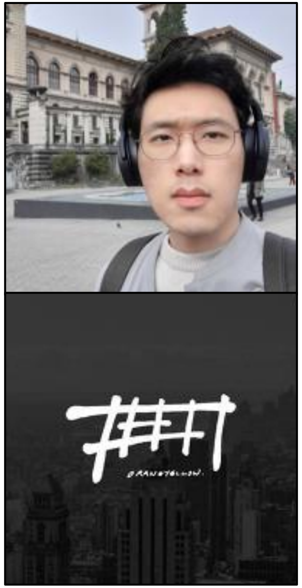
The Chronicle of
Software Vulnerability Detection

Gwangmu Lee hexhive

16.06.23 @ KYUNG HEE UNIVERSITY

Who Is This Guy?



Gwangmu Lee

Switched the field a few times.

- BS: Physics @ POSTECH
- MS: Compiler & Compiler Architecture @ POSTECH
- PhD: Computer Security @ SNU

Me back then:
What is a vulnerability?

Now settled in Computer Security.

- Currently a **post-doc** researcher @ EPFL (Switzerland)
  ("HexHive" led by Prof. Mathias Payer)

Some relevant addresses:

- https://hexhive.epfl.ch (lab website)
- iss300@gmail.com (my email)

About The **Vulnerability**

What Is A Vulnerability per Wikipedia

Vulnerabilities

Flaws in a system, which can be exploited by an attacker to perform unauthorized actions.

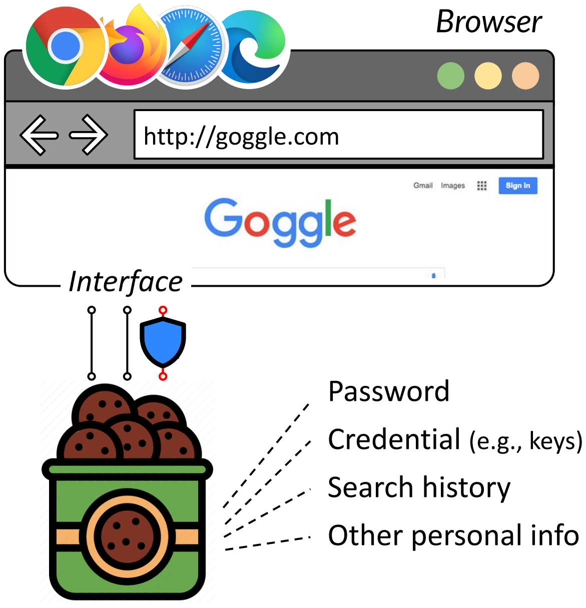
⊂

Software Bugs

Errors, flaws or faults in software that causes incorrect or unexpected behaviors.

Vulnerabilities in Action

Let's take an example from a web browser.



Password

Credential (e.g., keys)

Search history

Other personal info

Cookie storage

*Browsers hoard **tasty information** in its cookie storage.*

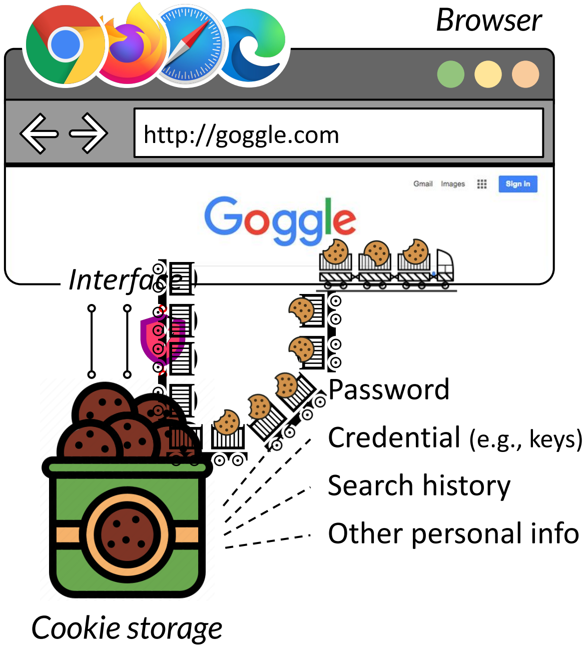
- Useful if used well, but critical if exposed.
- Browsers control access to cookies to prevent that.

*Now suppose **your browser has a bug**.*

- Some obscure site may try to take advantage of it.
- But if a bug doesn't meet some requirements, that attempt ought to be thwarted in the end.

Vulnerabilities in Action

Let's take an example from a web browser.



Browsers hoard ***tasty information*** *in its cookie storage.*

- Useful if used well, but critical if exposed.
- Browsers control access to cookies to prevent that.

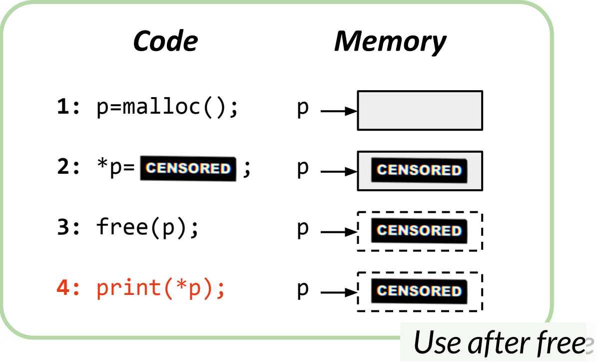
Now suppose ***your browser has a bug.***

- Some obscure site may try to take advantage of it.
- But if a bug doesn't meet some requirements, that attempt ought to be thwarted in the end.

Imagine this bug manages to ***open a way to cookies.***

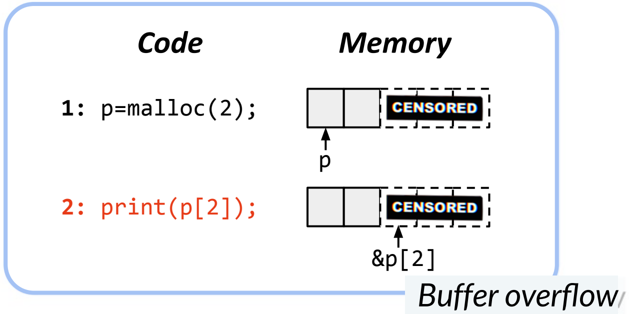
- Then this site can **exploit** this bug to steal data.
- Now this bug is called **vulnerability**.

Examples of Vulnerabilities Memory Bugs



```
1: p=malloc();
2: *p= CENSORED ;
3: free(p);
4: print(*p);
```

Use after free



```
1: p=malloc(2);
2: print(p[2]);
```

Buffer overflow

*Software itself is controlled by **memory.***
*Obviously, **memory bugs** are destined to be critical.*

- Collectively called *memory bugs* if it involves illegal read/write to memory.

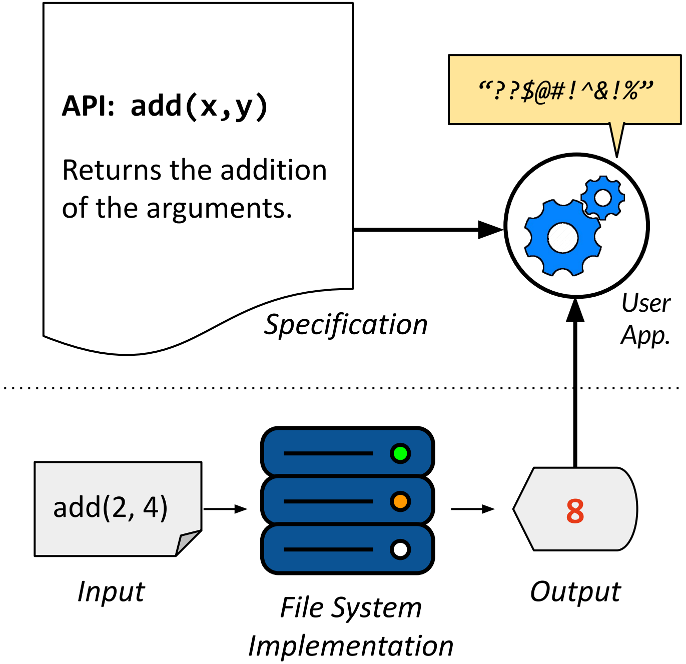
*Some examples of **illegal** memory access are;*

- Use after free (UAF): accessing freed memory.
- Buffer overflow (BO): accessing out of bound.

Repercussion 💀

- Stealing in-memory data (e.g., security keys).
- Hijacking the control to making it a *puppet*.
- …

Examples of Vulnerabilities Semantic Bugs



*Perfectly legal memory access can also wreak havoc, if it violates **high-level specifications.** (i.e. semantics)*

- Example: wrong return values from library APIs.
- "**`add(x,y)`** returned **x * y**"
- What if the caller acts up weirdly because of it?

