Software Protection: Security Vulnerability Detection

Shuai Wang

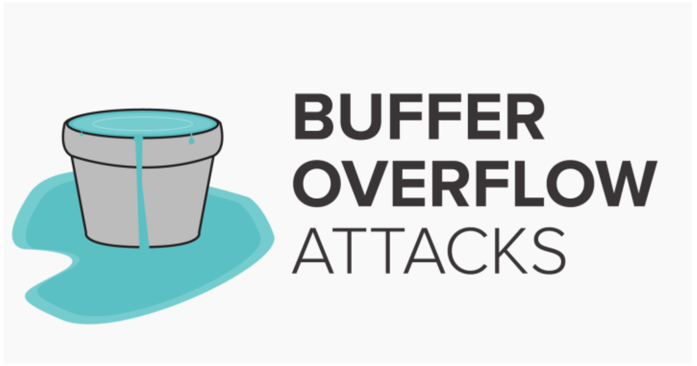
香 港 科 技 大 學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Some of the slides are written by Suman Jana.

How to find vulnerabilities in software?

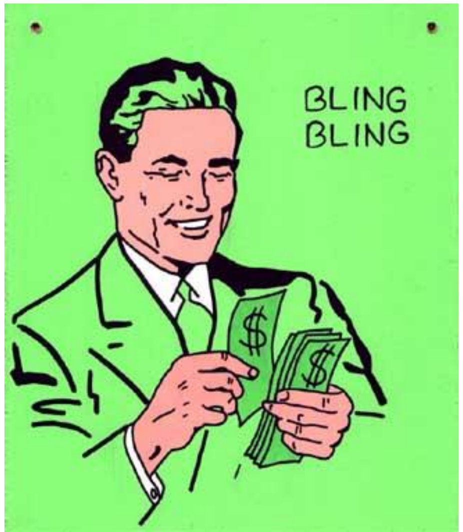
```
#include <stdio.h>
#define MAX_IP_LENGTH 15
int main(void) {
  char file_name[] = "ip.txt";
  FILE *fp;
  fp = fopen(file_name, "r");
  char ch;
  int counter = 0;
  char buf[MAX_IP_LENGTH];
  while((ch = fgetc(fp)) != EOF) {
    buf[counter++] = ch;
  }
```

Buffer overflow if “ip.txt” has more than 15 bytes.



Vulnerability Finding Today

- Security bugs can bring \$500-\$100,000 on the open market
- Good bug finders make \$180-\$250/hr consulting
- Few companies can find good people, many don't even realize this is possible.
    - Google: Team Zero; Tencent: Keen Lab; …
- Still largely a black art



Security Vulnerabilities

- What can an attacker do with Security bugs?
    - avoid authentication
    - privilege escalation
    - bypass security check
    - deny service
    - run code remotely
- Basically whatever you want.

Automatic Vulnerability Detection

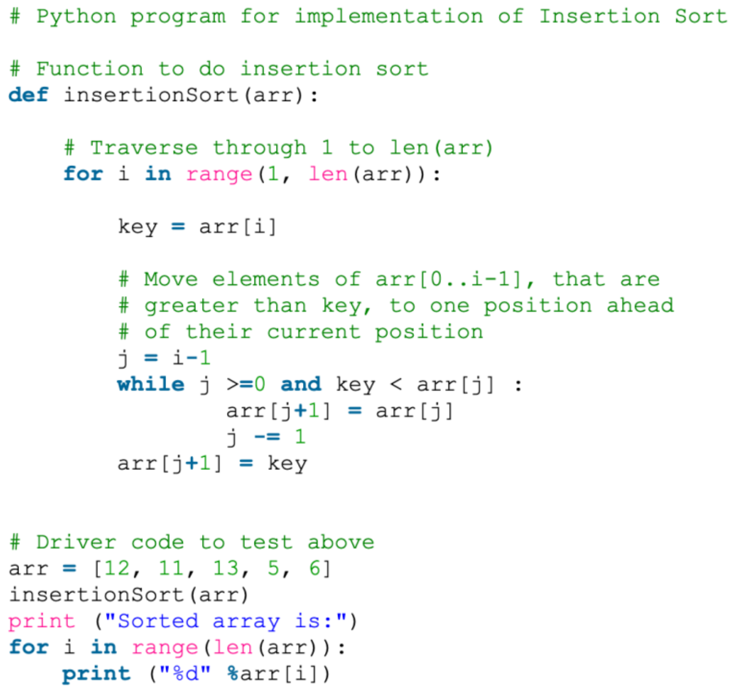
Find a needle in a haystack

Automatic Vulnerability Detection

- Fuzz testing ← discuss today.
- Taint analysis (information flow) ← discuss today
- Concolic execution ← *more comprehensive than testing*
- Symbolic execution ← *more comprehensive than testing*
- Type system ← brief
- Formal verification ← brief
- …

Formal Verification

- Formal verification can (ideally) completely eliminate vulnerabilities.
  - Mathematically prove the absence of bugs.
- How to I know the insertion sort will always return a sorted list?



