Program Analysis

Mustakimur R. Khandaker

Program Analysis

Software bugs are serious problems

07-31-2010, 12:57 PM

911crashes
Junior Member

Join Date: Jul 2010
Posts: 2

[OFFLINE]

calling 911 crashes my HTC evo 4G, every time

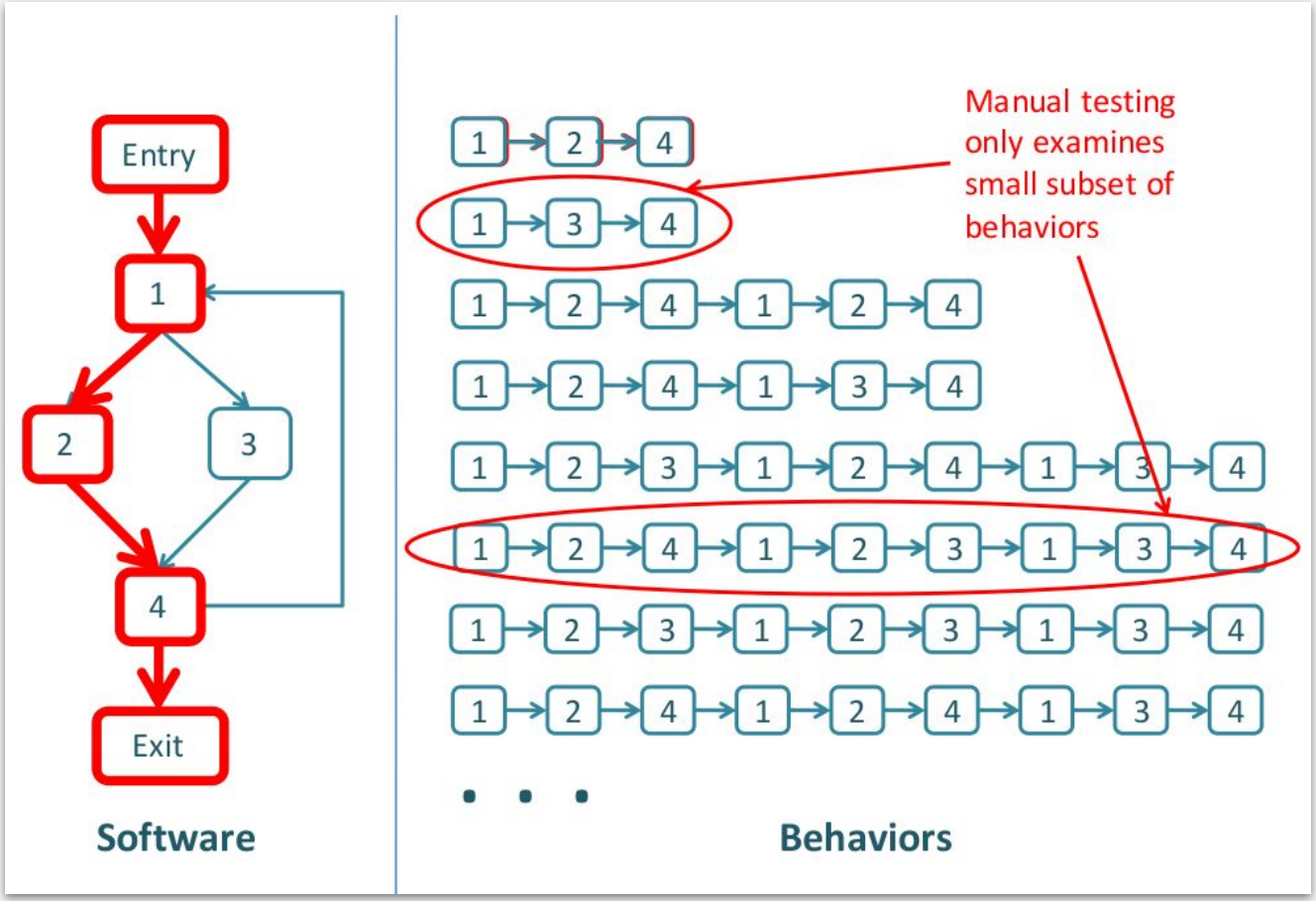
I happen to need to call 911 one night, found that my phone crashes every time I dial 911.

my wife's phone does not do that, any thought?

