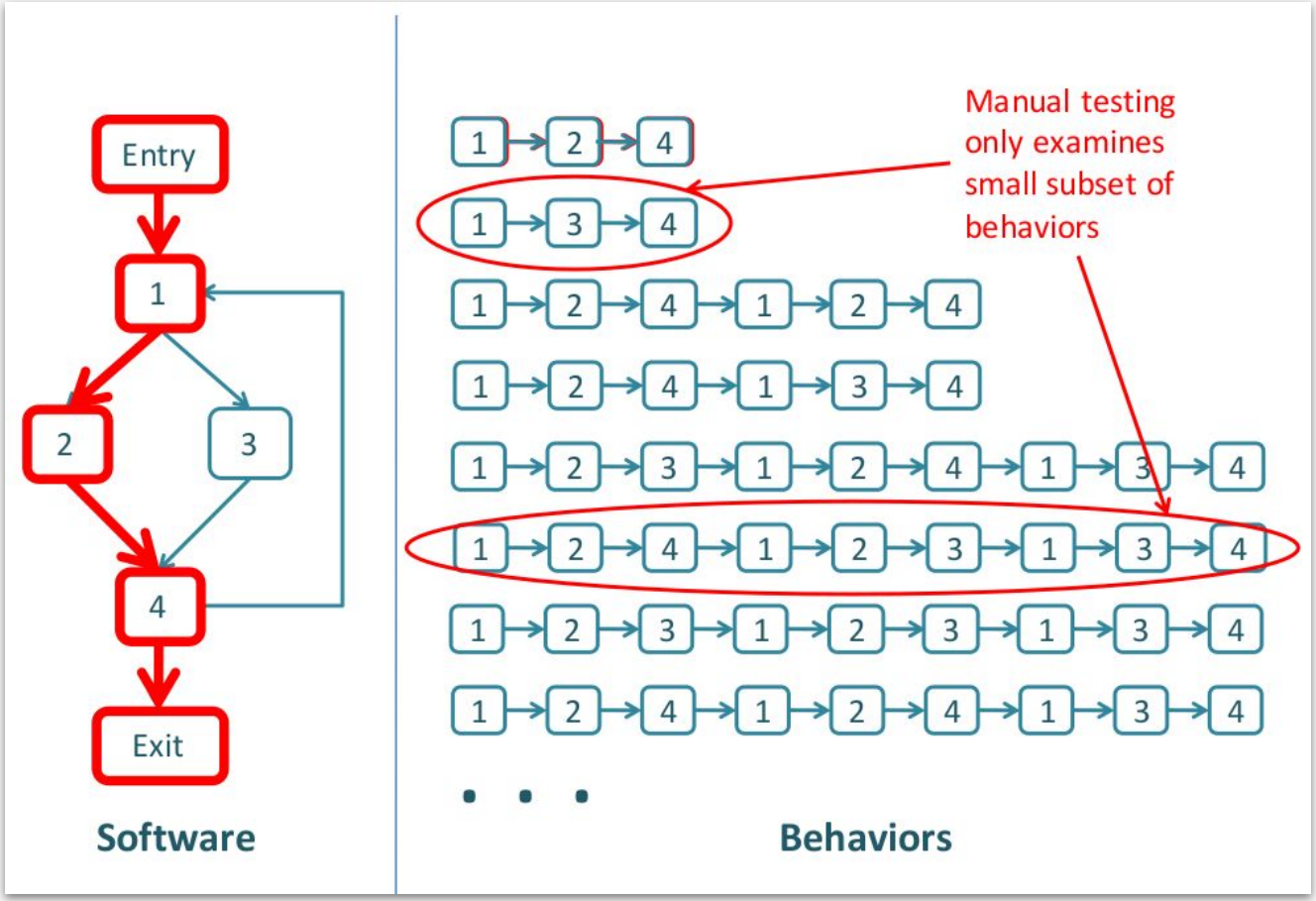