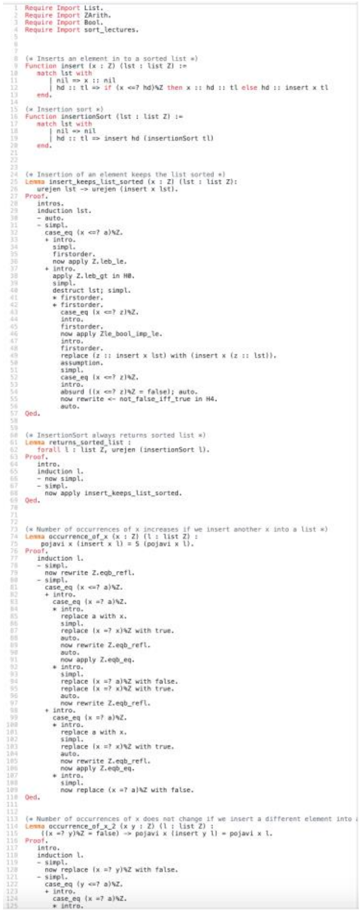
```python
# Python program for implementation of Insertion Sort

# Function to do insertion sort
def insertionSort(arr):

    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):

        key = arr[i]

        # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i-1
        while j >=0 and key < arr[j] :
                arr[j+1] = arr[j]
                j -= 1
        arr[j+1] = key


# Driver code to test above
arr = [12, 11, 13, 5, 6]
insertionSort(arr)
print ("Sorted array is:")
for i in range(len(arr)):
    print ("%d" %arr[i])
```

Formal Verification

- You can prove it, as how you prove some Euclidean geometry properties.

```python
# Python program for implementation of Insertion Sort

# Function to do insertion sort
def insertionSort(arr):

    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):

        key = arr[i]

        # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i-1
        while j >=0 and key < arr[j] :
                arr[j+1] = arr[j]
                j -= 1
        arr[j+1] = key


# Driver code to test above
arr = [12, 11, 13, 5, 6]
insertionSort(arr)
print ("Sorted array is:")
for i in range(len(arr)):
    print ("%d" %arr[i])
```

Proof



```coq
Require Import List.
Require Import ZArith.
Require Import Bool.
Require Import sort_lectures.



(* Inserts an element in to a sorted list *)
Function insert (x : Z) (lst : list Z) :=
   match lst with
       | nil => x :: nil
       | hd :: tl => if (x <=? hd)%Z then x :: hd :: tl else hd :: insert x tl
   end.

(* Insertion sort *)
Function insertionSort (lst : list Z) :=
   match lst with
       | nil => nil
       | hd :: tl => insert hd (insertionSort tl)
   end.



(* Insertion of an element keeps the list sorted *)
Lemma insert_keeps_list_sorted (x : Z) (lst : list Z):
   urejen lst -> urejen (insert x lst).
Proof.
   intros.
   induction lst.
   - auto.
   - simpl.
     case_eq (x <=? a)%Z.
     + intro.
       simpl.
       firstorder.
       now apply Z.leb_le.
     + intro.
       apply Z.leb_gt in H0.
       simpl.
       destruct lst; simpl.
       * firstorder.
       * firstorder.
         case_eq (x <=? z)%Z.
         intro.
         firstorder.
         now apply Zle_bool_imp_le.
         intro.
         firstorder.
         replace (z :: insert x lst) with (insert x (z :: lst)).
         assumption.
         simpl.
         case_eq (x <=? z)%Z.
         intro.
         absurd ((x <=? z)%Z = false); auto.
         now rewrite <- not_false_iff_true in H4.
         auto.
Qed.


(* InsertionSort always returns sorted list *)
Lemma returns_sorted_list :
   forall l : list Z, urejen (insertionSort l).
Proof.
   intro.
   induction l.
   - now simpl.
   - simpl.
     now apply insert_keeps_list_sorted.
Qed.



(* Number of occurrences of x increases if we insert another x into a list *)
Lemma occurrence_of_x (x : Z) (l : list Z) :
    pojavi x (insert x l) = S (pojavi x l).
Proof.
   induction l.
   - simpl.
     now rewrite Z.eqb_refl.
   - simpl.
     case_eq (x <=? a)%Z.
     + intro.
       case_eq (x =? a)%Z.
       * intro.
         replace a with x.
         simpl.
         replace (x =? x)%Z with true.
         auto.
         now rewrite Z.eqb_refl.
         auto.
         now apply Z.eqb_eq.
       * intro.
         simpl.
         replace (x =? a)%Z with false.
         replace (x =? x)%Z with true.
         auto.
         now rewrite Z.eqb_refl.
     + intro.
       case_eq (x =? a)%Z.
       * intro.
         replace a with x.
         simpl.
         replace (x =? x)%Z with true.
         auto.
         now rewrite Z.eqb_refl.
         now apply Z.eqb_eq.
       * intro.
         simpl.
         now replace (x =? a)%Z with false.
Qed.


(* Number of occurrences of x does not change if we insert a different element into
Lemma occurrence_of_x_2 (x y : Z) (l : list Z) :
    ((x =? y)%Z = false) -> pojavi x (insert y l) = pojavi x l.
Proof.
   intro.
   induction l.
   - simpl.
     now replace (x =? y)%Z with false.
   - simpl.
     case_eq (y <=? a)%Z.
     + intro.
       case_eq (x =? a)%Z.
       * intro.
```

