Lec 22: Fuzzing

CS492E: Introduction to Software Security

Sang Kil Cha
Software Bugs

• Bugs are plentiful
• Some bugs are memory corruption, some bugs are not
• Bugs are bad: attackers exploit bugs
Build a System that Finds Bugs

Program → System → Bugs

a.k.a. analyzer, fuzzer, etc.
Precision Matters

Program → System → Bugs

How *precise* can we make our system?
Given an arbitrary program, can we build a system that decides whether the program is buggy or not?
Informal Proof

Define a function \texttt{isBuggy} that takes a program as input, and outputs true if the program has at least one bug, and false if otherwise. Let’s assume that this function exists:

\begin{verbatim}
def isBuggy(prog):
    ... # somehow test prog and returns true or false
\end{verbatim}
Informal Proof

Define a function \texttt{myProg}:

\begin{verbatim}
def myProg(): # consider myProg as a program
    if isBuggy(myProg):
        return # do nothing (normal)
    else:
        corruptMemory()
        showBuggyBehavior()
        return
\end{verbatim}

Self contradictory
Building a Perfect Analyzer is Impossible

But, we can try to find as many bugs as possible.

For example,
• Bounded model checking
• Static analysis
• Software testing
• Etc.
Defining Precision (Soundness vs. Completeness)

If an analyzer is **sound**:

![Diagram showing the relationship between Truth and What I say]
Defining Precision
(Soundness vs. Completeness)

If an analyzer is \textit{complete}:

What I say

Truth
Defining Precision (Soundness vs. Completeness)

If an analyzer is *sound and complete* (= perfect):

What I say = Truth
Precision, Recall, and Accuracy

$\frac{TP}{TP + FP}$

$\frac{TP}{FN + TP}$

$\frac{TP + TN}{U}$

- Precision
- Recall
- Accuracy
False-Positive Rate vs. False-Negative Rate

- **FP Rate**
  \[ \text{FP Rate} = \frac{FP}{TP + FP} \]
- **FN Rate**
  \[ \text{FN Rate} = \frac{FN}{FN + TN} \]

- **FP**
- **TN**
- **TP**
- **FN**

- What I say

- Truth

U
Fuzzing?
A software testing technique for finding software bugs
History of Fuzzing

The original work was inspired by being logged on to a modem during a storm with lots of line noise. And the line noise was generating junk characters that seemingly were causing programs to crash. The noise suggested the term *fuzz*.

The term was coined by *Barton Miller* in 1990.
Fuzzing in 1990s

An Empirical Study of the Reliability of UNIX Utilities, *CACM 1990*
Fuzzing in 1990s

An Empirical Study of the Reliability of UNIX Utilities, CACM 1990
Fuzzing in 1990s

An Empirical Study of the Reliability of UNIX Utilities,
*CACM 1990*

- Only printable characters?
- Generate NULL character or not?
- Etc.
Fuzzing is ...

- *Simple*, and popular way to find security bugs
- Used by security practitioners

- But, *not studied systematically until recently (~2013)*
  - Why fuzzing works so well in practice?
  - Are we maximizing the ability of fuzzing?

Can we answer these questions?
Rough History of Fuzzing

* Visit https://fuzzing-survey.org/ to learn more
Fuzzing is an Overloaded Term

- White-box, black-box, grey-box fuzzing
- Directed fuzzing, Feedback-driven fuzzing
- Generational fuzzing
- Mutational fuzzing
- Grammar-based fuzzing
- Seed-based fuzzing
- Model-based, model-less fuzzing
- Etc.
Black-box vs. White-box Fuzzing
Grey-Box Fuzzing

• White-box fuzzing (strictly speaking)

• Obtain some partial information about the program execution
Mutation- vs. Generation-based Fuzzing

- **Seed**: an input to a program
- Mutation: mutate a given seed to generate test cases
- Generation: generate test cases from a model
Why Mutation?

Random inputs are likely to be rejected
Many Questions Remain

• Given a seed, how do we mutate the seed?

• How much portion do we mutate from the seed?

• How do we obtain seeds?
Why Generation?

Empty model = Random fuzzing

Random inputs are likely to be rejected!
Grammar-based Fuzzing

- Fuzzing compiler/interpreter
- Fuzzing VMs (Virtual Machines)
Fuzzing Algorithm
Key Properties of Fuzzing

• Generate test cases

• Run the program under test with the test cases

• Check if the program crashes
Definitions

- *Fuzzing* is the execution of the program using input(s) sampled from an input space that protrudes the expected input space of the PUT.

- *Fuzz testing* is the use of fuzzing to test if a program violates a correctness policy (e.g., security policy).
Definitions

• A **fuzz configuration** of a fuzz algorithm comprises the parameter value(s) that control(s) the fuzz algorithm.

• A **bug oracle** \((O_{\text{bug}})\) is a program, perhaps as part of a fuzzer, that determines whether a given execution of the program violates a specific security policy.
Fuzzing Algorithm

Conf_1 → Preprocess → Conf_1

Schedule

Bug Oracle
Conf

InputGen → InputEval

Test Case

ConfUpdate

Exec Info.

ok

crash
Fuzzing Algorithm

Algorithm 1: Fuzz Testing

\textbf{Input: } \mathcal{C}, t_{\text{limit}}

\textbf{Output: } \mathcal{B}  // a finite set of bugs

1. \( \mathcal{B} \leftarrow \emptyset \)
2. \( \mathcal{C} \leftarrow \text{Preprocess}(\mathcal{C}) \)
3. \textbf{while} \( t_{\text{elapsed}} < t_{\text{limit}} \wedge \text{Continue}(\mathcal{C}) \) \textbf{do}
   4. \( \text{conf} \leftarrow \text{Schedule}(\mathcal{C}, t_{\text{elapsed}}, t_{\text{limit}}) \)
   5. \( \text{tc} \leftarrow \text{InputGen}(\text{conf}) \)
      \hspace{1cm} \text{// } O_{\text{bug}} \text{ is embedded in a fuzzer}
   6. \( \mathcal{B}', \text{execinfo} \leftarrow \text{InputEval}(\text{conf}, \text{tc}, O_{\text{bug}}) \)
   7. \( \mathcal{C} \leftarrow \text{ConfUpdate}(\mathcal{C}, \text{conf}, \text{execinfo}) \)
   8. \( \mathcal{B} \leftarrow \mathcal{B} \cup \mathcal{B}' \)
4. \textbf{return} \( \mathcal{B} \)
Fuzzing is AI!

Finding paths in a maze
1. Move the agent based on the knowledge
2. Observe the environment (walls, passages, etc.)
3. Update the learnt knowledge
4. Goto 1
Research Challenges?

Conf₁ → Preprocess → Conf₁

Schedule → Bug Oracle

Conf → InputGen

ConfUpdate → Exec Info.

InputEval → Test Case

ok

crash
Questions?