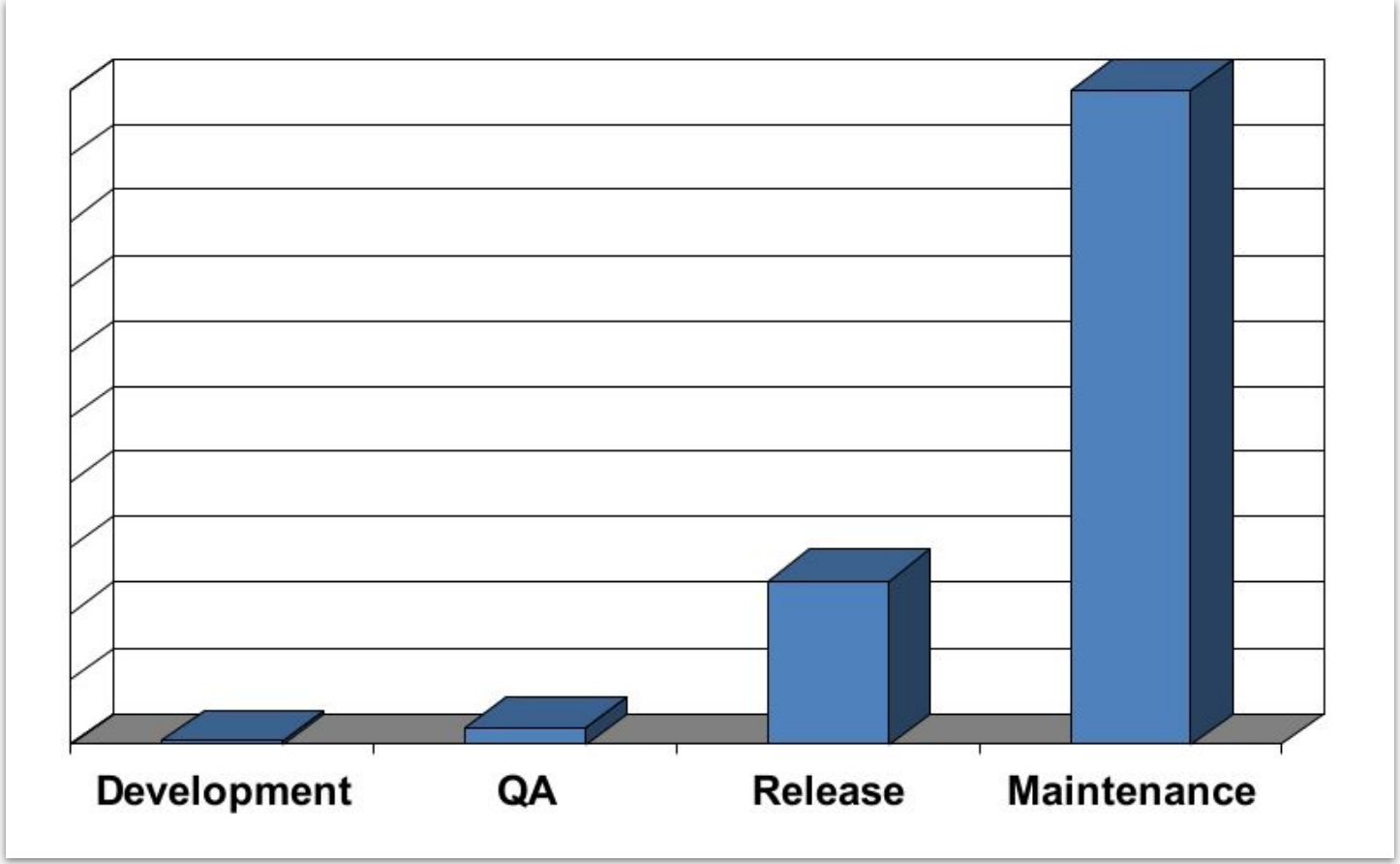
Program Analysis