Computer will check the correctness

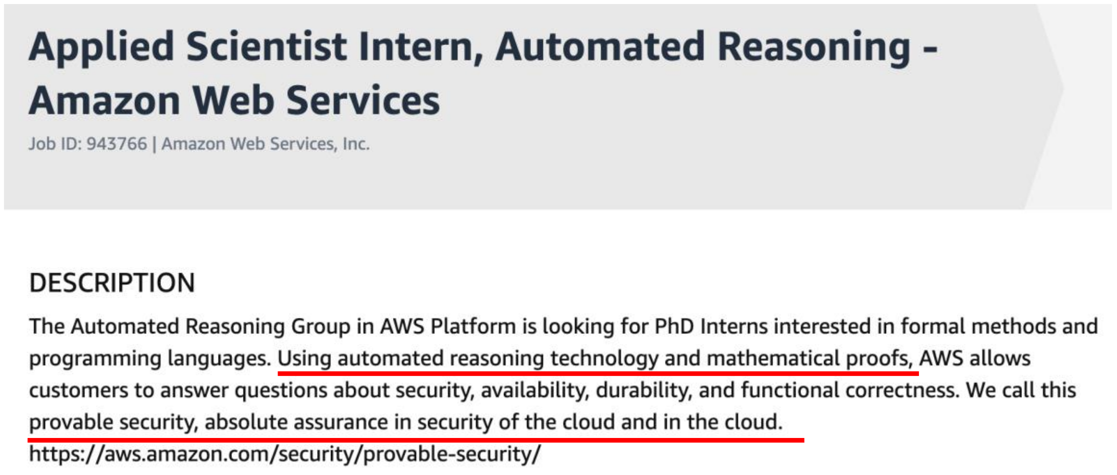
Haven’t finished yet..

Formal Verification

- Two important elements in a formal verification:
  - Specification ← what properties I would like to prove?
    - Sorting? Or memory safety?
  - Inductive proofs
    - You write your proof as code, and computers will check the correctness of your proof.
      - Difficult to manually write the proof, but relatively easy to verify a given proof.

But no solution is perfect! Even formal verification

- But impractical in general…
  - Formal verification is hard in general, impossible for big things.
  - Take a Ph.D. several years, write million lines (?) of proof code, to verify (simple) memory safety properties of a tiny OS kernel component
  - 2~5 x N lines of proof for N lines of code, or even worse.
- Adoption of formal verification in the industry



Applied Scientist Intern, Automated Reasoning - Amazon Web Services

Job ID: 943766 | Amazon Web Services, Inc.

DESCRIPTION

The Automated Reasoning Group in AWS Platform is looking for PhD Interns interested in formal methods and programming languages. Using automated reasoning technology and mathematical proofs, AWS allows customers to answer questions about security, availability, durability, and functional correctness. We call this provable security, absolute assurance in security of the cloud and in the cloud.

https://aws.amazon.com/security/provable-security/

Type System

- Type system (safe language)
  - Provide strong guarantee on well-typed programs
    - Suppose we have two variables N and M in our financial trading software.
    - N x M ← seems all right
    - N x M ← but not correct if N's type is US Dollars and M's type is HK Dollars
  - this doesn't eliminate **all** problems, although with such "annotations" we can quickly pinpoint some defects.
    - But type check is usually super fast, and does not require much extra cost.
    - Performance, existing code base, flexibility, programmer capability, etc.
  - Financial trading companies are major sponsors of the safe language communities
    - Java; C#; Scala; OCaml; Haskell
    - Jane Street; Goldman Sachs

Automatic Vulnerability Detection

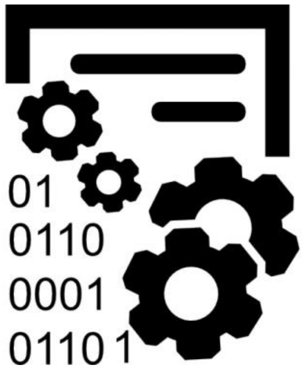
- Fuzz testing ← discuss today.
- Taint analysis (information flow) ← discuss today
- Concolic execution ← *more comprehensive than testing*
- Symbolic execution ← *more comprehensive than testing*
- Type system
- Formal verification
- …

Fuzz Testing

A simple but very well-performing testing method to find vulnerabilities.

- Automatically generate test cases
- Many abnormal/random/corner-case test cases are input into a target
- Application is monitored for errors

Random input →



Test program

Miller et al. '89

Fuzz Testing

- Given a program simply feed **random** inputs **to execute** and see whether it trigger errors **(usually "crash"; see the next slide)**
  - To execute:
    - We find a bug and we know how to trigger it at the same time
    - Why is this so critical?
      - Very likely we (attacker) can **control** the bug!
  - Random:
    - We find many corner cases by using invalid inputs
      - Invalid inputs: initial design goal, but extended with "grammar-aware" fuzzing later.
    - In contrast, many of the software testing techniques assert correct functionality with well-formed inputs.

Fuzz Testing

- Given a program simply feed **random** inputs **to execute** and see whether it trigger errors
  - Errors
    - Crash: highly like indicates memory access errors and therefore cause memory overread/overwrite vulnerabilities
    - Hang: system lost response (not "available" anymore)
    - Race condition: need to enable dynamic error detector…
    - Other errors: user-after free; double-free (*introduce later this semester*)
      - More stealthy than direct crash
      - Run program under dynamic memory error detector
    - But fuzz testing is usually not used to find logic errors.

Software Testing vs. Fuzzing

A very high-level comparison between software testing (e.g., regression testing) and fuzz testing.

	Software Testing	**Fuzzing**
Definition	Run program on many normal inputs, look for violation of testing oracle	Run program on many abnormal inputs, look for badness
Goals	Prevent normal users from encountering errors	Prevent attackers from identifying exploitable errors

