Recent Trends in Program Analysis for Bug Hunting and Exploitation

ben-holland.com

~~Recent~~ Two Positive Trends in Program Analysis for Bug Hunting and Exploitation

ben-holland.com

$ whoami

- 2005 – 2010
  - B.S. in Computer Engineering
  - Internship: Wabtec Railway Electronics, Ames Lab, Rockwell Collins
- 2010 – 2011
  - B.S. in Computer Science
  - Internship: Rockwell Collins
- 2010 – 2012
  - M.S. in Computer Engineering (Co-major Information Assurance)
  - Internship: MITRE
  - Thesis: Enabling Open Source Intelligence (OSINT) in private social networks
- 2012 – 2015
  - Research Associate → Assistant Scientist
  - DARPA's APAC and STAC programs
    - Demands impactful and practical software solutions for open security problems
    - Fast-paced, high-stakes, adversarial engagement challenges
- 2015 – Present
  - Ph.D. in Computer Engineering
  - Graduating: 2018 🤞🎓

Disclaimer

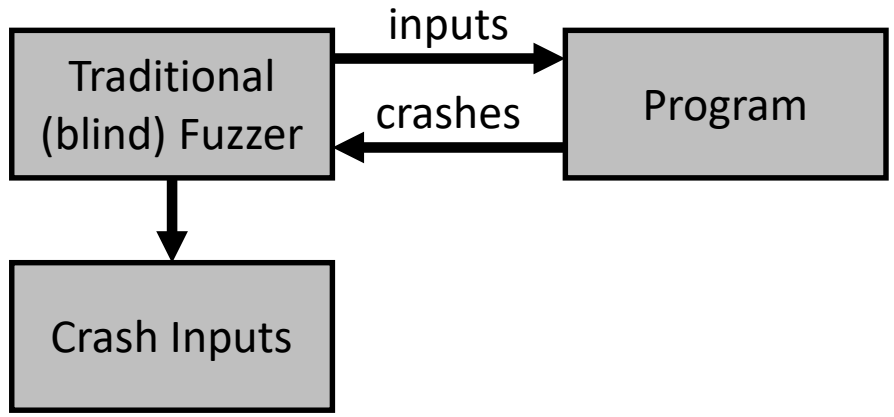
- Nobody is endorsing me to say any of the things I am about to say
- What I am going to say is my opinion and may be controversial among experts
- I am somewhat unavoidably biased towards certain approaches
- I'll probably ask more questions than I have answers
- I'll probably even get a few things wrong…

How do we analyze a program?

- Two main camps
  - Dynamic analysis
    - Run the program with some inputs and see what it does
    - Advantage: Everything we observe is feasible (we just saw it happen)
    - Concern: Input space is HUGE
    - Concern: Did we test the *interesting* inputs?
  - Static analysis
    - Don’t run the program, dissect the logic and examine program artifacts
    - Advantage: Bird’s eye view of everything that could possibly happen during execution
    - Concern: Number of program behaviors is HUGE
    - Concern: Is it feasible to reach/trigger an artifact of concern?
- What are we looking for?
  - Bugs: Memory corruption, rounding errors, null pointers, infinite loops, stack overflows, race conditions, memory leaks, business logic flaws, …
  - Not every issue translates to a crash!

Traditional/Dumb (Blind) Fuzzing

- Start with a test corpus of well formed program inputs
- Apply random or systematic mutations to program inputs
- Run program with mutated inputs and observe whether or not the program crashes
- Repeat until the program crashes
- Input space
  - Reading data in a loops could make the input space infinite
  - There are 2^n possible inputs for a binary input of length *n*



This is about all we can do without examining program artifacts…

Building a Static Analysis Tool

- Compiler: Lexer → Parser → AST → Semant → Code Generation
- Static Analysis: Lexer → Parser → AST → Semant → Graph of Program Artifacts → Graph Queries of Concerning Program Relationships

- Demo Eclipse Plugin: http://ben-holland.com/AtlasBrainfuck/updates

Brainf*ck Language

- Designed by Urban Müller in 1992 with the goal of implementing the smallest possible compiler.
- Compiler can be implemented in less than 100 bytes
- Implements a Turing machine
- https://en.wikipedia.org/wiki/Brainfuck