Mustakimur R. Khandaker

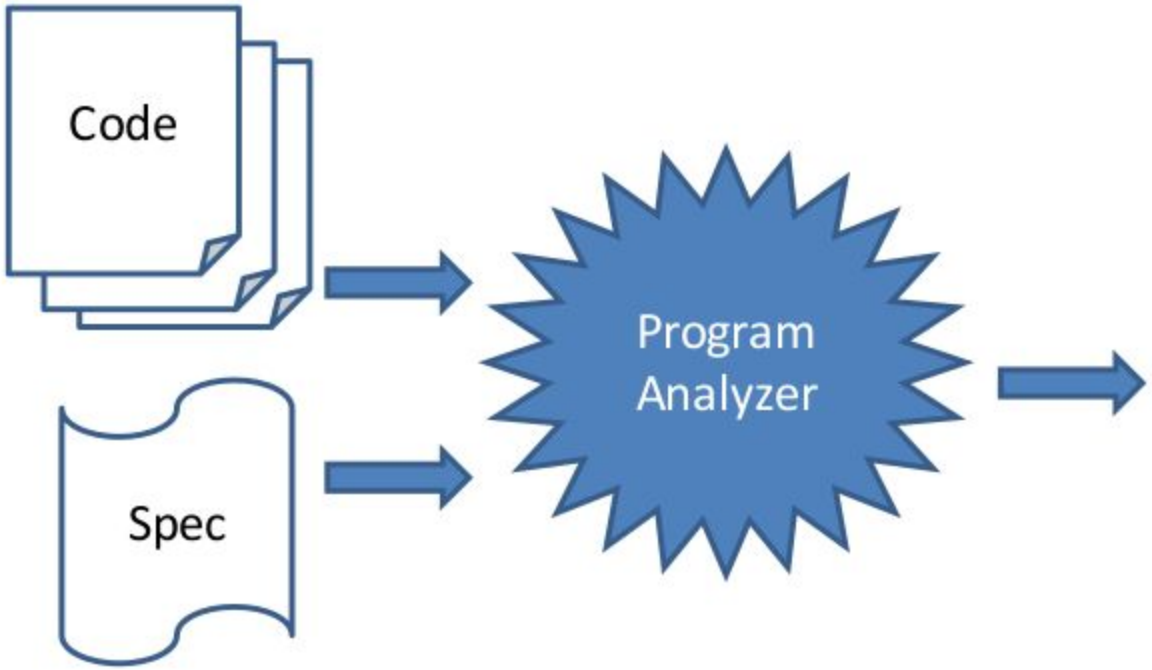
Manual Testing



Cost of Fixing a Defect



Program Analyzers



Report	Type	Line
1	mem leak	324
2	buffer oflow	4,353,245
3	sql injection	23,212
4	stack oflow	86,923
5	dang ptr	8,491
...	...	...
10,502	info leak	10,921

Options

Static Analysis

- Inspect code or run automated method to find errors or gain confidence about their absence.

Dynamic Analysis

- Run code with sample test input, possibly under instrumented conditions, to see if there are likely problems.

Concolic Analysis

- hybrid program verification technique that performs symbolic execution, a classical technique that treats program variables as symbolic variables, along a concrete execution (similar to dynamic analysis) path.

Static Code Analysis

Static Analysis

Analyzing code before executing it.

- Analogy: Spell checker.
- e.g. FindBugs, Fortify, Coverity, MS tools, etc.

Suited to problem identification because:

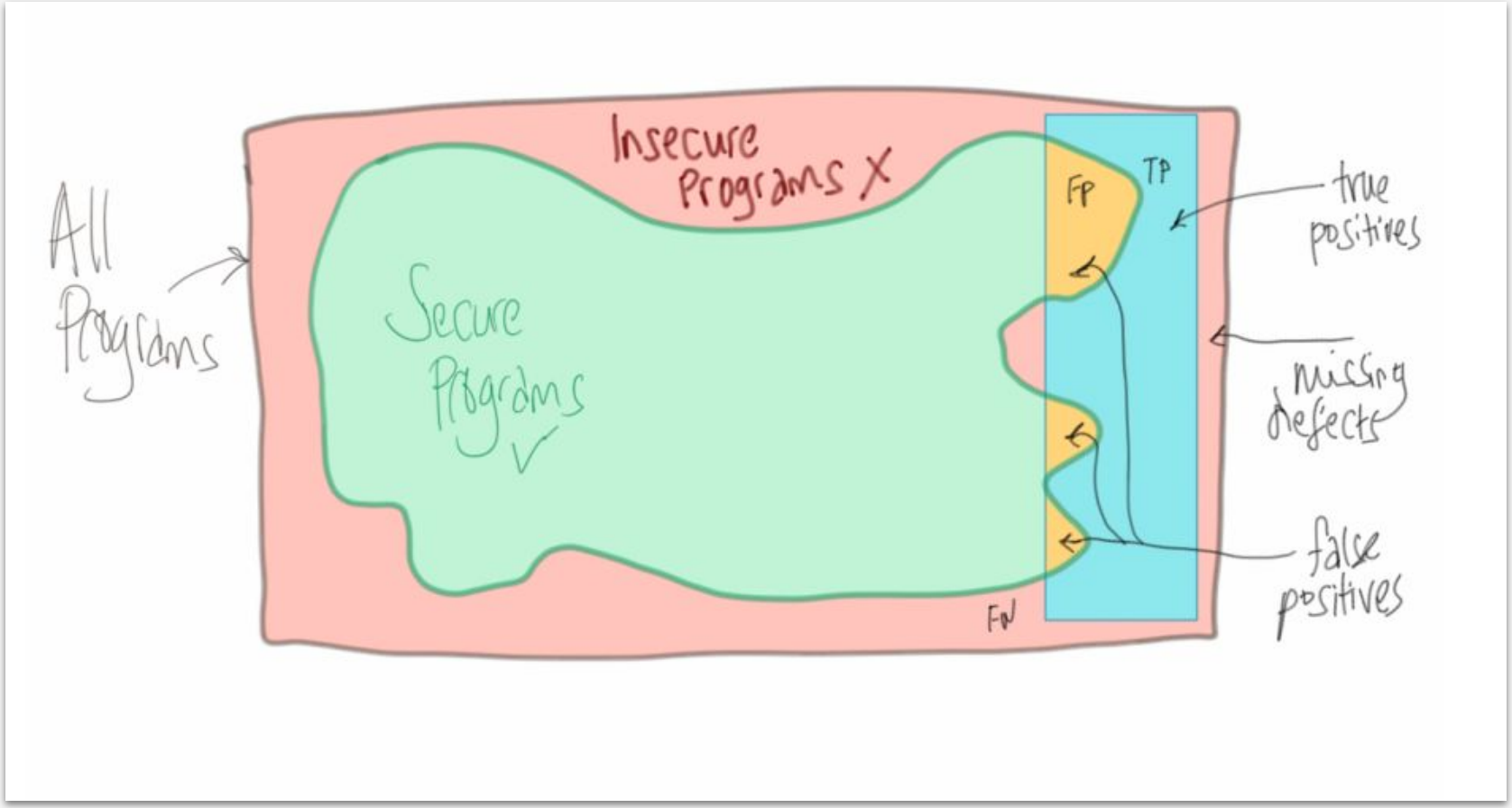
- Checks thoroughly and consistently.
- Can point to the root cause of the problem.
  - e.g., presence of buffer overflow.
- Help find errors/bugs early in the development.
  - reduce cost.
- New information can be easily incorporated to recheck a given program.