Fuzz Testing Techniques

Knowledge of inputs

Knowledge of the fuzzed software

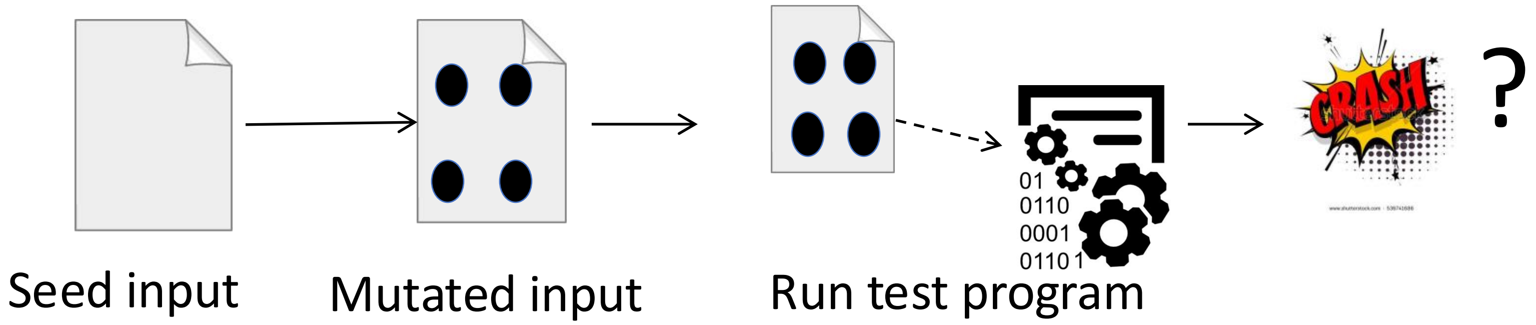
Mutation strategies

	Black Box (old)	Gray box (State of the art)
Random-mutation	Easy but ineffective	Industry Strength
Generation-based mutation	Not bad	Industry Strength

However, it needs input specifications which are often not available.

Mutation-Based fuzzing

- Take a ~~well-formed~~ well-formed or random inputs, randomly perturb (flipping bit, etc.)
    - although "well-formed" inputs are generally more desired and empirically better.
- Little or no knowledge of the structure of the inputs is assumed
- Anomalies are added to existing inputs
    - Anomalies may be completely random or follow some heuristics (e.g., remove NULL, shift character forward)
- Examples: ZZUF, Taof, GPF, ProxyFuzz, FileFuzz, Filep, etc.



Example: fuzzing a PDF viewer

- Google for .pdf (about 1 billion results)
- Crawl pdf files.
- Use fuzzing tool (or script)
  - Collect seed PDF files
  - Randomly mutate a file
  - Feed it to the program
  - Record if it crashed (and input that crashed it)

Mutation-based fuzzing

- Super easy to setup and automate
- Little or no input format knowledge is required
- May fail at the early stage:
  - Parsing; format checking; checksum; …
  - Good or bad?

Fuzz Testing Techniques

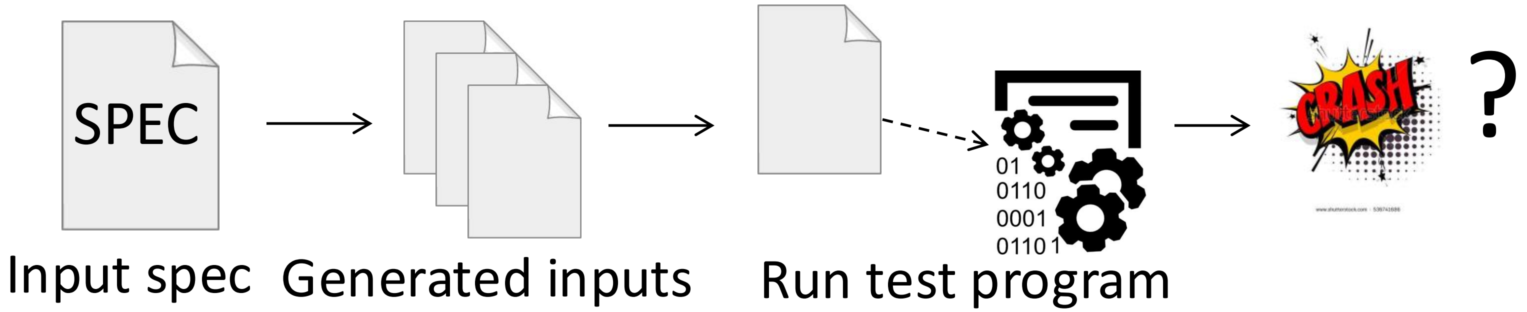
Mutation strategies

	Black Box (old)	Gray box (State of the art)
Random-mutation	Easy but ineffective	Industry Strength
Generation-based mutation	Not bad	Industry Strength

