定向覆盖模糊测试工具的设计与实现

毕业设计中期检查

雷尚远

南京邮电大学计算机学院

2023 年 4 月 17 日

What Fuzzing is?

Defination[1]

- **Fuzzing** Fuzzing is the execution of the PUT using input(s) sampled from an input space (the “fuzz input space”) that protrudes the expected input space of the PUT.
  - PUT: Program Under Test
- **Fuzz testing** Fuzz testing is the use of fuzzing to test if a PUT violates a correctness policy.
- **Fuzzer** A fuzzer is a program that performs fuzz testing on a PUT.
- **Bug Oracle** A bug oracle is a program, perhaps as part of a fuzzer, that determines whether a given execution of the PUT violates a specific correctness policy.
- **Fuzz Configuration** A fuzz configuration of a fuzz algorithm comprises the parameter value(s) that control(s) the fuzz algorithm.
- **Seed** A seed is a (commonly well-structured) input to the PUT, used to generate test cases by modifying it.

What Fuzzing is?

Fuzz Testing

Input: $\mathbb{C}$, $\mathtt{t}_{\text{limit}}$
Output: $\mathbb{B}$ // a finite set of bugs

1 $\mathbb{B} \leftarrow \varnothing$
2 $\mathbb{C} \leftarrow$ **Preprocess**($\mathbb{C}$)
3 **while** $\mathtt{t}_{\text{elapsed}} < \mathtt{t}_{\text{limit}} \wedge$ **Continue**($\mathbb{C}$) **do**
4 conf $\leftarrow$ **Schedule**($\mathbb{C}$, $\mathtt{t}_{elapsed}$, $\mathtt{t}_{limit}$)
5 tcs $\leftarrow$ **InputGen**(*conf*)
  // $\mathtt{O}_{\text{bug}}$ is embedded in a fuzzer
6 $\mathbb{B}'$, execinfos $\leftarrow$ **InputEval**(*conf*, *tcs*, $\mathtt{O}_{bug}$)
7 $\mathbb{C} \leftarrow$ **ConfUpdate**($\mathbb{C}$, *conf*, *execinfos*)
8 $\mathbb{B} \leftarrow \mathbb{B} \cup \mathbb{B}'$
9 **return** $\mathbb{B}$

Fuzzing Algorithm

```
1  Input: ℂ, t_limit
2  Output: 𝔹 // a finite set of bugs
3  𝔹 ← ∅
4  ℂ ← Preprocess(ℂ)
5  while t_elapsed < t_limit ∧ Continue(ℂ) do
6      conf ← Schedule(ℂ, t_elapsed, t_limit)
7      tcs ← InputGen(conf)
       // O_bug is embedded in a fuzzer
8      𝔹′, execinfos ← InputEval(conf, tcs, O_bug)
9      ℂ ← ConfUpdate(ℂ, conf, execinfos)
10     𝔹 ← 𝔹 ∪ 𝔹′
11 return 𝔹
```

- $\mathbb{C}$:a set of fuzz configurations
- t_{limit}: timeout
- $\mathbb{B}$: a set of discovered bugs

Fuzzing Algorithm

```
   Input: ℂ, t_limit
   Output: 𝔹 // a finite set of bugs
1  𝔹 ← ∅
2  ℂ ← Preprocess(ℂ)
3  while t_elapsed < t_limit ∧ Continue(ℂ) do
4      conf ← Schedule(ℂ, t_elapsed, t_limit)
5      tcs ← InputGen(conf)
       // O_bug is embedded in a fuzzer
6      𝔹′, execinfos ← InputEval(conf, tcs, O_bug)
7      ℂ ← ConfUpdate(ℂ, conf, execinfos)
8      𝔹 ← 𝔹 ∪ 𝔹′
9  return 𝔹
```

PREPROCESS $(\mathbb{C}) \rightarrow \mathbb{C}$

- **Instrumentation**
  - grey-box and white-box fuzzers
  - static/dynamic(INPUTEVAL)
- **Seed Selection**
  - weed out potentially redundant configurations
- **Seed Trimming**
  - reduce the size of seeds
- **Preparing a Driver Application**
  - library Fuzzing, kernal Fuzzing