Key Issues

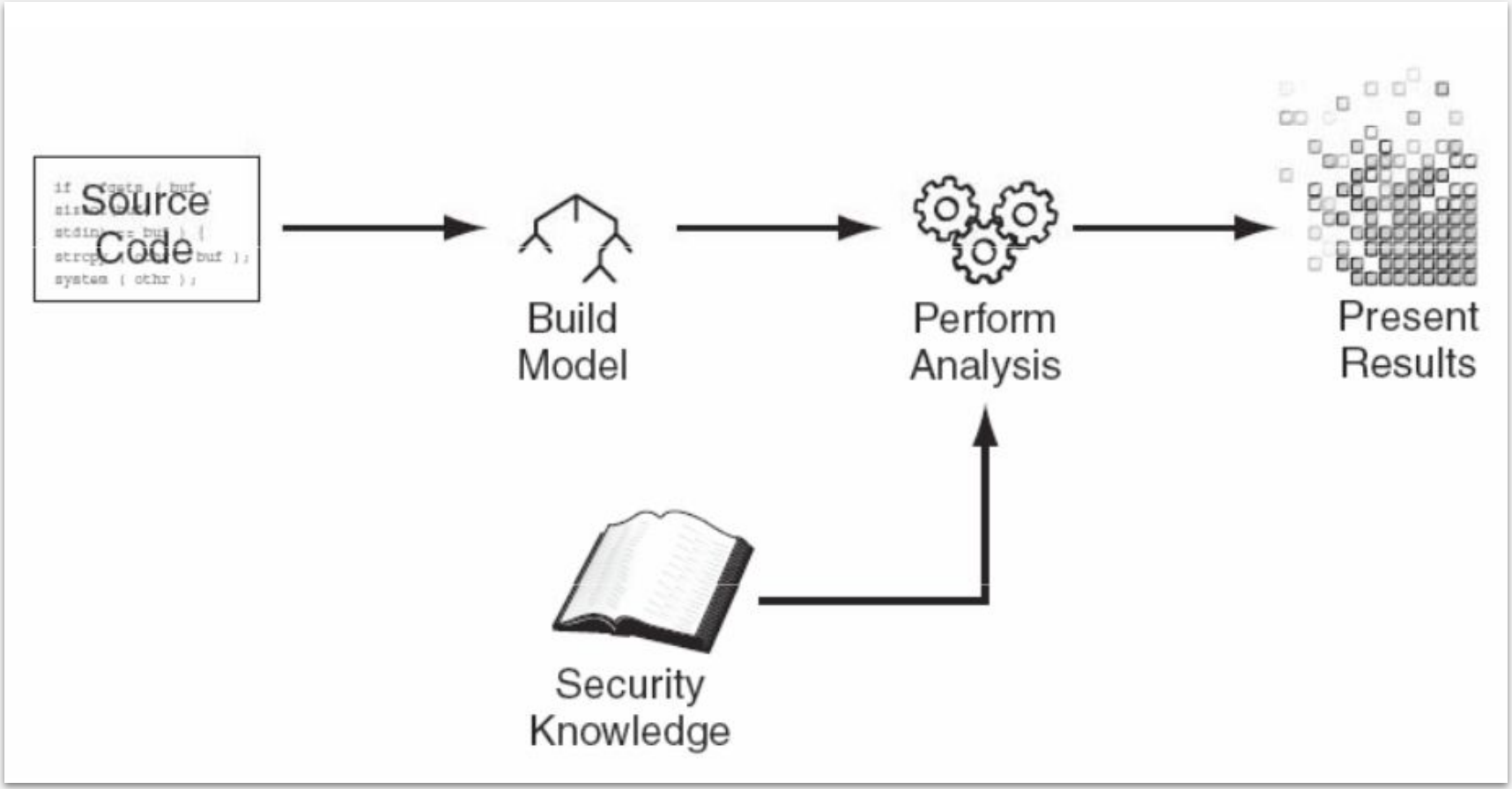
- Can give a lot of noise!
  - Path exploration issues (Completeness).
- False Positives & False Negative.
  - Which is worse? Need to balance (Soundness) the FP and FN.
- Defects must be visible to the tool.

