Software Testing in a Data-driven Approach

Dongdong She
Hong Kong University of Science and Technology
• 1966, MIT Computation Center
• IBM 7094, Ancient OS CTSS (before UNIX)
• A technical issue leaked all user’s passwords in plain text

One of the earliest cybersecurity vulnerability

Source: https://multicians.org/thvv/7094.html
What is Security Vulnerability?

• Software code flaws or system misconfigurations
• Lead to unauthorized access/control of computer systems
• Huge real-world impact on our lives
Global Ransomware Attacks

Global Ransomware Damage Costs*

- **2015:** $325 Million
- **2017:** $5 Billion
- **2021:** $20 Billion
- **2024:** $42 Billion
- **2026:** $71.5 Billion
- **2028:** $157 Billion
- **2031:** $265 Billion

*Ransomware is expected to attack a business, consumer, or device every 2 seconds by 2031, up from every 11 seconds in 2021.*

*SOURCE: CYBERSECURITY VENTURES*

Global ransomware attacks cost **billions of dollars** every year.
Confidential Data Breach

Account leaked: 12 billion[1]

Online population: 5 billion[2]

Every people has an average of 2.4 accounts leaked online

Twitter in data-protection probe after '400 million' user details up for sale

[1] https://haveibeenpwned.com/
Why Do Vulnerabilities Exist?

- Humans write code
- Humans inevitably make the mistake
- Current AI-code completion still contains vulnerabilities[1]

It is hard to eliminate all the bugs

Automated Approaches to Find Vulnerabilities

• Fuzzing
• Static/Dynamic analysis
• Formal verification
• Symbolic execution

• Simple and effective
• Light-weight and scalable
• Widely-used in industry

Linus Torvalds says targeted fuzzing is improving Linux security

Linux 4.14 release candidate five is out. "Go out and test," says Linus Torvalds.
Limitation of Existing Approaches

Rule-based design: rely on a set of **static rules or heuristics**.

- Rule 1: Schedule the seed by file size
- Rule 2: Schedule the seed by execution time
- Rule 3: Randomly mutate the first byte of the seed
- Rule N: ...

- Good heuristics are expensive
- Often **fail to generalize** on diverse programs
Rule-Based vs. Data-Driven

Human experience/Hand-crafted heuristics

Rule 1
Rule 2
... Rule n

Rule-based system

$f(input)$ Adaptive to data

Data-driven system

Data-driven approach is adaptive and effective
My Research

- Part 1: Data-driven mutation
- Part 2: Data-driven scheduling

Data-driven mutation:
- Neuzz[SP’19]
- Neutaint[SP’20]
- MTFuzz[FSE’20]

Data-driven scheduling:
- K-Scheduler[SP’22]
NEUZZ: Data-Driven Mutation

Background: Fuzzing is a search problem aimed at discovering testcases that can trigger vulnerabilities

Problem: How to effectively search for interesting testcases

Existing works: rule-based mutation
Our solution: data-driven mutation
  - Fuzzing as an optimization problem => Gradient-guided mutation
Input Space of Program

High-dimensional and discrete input space

\[ X = [x_1, x_2, x_3, ..., x_n] \]

\begin{align*}
x_1 & \quad 0 & 1 & 2 & 3 & 4 & . & . & . & . & . & 255 \\
x_2 & \quad 0 & 1 & 2 & 3 & 4 & . & . & . & . & . & 255 \\
\vdots & \quad \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
x_n & \quad 0 & 1 & 2 & 3 & 4 & . & . & . & . & . & 255 \\
\end{align*}

n is the length of the input

Total possible inputs = \(256^n\)

Random mutation in huge search space is inefficient
Overview of NEUZZ

Program Inputs → EXE → Program Behaviors

Gradient-guided mutation

Program Inputs → NN → Program Behaviors

Smooth Approximation

Discrete and Non-differentiable

Smooth and Differentiable
K-Scheduler: Data-Driven Scheduling

Background: fuzzing needs to **choose a seed** during the search

Problem: How to choose a **promising seed** from seed corpus

Existing work: rule-based selection
Our approach: Data-driven scheduling
  - Fuzzing as an **influence analysis** problem => Graph centrality analysis
Overview of K-Scheduler

We use graph centrality score to estimate the search gain of each seed.
Future Directions

• Neural-symbolic software testing
• LLM-assisted program analysis
Neural-Symbolic Software Testing

Software testing with domain knowledge

- Smart contract, Network protocol, Autonomous driving, Deep Learning API

Symbolic Module
- Domain knowledge
  - Temporal Logic
  - Fuzzy Logic
  - Solver-based techniques

Neural Module
- Learn from data
  - NN approximation
  - Optimizations
  - Sampling-based techniques

Neural-Symbolic Software Testing

Explore the domain-specific software testing in a neural-symbolic way

Robustness
Interpretability

Expressiveness
Scalability
LLM-Assisted Program Analysis

Leverage LLM’s capability of **code comprehension** and **code summary** to boost traditional program analysis tasks

- Dataflow analysis, vulnerability detection (e.g., race condition, memory corruption, integer overflow), software testing (e.g., fuzzing)

• Task decomposition
• Automatic prompt generation
• Retrieval augmented generation