Reinforcement Learning-based Hierarchical Seed Scheduling for Greybox Fuzzing

NSE ML+Security Reading Group

Kim Hammar

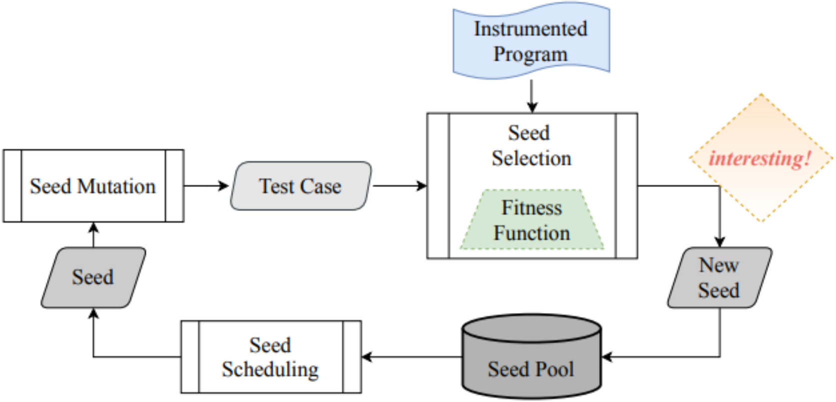
kimham@kth.se

Division of Network and Systems Engineering
KTH Royal Institute of Technology

March 18, 2022

The Context and Key Points of the Paper

- The paper proposes a new architecture for greybox fuzzing
  - Uses hierarchical coverage metric
  - Models seed scheduling as an MAB problem
  - Learns scheduling of seeds through reinforcement learning

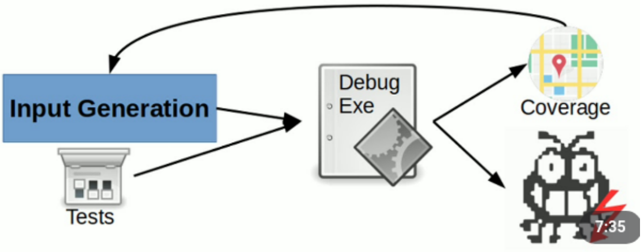


Outline

- **Background**
  - Greybox fuzzing
  - Evaluation datasets: Cyber Grand Challenge
  - Multi-armed bandits
- **The Paper**
  - Approach & Contributions
  - Proposed architecture
  - Evaluation
- **Strong points and weak points of the paper and Discussion**
  - Strong points
  - Limitations of the paper
  - Discussion about future work
- **Conclusions**

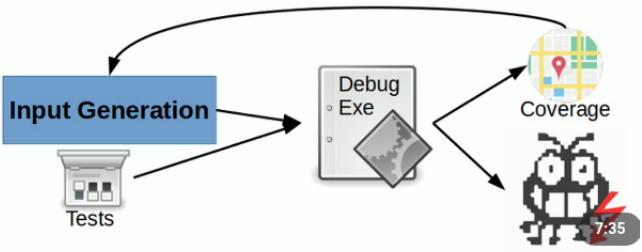
Background: Fuzzing

- Fuzzing is a method to test software, systems, networks, etc.
- Generates random inputs to the program to find crashes/bugs/memory leaks etc.



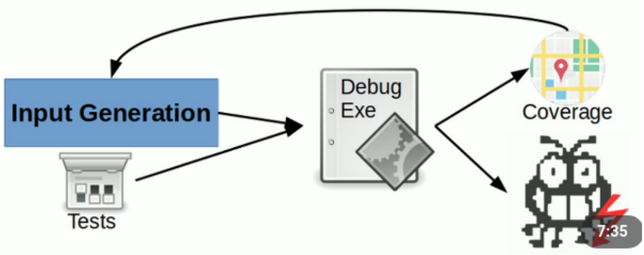
Comparison to other types of Tests

- **Unit tests and integration tests** define an execution of the program and verifies its result with assertions.
  - In fuzzing, we don't specify the execution nor the verification process
  - We run the program with random inputs and checks if it crashes
- **Property-based tests** specify properties that should be true for classes of inputs. For exampe:
  $x + y \in \mathcal{D} \quad \forall x \in \mathcal{X}, y \in \mathcal{Y}$.
  - Many similarities with fuzzing
  - Fuzzing in general is less structured and more random (i.e. don't even specify $\mathcal{X}$ and $\mathcal{Y}$)



