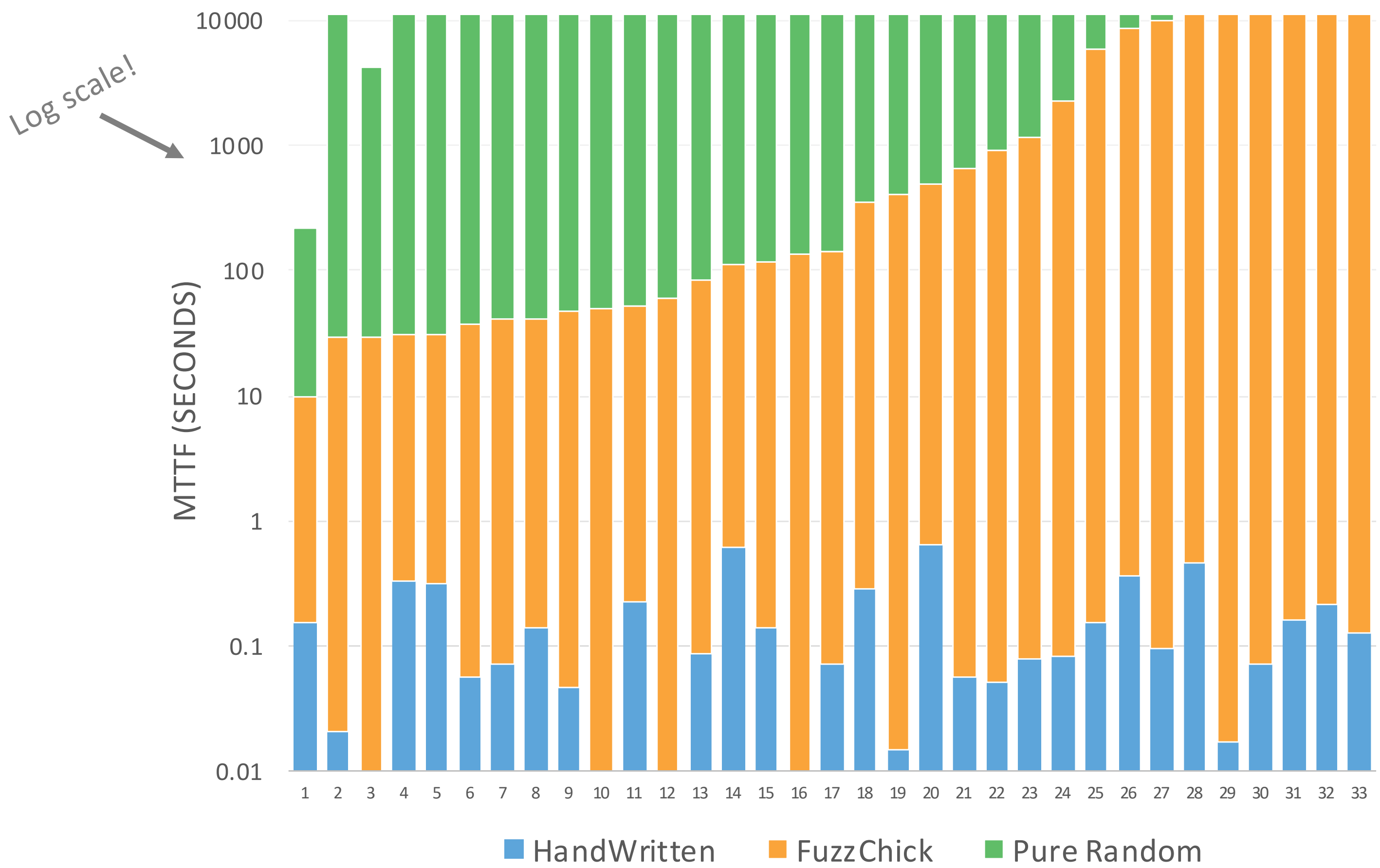
Noninterference – Property

```
Definition prop_noninterference (m1 m2 : machine) : bool :=
    indistinguishable m1 m2 ==>
    indistinguishable (step m1) (step m2).
```