Property	Definition
Soundness	"Sound for reporting correctness" Analysis says no bugs 🎭 No bugs or equivalently There is a bug 🎭 Analysis finds a bug
Completeness	"Complete for reporting correctness" No bugs 🎭 Analysis says no bugs

Continue ...


All Programs
Insecure Programs X
Secure Programs ✓
FP
TP
FN
true positives
missing defects
false positives

Under the Hood



Pointer Analysis

Two variables are aliases if:
- they reference the same memory location.

More useful
- prove variables reference different locations.

```
int x,y;
int *p = &x;
int *q = &y;
int *r = p;
int **s = &q;
```

Issues:
- Decide for every pair of pointers at every program point:
  - do they point to the same memory location?
- Correctness:
  - report all pairs of pointers which do/may alias.
- Ambiguous:
  - two pointers which may or may not alias.

Alias sets:

{x, *p, *r}

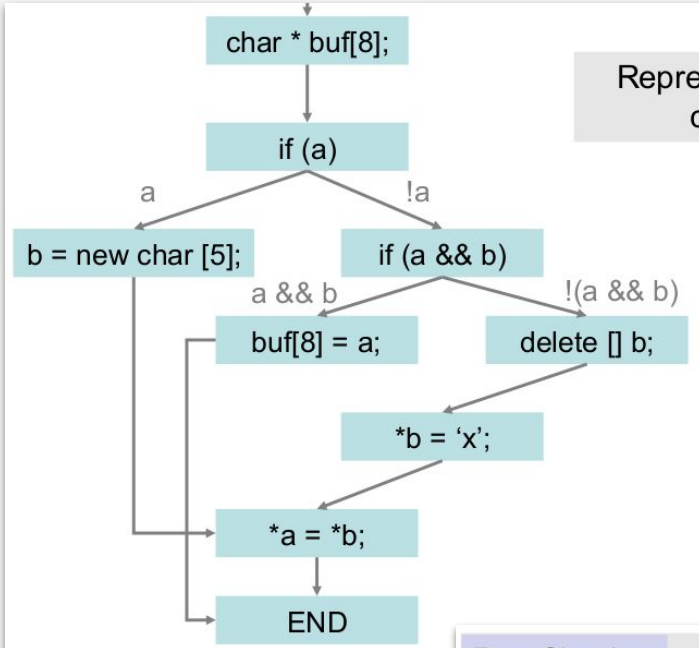
{y, *q, **s}

{q, *s}

p and q point to different locs

Finding Local Bugs

```
#define SIZE 8
void set_a_b(char * a, char * b) {
  char * buf[SIZE];
  if (a) {
    b = new char[5];
  } else {
    if (a && b) {
      buf[SIZE] = a;
      return;
    } else {
      delete [] b;
    }
    *b = 'x';
  }
  *a = *b;
}
```



Represent logical structure of code in graph form

Checker

- Defined by a state diagram, with state transitions and error states

Run Checker

- Assign initial state to each program var
- State at program point depends on state at previous point, program actions
- Emit error if error state reached

Sanitizer

Sanitizers

Add instrumentation to detect unsafe behaviour!

We will look at 3 tools:

- ❏ ASan (Address Sanitizer).
- ❏ MSan (Memory Sanitizer).
- ❏ UBSan (Undefined Behaviour Sanitizer).

Address Sanitizer

One of the leading causes of errors in C is memory corruption:

- Out-of-bounds array accesses.
- Use pointer after call to free().
- Use stack variable after it is out of scope.
- Double-frees or other invalid frees.
- Memory leaks.

AddressSanitizer instruments code to detect these errors.

- Need to recompile.
- Adds runtime overhead.
  - Use it while developing.

Built into gcc and clang!
Compile with -fsanitize=address.

Instrumentation

Original code:

```
*addr = 42;
```

Instrumented pseudocode:

```
if (!is_ok_to_use(addr))
    print_report_and_crash();
// memory is ok to use:
*addr = 42;
```

A state of every aligned 8 bytes of memory is stored in a single shadow byte.
Simple shadow address calculation:

```
Shadow_addr = addr / 8 + offset
```

Allows very simple instrumentation, performed at LLVM IR level.

Example: stack-buffer-overflow

```
ERROR: AddressSanitizer stack-buffer-overflow
 on address 0x7f5620d981b4
 at pc 0x4024e8 bp 0x7fff101cbc90 sp 0x7fff101cbc88
READ of size 4 at 0x7f5620d981b4 thread T0
 #0 0x4024e8 in main example_StackOutOfBounds.cc:4
 #1 0x7f5621db6c4d in __libc_start_main ??:0
 #2 0x402349 in _start ??:0
Address 0x7f5620d981b4 is located at offset 436 in frame <main>
of T0's stack:
 This frame has 1 object(s):
 [32, 432) 'stack_array'
```

Memory Sanitizer

Both local variable declarations and dynamic memory allocation via malloc() do not initialize memory:

```c
#include <stdio.h>

int main(void) {
  int x[10];
  printf("%d\n", x[0]); // uninitialized
  return 0;
}
```

- Accesses to uninitialized variables are undefined.
- ASan does not catch uninitialized memory accesses.
- Memory sanitizer (MSan) does check for uninitialized memory accesses.