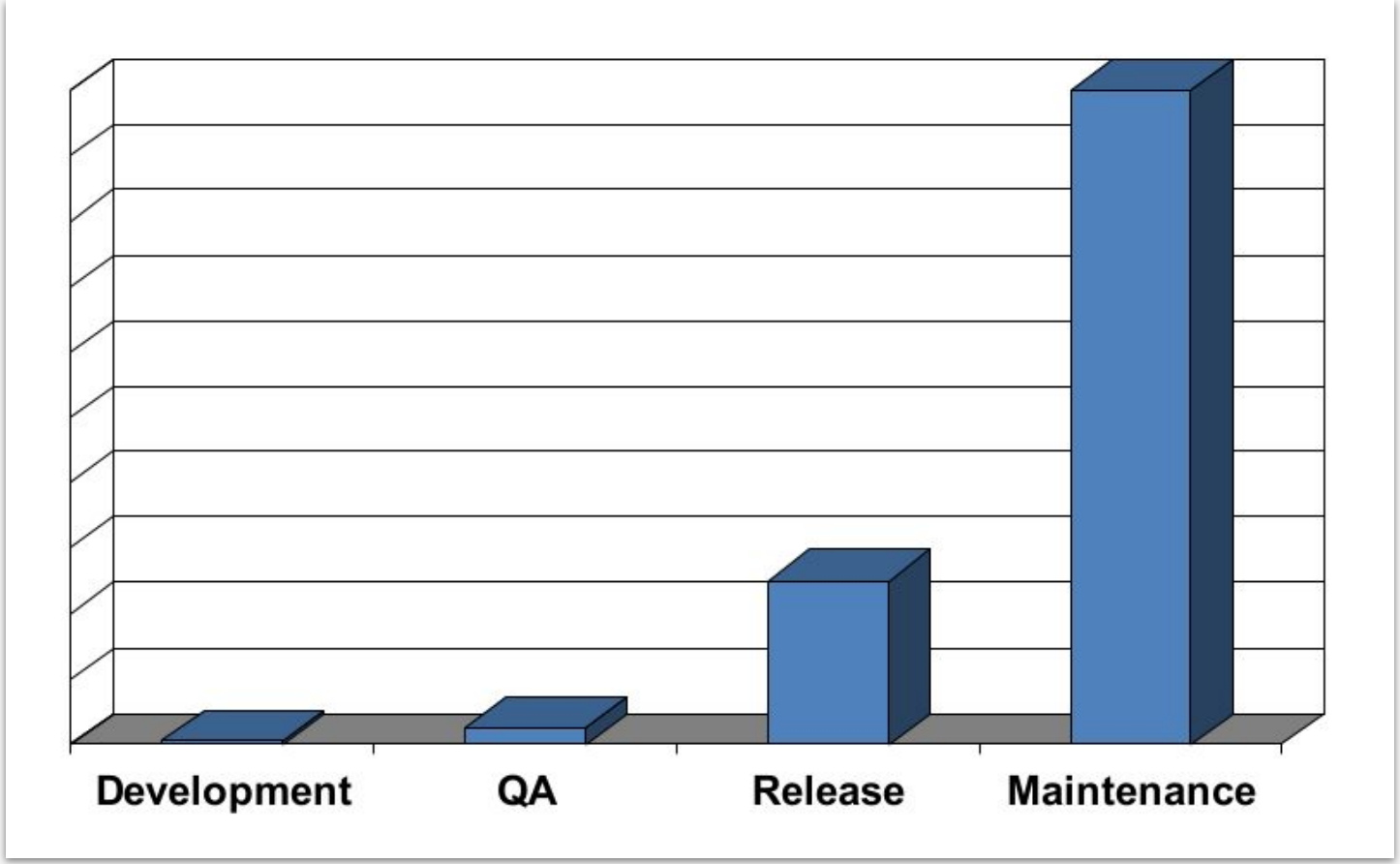
by the way, it is hard to test this problem due to the sensitivity of calling 911 repeatedly.
thanks,

