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

- Completely Random Fuzzing
- Coverage-Guided Fuzzing (e.g., AFL)
- "Smart" Coverage-Guided Fuzzing (e.g., Driller, VUzzer)
- Fuzzing with Custom Input Generators / Grammars (e.g., libfuzzer, IMF, FuzzM)
Coverage-Guided Fuzzing

Initial Seed → Mutator

Mutated seed → Seed Pool

Fuzzer

Random bits → Mutator

Yes → New Paths?

No → Throw away

Coverage info → Program Under Test

Instrumentation

Binary Input

Crash → Report to User
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
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

- RNG
  - Random bits
  - Random structured data
  - SUT + Property
  - Failure
  - Report to User

Success/Discard
QuickChick
Property-Based Testing in Coq

Leonidas Lampropoulos
Benjamin C. Pierce

- 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**

\[
\text{prop_insert_correct} \ (x : \text{nat}) \ (l : \text{list nat}) : \text{bool} := \\
\text{is_sorted} \ l \implies \text{is_sorted} \ (\text{insert} \ x \ l).
\]

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

User Effort vs. Bug-Finding Efficiency

- Naïve Random Testing
- Hand-Written Generators
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

Generator

Random.

bit

“Semantic
Mutator”

Structured
data

Mutated
structured
data

Seed Pool

New
Paths?

Yes

No

Throw away

Instrumentation

SUT + Property

Failure

Success/
Discard

Report to User

Coverage info
Semantic Mutators

All “stepwise variants”*

* Actually, a probability distribution over all stepwise variants...
Semantic Mutators: Modification

1 -> 0
2 -> 1
3 -> 1
4 -> 2
Etc.
Semantic Mutators: Deletion

![Diagram showing semantic mutators: deletion with nodes 1, 2, 3, and 4 connected in a network, with deletions resulting in new configurations and an 'Etc.' symbol]
Semantic Mutators: Addition

![Diagram showing semantic mutators for addition]

Etc.
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 noninterference

• 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

Register File

- r0: 0
- r1: 42
- r2: 1
- ...

Heap

- ...
- 2
- 3
- 17
- ...

...
Noninterference – Security Labels

Register File

- r0: 0 @Public
- r1: 42 @Public
- r2: 1 @Secret
- ...

Heap

- ...
- 2 @Public
- 3 @Public
- 17 @Secret
- ...
- ...
Noninterference – Indistinguishability

<table>
<thead>
<tr>
<th>Register File</th>
<th>Heap</th>
<th>Register File</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0: 0 @Public</td>
<td>...</td>
<td>r0: 0 @Public</td>
<td>2 @Public</td>
</tr>
<tr>
<td>r1: 42 @Public</td>
<td>3 @Public</td>
<td>r1: 42 @Public</td>
<td>3 @Public</td>
</tr>
<tr>
<td>r2: 1 @Secret</td>
<td>17 @Secret</td>
<td>r2: 1 @Secret</td>
<td>17 @Secret</td>
</tr>
</tbody>
</table>
Noninterference – Indistinguishability

### Register File
- **r0**: 0 @Public
- **r1**: 42 @Public
- **r2**: 1 @Secret

### Heap
- ... 2 @Public
- ... 3 @Public
- ... 17 @Secret

### Register File
- **r0**: 0 @Public
- **r1**: 42 @Public
- **r2**: 0 @Secret

### Heap
- ... 2 @Public
- ... 3 @Public
- ... 17 @Secret
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
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

Log scale!

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?
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
Bug-Finding Efficiency

User Effort

Naïve Random Testing

Hand-Written Generators

FuzzChick
Future work: Import more ideas from fuzzing!

- Completely Random Fuzzing
- User Effort
- Bug-Finding Efficiency
- Coverage-Guided Fuzzing (e.g. AFL)
- “Smart” Coverage-Guided Fuzzing (e.g. Driller, VUzzer)

Fuzzing with Custom Input Generators / Grammars (e.g., libfuzzer, IMF)

lots of other interesting points in this space...!
• 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