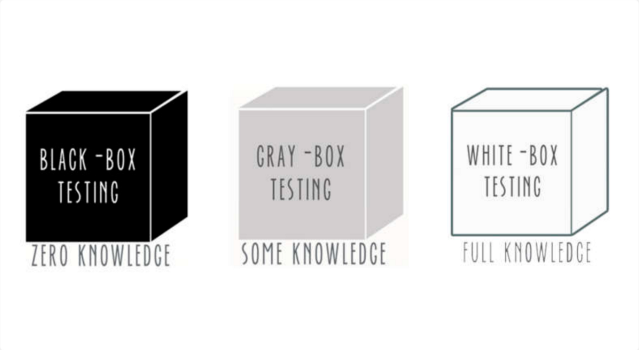
Comparison to other types of Tests

- **Unit tests and integration tests** define an execution of the program and verifies its result with assertions.
  - In fuzzing, we don't specify the execution nor the verification process
  - We run the program with random inputs and checks if it crashes
- **Property-based tests** specify properties that should be true for classes of inputs. For exampe:
  $x + y \in \mathcal{D} \quad \forall x \in \mathcal{X}, y \in \mathcal{Y}$.
  - Many similarities with fuzzing
  - Fuzzing in general is less structured and more random (i.e. don't even specify $\mathcal{X}$ and $\mathcal{Y}$)



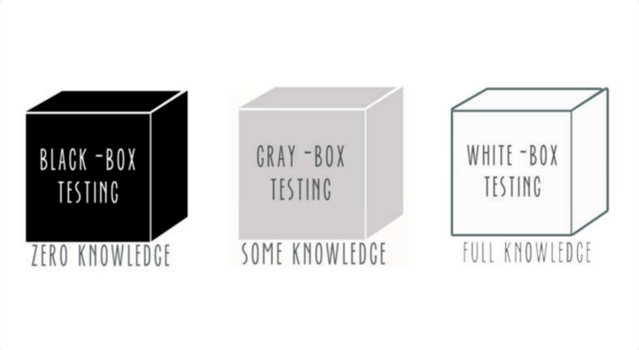
Different Types of Fuzzing

- This paper focuses on greybox fuzzing:
  - Use instrumentation to measure how the input causes the program to exercise different code paths
  - Try to generate inputs that maximize coverage
- Other types of fuzzing:
  - Black box: no instrumentation (random search)
  - White box: leverage static program analysis to generate inputs that maximize coverage



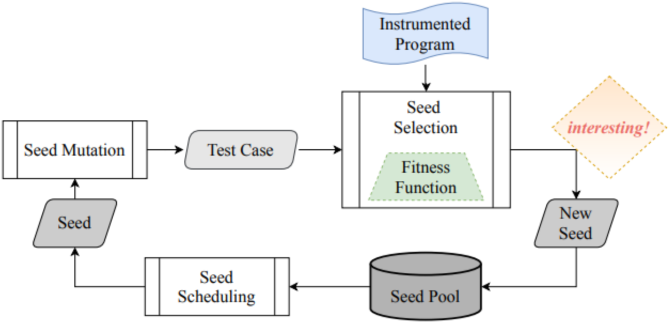
Different Types of Fuzzing

- This paper focuses on greybox fuzzing:
  - Use instrumentation to measure how the input causes the program to exercise different code paths
  - Try to generate inputs that maximize coverage
- Other types of fuzzing:
  - Black box: no instrumentation (random search)
  - White box: leverage static program analysis to generate inputs that maximize coverage



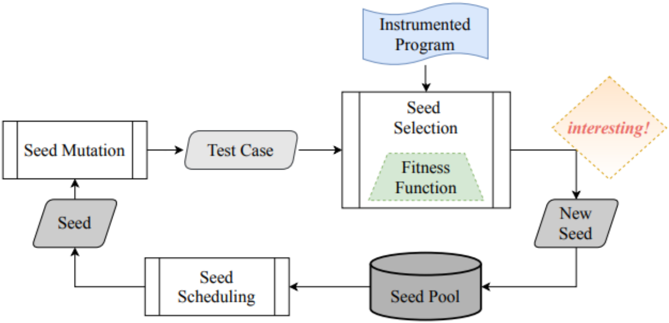
Background: Greybox Fuzzing

1. Start with an initial seed
2. Generate test inputs from the seed using some algorithm
3. Run the tests and measure coverage and bugs
4. If you improve coverage or find a bug, add the test case as a new seed
5. Repeat