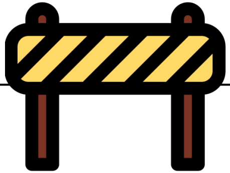
Repercussion 💀

- Data loss (i.e., attacker-controlled data corruption).
- Denial of service, and so on.

How to Mitigate Them?

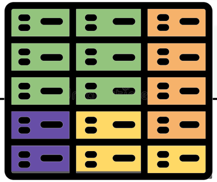
These are some representative software-based approaches.

Let's talk about this.



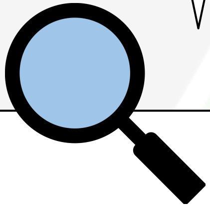
Runtime Defense

Detect weird behaviors at runtime and stop them to go further. (e.g., by terminating it)



Compartmentalize

Confine the impact of one vulnerability to a subset of the entire program.

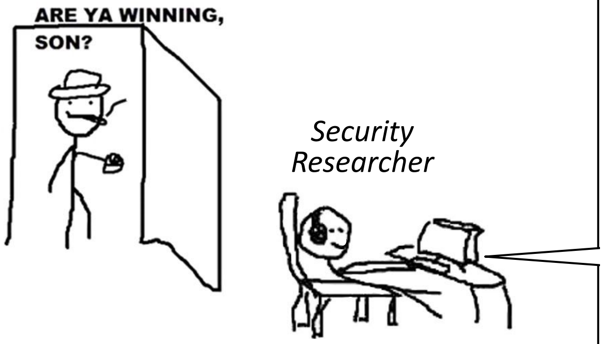


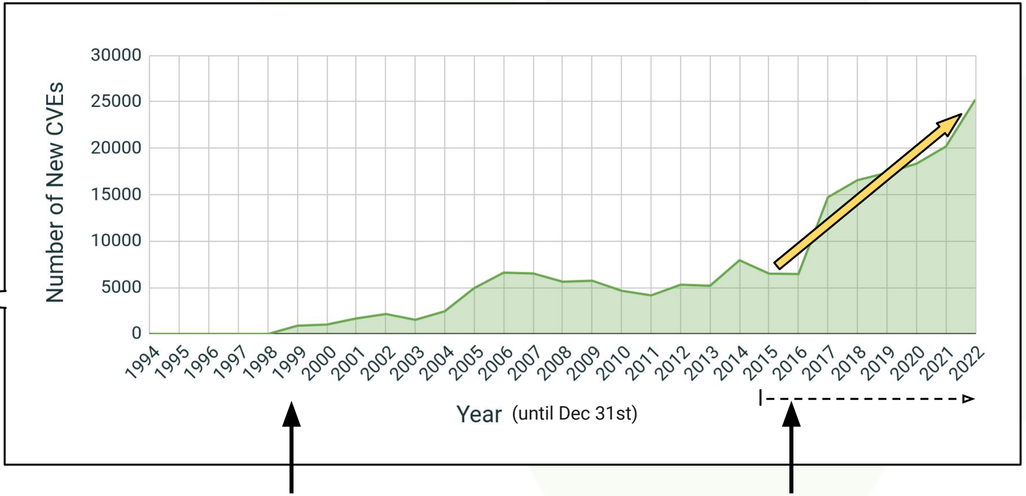
Early Detection

Detect and eradicate vulnerabilities as early as possible, before attackers.

Vulnerability Detection: Are We Winning?

Let's see whether vulnerability detection is paying off.





When ***CVE*** *was introduced. (1999)*

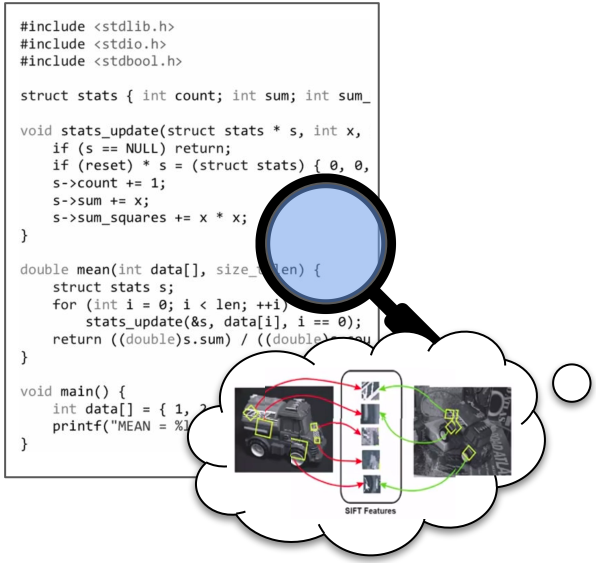
- Short for "Common Vulnerabilities and Exposures."
  - Roughly, recognized vulnerabilities in the wild.
- Mostly discovered and reported by researchers first.

Increasing Trend *(mid-2010~)*

- In 2017, even tripled in a year.
  - What happened here?

The **History** of Vulnerability Detection

Let’s Go Back in Time. In Early Years...



```
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>

struct stats { int count; int sum; int sum_

void stats_update(struct stats * s, int x,
    if (s == NULL) return;
    if (reset) * s = (struct stats) { 0, 0,
    s->count += 1;
    s->sum += x;
    s->sum_squares += x * x;
}

double mean(int data[], size_t len) {
    struct stats s;
    for (int i = 0; i < len; ++i)
        stats_update(&s, data[i], i == 0);
    return ((double)s.sum) / ((double)s.cou
}

void main() {
    int data[] = { 1, 2
    printf("MEAN = %l
}
```

Suppose you want to find vulnerabilities in code.

- A vulnerability is effectively **a set of rules**.
  (e.g., use after free; find uses after frees)

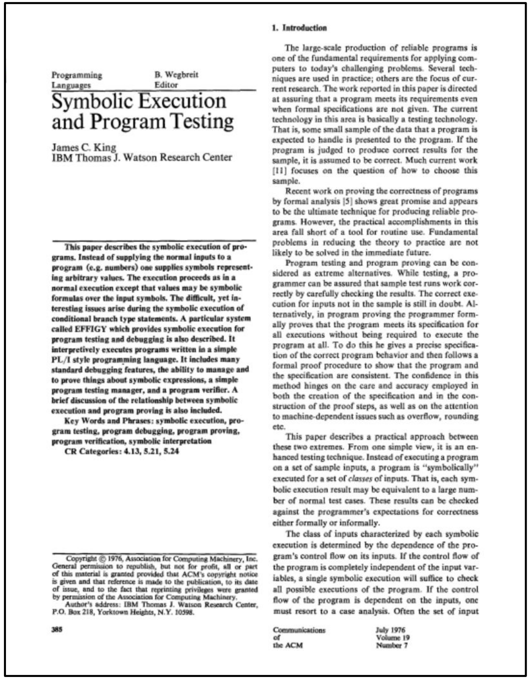
*Maybe? Can we just look into code and **analyze** it?*

- “Analytical approach”, that’s the most orthodoxical approach if it *seems* to be clear what to find.
- Similar to how CV started off with this approach.
  (like, “scale-invariant feature transformation”)

Two major analytical approaches

1) **Symbolic execution**
2) **Static analysis** (e.g., abstract interpretation)

Symbolic Execution Proposal



Programming Languages

B. Wegbreit Editor

Symbolic Execution and Program Testing

James C. King
IBM Thomas J. Watson Research Center

This paper describes the symbolic execution of programs. Instead of supplying the normal inputs to a program (e.g. numbers) one supplies symbols representing arbitrary values. The execution proceeds as in a normal execution except that values may be symbolic formulas over the input symbols. The difficult, yet interesting issues arise during the symbolic execution of conditional branch type statements. A particular system called EFFIGY which provides symbolic execution for program testing and debugging is also described. It interpretively executes programs written in a simple PL/I style programming language. It includes many standard debugging features, the ability to manage and to prove things about symbolic expressions, a simple program testing manager, and a program verifier. A brief discussion of the relationship between symbolic execution and program proving is also included.

Key Words and Phrases: symbolic execution, program testing, program debugging, program proving, program verification, symbolic interpretation

CR Categories: 4.13, 5.21, 5.24

1. Introduction

The large-scale production of reliable programs is one of the fundamental requirements for applying computers to today's challenging problems. Several techniques are used in practice; others are the focus of current research. The work reported in this paper is directed at assuring that a program meets its requirements even when formal specifications are not given. The current technology in this area is basically a testing technology. That is, some small sample of the data that a program is expected to handle is presented to the program. If the program is judged to produce correct results for the sample, it is assumed to be correct. Much current work [11] focuses on the question of how to choose this sample.