Character	Meaning
>	increment the data pointer (to point to the next cell to the right).
<	decrement the data pointer (to point to the next cell to the left).
+	increment (increase by one) the byte at the data pointer.
-	decrement (decrease by one) the byte at the data pointer.
.	output the byte at the data pointer.
,	accept one byte of input, storing its value in the byte at the data pointer.
[	if the byte at the data pointer is zero, then instead of moving the instruction pointer forward to the next command, jump it *forward* to the command after the *matching*] command.
]	if the byte at the data pointer is nonzero, then instead of moving the instruction pointer forward to the next command, jump it *back* to the command after the *matching* [command.

Brainf*ck (Hello World)

```
++++++++[>++++[>++>+++>+++>+<<<<-]>+>+>->>+[<]<-]>>.>---
.+++++++..+++.>>.<-.<.+++.------.--------.>>+.>++.
```

Brainf*ck Lexical Analysis

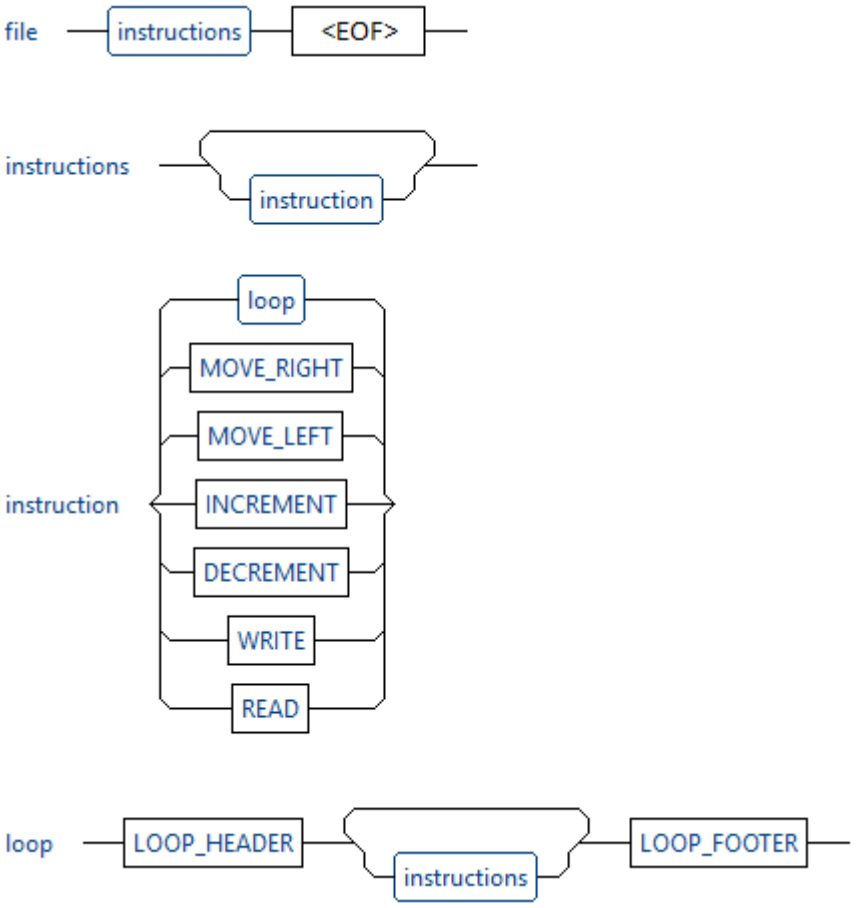
```
MOVE_RIGHT: '>';
MOVE_LEFT: '<';
INCREMENT: '+';
DECREMENT: '-';
WRITE: '.';
READ: ',';
LOOP_HEADER: '[';
LOOP_FOOTER: ']';
```

Program: ***++[>+[+]].***

Program Tokens: **INCREMENT INCREMENT LOOP_HEADER MOVE_RIGHT INCREMENT LOOP_HEADER INCREMENT LOOP_FOOTER LOOP_FOOTER WRITE <EOF>**