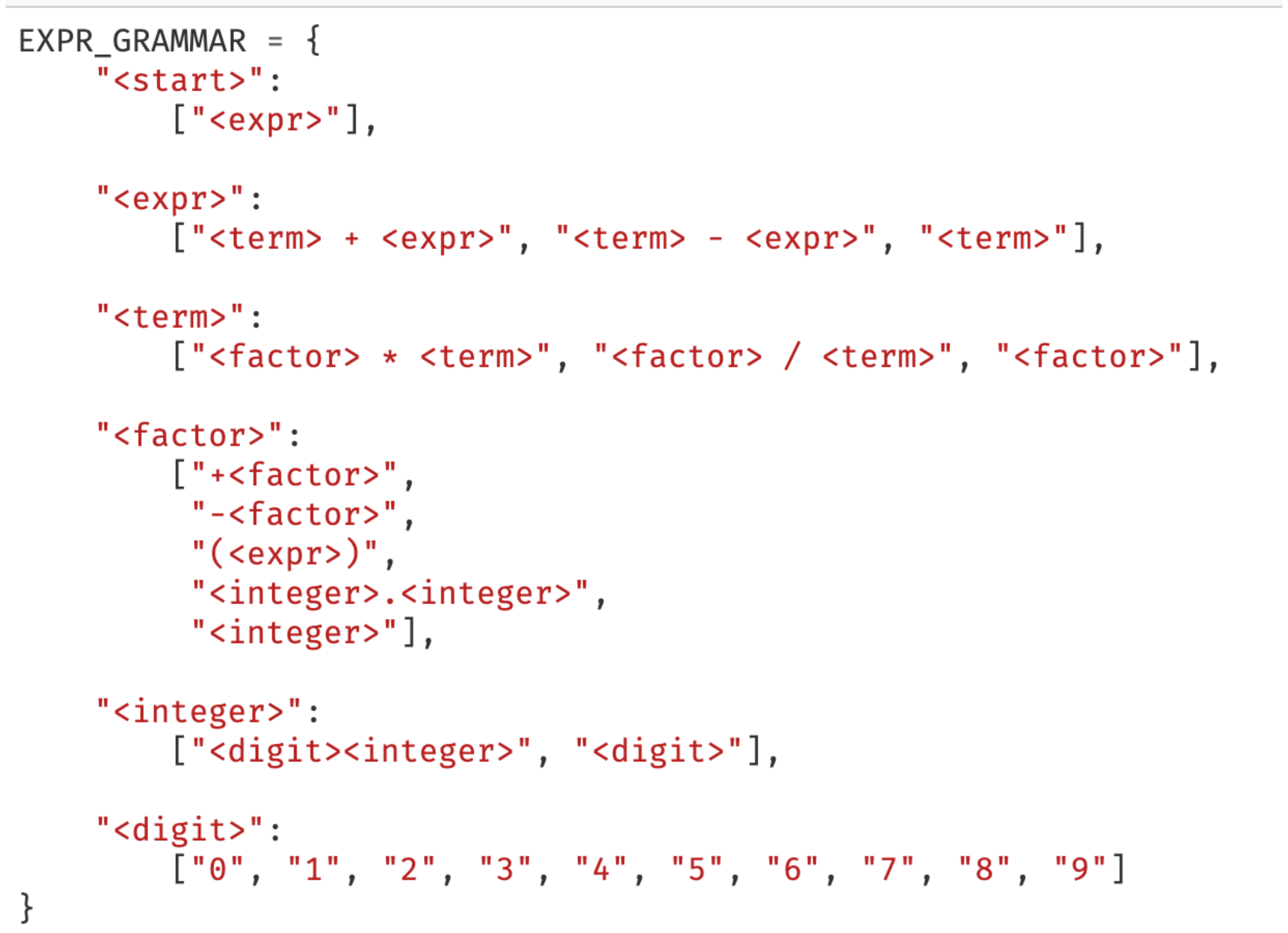
However, it needs input specifications which are often not available.

Generation-Based (Grammar-Aware) Fuzzing

- Test cases are generated from some description of the input format: documentation, etc.
  - Using specified protocols/file format info
- Pre-knowledge of protocol/format should give better results than random fuzzing
  - By being able to directly generate many structured inputs
  - Often work together with random mutation to explore different components.



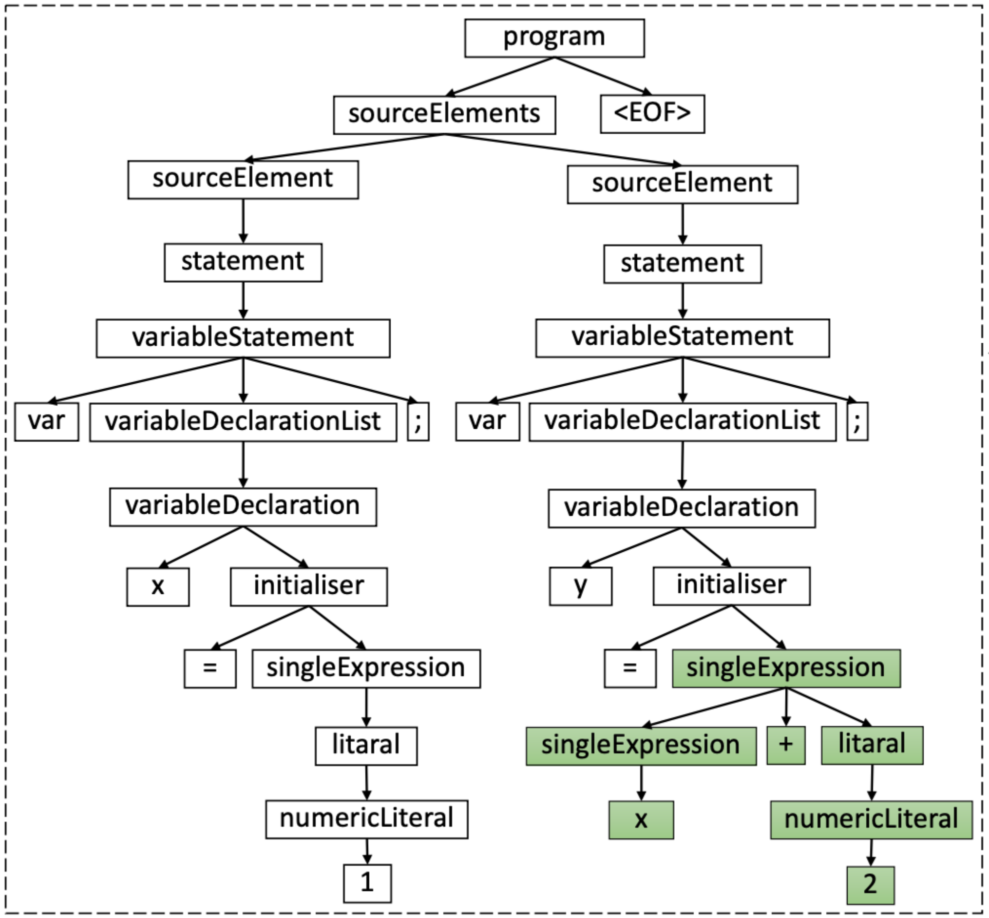
Generation-Based Fuzzing



```
EXPR_GRAMMAR = {
    "<start>":
        ["<expr>"],

    "<expr>":
        ["<term> + <expr>", "<term> - <expr>", "<term>"],

    "<term>":
        ["<factor> * <term>", "<factor> / <term>", "<factor>"],

    "<factor>":
        ["+<factor>",
         "-<factor>",
         "(<expr>)",
         "<integer>.<integer>",
         "<integer>"],

    "<integer>":
        ["<digit><integer>", "<digit>"],

    "<digit>":
        ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]
}
```