Recent work on proving the correctness of programs by formal analysis [5] shows great promise and appears to be the ultimate technique for producing reliable programs. However, the practical accomplishments in this area fall short of a tool for routine use. Fundamental problems in reducing the theory to practice are not likely to be solved in the immediate future.

Program testing and program proving can be considered as extreme alternatives. While testing, a programmer can be assured that sample test runs work correctly by carefully checking the results. The correct execution for inputs not in the sample is still in doubt. Alternatively, in program proving the programmer formally proves that the program meets its specification for all executions without being required to execute the program at all. To do this he gives a precise specification of the correct program behavior and then follows a formal proof procedure to show that the program and the specification are consistent. The confidence in this method hinges on the care and accuracy employed in both the creation of the specification and in the construction of the proof steps, as well as on the attention to machine-dependent issues such as overflow, rounding etc.

This paper describes a practical approach between these two extremes. From one simple view, it is an enhanced testing technique. Instead of executing a program on a set of sample inputs, a program is "symbolically" executed for a set of *classes* of inputs. That is, each symbolic execution result may be equivalent to a large number of normal test cases. These results can be checked against the programmer's expectations for correctness either formally or informally.

The class of inputs characterized by each symbolic execution is determined by the dependence of the program's control flow on its inputs. If the control flow of the program is completely independent of the input variables, a single symbolic execution will suffice to check all possible executions of the program. If the control flow of the program is dependent on the inputs, one must resort to a case analysis. Often the set of input

---

Author's address: IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, N.Y. 10598.

In the mid-70's, a series of papers proposed ***symbolically*** *executing programs. (as in, no concrete input values)*

- Input bytes as **symbols**, like mathematical variables.
- Describe a program state as **a function of those symbols**.
- Find if illegal program states are possible.

Rough mechanism sketch

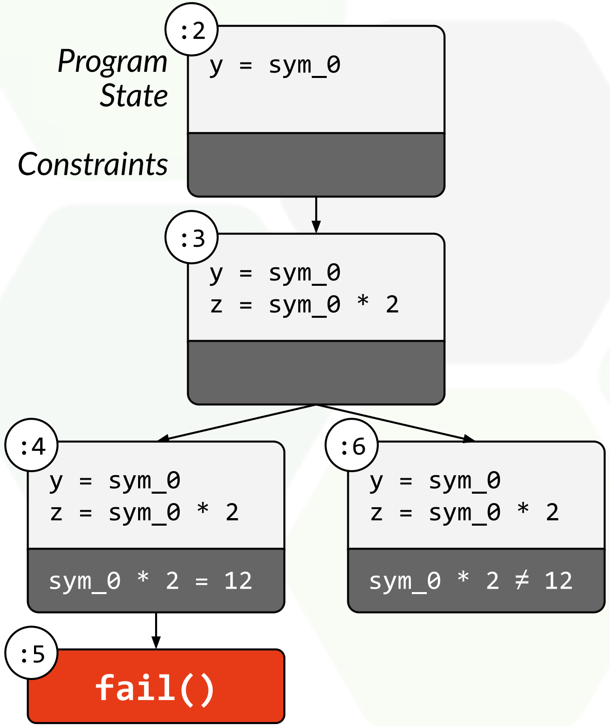
- Program state ⇒ a **function** of symbols.
- Branch (e.g., "if") ⇒ a **constraint** on those functions.
    - If a constraint is *satisfiable*,
      the following program state is also *possible*.
- See if some possible states are **illegal**.
  (e.g., an offset larger than the buffer size)

Symbolic Execution Example

Code stolen from Wikipedia ("Symbolic Execution")

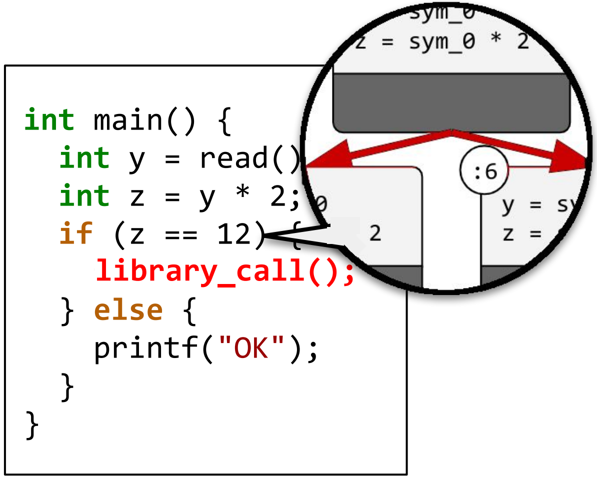
```
  1  int main() {
● 2    int y = read();
● 3    int z = y * 2;
● 4    if (z == 12) {
● 5      fail();
● 6    } else {
  7      printf("OK");
  8    }
  9  }
```

Program



Program State Graph

Symbolic Execution Ups and Downs



```
int main() {
  int y = read()
  int z = y * 2;
  if (z == 12) {
    library_call();
  } else {
    printf("OK");
  }
}
```

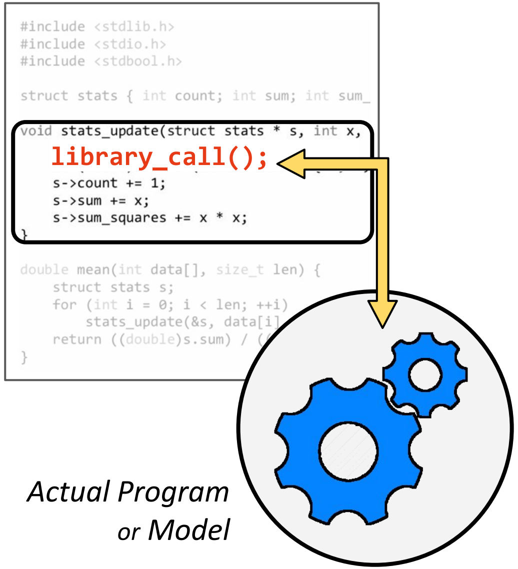
*Perfect and ideal, **if done faithfully.***

- Theoretically, you can completely investigate (almost) every single program state *before actually running it.*
- Works well with small simple programs.

*The caveat here is “faithfully”, because **we may not.***

1) Increasing program states against branches, **<u>exponentially</u>.** (i.e., one branch doubles up the # of states)
2) Non-analyzable code. (e.g., library calls)

Symbolic Execution Development



```
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>

struct stats { int count; int sum; int sum_

void stats_update(struct stats * s, int x,
   library_call();
    s->count += 1;
    s->sum += x;
    s->sum_squares += x * x;
}

double mean(int data[], size_t len) {
    struct stats s;
    for (int i = 0; i < len; ++i)
        stats_update(&s, data[i]
    return ((double)s.sum) / ((
}
```

Actual Program
or *Model*

Improvement mostly made in the early 2010's.

*Analyze **less branches** to avoid exploding states.*

1) Don't analyze *the entire* program; do it on a **function**.
   ("Under-constrained symbolic execution")
2) Just use it for a **part** of a program. (e.g., part of the OS kernel)
3) Solve the branches **along the <u>conc</u>rete execution path**.
   ("<u>Conc</u>olic execution"; that's the actual term!)

*Learn from the **real behavior** of non-analyzable code.*

1) Request **the actual outcome** to the code. (e.g., S2E)
2) Use the **model** of the code. (e.g., KLEE)

Static Analysis Proposal

ABSTRACT INTERPRETATION : A UNIFIED LATTICE MODEL FOR STATIC ANALYSIS
OF PROGRAMS BY CONSTRUCTION OR APPROXIMATION OF FIXPOINTS

Patrick Cousot* and Radhia Cousot**

Laboratoire d'Informatique, U.S.M.G., BP. 53
38041 Grenoble cedex, France

1. Introduction

A program denotes computations in some universe of objects. Abstract interpretation of programs consists in using that denotation to describe computations in another universe of abstract objects, so that the results of abstract execution give some informations on the actual computations. An intuitive example (which we borrow from Sintzoff [72]) is the rule of signs. The text $-1515 * 17$ may be understood to denote computations on the abstract universe $\{(+), (-), (\pm)\}$ where the semantics of arithmetic operators is defined by the rule of signs. The abstract execution $-1515 * 17 \Longrightarrow -(+) * (+) \Longrightarrow (-) * (+) \Longrightarrow (-)$, proves that $-1515 * 17$ is a negative number. Abstract interpretation is concerned by a particular underlying structure of the usual universe of computations (the sign, in our example). It gives a summary of some facets of the actual executions of a program. In general this summary is simple to obtain but inaccurate (e.g. $-1515 + 17 \Longrightarrow -(+) + (+) \Longrightarrow (-) + (+) \Longrightarrow (\pm)$). Despite its fundamentally incomplete results abstract interpretation allows the programmer or the compiler to answer questions which do not need full knowledge of program executions or which tolerate an imprecise answer, (e.g. partial correctness proofs of programs ignoring the termination problems, type checking, program optimizations which are not carried in the absence of certainty about their feasibility, ...).