heartboken

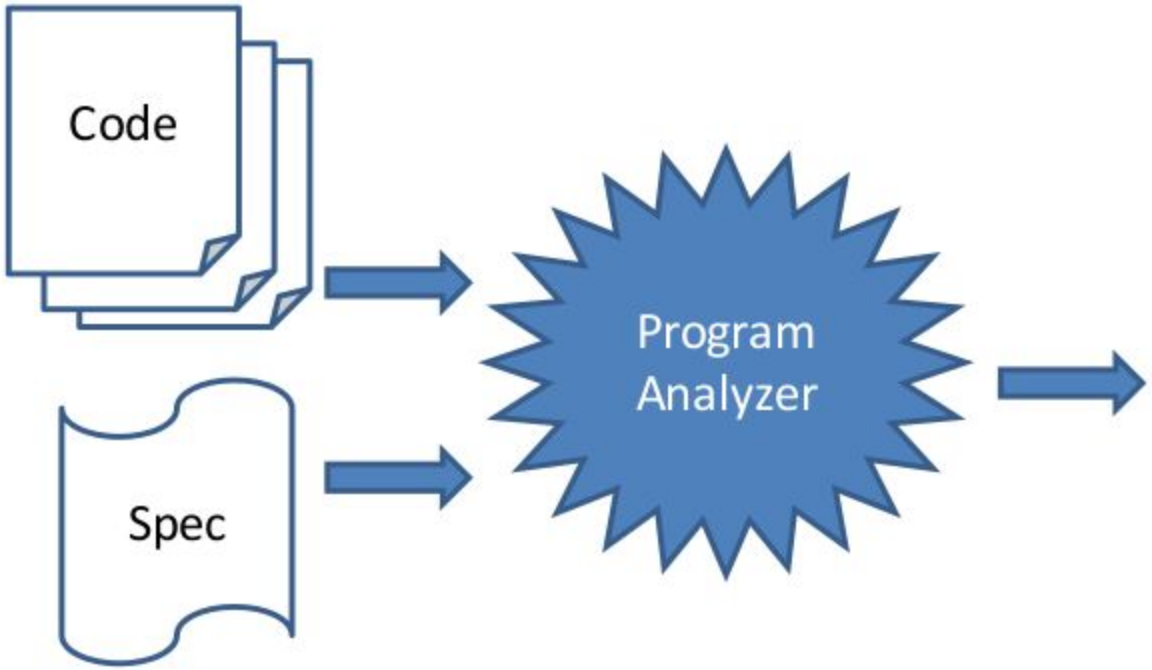
Manual Testing



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:

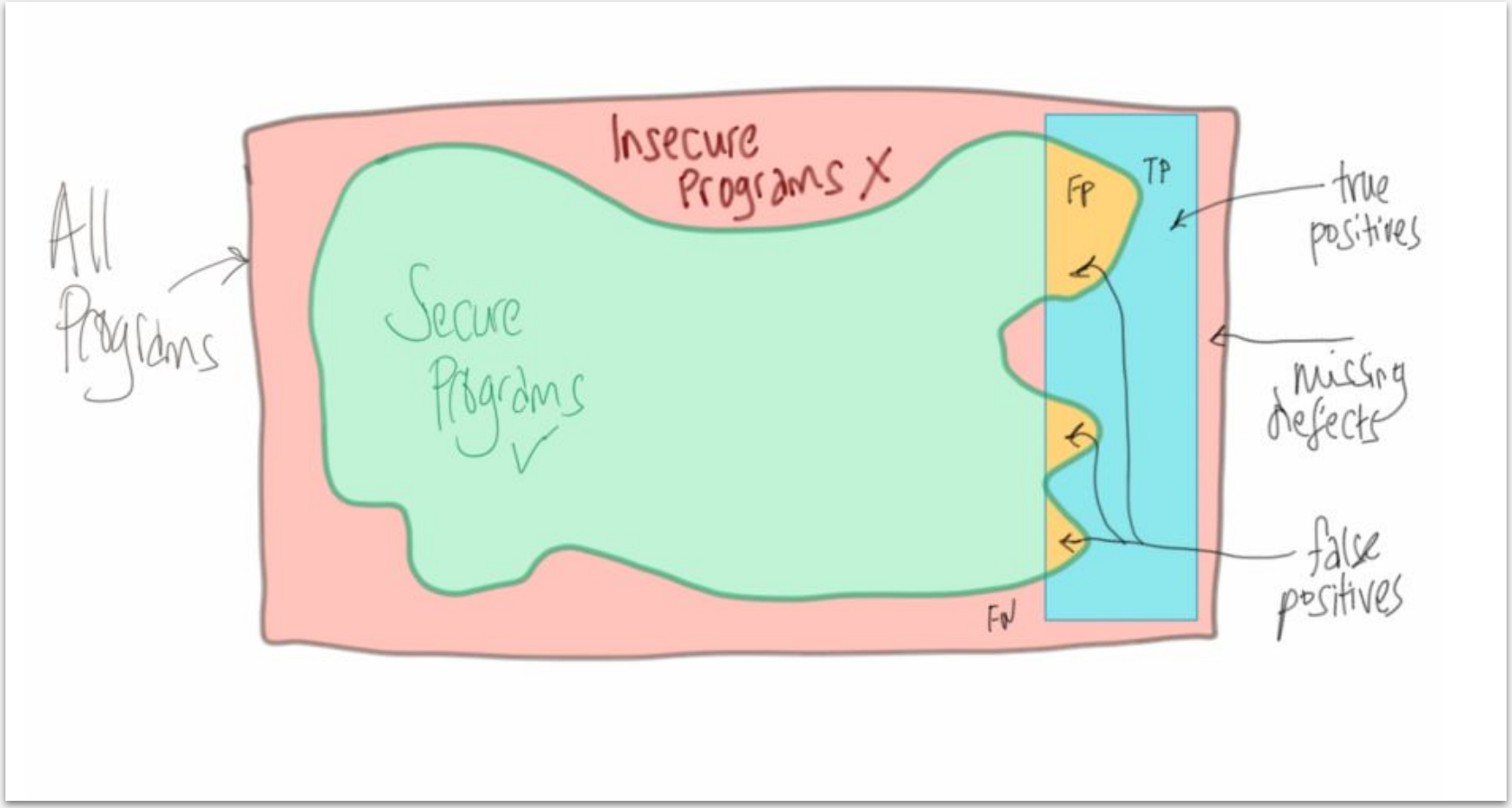
- Checks thoroughly and consistently.
- Can point to the root cause of the problem.
  - e.g., presence of buffer overflow.
- Help find errors/bugs early in the development.
  - reduce cost.
- New information can be easily incorporated to recheck a given program.