Three approaches:

1. Naïve automatic generate-and-test
2. FuzzChick with an almost trivial random seed generator
3. Optimized handwritten generators (ICFP 2013)

Results



Numbers on x axis denote buggy variants of a correct IFC enforcement mechanism, sorted by height of the orange bar (effectiveness of FuzzChick)

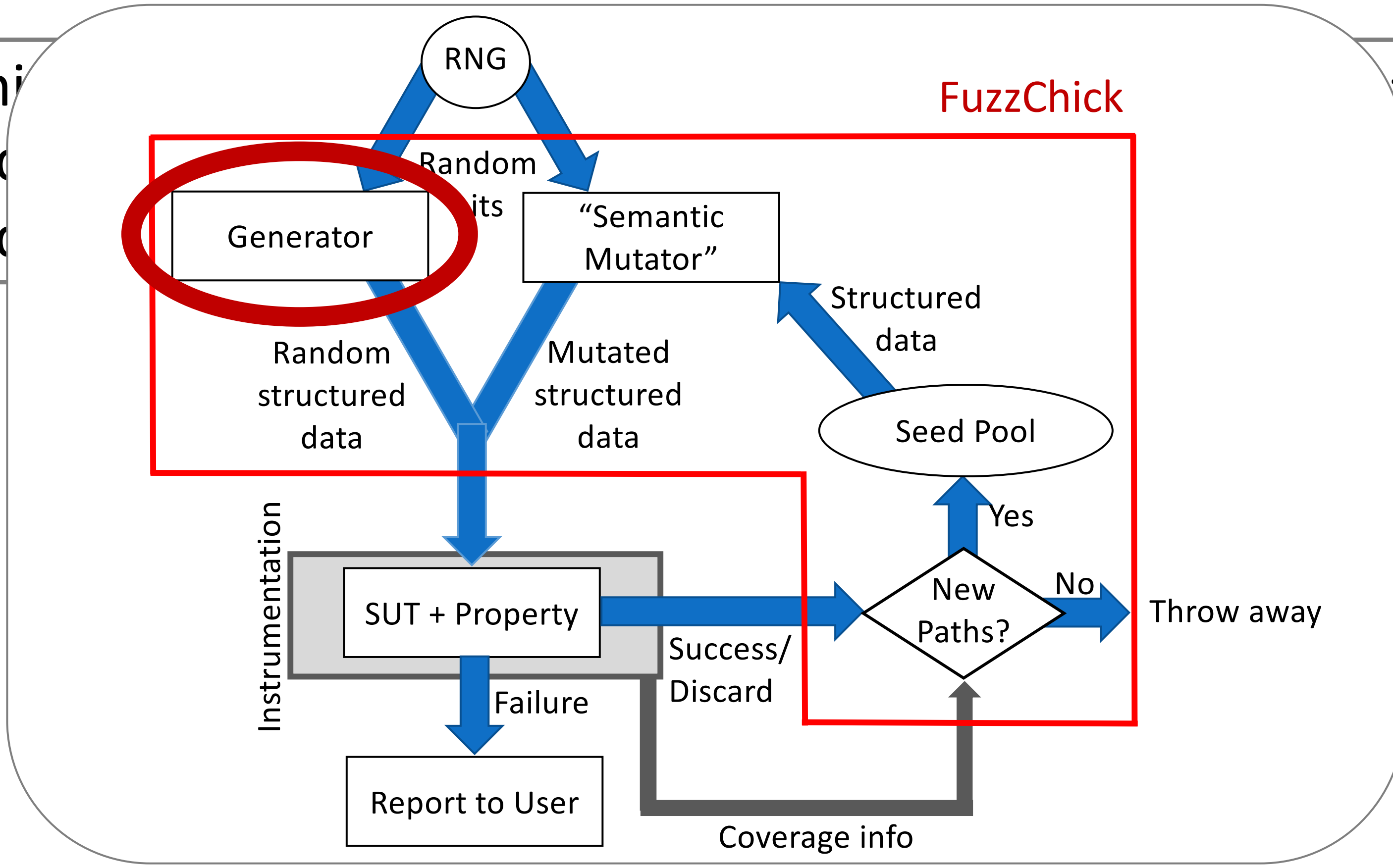
What does “almost automatic” mean?