2. Summary

Section 3 describes the syntax and mathematical semantics of a simple flowchart language, Scott and Strachey[71]. This mathematical semantics is used in section 4 to built a more abstract model of the semantics of programs, in that it ignores the sequencing of control flow. This model is taken to be the most concrete of the abstract interpretations of programs. Section 5 gives the formal definition of the abstract interpretations of a program. Abstract program properties are modeled by a complete semilattice, Birkhoff[61]. Elementary program constructs are locally interpreted by order preserving functions which are used to associate a system of recursive equations with a program. The program global properties are then defined as one of the extreme fixpoints of that system, Tarski[55]. The abstraction process is defined in section 6. It is shown that the program properties obtained by an abstract interpretation of a program are consistent with those obtained by a more refined interpretation of that program. In particular, an abstract interpretation may be shown to be consistent with the formal semantics of the language. Levels of abstraction are formalized by showing that consistent abstract interpretations form a lattice (section 7). Section 8 gives a constructive definition of abstract properties of programs based on constructive definitions of fixpoints. It shows that various classical algorithms such as Kildall [73], Wegbreit[75] compute program properties as limits of finite Kleene[52]'s sequences. Section 9 introduces finite fixpoint approximation methods to be used when Kleene's sequences are infinite, Cousot[76]. They are shown to be consistent with the abstraction process. Practical examples illustrate the various sections. The conclusion points out that abstract interpretation of programs is a unified approach to apparently unrelated program analysis techniques.

3. Syntax and Semantics of Programs

We will use finite flowcharts as a language independent representation of programs.

3.1 Syntax of a Program

A program is built from a set "Nodes". Each node has successor and predecessor nodes :

$$\underline{\text{n-succ}}, \underline{\text{n-pred}} : \text{Nodes} \to 2^{\text{Nodes}} \mid (n \in \underline{\text{n-succ}}(n)) \Longleftrightarrow (n \in \underline{\text{n-pred}}(n))$$

Hereafter, we note $|S|$ the cardinality of a set S. When $|S| = 1$ so that $S = \{x\}$ we sometimes use S to denote x.

The node subsets "Entries", "Assignments", "Tests", "Junctions" and "Exits" partition the set Nodes.

- An entry node ($n \in$ Entries) has no predecessors and one successor, $((\underline{\text{n-pred}}(n) = \emptyset)$ $\underline{\text{and}}$ $(|\underline{\text{n-succ}}(n)| = 1))$.

---

* Attaché de Recherche au C.N.R.S., Laboratoire Associé n° 7.

** This work was supported by IRIA-SESORI under grants 75-035 and 76-160.

Wait. There's another analytical approach,
*called **Abstract Interpretation,** also from 70's.*

- Similar to Symbolic Execution, but a little relaxed.
- "Examine every ****possible**** states."
- If things get too complex or uncertain (e.g., library calls), it just ***glosses over*** or ***assumes conservatively***.

Rough mechanism sketch

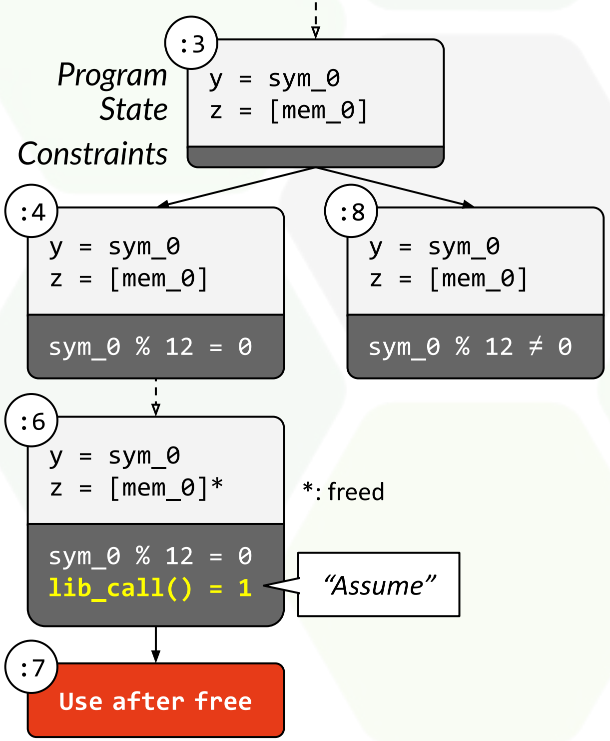
- Track execution paths. (just like Symbolic Execution)
- Approximate or assume states/constraints if needed.
- Try matching vulnerability patterns to execution paths. (e.g., use after free; first free, then use the memory)

Static Analysis Example

“Abstract Interpretation”, to be specific in this example.

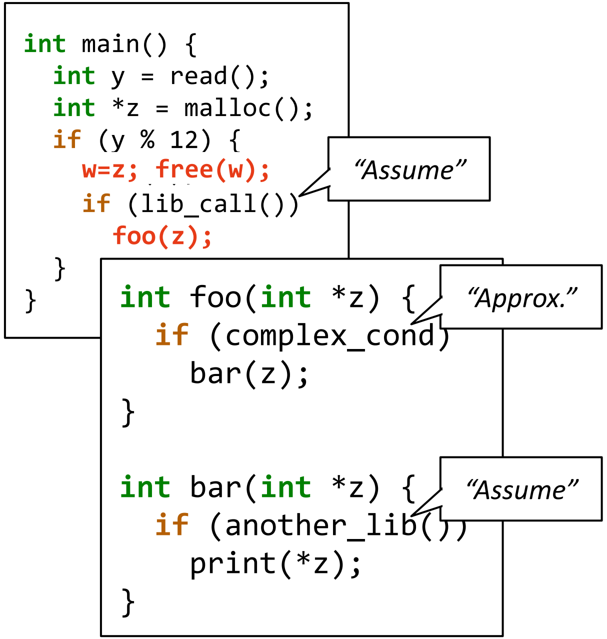
```
int main() {
  int y = read();
  int *z = malloc();
  if (y % 12) {
    free(z);
    if (lib_call())
      print(*z);
  }
}
```

Program



Program State Graph

Static Analysis Ups and Downs



```
int main() {
  int y = read();
  int *z = malloc();
  if (y % 12) {
    w=z; free(w);
    if (lib_call())
      foo(z);
  }
}
```

"Assume"

```
int foo(int *z) {
  if (complex_cond)
    bar(z);
}
```

"Approx."

```
int bar(int *z) {
  if (another_lib())
    print(*z);
}
```

"Assume"

*Very effective for **shallow, straightforward** vulnerabilities.*

- "Shallow": close to the entry point (e.g., `main()`),
- "Straightforward": the info. that should be tracked is clear.

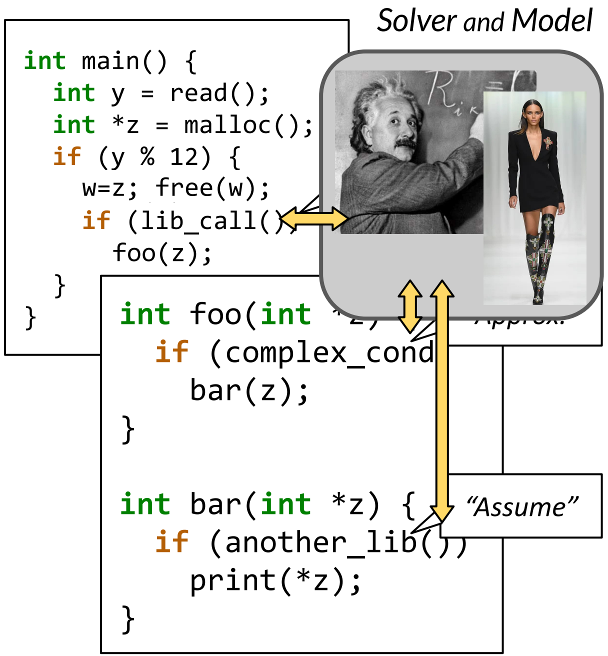
*Problem 1: many **false positives**.*

- Assumptions may be wrong, let alone when it's **accumulated**.
- Easily happen for non-shallow code.