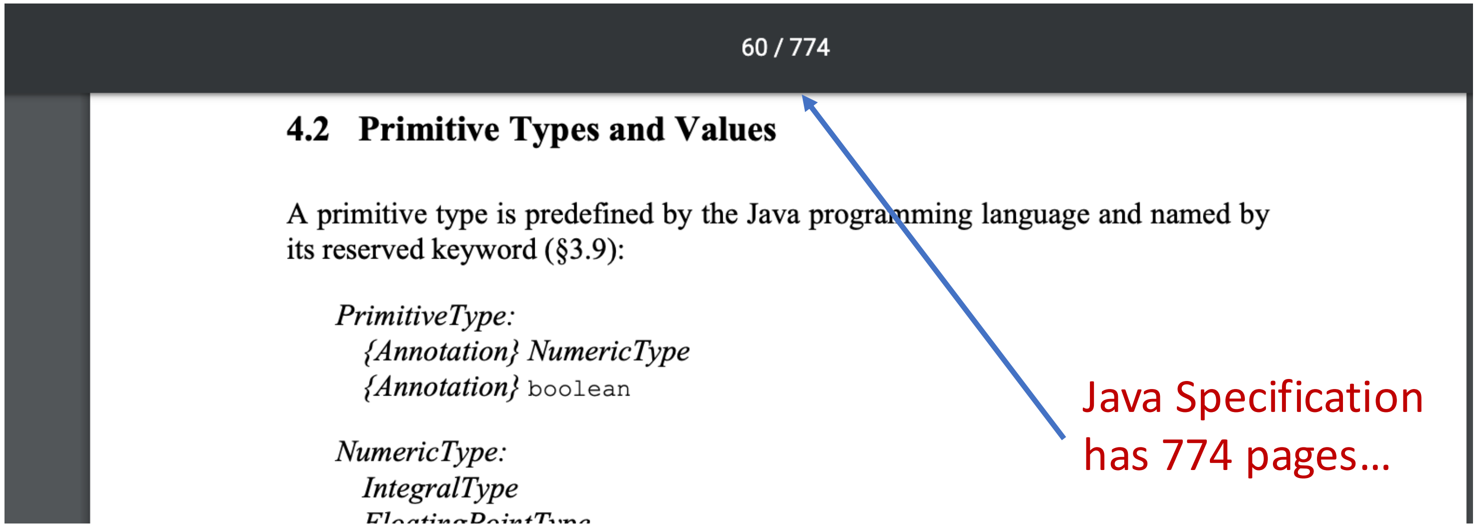
Grammar of a simple calculator

Structure-Based Fuzzers



How to fuzz Python/JavaScript Interpreter?
--> tree structure mutation of syntax valid programs

But grammars could be very complex



60 / 774

4.2 Primitive Types and Values

A primitive type is predefined by the Java programming language and named by its reserved keyword (§3.9):

PrimitiveType:
  {Annotation} NumericType
  {Annotation} `boolean`

NumericType:
  IntegralType

Java Specification has 774 pages…

It could be difficult to define good generation rules for real-world cases..

➔ State-of-the-art: use deep learning (language models) to “learn” this knowledge…

Mutation-based vs. Generation-based

- Mutation-based fuzzer
  - Pros: Easy to set up and automate, little to no knowledge of input format required
  - Cons: Limited by initial input seeds, may fall for protocols with checksums and other hard checks
- Generation-based fuzzers
  - Pros: completeness, theoretically can deal with complex dependencies
  - Cons: writing generators is hard, performance depends on the quality of the SPEC ➔ very often difficult to get SPEC
- Luckily, “structure/grammar” information could be partially inferred on the fly!!!
  - With some genetic algorithm ➔ grey box fuzzing.

Fuzz Testing Techniques

Knowledge of inputs

Knowledge of the fuzzed software

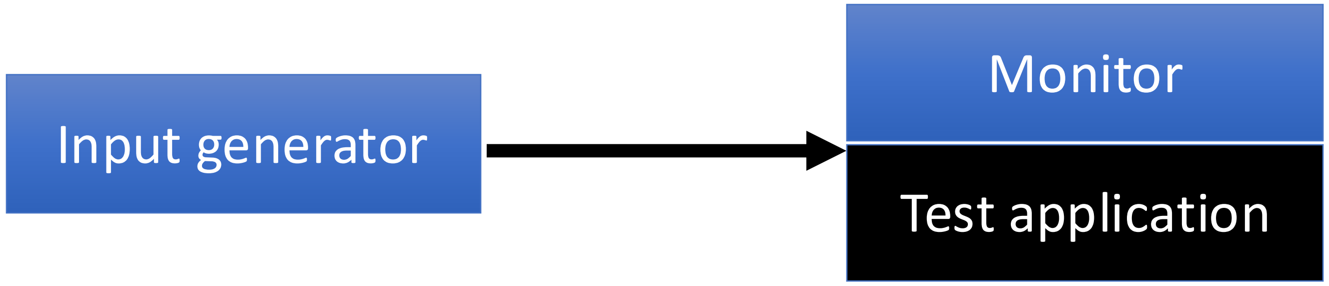
Mutation strategies

	Black Box (old)	Gray box (State of the art)
Random-mutation	Easy but ineffective	Industry Strength
Generation-based mutation	Not bad	Industry Strength

However, it needs input specifications which are often not available.