Fuzzing Algorithm

```
Input: ℂ, t_limit
Output: 𝔹 // a finite set of bugs
1 𝔹 ← ∅
2 ℂ ← Preprocess(ℂ)
3 while t_elapsed < t_limit ∧ Continue(ℂ) do
4     conf ← Schedule(ℂ, t_elapsed, t_limit)
5     tcs ← InputGen(conf)
      // O_bug is embedded in a fuzzer
6     𝔹′, execinfos ← InputEval(conf, tcs, O_bug)
7     ℂ ← ConfUpdate(ℂ, conf, execinfos)
8     𝔹 ← 𝔹 ∪ 𝔹′
9 return 𝔹
```

Stop Condition

- $t_{elapsed} < t_{limit}$
- CONTINUE $(\mathbb{C}) \rightarrow \{True, False\}$
  - Determine whether a new fuzz iteration should occur

Fuzzing Algorithm

```
Input: ℂ, t_limit
Output: 𝔹 // a finite set of bugs
1  𝔹 ← ∅
2  ℂ ← Preprocess(ℂ)
3  while t_elapsed < t_limit ∧ Continue(ℂ) do
4      conf ← Schedule(ℂ, t_elapsed, t_limit)
5      tcs ← InputGen(conf)
       // O_bug is embedded in a fuzzer
6      𝔹′, execinfos ← InputEval(conf, tcs, O_bug)
7      ℂ ← ConfUpdate(ℂ, conf, execinfos)
8      𝔹 ← 𝔹 ∪ 𝔹′
9  return 𝔹
```

SCHEDULE ($\mathbb{C}$, $t_{elapsed}$, t_{limit}) $\rightarrow$ conf

- **Function**
  - Pick important information(conf)
- **FCS Problem**
  - *exploration*:Spent time on gathering more accurate information on each configuration to inform future decisions
  - *exploitation*:Spent time on fuzzing the configurations that are currently believed to lead to more favorable outcomes

Fuzzing Algorithm

```
Input: ℂ, t_limit
Output: 𝔹 // a finite set of bugs
1  𝔹 ← ∅
2  ℂ ← Preprocess(ℂ)
3  while t_elapsed < t_limit ∧ Continue(ℂ) do
4      conf ← Schedule(ℂ, t_elapsed, t_limit)
5      tcs ← InputGen(conf)
       // O_bug is embedded in a fuzzer
6      𝔹′, execinfos ← InputEval(conf, tcs, O_bug)
7      ℂ ← ConfUpdate(ℂ, conf, execinfos)
8      𝔹 ← 𝔹 ∪ 𝔹′
9  return 𝔹
```

InputGen (conf)→ tcs

- **function**
  - Generate testcases
- **classification**
  - Generation-based(*Model-based*)
  - Mutation-based(*Model-less*)
  - White-box Fuzzers: symbolic execution

Fuzzing Algorithm

```
   Input: ℂ, t_limit
   Output: 𝔹 // a finite set of bugs
1  𝔹 ← ∅
2  ℂ ← Preprocess(ℂ)
3  while t_elapsed < t_limit ∧ Continue(ℂ) do
4      conf ← Schedule(ℂ, t_elapsed, t_limit)
5      tcs ← InputGen(conf)
       // O_bug is embedded in a fuzzer
6      𝔹′, execinfos ← InputEval(conf, tcs, O_bug)
7      ℂ ← ConfUpdate(ℂ, conf, execinfos)
8      𝔹 ← 𝔹 ∪ 𝔹′
9  return 𝔹
```

InputEval (conf, tcs, O_{bug}) $\rightarrow \mathbb{B}'$, execinfos

- **Fuzzing PUT**
  - tcs
  - $\mathbb{B}'$
- **Feedback Information**
  - conf, tcs
  - execinfos (tcs,crashes,stack backtrace hash,edge coverage,etc.)