*Problem 2: many **false negatives**.*

- It should keep **relevant information** from approximated out.
  (e.g., memory allocation/free states in use after free)
- But how would you know **which information is relevant?**
  (e.g., pointer transfer in use after free)

Static Analysis Development



```
int main() {
  int y = read();
  int *z = malloc();
  if (y % 12) {
    w=z; free(w);
    if (lib_call())
      foo(z);
  }
}
```

```
int foo(int *z)
  if (complex_cond
    bar(z);
}

int bar(int *z) {
  if (another_lib())
    print(*z);
}
```

Also improved mostly in the early 2010’s and onwards.

*Make it **less** relaxed.*

1) Incorporate constraint solvers (e.g., Z3) from Symbolic Analysis.
2) Use a model for non-analyzable code (e.g., Clang Static Analyzer).

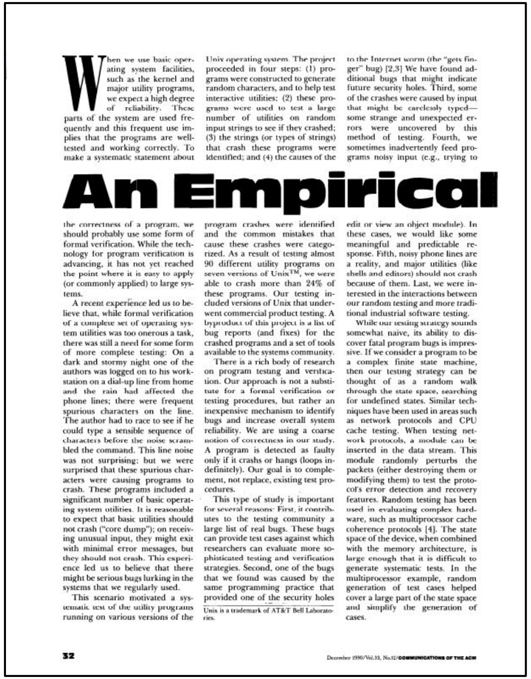
***Add/create/revise patterns** until it’s fair enough.*

1) Make pattern creation as easy as possible. (e.g., CodeQL, Joern)
2) Include many patterns ~~and sell it~~. (e.g., SonarQube, Coverity)

Why not combine it to Symbolic Analysis?

- Rough investigation with Static Analysis,
  and through verification with Symbolic Execution.

Meanwhile, Not Every Approach Was Analytical



An Empirical

When we use basic operating system facilities, such as the kernel and major utility programs, we expect a high degree of reliability. These parts of the system are used frequently and this frequent use implies that the programs are well-tested and working correctly. To make a systematic statement about the correctness of a program, we should probably use some form of formal verification. While the technology for program verification is advancing, it has not yet reached the point where it is easy to apply (or commonly applied) to large systems.

A recent experience led us to believe that, while formal verification of a complete set of operating system utilities was too onerous a task, there was still a need for some form of more complete testing: On a dark and stormy night one of the authors was logged on to his workstation on a dial-up line from home and the rain had affected the phone lines; there were frequent spurious characters on the line. The author had to race to see if he could type a sensible sequence of characters before the noise scrambled the command. This line noise was not surprising; but we were surprised that these spurious characters were causing programs to crash. These programs included a significant number of basic operating system utilities. It is reasonable to expect that basic utilities should not crash ("core dump"); on receiving unusual input, they might exit with minimal error messages, but they should not crash. This experience led us to believe that there might be serious bugs lurking in the systems that we regularly used.

This scenario motivated a systematic test of the utility programs running on various versions of the Unix operating system. The project proceeded in four steps: (1) programs were constructed to generate random characters, and to help test interactive utilities; (2) these programs were used to test a large number of utilities on random input strings to see if they crashed; (3) the strings (or types of strings) that crash these programs were identified; and (4) the causes of the program crashes were identified and the common mistakes that cause these crashes were categorized. As a result of testing almost 90 different utility programs on seven versions of Unix$^{\text{TM}}$, we were able to crash more than 24% of these programs. Our testing included versions of Unix that underwent commercial product testing. A byproduct of this project is a list of bug reports (and fixes) for the crashed programs and a set of tools available to the systems community.

There is a rich body of research on program testing and verification. Our approach is not a substitute for a formal verification or testing procedures, but rather an inexpensive mechanism to identify bugs and increase overall system reliability. We are using a coarse notion of correctness in our study. A program is detected as faulty only if it crashs or hangs (loops indefinitely). Our goal is to complement, not replace, existing test procedures.

This type of study is important for several reasons: First, it contributes to the testing community a large list of real bugs. These bugs can provide test cases against which researchers can evaluate more sophisticated testing and verification strategies. Second, one of the bugs that we found was caused by the same programming practice that provided one of the security holes to the Internet worm (the "gets finger" bug) [2,3] We have found additional bugs that might indicate future security holes. Third, some of the crashes were caused by input that might be carelessly typed—some strange and unexpected errors were uncovered by this method of testing. Fourth, we sometimes inadvertently feed programs noisy input (e.g., trying to edit or view an object module). In these cases, we would like some meaningful and predictable response. Fifth, noisy phone lines are a reality, and major utilities (like shells and editors) should not crash because of them. Last, we were interested in the interactions between our random testing and more traditional industrial software testing.

While our testing strategy sounds somewhat naive, its ability to discover fatal program bugs is impressive. If we consider a program to be a complex finite state machine, then our testing strategy can be thought of as a random walk through the state space, searching for undefined states. Similar techniques have been used in areas such as network protocols and CPU cache testing. When testing network protocols, a module can be inserted in the data stream. This module randomly perturbs the packets (either destroying them or modifying them) to test the protocol's error detection and recovery features. Random testing has been used in evaluating complex hardware, such as multiprocessor cache coherence protocols [4]. The state space of the device, when combined with the memory architecture, is large enough that it is difficult to generate systematic tests. In the multiprocessor example, random generation of test cases helped cover a large part of the state space and simplify the generation of cases.

Unix is a trademark of AT&T Bell Laboratories.

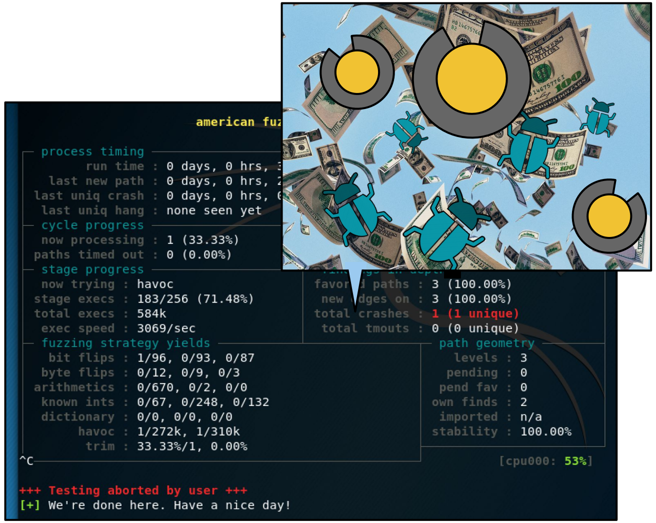
In 1990, an ***empirical*** *approach revealed many bugs in UNIX utilities. (e.g., tee and nm)*

- *Literally* empirical; “put random bytes to programs.”
- Found many undiscovered bugs by then.
- Deemed as a precursor of modern-day **fuzzing**.

Results were promising, but it had obvious ***drawbacks****.*

- Random inputs cannot explore, or even *reach* a deeper part of a program.
- Pushed back to a backseat ever since, used by researchers and hackers behind the scenes.