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

Soundness vs Completeness

Types

Different types of Static analysis:

- **Type checking:** part of language.
- **Style checking:** ensuring good practice.
- **Program understanding:** inferring meaning.
- **Program verification:** ensuring correct behaviour.
- **Property checking:** ensuring no bad behaviour.
- **Bug finding:** detecting likely errors.

General tools in each category may be useful for security. Dedicated static security analysis tools also exist.

Type Checking

Example 2.1 A type-checking false positive: These Java statements do not meet type safety rules even though they are logically correct.

```
10  short s = 0;
11  int i = s;   /* the type checker allows this */
12  short r = i; /* false positive: this will cause a
13                  type checking error at compile time */
```

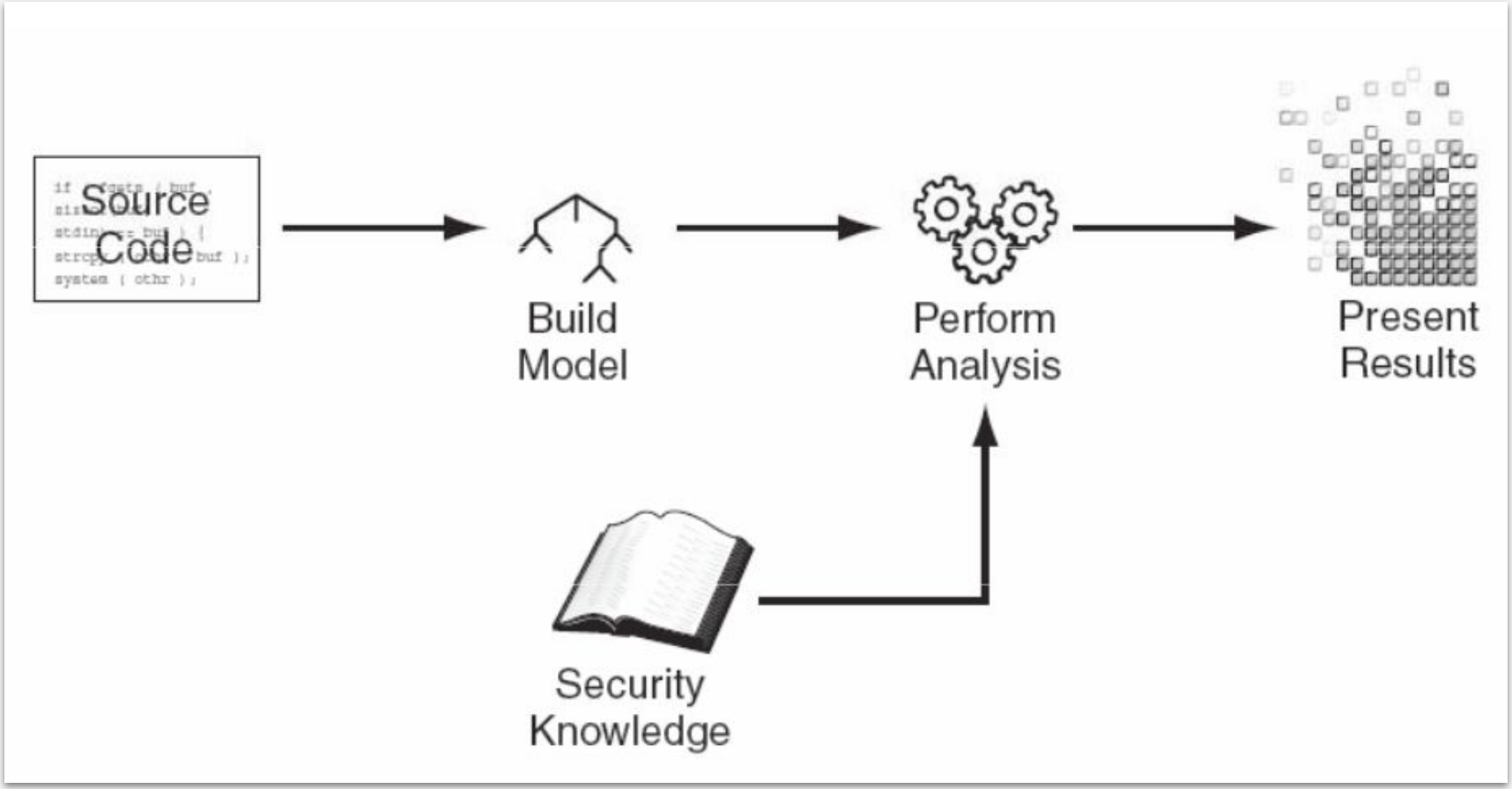
Example 2.2 Output from the Java compiler demonstrating the type-checking false positive.

```
$ javac bar.java
bar.java:12: possible loss of precision
found   : int
required: short
  short r = i; /* false positive: this will cause a
             ^
1 error
```