Fuzzing Algorithm

```
Input: ℂ, t_limit
Output: 𝔹 // a finite set of bugs
1  𝔹 ← ∅
2  ℂ ← Preprocess(ℂ)
3  while t_elapsed < t_limit ∧ Continue(ℂ) do
4      conf ← Schedule(ℂ, t_elapsed, t_limit)
5      tcs ← InputGen(conf)
       // O_bug is embedded in a fuzzer
6      𝔹′, execinfos ← InputEval(conf, tcs, O_bug)
7      ℂ ← ConfUpdate(ℂ, conf, execinfos)
8      𝔹 ← 𝔹 ∪ 𝔹′
9  return 𝔹
```

- ConfUpdate ($\mathbb{C}$, conf, execinfos) $\rightarrow \mathbb{C}$
  - Update Fuzz Configuration(distinguishablity)
  - Seed Pool Update
- $\mathbb{B} \cup \mathbb{B}' \rightarrow \mathbb{B}$
  - Update Bugs Set

Fuzzing Algorithm

```
Input: ℂ, t_limit
Output: 𝔹 // a finite set of bugs
1  𝔹 ← ∅
2  ℂ ← Preprocess(ℂ)
3  while t_elapsed < t_limit ∧ Continue(ℂ) do
4      conf ← Schedule(ℂ, t_elapsed, t_limit)
5      tcs ← InputGen(conf)
       // O_bug is embedded in a fuzzer
6      𝔹′, execinfos ← InputEval(conf, tcs, O_bug)
7      ℂ ← ConfUpdate(ℂ, conf, execinfos)
8      𝔹 ← 𝔹 ∪ 𝔹′
9  return 𝔹
```

stop condition

- $t_{elapsed} < t_{limit}$
- CONTINUE $(\mathbb{C}) \rightarrow \{True, False\}$
  - Determine whether a new fuzz iteration should occur

Classification

The amount of collected information defines the color of a fuzzer[1].

```
Input: ℂ, t_limit
Output: 𝔹 // a finite set of bugs
1 𝔹 ← ∅
2 ℂ ← Preprocess(ℂ)
3 while t_elapsed < t_limit ∧ Continue(ℂ) do
4     conf ← Schedule(ℂ, t_elapsed, t_limit)
5     tcs ← InputGen(conf)
      // O_bug is embedded in a fuzzer
6     𝔹′, execinfos ← InputEval(conf, tcs, O_bug)
7     ℂ ← ConfUpdate(ℂ, conf, execinfos)
8     𝔹 ← 𝔹 ∪ 𝔹′
9 return 𝔹
```

- **program instrumentation**
  - static
  - dynamic
- processor traces
- system call usage
- etc.

Classification

```
Input: ℂ, t_limit
Output: 𝔹 // a finite set of bugs
1 𝔹 ← ∅
2 ℂ ← Preprocess(ℂ)
3 while t_elapsed < t_limit ∧ Continue(ℂ) do
4     conf ← Schedule(ℂ, t_elapsed, t_limit)
5     tcs ← InputGen(conf)
      // O_bug is embedded in a fuzzer
6     𝔹′, execinfos ← InputEval(conf, tcs, O_bug)
7     ℂ ← ConfUpdate(ℂ, conf, execinfos)
8     𝔹 ← 𝔹 ∪ 𝔹′
9 return 𝔹
```

Program Instrumentation

- Static
  - source code
  - intermediate code
  - binary-level
- Dynamic

Classification

```
   Input: ℂ, t_limit
   Output: 𝔹 // a finite set of bugs
1  𝔹 ← ∅
2  ℂ ← Preprocess(ℂ)
3  while t_elapsed < t_limit ∧ Continue(ℂ) do
4      conf ← Schedule(ℂ, t_elapsed, t_limit)
5      tcs ← InputGen(conf)
       // O_bug is embedded in a fuzzer
6      𝔹′, execinfos ← InputEval(conf, tcs, O_bug)
7      ℂ ← ConfUpdate(ℂ, conf, execinfos)
8      𝔹 ← 𝔹 ∪ 𝔹′
9  return 𝔹
```

Program Instrumentation

- Static
- Dynamic
  - dynamically-linked libraries
  - execution feedback: branch coverage, new path, etc.
  - race condition bugs: thread scheduling