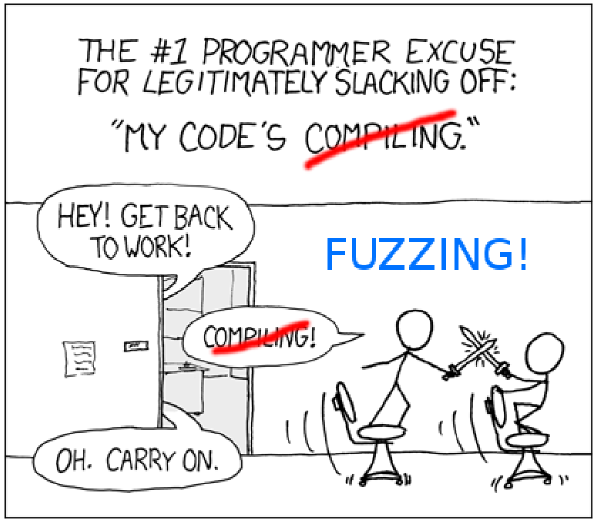
Blackbox Fuzzing

- Keep generating mutated inputs and feed to the test application.
- Application is monitored for errors



Pros/Cons of Blackbox Fuzzing

- Advantage: easy, low programmer cost
  - Well, every "script kid" know blackbox fuzzing from day one…
- Disadvantage: inefficient (particularly for random mutation)
  - Do your best and need some luck
  - Inputs often require structures, random inputs are likely to be mal-formed and rejected at the early stage
  - Inputs that trigger an incorrect behavior is a very small fraction, probably of getting lucky is very low


THE #1 PROGRAMMER EXCUSE FOR LEGITIMATELY SLACKING OFF:
"MY CODE'S ~~COMPILING~~."
HEY! GET BACK TO WORK!
FUZZING!
~~COMPILING~~!
OH. CARRY ON.

How much fuzzing is enough?

- Fuzzer may generate an infinite number of test cases. When has the fuzzer run long enough?
    - Code coverage is used as the “feedback” to guide fuzzer.
- Code coverage is a metric that can be used to determine how much code has been executed.
    - Data can be obtained using a variety of profiling tools. e.g. gcov, lcov
    - Line coverage; branch coverage; path coverage;

Line coverage

- Line/block coverage: Measures how many lines of source code have been executed.
- For the code on the right, how many test cases (values of pair (a,b)) needed for full (100%) line coverage?

```
if( a >  2  )
      a =  2;
if( b >2  )
      b =  2;
```

Branch coverage

- Branch coverage: Measures how many branches in code have been taken (conditional jmps)
- For the code on the right, how many test cases needed for full branch coverage?

```
if( a >  2  )
     a =  2;
if( b >2  )
     b =  2;
```

Path coverage

- Path coverage: Measures how many paths have been taken
- For the code on the right, how many test cases needed for full path coverage?

```
if( a >  2  )
     a =  2;
if( b >2  )
     b =  2;
```

Benefits of Code coverage

- Can answer the following questions
  - How good is an initial file?
  - Am I getting stuck somewhere?

```
if (packet[0x10] < 7) {
                //hot path
  } else {
              //cold path
  }
```

- How good is fuzzerX vs. fuzzerY
- Am I getting benefits by running multiple fuzzers?