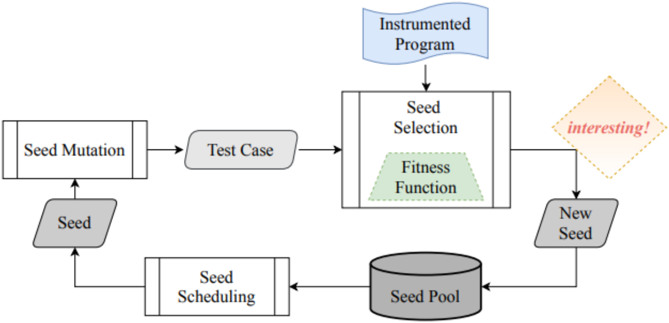
Background: Greybox Fuzzing

- **Generation of inputs (genetic process):**:
  - Start with some seed.
  - Generate new inputs through mutation and crossover.
  - Test the new inputs
  - Save inputs with strongest fitness as new seeds
  - Most common fitness function: edge coverage
  - Measure "hit counts" on branches in the code



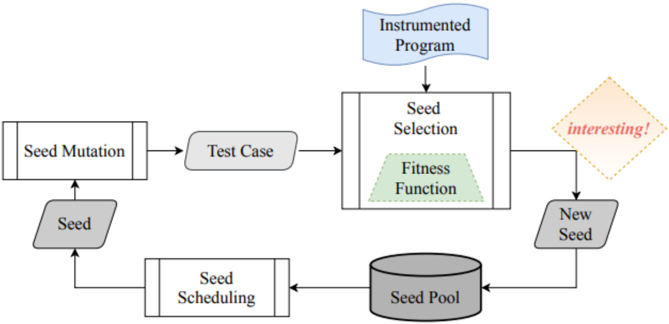
Background: Greybox Fuzzing

- **Trade-Off**:
  - Using a more sensitive/detailed coverage metric, the fuzzing can find more bugs by saving more critical "waypoints"
  - However this also leads to many more potential inputs to test (seeds)



Background: Greybox Fuzzing

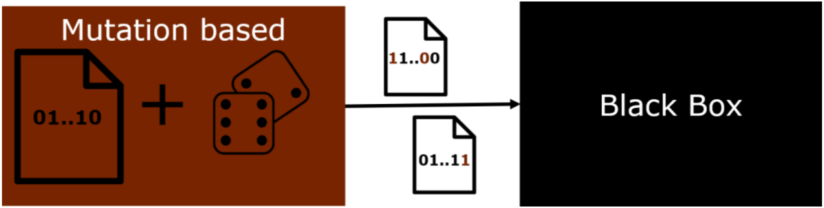
- **Problem**: may have to run the program thousands of times
- Cannot try all possible seeds (Seed exploision)
- Need some algorithm to schedule the seeds (i.e prioritize which seeds to use first)
- This paper proposes a novel approach for dealing with this problem.



Background: Other types of Fuzzing

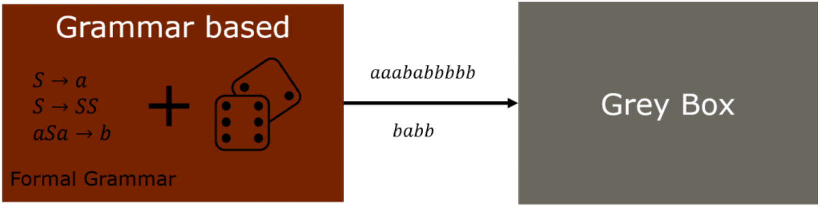
- Seed-based, aka mutational-based fuzzing is not the only type of fuzzing..
- Also exist generational fuzzing, aka model-based fuzzing.

- **Mutational fuzzing** incrementally performs arbitrary mutations to the data (does not take into account the structure of the data).



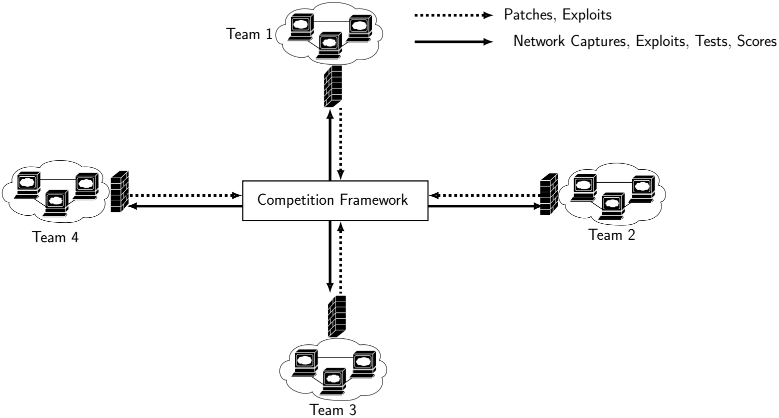
Background: Other types of Fuzzing

- **Generational fuzzing** mutates the data according to some specific structure (e.g. described by a grammar). Does not generate inputs incrementally but rather generates inputs from scratch every time.