Defini

inc

inc

=

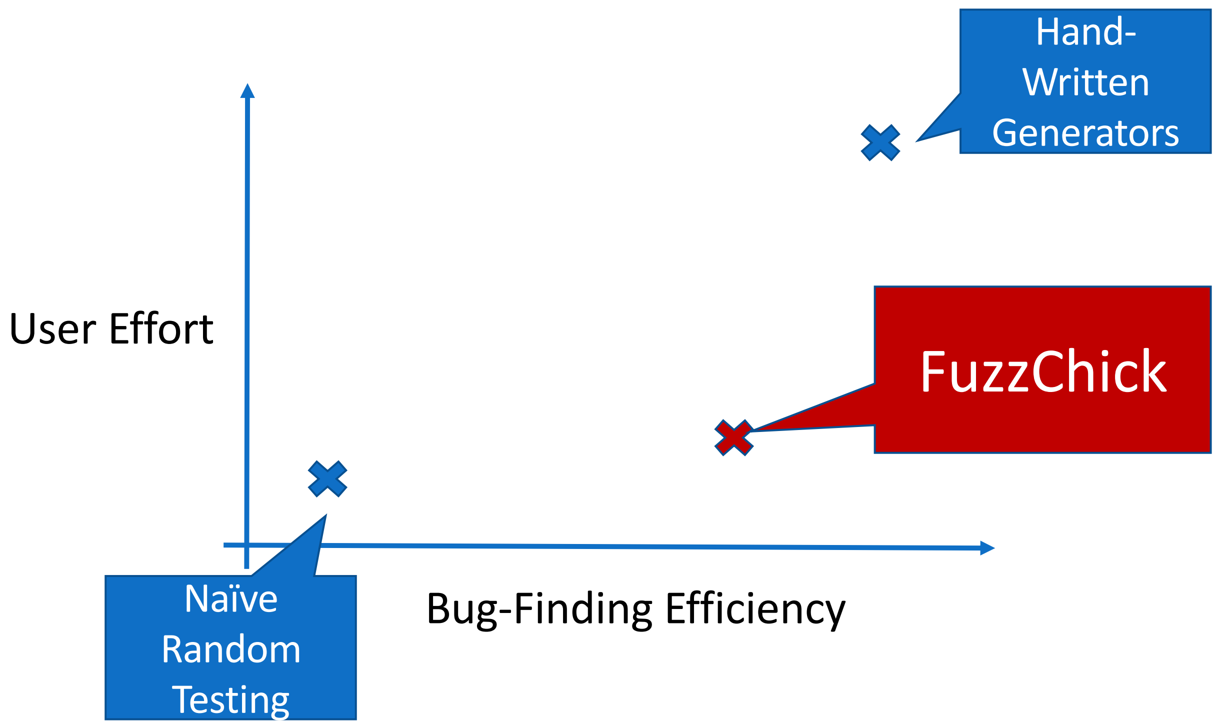


Initial random seed = Pair of machines

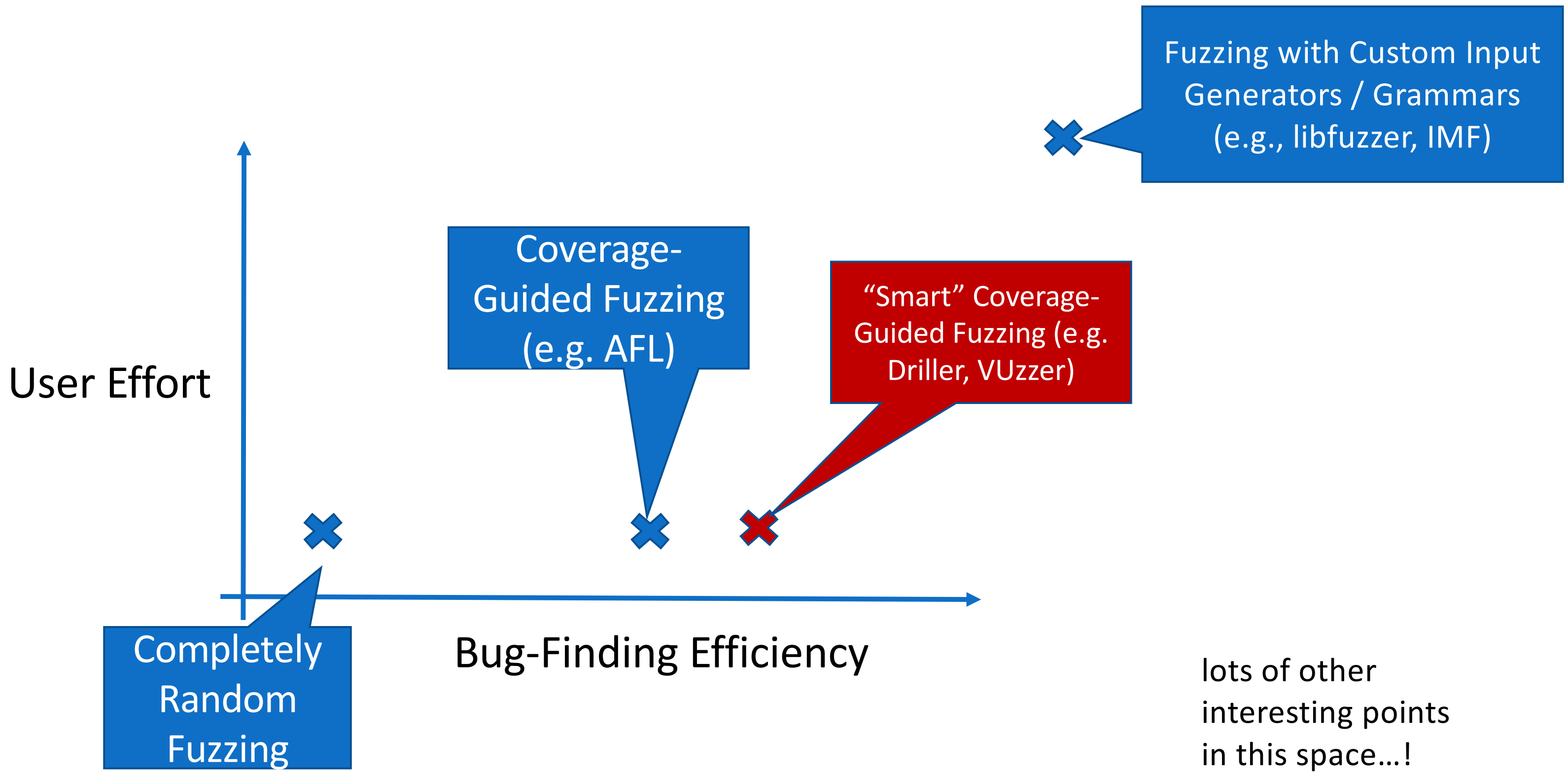
Approaches to finding “interesting” pairs of low-indistinguishable machine states:

1. Generate two random states. Mutate them until they become low-indistinguishable.
2. Generate one random state. Copy it. Mutate until it becomes interesting.

Conclusion


Hand-
Written
Generators
User Effort
FuzzChick
Naïve
Random
Testing
Bug-Finding Efficiency

Future work: Import more ideas from fuzzing!



lots of other interesting points in this space…!

Summary

- We introduced *coverage guided, property based testing* (CGPT), a novel combination of specification-based random testing and coverage-guided fuzzing
- We implemented this technique in FuzzChick, a redesign of QuickChick
- We evaluated FuzzChick by using it to test an existing formalized development of low-level information-flow tracking
- On this challenging application domain, FuzzChick significantly outperforms QuickChick
  - not nearly as good as carefully hand-written generators
  - but requires almost no effort to use