Example 2.3 These Java statements meet type-checking rules but will fail at runtime.

```
Object[] objs = new String[1];
objs[0] = new Object();
```

Under the Hood



Data Flow

Specify

- Security properties.
- Behavior of library code.

```
buff = getInputFromNetwork();
copyBuffer(newBuff, buff);
exec(newBuff);
```

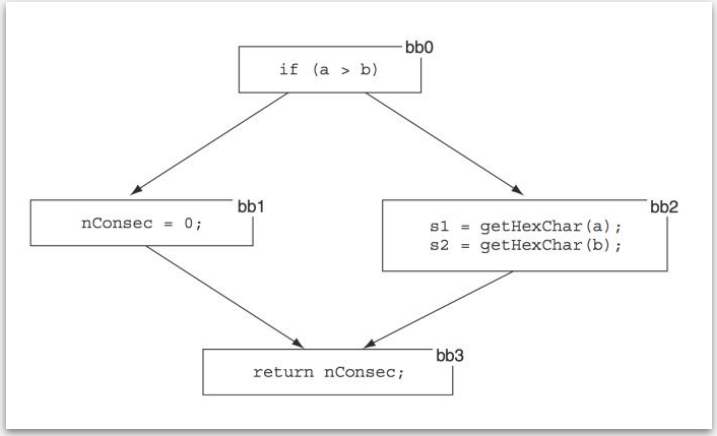
Three rules to detect the command injection vulnerability:

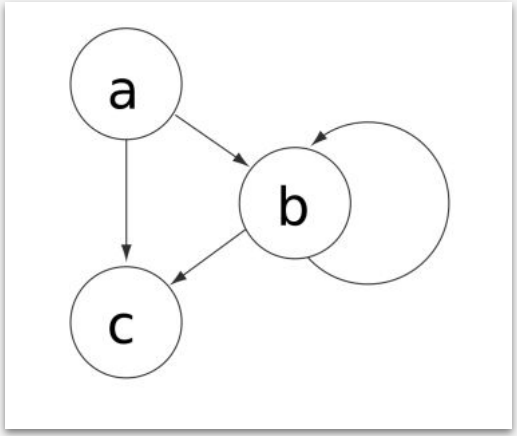
1. **getInputFromNetwork()** postcondition: *return* value is tainted.
2. **copyBuffer(arg1, arg2)** postcondition: *arg1* array values set to *arg2* array values.
3. **exec(arg)** precondition: *arg* must not be tainted.

Control Flow

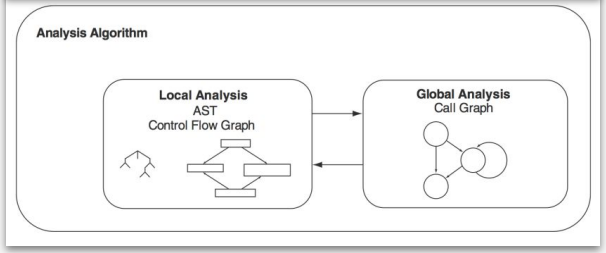
```
if (a > b) {
  nConsec = 0;
} else {
  s1 = getHexChar(1);
  s2 = getHexChar(2);
}
return nConsec;
```

```
int a(int x) {
  if (x) { b(1); } else { c(); }
}
int b(int y) {
  if (y) { c(); b(0); } else { c(); }
}
int c() { /* empty */ }
```