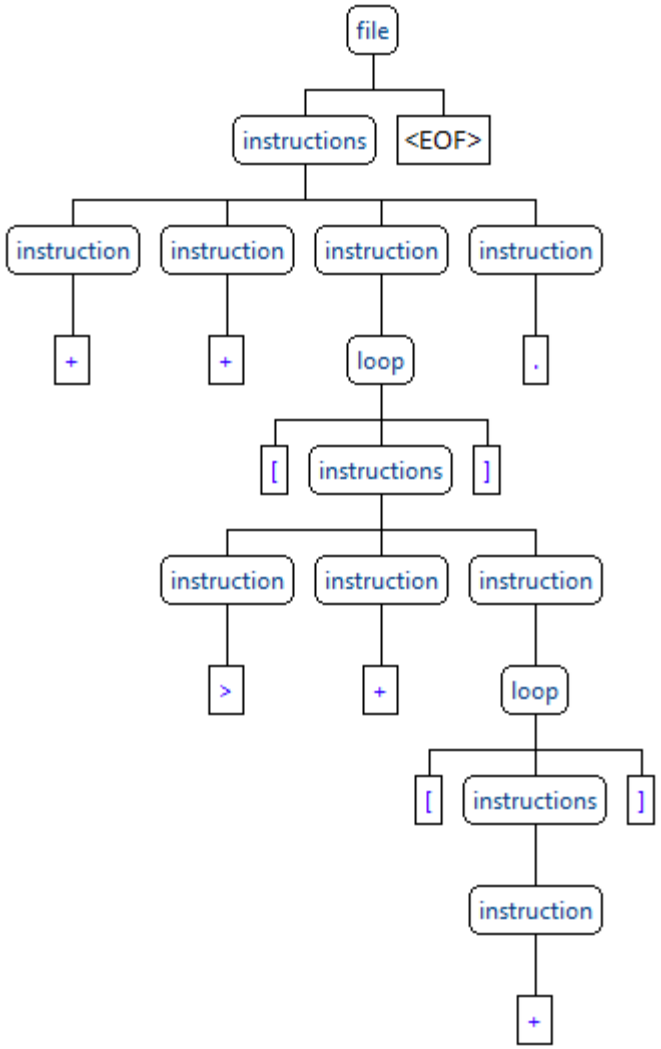
Brainf*ck Parsing Rules

```
file: instructions EOF;

instructions: instruction+;

instruction: loop
           | MOVE_RIGHT
           | MOVE_LEFT
           | INCREMENT
           | DECREMENT
           | WRITE
           | READ
           ;

loop: LOOP_HEADER instructions+ LOOP_FOOTER;
```