Background: Cyber Grand Challenge (CGC)

- To evaluate their fuzzing techniques, the CGC dataset is used.
- The CGC dataset is a dataset of software programs with mainly **memory corruption vulnerabilities**, .e.g buffer-overflows and memory disclosures.
- Written in C or C++ for the DECREE operating system.



Example: Heartbleed

- **A security bug in the OpenSSL library**
  - Released 2012
  - Disclosed 2014
- **Affected software:** most implementations of TLS
- **How it works:**
  - A sender in OpenSSL can send a heartbeat msg with payload+length
  - The receiver allocates a memory buffer according to the length without verifying the length
  - The receiver writes the payload to the buffer
  - The receiver sends back the content of the buffer to the sender
  - Since the buffer size can be larger than the payload (it is not verified) the sender may send back more data than the original payload - possibly sensitive data.

Background: Multi-arm Bandits

- Finite set of actions (arms) $\mathcal{A}$
- Each time an action is chosen, some reward $r \in \mathbb{R}$ is received.
- The rewards follow an unknown i.i.d distribution $p(\cdot|a)$.
- Denote the expected reward of a as $q(a) = \mathbb{E}[r|a]$
- Goal: learn in an **online** fashion to select the actions that minimize the regret:

$$R_T = \underbrace{q^* T}_{\text{Optimal reward}} - \underbrace{\sum_{t=1}^{T} r_t}_{\text{Our reward}}$$

- Algorithm should have sub-linear regret, i.e. $\lim_{T \to \infty} R_T = 0$

Background: Multi-arm Bandits

procedure UCB MULTI-ARMED BANDIT
  $N(a) \rightarrow 0, Q(a) \rightarrow 0$ ▷ Initialization
  for $t \in \{1, \ldots T\}$ **do**
    $a = \arg\max_a Q(a) + c\sqrt{\frac{\log t}{N(a)}}$
    $r \leftarrow reward(a)$
    $N(a) \leftarrow N(a) + 1$
    $Q(a) \leftarrow Q(a) + \frac{1}{N(a)}(r - Q(a))$
  end for
end procedure

- Uses principle of optimism in the face of uncertainty
- Greedily select actions based on highest expected reward or if the actions we have not been tried before

Background: Multi-arm Bandits

- The term: $\sqrt{\frac{\log t}{N(a)}}$ comes from Hoeffding's inequality:

$$\mathbb{P}[\bar{X} \leq \mathbb{E}[\bar{X}] + \epsilon] \leq e^{-2n\epsilon^2}$$

- We use Hoeffding's inequality to bound empirical reward $Q(a)$ from actual mean $Q^*(a)$:

$$\mathbb{P}[Q(a_t) \leq Q^*(a_t) + \epsilon] \leq e^{-2n\epsilon^2} \tag{1}$$

- Want to select ϵ such that this inequality holds with a some probability, e.g. $\leq \frac{\delta}{2}$

Background: Multi-arm Bandits

- We get:

$$
\begin{aligned}
e^{-2n\epsilon^2} &= \frac{\delta}{t^2} \\
\implies -2n\epsilon^2 &= \ln(\frac{\delta}{t^2}) \\
\implies \epsilon^2 &= \frac{1}{2n}\ln(-\frac{\delta}{t^2}) \\
\implies \epsilon^2 &= \frac{1}{2n}\ln(\frac{t^2}{\delta}) \\
\implies \epsilon^2 &= \frac{\ln(\frac{t^2}{\delta})}{2n} \\
\implies \epsilon &= \sqrt{\frac{\ln(\frac{t^2}{\delta})}{2n}}
\end{aligned}
$$

- Continuing this derivation we obtain the UCB term, which we can show gives sub-linear regret.