Putting them together: local and global



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}

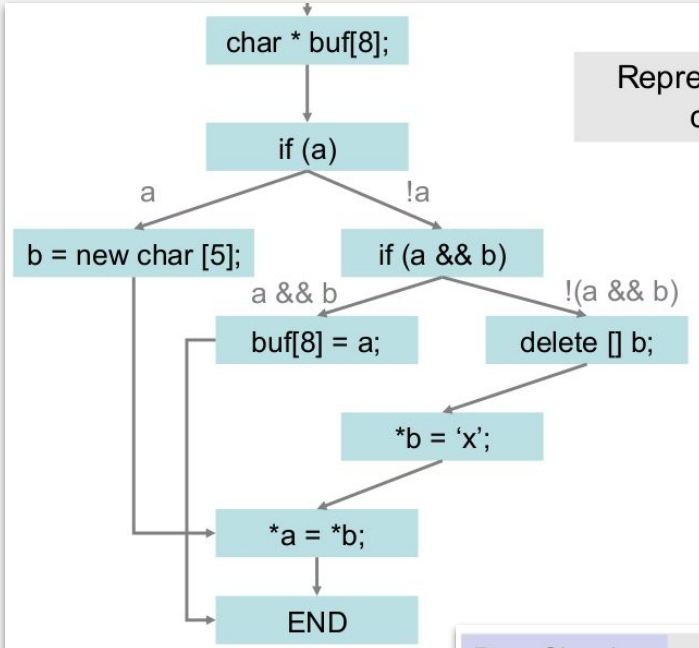
{y, *q, **s}

{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

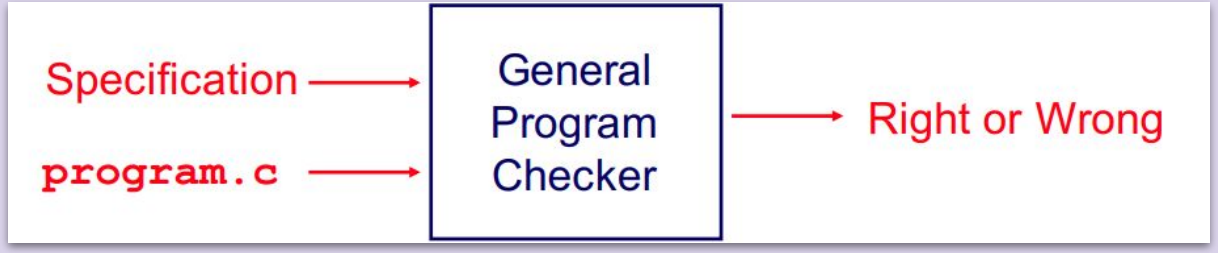
Run Checker

- 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

Program Verification

Accepts a specification and associated Code.

- Aims to prove that the code is faithful implementation.
- “equivalence checking” to check the two match.



Proof Systems

- Perform reasoning using logic formulas and rules of inference.

Automatic verification techniques

- Program assertions are derived automatically.
- Model checkers can find proofs and generate counterexamples.

Summary

Static analysis suffers from:
- Soundness.
- Completeness.

Most critical component of static analysis is constructing model:
- Data flow.
- Control flow.
- Pointer analysis.

In security, static analysis is used to:
- Finding bugs.
- Verifying program correctness.

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
```

Dynamic Code Analysis

Static vs Dynamic Analysis

Static Analysis

- Consider all possible inputs (in summary form).
- Find bugs and vulnerabilities.
- Can prove absence of bugs, in some cases.

Dynamic Analysis

- Need to choose sample test input.
- Can find bugs vulnerabilities.
- Cannot prove their absence.
- Instrument code for testing.
- Black-box testing:
    - Fuzzing and penetration testing.

Valgrind

Valgrind is a general-purpose dynamic analysis tool.

- Mainly used for memory debugging, memory leak detection and profiling.
- Essentially runs programs on a virtual machine, allowing tools to do arbitrary transformations on the program before execution.
  - Heavy-weight binary instrumentation.
- Extremely high overhead.
  - Designed to shadow all program values: registers and memory.
  - Shadowing requires serializing threads.

Use cases

- ❑ Complex memory bugs that are not detected by simpler tools like the address sanitizer.
- ❑ Complex profiling tasks.

Valgrind memcheck

Validates memory operations in a program.

- Each allocation is freed only once.
- Each access is to a currently allocated space.
- All reads are to locations already written.
- 10 – 20x overhead.

```
valgrind --tool=memcheck <prog...>
```

```
...
==29991== HEAP SUMMARY:
==29991==     in use at exit: 2,694,466,576 bytes in 2,596 blocks
==29991==   total heap usage: 16,106 allocs, 13,510 frees, 3,001,172,305 bytes allocated
==29991==
==29991== LEAK SUMMARY:
==29991==    definitely lost: 112 bytes in 1 blocks
==29991==    indirectly lost: 0 bytes in 0 blocks
==29991==      possibly lost: 7,340,200 bytes in 7 blocks
==29991==    still reachable: 2,687,126,264 bytes in 2,588 blocks
==29991==         suppressed: 0 bytes in 0 blocks
```

Intel PIN

CompArch research project, now **Intel** tool.

- Binary instrumentation tool framework.
  - “Low” overhead.
  - Provides many sample tools.

Given its architecture roots, it is best suited to specific architectural questions about a program.

- What is the instruction mix?
- What memory addresses does it access?

Pin acts as a virtual machine.

- It reassembles the instructions with appropriate instrumentation.

Instrumentation Granularity

- Instruction.
- Basic Block.
  - A sequence of instructions.
  - Single entry, single exit.
  - Terminated with one control flow instruction.
- Trace.
  - A sequence of executed basic blocks.
  - May span multiple functions.

- $ pin -t pin/source/tools/ManualExamples/obj-intel64/inscount_tls.so -- ./paraGraph bfs -t 8 -r soc-pokec_30m.graph
- $ cat inscount_tls.out