Compile with `-fsanitize=memory`.

Example

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

int main(int argc, char **argv) {
 int a[10];
 a[2] = 0;

 if (a[argc]) {
   printf("print something\n");
 }

 return 0;
}
```

1. Stack allocate array on line 5.
2. Partially initialize it on line 6.
3. Access it on line 8.
4. This might or might not be initialized.

Undefined Behaviour Sanitizer

There is lots of non-memory-related undefined behaviour in C:

- Signed integer overflow.
- Dereferencing null pointers.
- Pointer arithmetic overflow.
- Dynamic arrays whose size is non-positive.

Undefined Behaviour Sanitizer (UBSan) instruments code to detect these errors.

Adds runtime overhead.

- Typical overhead of 20%.

Built into gcc and clang!
Compile with `-fsanitize=undefined`

Example

foo.cpp

```cpp
#include <iostream>
int main() {
    int a; int b;
    std::cin >> a >> b;
    int c = a * b;
    std::cout << c << std::endl;
    return 0;
}
```

```
$ g++ -std=c++17 -g -fsanitize=undefined foo.cpp -o foo
$ ./foo
123456
789123
foo.cpp:7:9: runtime error: signed integer overflow: 123456 * 789123
↪  cannot be represented in type 'int'
-1362278720
```

Fuzzing

Fuzzing

Fuzzing is an automated software testing technique

- Generate invalid, unexpected, or random inputs to the program.
  Inputs are often file based or network based.
- The program is monitored for errors, Crashes, assertions, sanitizers…

Dumb fuzzing:

- Blindly mutate existing valid inputs.

Smart fuzzing:

- **Generation based:** generate inputs according to protocol specification.
- **Guided fuzzing:** collect feedback to guide the next round of mutation.

Mutation Based Fuzzing

Little or no knowledge of the structure of the inputs is assumed.

Anomalies are added to existing valid inputs.

Anomalies may be completely *random or follow some heuristics*.

Examples:

- **interest:** -1, 0x8000000, 0xffff, etc.
- **bitflip:** flipping 1,2,3,4,8,16,32 bits.
- **havoc:** random tweak in fixed length.
- **extra:** dictionary, remove Null, etc..

Example Tools:

- Taof, GPF, ProxyFuzz, Peach Fuzzer, etc.

Example: Fuzzing a PDF Viewer

- Google for .pdf (about 1 billion results).
- Crawl pages to build a corpus.
- Use fuzzing tool (or script to).
  - Grab a file.
  - Mutate that file.
  - Feed it to the program.
  - Record if it crashed (and input that crashed it).

Advantage & Disadvantage

```
bool is_ELF(Elf32_Ehdr eh)

{

    /* ELF magic bytes are 0x7f,'E','L','F'

     * Using  octal escape sequence to represent 0x7f

     */

    if(!strncmp((char*)eh.e_ident, "\177ELF", 4)) {

        printf("ELFMAGIC \t= ELF\n");

        /* IS a ELF file */

        return 1;

    } else {

        printf("ELFMAGIC mismatch!\n");

        /* Not ELF file */

        return 0;

    }

}
```

Strengths

- Super easy to setup and automate.
- Little to no protocol knowledge required.

Weaknesses

- Limited by initial corpus.
- Limited code coverage.
- May fail for protocols with checksums, those which depend on challenge response, etc.

Generation-based Fuzzing

Test cases are generated from specification of input format.

- e.g., RFC, documentation, etc.
- Wireshark has a whole family of protocol specification.

Anomalies are added to each possible spot in the inputs.

```
//png.spk
//author: Charlie Miller

// Header - fixed.
s_binary("89504E470D0A1A0A");

// IHDRChunk
s_binary_block_size_word_bigendian("IHDR"); //size of data field
s_block_start("IHDRcrc");
        s_string("IHDR");  // type
        s_block_start("IHDR");
// The following becomes s_int_variable for variable stuff
// 1=BINARYBIGENDIAN, 3=ONEBYE
                s_push_int(0x1a, 1);    // Width
                s_push_int(0x14, 1);    // Height
                s_push_int(0x8, 3);     // Bit Depth - should be 1,2,4,8,16, based on colortype
                s_push_int(0x3, 3);     // ColorType - should be 0,2,3,4,6
                s_binary("00 00");      // Compression || Filter - shall be 00 00
                s_push_int(0x0, 3);     // Interlace - should be 0,1
        s_block_end("IHDR");
s_binary_block_crc_word_littleendian("IHDRcrc"); // crc of type and data
s_block_end("IHDRcrc");
...
```

Knowledge of protocol should give better results than random fuzzing.

Continue …