But Then, There Was A Breakthrough



```
                     american fuz

 ─ process timing ─────────────
        run time : 0 days, 0 hrs, 3
   last new path : 0 days, 0 hrs, 2
 last uniq crash : 0 days, 0 hrs, 0
  last uniq hang : none seen yet
 ─ cycle progress ─────────────
  now processing : 1 (33.33%)
 paths timed out : 0 (0.00%)
 ─ stage progress ─────────────
  now trying : havoc                 favor  d paths : 3 (100.00%)
 stage execs : 183/256 (71.48%)       new  dges on : 3 (100.00%)
 total execs : 584k                  total crashes : 1 (1 unique)
  exec speed : 3069/sec               total tmouts : 0 (0 unique)
 ─ fuzzing strategy yields ─────────────────────────── path geometry ─
   bit flips : 1/96, 0/93, 0/87                        levels : 3
  byte flips : 0/12, 0/9, 0/3                         pending : 0
 arithmetics : 0/670, 0/2, 0/0                       pend fav : 0
  known ints : 0/67, 0/248, 0/132                   own finds : 2
  dictionary : 0/0, 0/0, 0/0                         imported : n/a
       havoc : 1/272k, 1/310k                       stability : 100.00%
        trim : 33.33%/1, 0.00%
^C────────────────────────────────────────────────
                                                     [cpu000: 53%]

+++ Testing aborted by user +++
[+] We're done here. Have a nice day!
```

*In 2013, the arrival of **AFL** revolutionized **fuzzing**.*

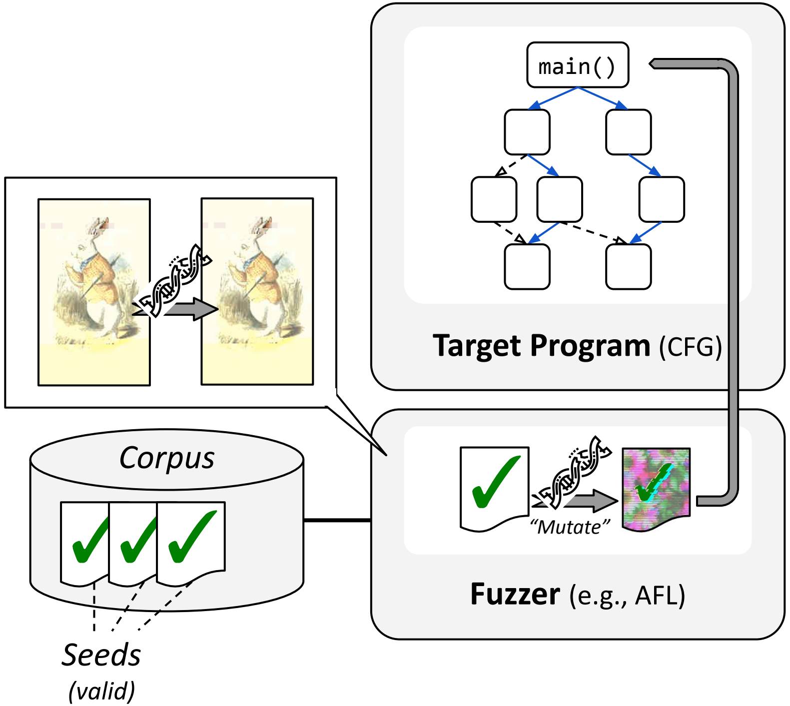
- Random nature didn't change, but it did it smart.
- **Mutation**: slightly modify valid inputs to create new ones.
- **Feedback**: make the target program report whether the last input was "interesting".

* Caveat: most probably they didn't do them the first time.

*The result was **remarkable**; tons of new vulnerabilities across all sort of programs.*

- Check out the official site (https://lcamtuf.coredump.cx/afl) for the list of bugs found by AFL. (it's quite a lot!)

Fuzzing How It Works



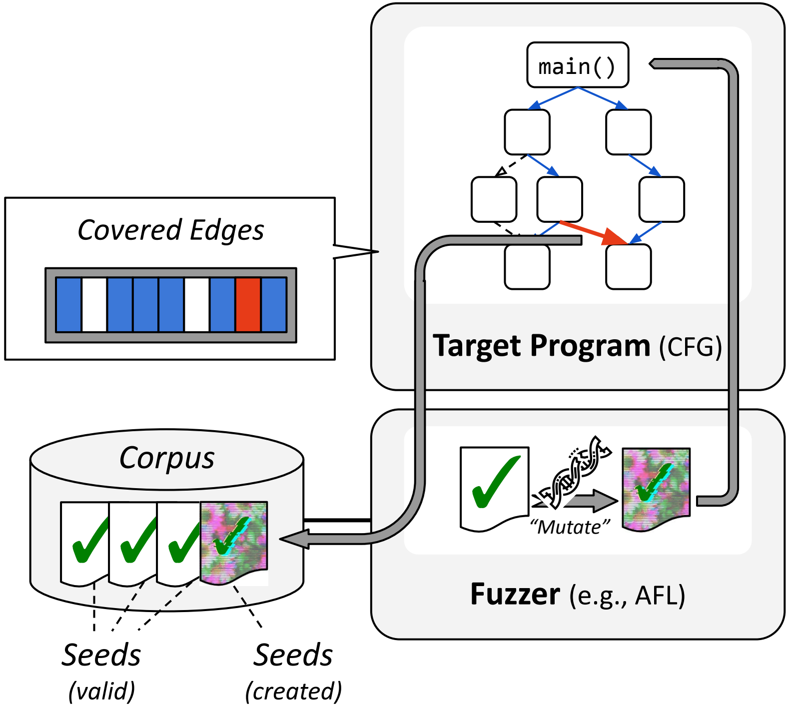
Basic Terminology (roughly)

- **Seed**: "interesting" inputs.
- **Corpus**: seed database.

***Two key weapons** in the arsenal.*

1) **Mutation**
   - Take one seed from the corpus.
   - Change some **part** of it randomly. (e.g., bit flip)

Fuzzing How It Works



Basic Terminology (roughly)

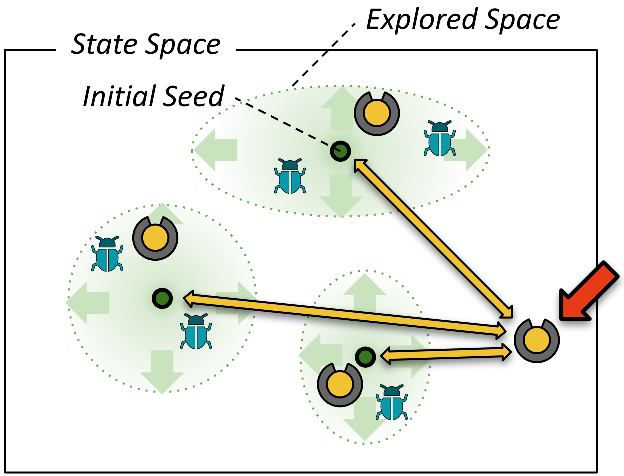
- **Seed**: "interesting" inputs.
- **Corpus**: seed database.

***Two key weapons** in the arsenal.*

1) **Mutation**
   - Take one seed from the corpus.
   - Change some **part** of it randomly. (e.g., bit flip)

2) **Feedback**
   - Check if the mutated input exhibited any **interesting** behavior. (e.g., triggering new edge)
   - If it is, add the mutated input to the corpus.

Fuzzing Ups and Downs



*Cons 1: **cannot** say “there’s **no bugs anymore**”*

(or academically put, “no guarantee on completeness”)

- There might be vulnerabilities that **we** couldn’t find, but **they** (e.g., attackers) may find.

*Cons 2: highly dependent on the **initial** seeds.*

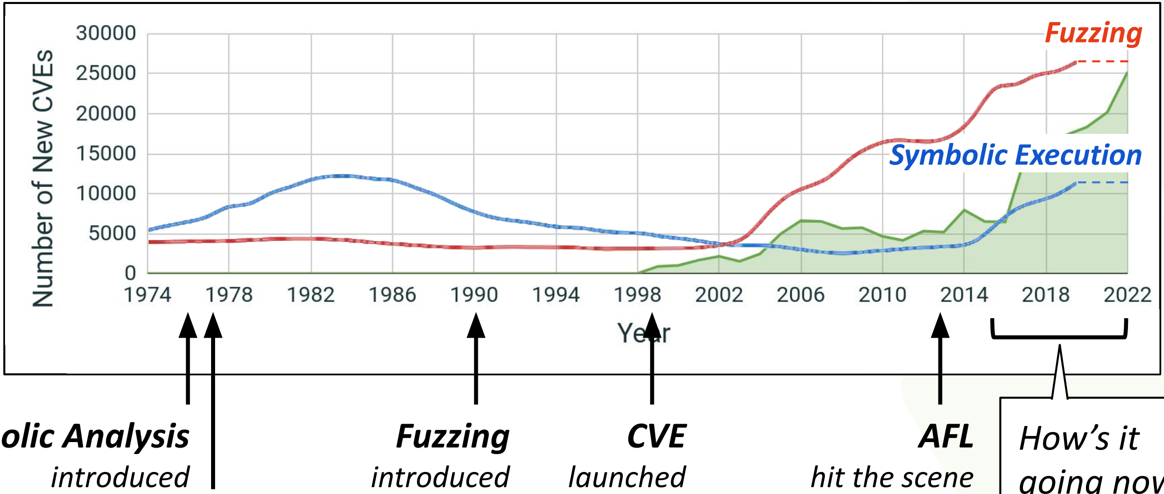
- From the perspective of the state space, mutation **can’t go too far** from the initial seeds.
- Why? Because mutation only **breaks** inputs.
- Bad initial seeds ⇒ bad fuzzing.

*But in practice, it was a **huge success**.*

- If the vulnerability is too obscure, anybody wouldn’t easily find it either (incl. attackers).