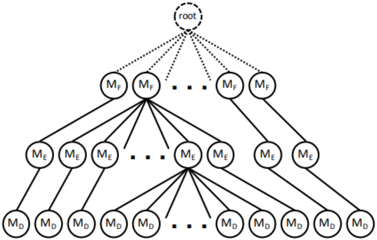
The Paper Approach and Contributions

- **Approach**:
  - Use hierarhical seed generation and multi-level coverage metric
  - Multi-level coverage metric allows to organize seeds for efficient scheduling
  - Model seed scheduling as a multi-armed bandit problem
  - Use reinforcement learning to find effective seed scheduling strategy
- **Contributions**:
  - New hierarhical coverage-metric function to instrument the fuzzing
  - Extensive evaluation of a standard multi-armed bandit algorithm

The Paper Approach and Contributions

- **Approach**:
  - Use hierarhical seed generation and multi-level coverage metric
  - Multi-level coverage metric allows to organize seeds for efficient scheduling
  - Model seed scheduling as a multi-armed bandit problem
  - Use reinforcement learning to find effective seed scheduling strategy
- **Contributions**:
  - New hierarhical coverage-metric function to instrument the fuzzing
  - Extensive evaluation of a standard multi-armed bandit algorithm

Multi-Level Coverage

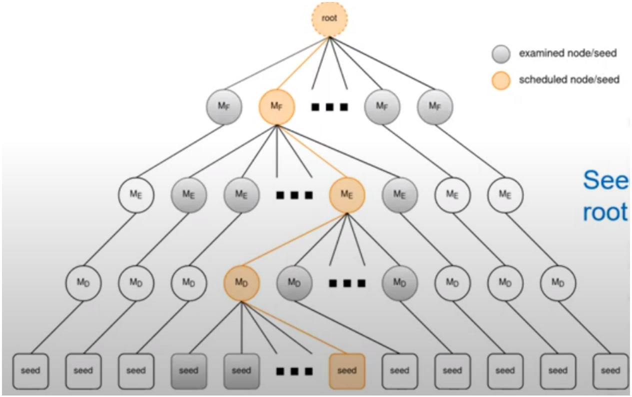


- A metric that consists of a sequence of coverage merasurements on different levels
  - Edge coverage
  - Function coverage
  - Hamming distance of comparison operands
- A way to organize coverage and seeds
- Trade-off sensitive coverage metrics and more coarse-grained coverage metrics
- I.e how long does the fuzzer mutate a given seed before giving up and scheduling other seeds? Exploitation vs exploration

Incremental Seed Clustering

- Use a clustering algorithm to group seeds that are similar.
- Similar in terms of coverage.
- A way to organize the seeds to facilitate intelligent scheduling of seeds.

Hierarchical Seed Scheduling



- Seed scheduling: seek path from root to leaf node
- Leaf node is selected as next seed to schedule

Modeling Seed Scheduling as a Multi-Armed Bandit

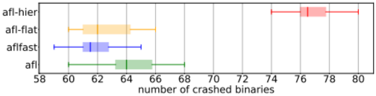
- **Model**:
  - Action: select seed to schedule
  - Reward: progress in terms of coverage/bugs
  - Balance exploitation/exploration

- **Training**:
  - Select nodes to schedule following the UCB1 algorithm
  - Reward seed selection based on how much progress was made in terms of coverage/bugs

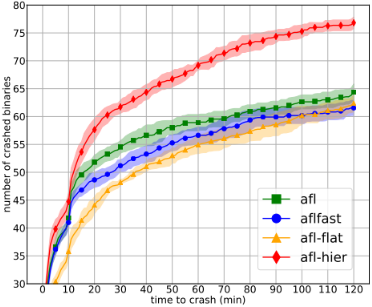
$$SeedReward(s, l, t) = \max_{F} rareness[F] \tag{2}$$

$$Reward(a^l, t) = \sqrt[n-l+1]{\prod_{k} SeedReward(s, k, t)} \tag{3}$$

Evaluation



(a) Number of crashed CGC binaries.



(b) Number of CGC binaries crashed over time.

- Outperforms state-of-the-at on the CGC dataset
- Does not outperform other fuzzers on other benchmarks

Strong points of the Paper

- **Extensive evaluation with clear benchmarks**
  - Outperforms state-of-the-art on the CGC dataset
  - Does not outperform other fuzzers on other benchmarks
- **Clever idea with multi-level coverage and seed clustering**
  - Deserves further study

Limitations of the Paper

- **Modeling of the multi-armed bandit**
  - Hard to follow
  - Lacks details and formal treatement
- **Evaluation**
  - No evaluation of the reinforcement learning algorithms

Conclusions

- Greybox fuzzing
- Use multi-level coverage metric and incremental seed clustering to organize sseds
- Schedule seeds based on a hierarchical structure
- Use multi-armed bandit to model the scheduling problem
- Learn scheduling strategy using reinforcement learning

Discussion

- Is fuzzing a MAB problem or MDP? Trade-offs?
- Opinions of the paper?
- Applications to your research?