Classification

Classification of Fuzzing

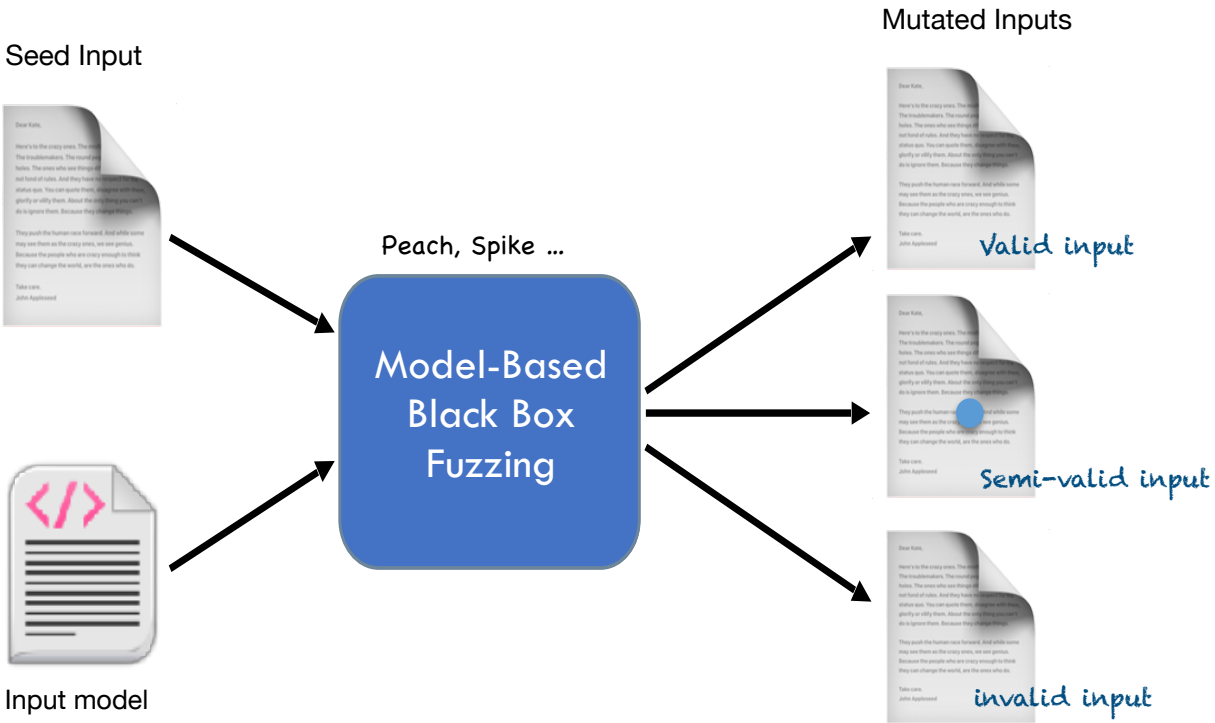
- **Black-box Fuzzing**
  - no program analysis, no feedback
- **White-box Fuzzing**
  - mostly program analysis
- **Grey-box Fuzzing**
  - no program analysis, but feedback

Why Grey-box Fuzzing ?

- Black-box Fuzzing

  Defination: techniques that do not see the internals of the PUT,and can observe only the input/output behavior of the PUT, treating it as a black-box[1].