Let’s Take A Look at A Timeline… Again

Google N-gram Search
(American English, ~2019)



Symbolic Analysis
introduced

Abstract Interpretation
(Static Analysis) introduced

Fuzzing
introduced

CVE
launched

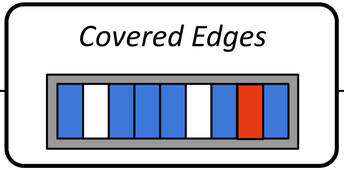
AFL
hit the scene

How’s it going now?

Development in **Fuzzing**

After The Initial Breakthrough

*Research on a fundamental level; "can we improve fuzzing **itself**?"*



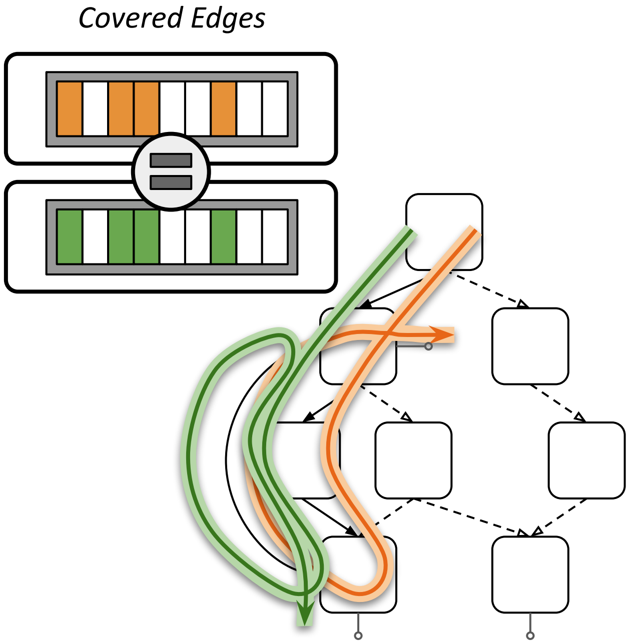
Feedback

Simply checking if CFG edges are covered (i.e., edge coverage) glosses over exec. too much.

Mutation

Randomly flipping bits and bytes can overly break the sanity of seeds.

Topic: Searching for Better Feedback



*Only checking CFG edges (e.g., "edge coverage") may **miss too much execution details.***

- The same edge can be entered differently.

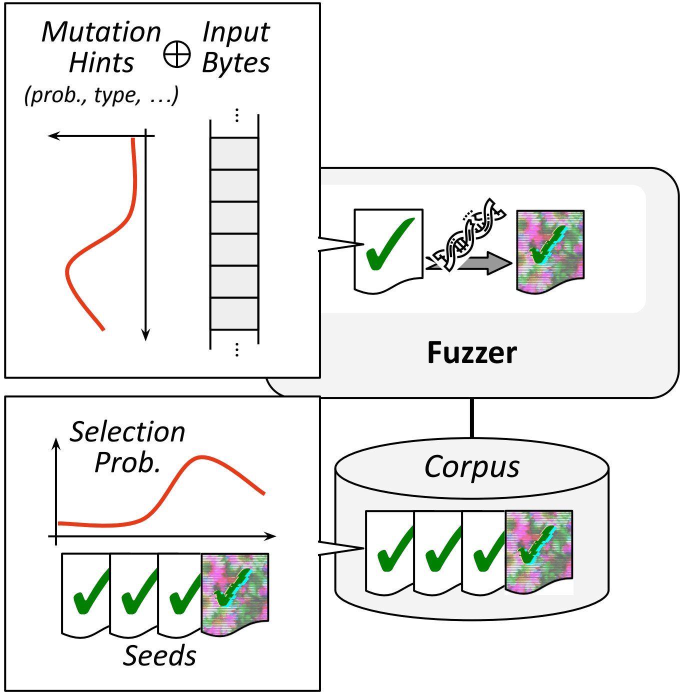
Some alternative proposals.

- Counting **how many times** a given edge is taken.
- Distinguish the **context** when it enters an edge. (e.g., previous N edges, call stack, …)
- Enhance with **data-flow** hints.

Issues and Status-quo

- Not super effective for an added complexity.
- Some side-effects. (e.g., too many "interesting" seeds)
- Currently, just plain edge coverage is dominant.

Topic: Improving Mutation and Seed Selection



Basic mutation and seed selection (=”what to mutate?”)

- **Randomly** changing bits and bytes.
- Also **randomly** choosing seeds.

Making ***mutation*** *smarter.*

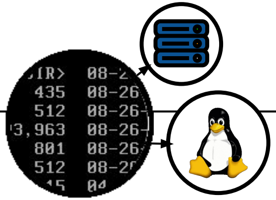
- Mutate the bytes affecting blocked branches.
- Mutate the bytes yielding better feedback.
- Identify the type of bytes and mutate accordingly.

Making ***seed selection*** *smarter.*

- Use gradient-descent or DL to prioritize seeds closer to the solutions of blocked branches.
- Use statistics to select generally high-yielding seeds.

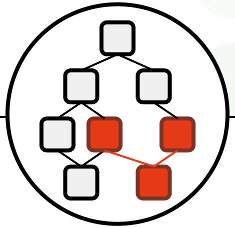
Entering Mature Stage

Going beyond the conventional fuzzing.



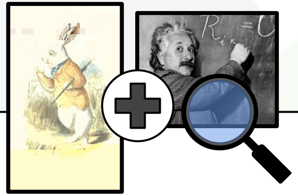
Extending Applicability

Can we fuzz other than the standard *byte-input, open-source* programs?



Specializing Purposes

Do we have to stick to discovering *vulnerabilities* in *every* part of the program?

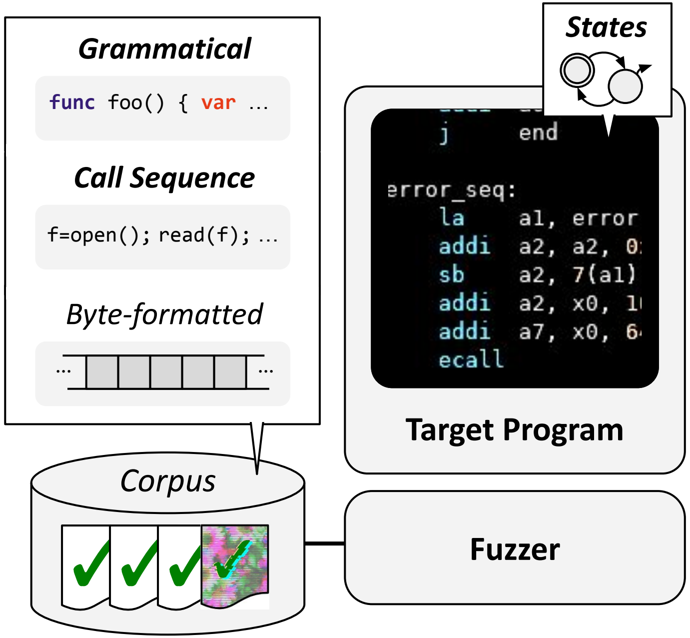


Hybrid Approaches

Do we have to rely on *pure randomness* in every stage of fuzzing?

* Not a definitive list.

Topic: Extending Applicability



```
    j      end

error_seq:
    la     a1, error
    addi   a2, a2, 0
    sb     a2, 7(a1)
    addi   a2, x0, 1
    addi   a7, x0, 6
    ecall
```

Conventional fuzzing works well with programs that;

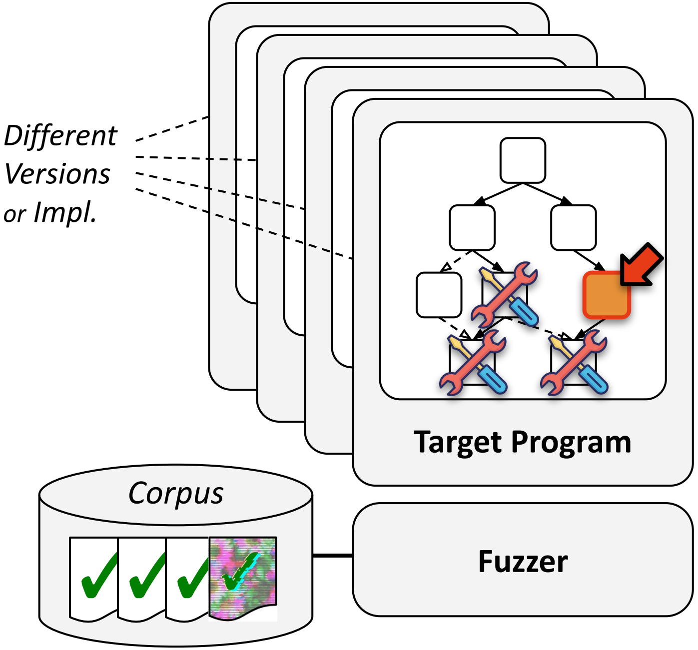
- Accept byte-formatted inputs.
- Have no inter-execution states.
- Are open-sourced.