```
<?xml encoding="utf-8"  version="1.0" ?>
<test>
<xslo:variable test="self::xhtml:pre"
select="seda:Contains/seda:Appraisal"  test="$xml2rfc-ext-strip-vbare='true'" />
<x:elements-xslt>     <vra-p:versionOf>     <xsl:copy>     &#x2203;     <wot:signed>
<x:param select="@dur2" />     &#x39D;     </wot:signed>     <jabberID>
<errorname>  <vra-p:versionOf>     &#x3D2;     &#x3B5;     &#x3B2;      
</vra-p:versionOf>  <w:endnote test="$generate.component.toc != 0" >
&#x2592;     &pt235;     <xsl:value-of test="empty($jpackageDoc)" />
&le;     &olarr;     </w:endnote>  </errorname>     <xsl:copy>
<body>     &pt456;     &#x251c;     &#x0342;     </body>
<seeie test="$mode = 'fit'" >     &pt13456;     &#xF4;
<people>  <axsl:choose>  <vra-p:versionOf>     &phi;     &#x03C1;
&#x202A;
</vra-p:versionOf>    </axsl:choose>    </people>
</seeie>  </xsl:copy>           </jabberID>        </xsl:copy>
</vra-p:versionOf>        </x:elements-xslt>    </test>
```

FIGURE 1: XML Fuzzing File Generated by Skyfire.

Continue ...

Strengths

- Completeness.
- Can deal with complex dependencies e.g. checksums.

Weaknesses

- Have to have spec of protocol.
  - It is possible to automatically extract protocol spec from program.
- Writing generator can be labor intensive for complex protocols.
- The spec is not the code.

Guided Fuzzing

Dumb fuzzing often hits the same code again and again (Black box) without real progress.
Generation based – stop eventually.

Guided fuzzing **collects feedback** to guide future test case generation.

- e.g., code-coverage guided fuzzing tries to explore new code with each new generated input.

Code coverage is a metric to determine how much code has been executed.

American Fuzzy Lop (AFL) is the most popular guided fuzzing tool.

Fuzzing Resources

AFL (American fuzzy lop):

- american fuzzy lop (2.52b).

Syzkaller: unsupervised coverage-guided kernel fuzzer.

- google/syzkaller: syzkaller is an unsupervised coverage-guided kernel fuzzer.

Driller: augmenting AFL with symbolic execution!

- shellphish/driller: Driller: augmenting AFL with symbolic execution!.
- used in Cyber Grand Challenge.

Awesome fuzzing:

- secfigo/Awesome-Fuzzing: A curated list of fuzzing resources (Books, courses - free and paid, videos, tools, tutorials and vulnerable applications to practice on) for learning Fuzzing and initial phases of Exploit Development like root cause analysis..

Fuzzing Rules of Thumb

Protocol specific knowledge is very helpful.
- Generational tends to beat random, better spec's make better fuzzers.