Fuzz Testing Techniques

Mutation strategies

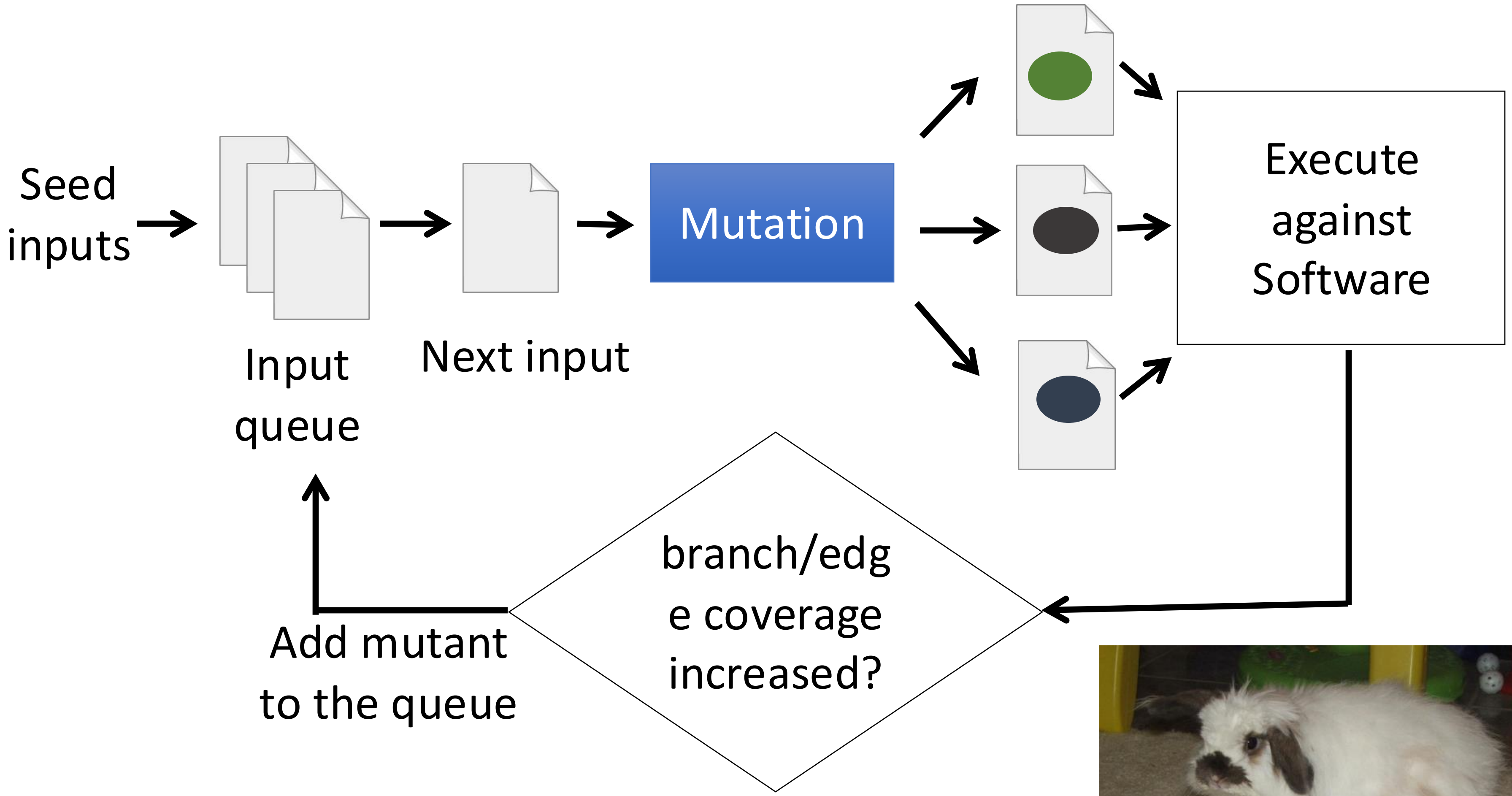
	Black Box (old)	Gray box (State of the art)
Random-mutation	Easy but ineffective	Industry Strength
Generation-based mutation	Not bad	Industry Strength

However, it needs input specifications which are often not available.

Coverage-guided GrayBox Fuzzing

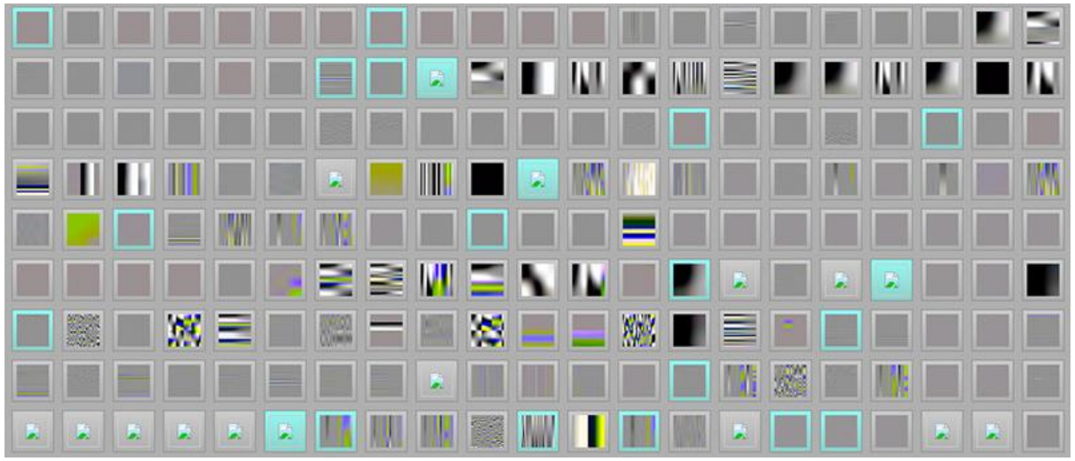
- The state of the art
  - Run mutated inputs and measure code coverage
  - Search for mutants that result in coverage increase
  - Often use (conceptually) a "genetic algorithm", i.e., try random mutations on test corpus and only keep mutants for further usage if coverage increases
  - Examples: AFL, libfuzzer

American Fuzzy Lop (AFL)



The powerful of genetic algorithm

Just random junk data…



Random images but with legit JPEG format!!!

Good coverage-guided fuzzer can gradually "synthesize" well-formed input format to certain degree, by keeping mutants with new coverage and proceed further.

The input with relatively more legitimate formats got more trials!

Fuzzing challenges

- How to seed a fuzzer?
  - (Ideally) seed inputs must cover different branches
  - Remove duplicate seeds covering the same branches
- Some branches might be very hard to get past as the # of inputs satisfying the conditions are very small
  - Manually/automatically transform/remove those branches

Hard to fuzz code

```
void test (int n) {
  if (n==0x12345678)
     crash();
}
```

needs 2^32 attempts
In the worst case

Make it easier to fuzz

```
void test (int n) {
   int dummy = 0;
   char *p = (char *)&n;
   if (p[3]==0x12) dummy++;
   if (p[2]==0x34) dummy++;
   if (p[1]==0x56) dummy++;
   if (p[0]==0x56) dummy++;
   if (dummy==4)
      crash();
}
```

The time complexity is largely reduced from 2^32 to roughly (2^8 + 2^8 + 2^8 + 2^8).

Fuzzing rules of thumb

- Input-format knowledge is very helpful
- Beat random with feedback guide, and better specs make better fuzzers
- Time matters
  - More fuzzing is obviously better
  - It is recommended to fuzz at least 12 hours for real-world applications.
- Best results come from guiding the process