*But there are *__many__* programs that;*

- Accept call sequences as inputs. (e.g., OS kernels, libraries)
- Have a strict grammar. (e.g., JS interpreters, hypervisors, …)
- Have inter-execution states. (e.g., network, bluetooth, …)
- Are closed-sourced. (e.g., firmware, …)

*They **all** have their own line of research.*

Topic: Specializing Purposes



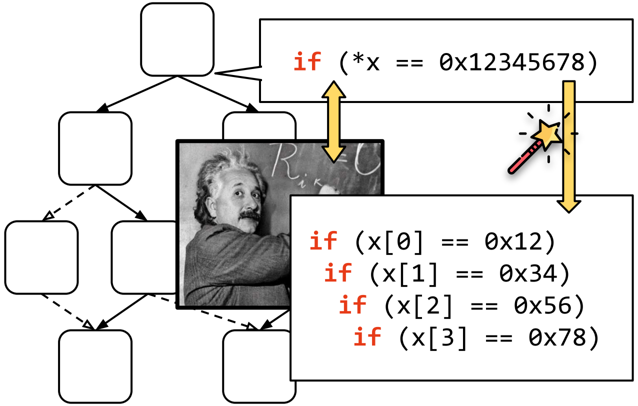
Conventional fuzzing aims at;

- Testing the **entire** program.
- Detecting **easy-to-detect** vulnerabilities.
  (e.g., memory errors)

Specializing purposes can improve efficiency.

- Targeting **a specific code location**
  ("Directed fuzzing")
- Targeting **patched code locations**.
  ("Regression fuzzing")
- Detecting the **semantic difference** between different versions or implementations.
  ("Differential fuzzing")

Topic: Hybrid Approaches



```
if (*x == 0x12345678)
```

```
if (x[0] == 0x12)
 if (x[1] == 0x34)
  if (x[2] == 0x56)
   if (x[3] == 0x78)
```

*Fuzzing is fundamentally **empirical** (i.e., trial-and-error), so it can easily stuck at difficult branches.*

- Example: "`if (*x == 0x12345678)`".
- Which one would be faster?
  - Guessing random numbers between 0x00000000 to 0xffffffff.
  - Solving the equation.

*Why not **combining** it to **analytical** approaches?*

- Resort to **Symbolic Execution** when a difficult branch needs to be solved.
- Resort to **Static Analysis** to make such a branch easy-to-solve by fuzzing.

Some **Future TODOs** for Fuzzing

1) *Detecting Semantic Vulnerabilities*
  - Fuzzing relies on **detection mechanism**.
  - Detecting **semantic vulnerabilities** is never easy. (remember the specification example?)
  - Some research has been done (e.g., file system), but never been generally solved yet.

2) *Providing Completeness Guarantee*
  - Fuzzing is an **empirical** process.
  - Implication; it cannot **guarantee** that there's no remaining vulnerability.
  - Very critical shortcoming for **mission-critical software**.
    (e.g., firmware on medical devices and aerospace vehicles)
  - Can we give some completeness guarantee in one way or another?

Conclusion

Software vulnerabilities *can do harm to software/systems/users.*

- ***Detecting vulnerabilities*** *is one way to counter that threat.*

Analytic approaches *were dominant at the early stage,*

- *but* ***fuzzing*** *eventually took over the mainstream.*

Research first attempted to improve fuzzing ***on a fundamental level****.*

- *Later research was diversified to such as about applicability and specialization.*

There are still some future tasks to solve.

Check out recent fuzzing papers at https://github.com/wcventure/FuzzingPaper.

*(caveat: ****not**** ***my repo****, but it's pretty extensive)*

Thanks for Listening

Special thanks to Chibin Zhang (chibin.zhang@epfl.ch)

Speaker: Gwangmu Lee (HexHive @ EPFL)

Slides available at https://gwangmu.github.io.

References

[pg. 11] “Vulnerabilities by Date”, https://www.cvedetails.com/browse-by-date.php, accessed on May 28, 2023.

[pg. 13] “Symbolic execution and program testing”, CACM, 1976.

[pg. 14] “Symbolic Execution”, https://en.wikipedia.org/wiki/Symbolic_execution, accessed on May 28, 2023.

[pg. 16] “Under-Constrained Symbolic Execution: Correctness Checking for Real Code”, Ramos et al., Usenix Security, 2015.
“HFL: Hybrid Fuzzing on the Linux Kernel”, Kim et al., NDSS, 2020.
“DART: Directed Automated Random Testing”, Godefroid et al., PLDI, 2015.
“The S2E Platform: Design, Implementation, and Applications”, Chipounov et al., ACM Trans. Comput., 2012.
"KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs", Cedar et al., OSDI, 2008.
“Sok: (State of) the art of war: Offensive techniques in binary analysis”, Shoshitaishvili et al., Oakland, 2016.

[pg. 17] “Abstract interpretation: a unified lattice model for static analysis…”, Cousot et al., POPL, 1977.

[pg. 20] “Z3: the theorem prover”, https://github.com/Z3Prover/z3, accessed on May 28, 2023.
“Clang Static Analyzer”, https://clang-analyzer.llvm.org/, accessed on May 28, 2023.
“Modeling and Discovering Vulnerabilities with Code Property Graphs”, Yamaguchi et al., Oakland, 2013.
“CodeQL”, https://codeql.github.com/, accessed on May 28, 2023.

References

[pg. 20] “Code Quality Tool and Secure Analysis with SonarQube”, https://www.sonarsource.com/products/sonarqube/, accessed on May 28, 2023.
“Coverity Scan - Static Analysis”, https://scan.coverity.com/, accessed on May 28, 2023.
“Sys: A Static/Symbolic Tool for Finding Good Bugs in Good (Browser) Code”, Brown et al., Usenix Security, 2020.

[pg. 21] “An empirical study of the reliability of UNIX utilities”, Miller et al., CACM, 1990.

[pg. 22] “american fuzzy lop”, https://lcamtuf.coredump.cx/afl/, accessed on May 28, 2023.

[pg. 29] “CollAFL: Path Sensitive Fuzzing”, Gan et al., Oakland, 2018.
“Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing”, Wang et al., RAID, 2019.
“GREYONE: Data Flow Sensitive Fuzzing”, Gan et al., Usenix Security, 2020.
“DatAFLow: Toward a Data-Flow-Guided Fuzzer”, Herrera et al., ACM TOSEM, 2023.

[pg. 30] “NEUZZ: Efficient Fuzzing with Neural Program Smoothing”, She et al., Oakland, 2019.
“Angora: Efficient Fuzzing by Principled Search”, Chen et al., Oakland, 2018.
“ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery”, Oakland, 2019.
“EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit”, Yue et al., Usenix Security, 2020.
“MOPT: Optimize Mutation Scheduling for Fuzzers”, Lyu et al., Usenix Security, 2019.

References

[pg. 32] “Syzkaller”, https://github.com/google/syzkaller, accessed on June 2, 2023.
“FuzzGen: Automatic Fuzzer Generation”, Ispoglou et al., USENIX Security, 2020.
“FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities”, Groß et al., NDSS, 2023.
“MundoFuzz: Hypervisor Fuzzing with Statistical Coverage Testing and Grammar Inference”, Myung et al., USENIX Security, 2022.
“Nyx-net: network fuzzing with incremental snapshots”, Schumilo et al., EuroSys, 2022.
“Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets”, Ruge et al., USENIX Security, 2020.
“TEEzz: Fuzzing Trusted Applications on COTS Android Devices”, Busch et al., Oakland, 2023.
“HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation”, Clements et al., USENIX Security, 2020.

[pg. 33] “Directed Greybox Fuzzing”, Böhme et al., CCS, 2017.
“Constraint-guided Directed Greybox Fuzzing”, Lee et al., USENIX Security, 2021.
“BEACON: Directed Grey-Box Fuzzing with Provable Path Pruning”, Huang et al., Oakland, 2022.
“Regression Greybox Fuzzing”, Zhu et al., CCS, 2021.
“NEZHA: Efficient Domain-Independent Differential Testing”, Petsios et al., Oakland, 2017.
“HyDiff: Hybrid Differential Software Analysis”, Noller et al., ICSE, 2020.

[pg. 34] “QSym : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing”, Yun et al., USENIX Security 2018.
“T-Fuzz: Fuzzing by Program Transformation”, Peng et al., Oakland, 2018.