More fuzzers are better.
- Each implementation will vary, different fuzzers find different bugs.

The longer you run, the more bugs you find.

Best results come from guiding the process.
- Code coverage can be very useful for guiding the process.

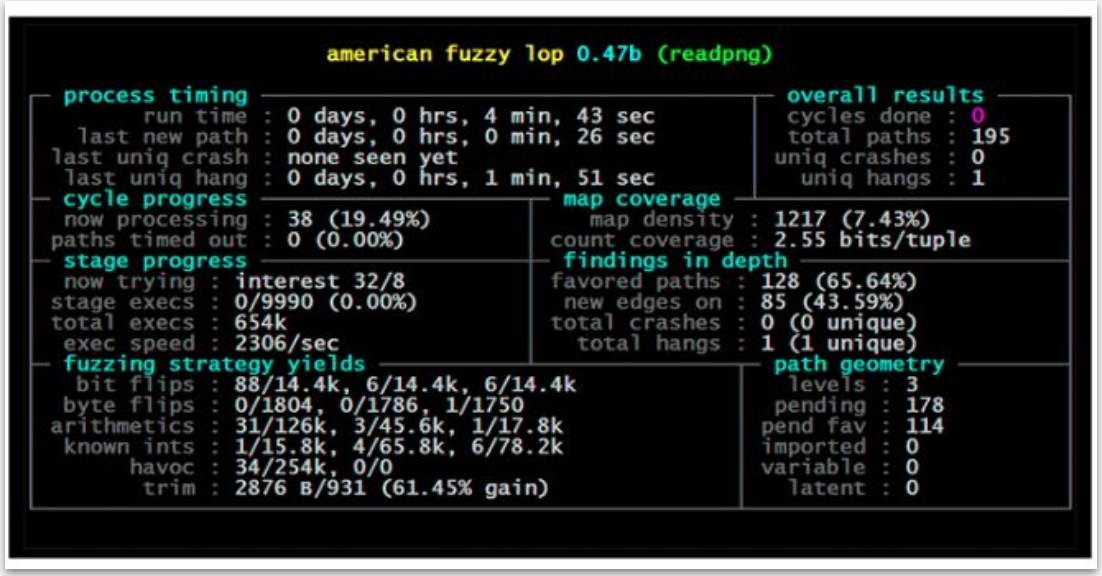
AFL

AFL

american fuzzy lop (2.52b)
AFL (american fuzzy lop) — AFL 2.53b documentation

Code coverage guided genetic fuzzer.

- A set of carefully research rules to mutate the inputs.
- It can synthesizing complex file semantics.
- It has a crash explorer, test case minimizer, and fault-triggering allocator, and syntax analyzer.



Key Idea

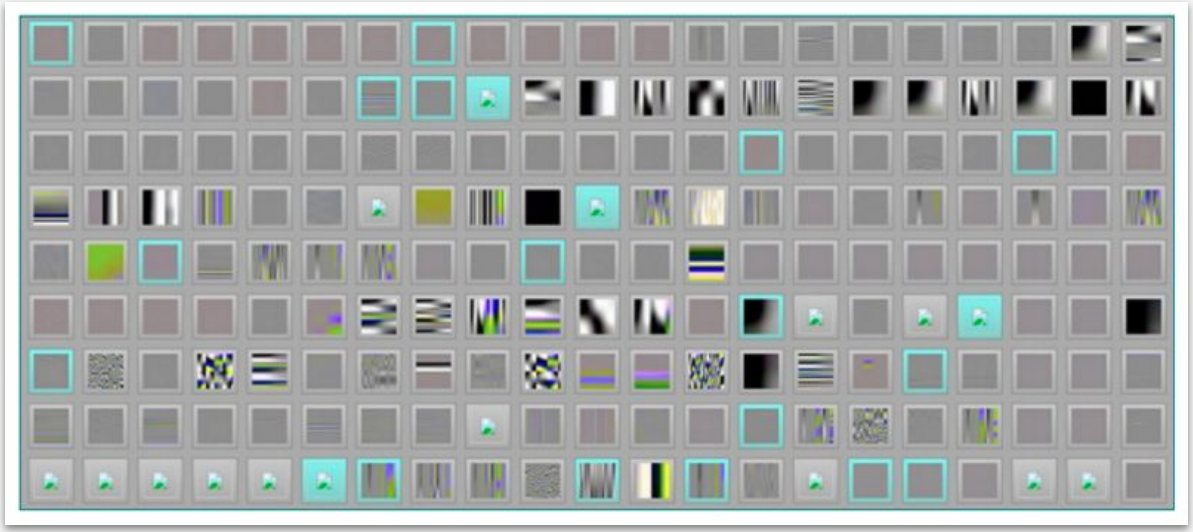
https://lcamtuf.coredump.cx/afl/technical_details.txt

Mapping input to state transitions.

- Instrumentation: Compiler (Source Code) or QEMU (Binary).

Avoiding redundant paths.

- If you see the duplicated state, throw out.
- If you see the new path, keep it for further exploration.



AFL Operations

Mutation strategies

- Highly deterministic at first – bit flips, add/sub integer values, and choose interesting integer values.
- Then, non-deterministic choices – insertions, deletions, and combinations of test cases.

afl-cmin takes a given folder of potential test cases, then runs each one and compares the feedback it receives to all rest of the testcases to find the best testcases which most efficiently express the most unique code paths.

afl-tmin works on only a specified file to avoid wasting CPU cycles fiddling with bits and bytes that are useless relative to the code paths the testcase might express.

Setup & Build

AFL User Guide — AFL 2.53b documentation

- Download and build AFL.
- Create initial test cases, the seeds.
- Build the program with afl-gcc/afl-g++ and run AFL.

```
$ cd afl-demo
$ mkdir testcases
$ cd $_
$ echo "your first test input" >01.txt
$ echo "your second test input" >02.txt
```

```
$ cd afl-demo
$ mkdir aflbuild
$ cd $_
$ CC=/path/to/afl/afl-2.51b/afl-gcc CXX=/path/to/afl/afl-2.51b/afl-g++ cmake ..
$ make
$ /path/to/afl/afl-2.51b/afl-fuzz -i ../testcases -o ../findings ./afldemo
```

AFL Output

Shows the results of the fuzzer.

- e.g., provides inputs that will cause the crash.

- File **“fuzzer_stats”** provides summary of stats – UI.
- File **“plot_data”** shows the progress of fuzzer.
- Directory **“queue”** shows inputs that led to paths.
- Directory **“crashes”** contains input that caused crash.
- Directory **“hangs”** contains input that caused hang.

HappY HackinG