```
Total number of threads = 9
Count[0]= 561617530
Count[1]= 16153
Count[2]= 44659367
Count[3]= 44863462
```

```
// Pin calls this function every time a new basic block is encountered.
// It inserts a call to docount.
VOID Trace(TRACE trace, VOID *v)
{
    // Visit every basic block  in the trace
    for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl))
    {
        // Insert a call to docount for every bbl, passing the number of
instructions.

        BBL_InsertCall(bbl, IPOINT_ANYWHERE, (AFUNPTR)docount,
IARG_FAST_ANALYSIS_CALL, IARG_UINT32, BBL_NumIns(bbl), IARG_THREAD_ID, IARG_END);
    }
}
```

Concolic Code Analysis

Symbolic Execution

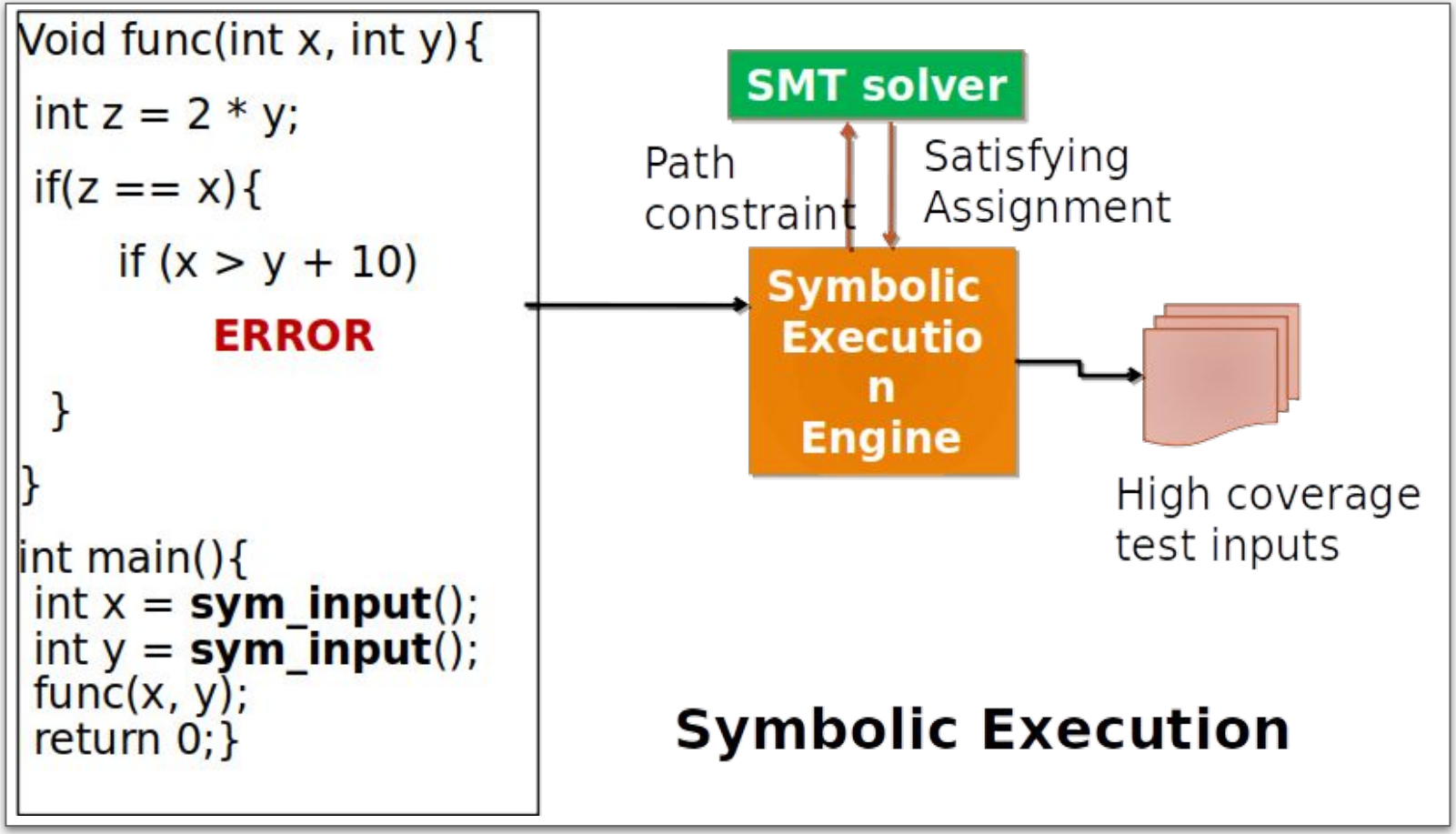
Symbolic Execution: Execute the program with symbolic valued inputs (Goal: good path coverage).

Instead of concrete state, the program **maintains symbolic states**, each of which maps variables to symbolic values.

Path condition is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far.

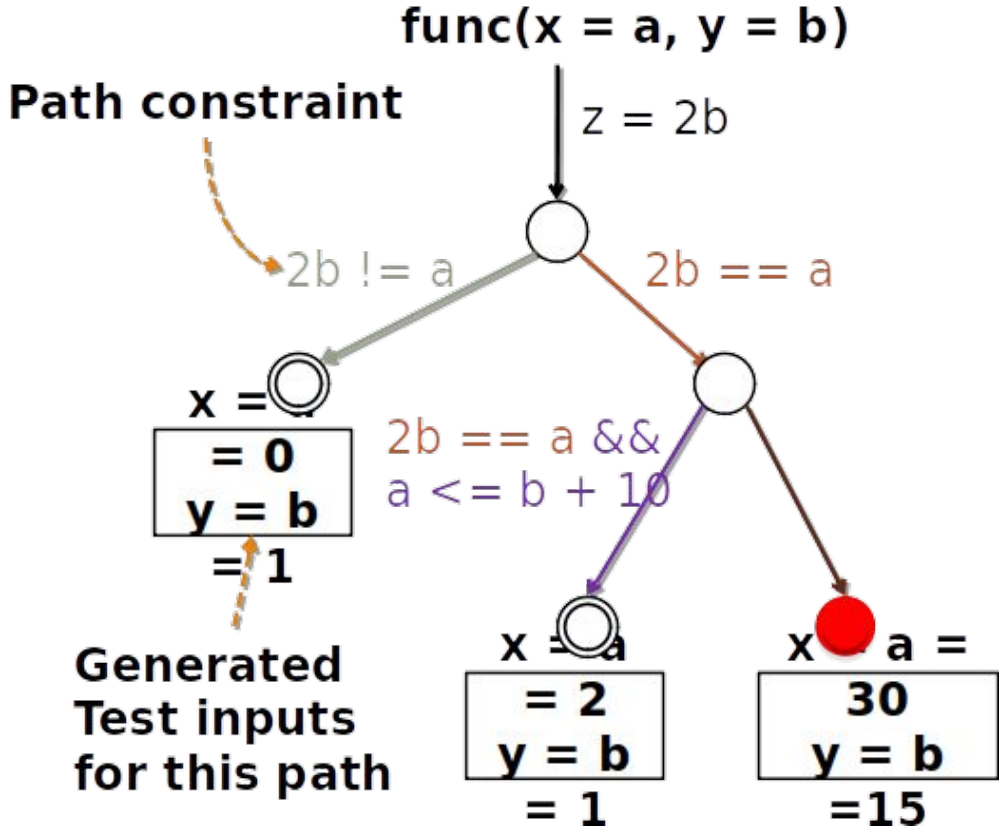
Symbolic execution implementations: KLEE, angr, Triton, Java PathFinder, etc.

Symbolic Engine



Constraint Solver

```
Void func(int x, int y){

 int z = 2 * y;

 if(z == x){
       if (x > y + 10)
            ERROR

 }

}

int main(){
 int x = sym_input();
 int y = sym_input();
 func(x, y);
 return 0;

}
```



Issues

Loops and recursions: infinite execution tree.
Path explosion: exponentially many paths.
Heap modeling: symbolic data structures and pointers.

SMT solver limitations: dealing with complex path constraints.

Environment modeling: dealing with native/system/library calls/file operations/network events .

Coverage Problem: may not reach deep into the execution tree, specially when encountering loops.

Solution: Concolic Execution

Concolic Execution: Combine concrete execution and symbolic execution.

Concrete + Symb**olic** = ***Concolic***

Also called *dynamic symbolic execution*.

The intention is to visit deep into the program execution tree.

- Program is simultaneously executed with concrete and symbolic inputs.
- Start off the execution with a random input.
- Specially useful in cases of remote procedure call.
- Can be focused on localized code coverage.

Concolic execution implementations: SAGE (Microsoft), CREST.

< Next: Fuzzing />