  -No program analysis, no feedback

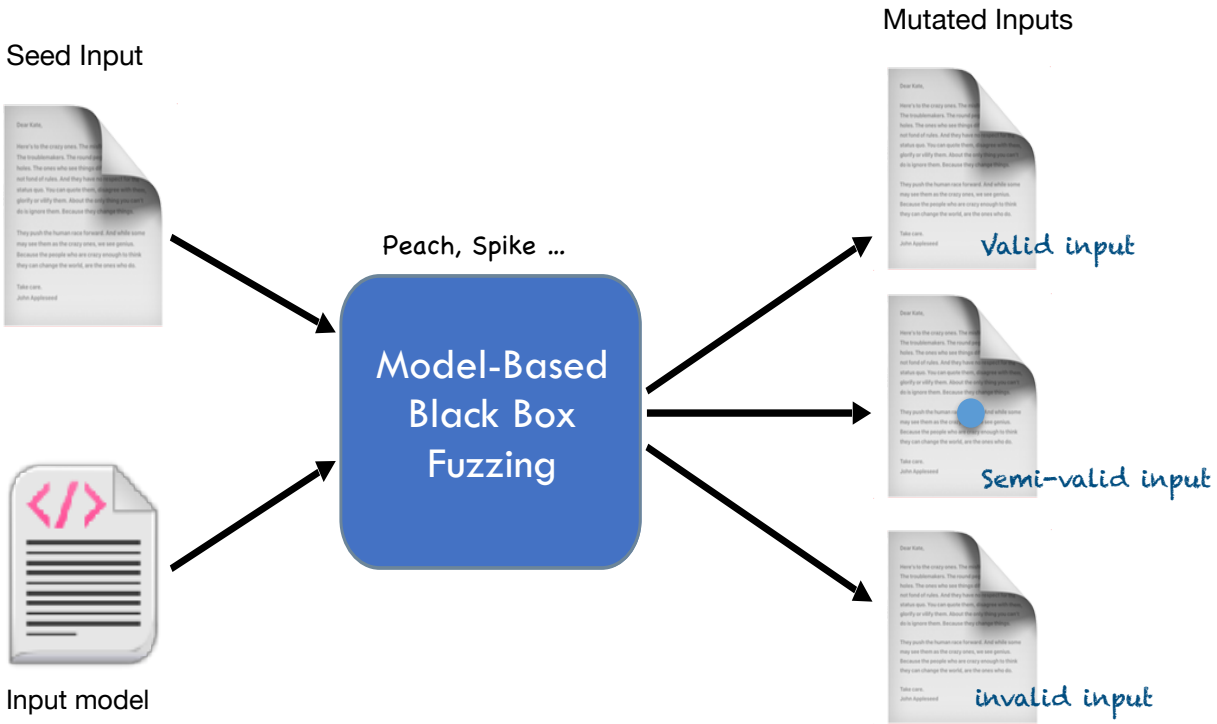


Why Grey-box Fuzzing ?

- Black-box Fuzzing

  Defination: techniques that do not see the internals of the PUT,and can observe only the input/output behavior of the PUT, treating it as a black-box[1].

  - No program analysis, no feedback



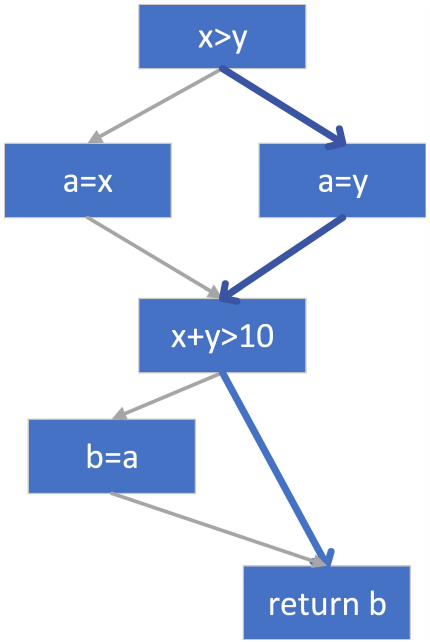
- You have no view of the PUT,but have some view of the input/output domain
- Fuzzing congfigurations are not changed according to some feedback - some fuzzer may add the testcases to seed pool
- Not effective

Why Grey-box Fuzzing ?

- White-box Fuzzing

  Defination: techniques that generates test cases by analyzing the internals of the PUT and the information gathered when executing the PUT[1].

  - Requires heavy-weight program analysis and constraint solving.



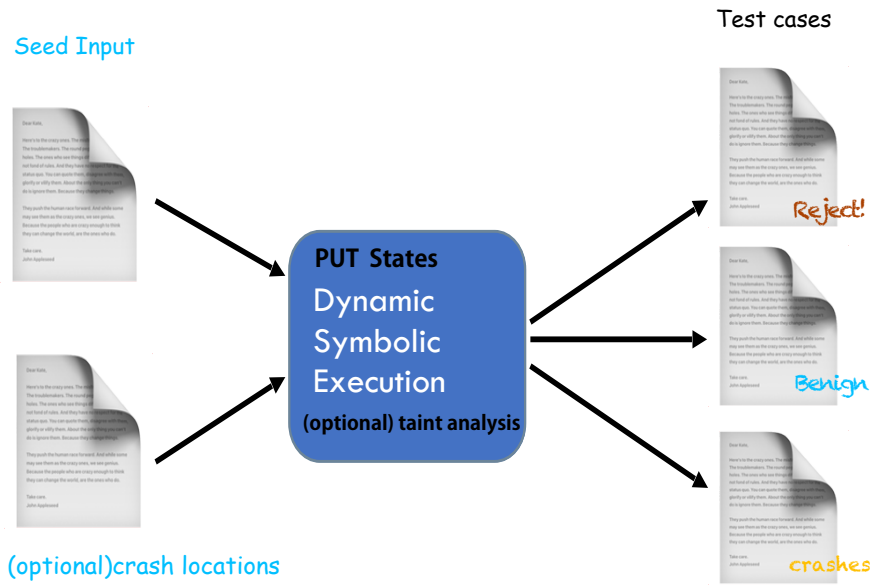
Cover more paths

$x \le y \wedge x+y \le 10$

$x \le y \wedge \neg(x+y \le 10)$

$\neg(x \le y)$

Program Analysis
- Symbolic Excution
- Constrains Satisfation

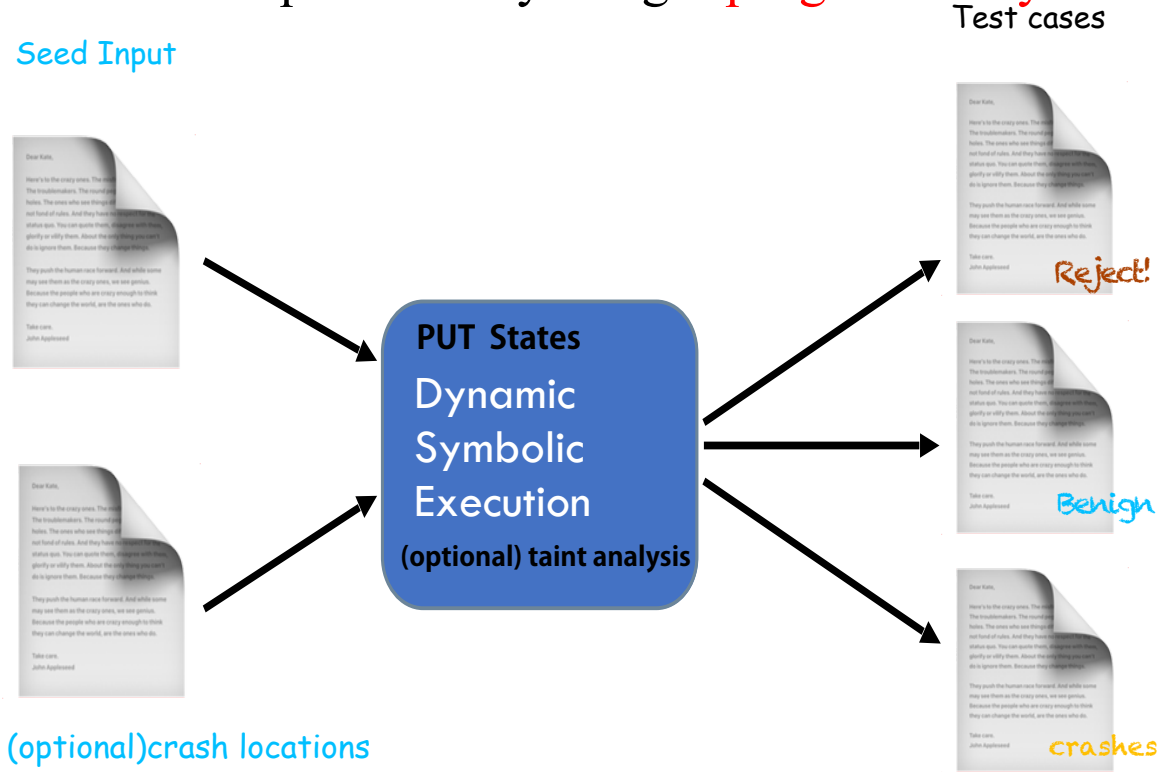


Why Grey-box Fuzzing ?

- White-box Fuzzing

  Defination: techniques that generates test cases by analyzing the internals of the PUT and the information gathered when executing the PUT[1].

  - Requires heavy-weight program analysis and constraint solving.



- You have the view of the PUT state(CFG,CG)
- Program analysis (effective but not efficient!)

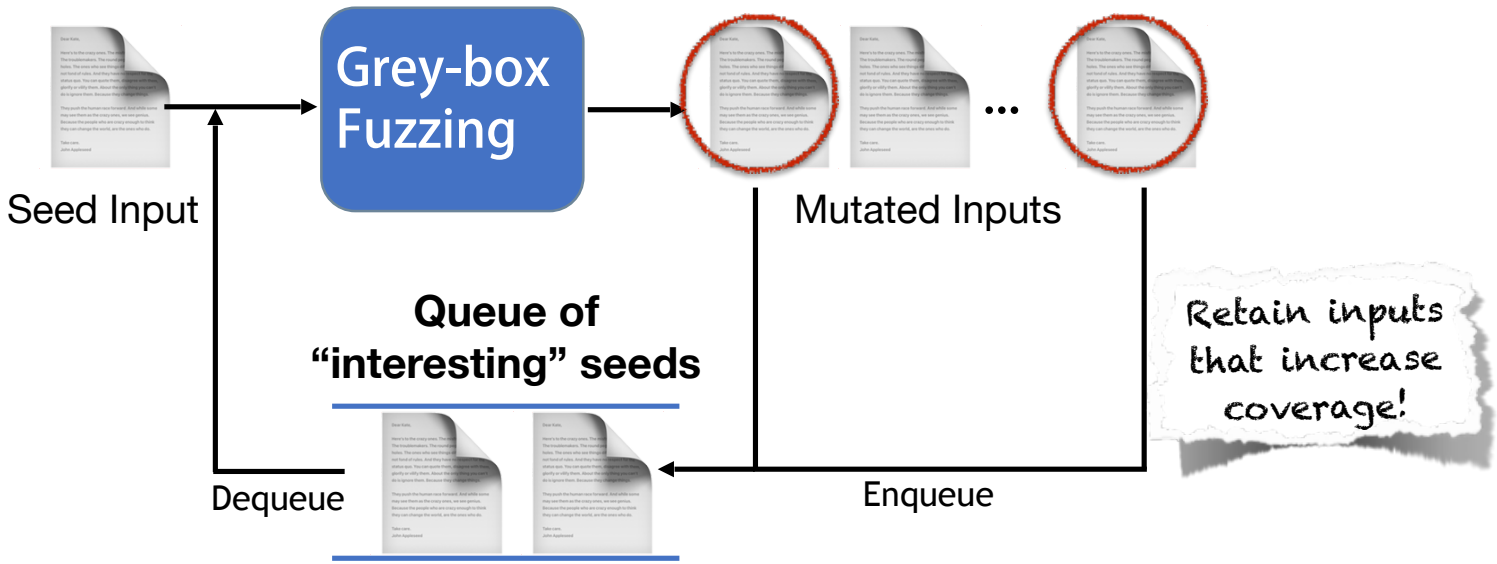
Why Grey-box Fuzzing ?

- Grey-box Fuzzing

  Defination: techniques that can obtain *some* information internal to the PUT and/or its executions to generates test cases[1].

  - Uses only lightweight instrumentation to glean some program structure

  - And coverage feedback



Why Grey-box Fuzzing ?

Grey-box Fuzzing is frequently used

- State-of-the-art in automated vulnerability detection
- Extremely efficient coverage-based input generation
  - All program analysis before/at instrumentation time.
  - Start with a seed corpus, choose a seed file, fuzz it.
  - Add to corpus only if new input increases coverage.

Why Directed Grey-box Fuzzing ?

Why Directed Grey-Box Fuzz?

- 大家都会 LaTeX，好多学校都有自己的 Beamer 主题

Why Directed Grey-Box Fuzz?

- 大家都会 LaTeX，好多学校都有自己的 Beamer 主题
- 中文支持请选择 XeLaTeX 编译选项

[1] MANÈS V J, HAN H, HAN C, et al. The art, science, and engineering of fuzzing: A survey[J]. IEEE Transactions on Software Engineering, 2019, 47(11): 2312–2331.

Thanks!