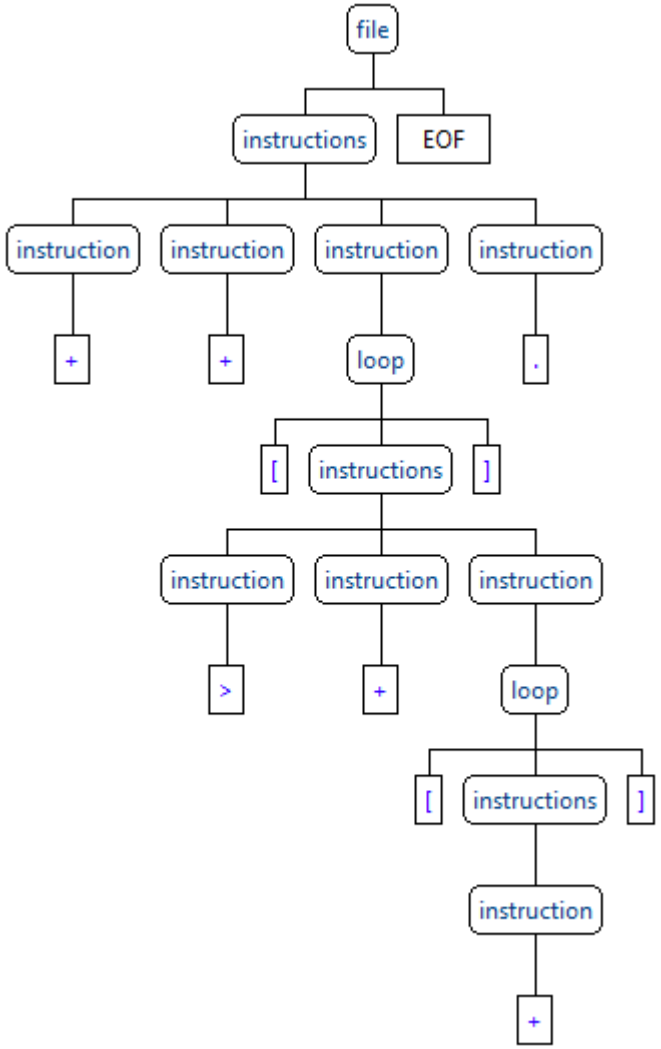


Brainf*ck Parse Tree

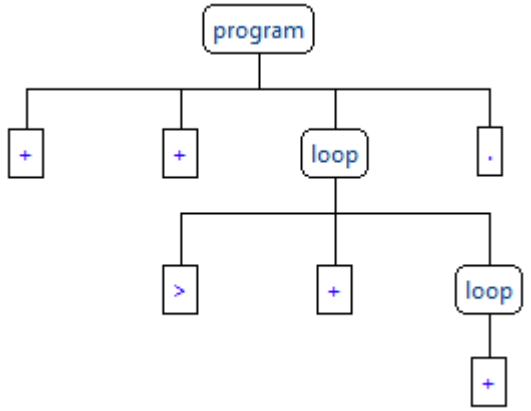
Program: ***++[>+[+]].***



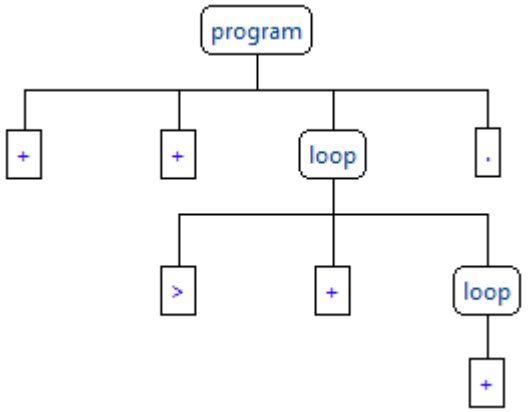
Brainf*ck Abstract Syntax Tree (AST)



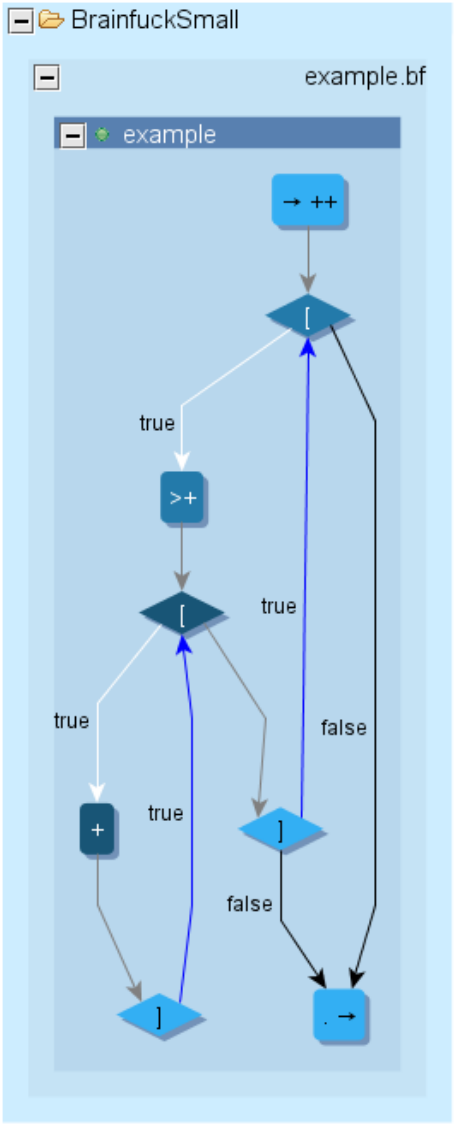
Parse Tree(s) to AST



Brainf*ck AST to Program Graph



Parse Tree(s) to AST

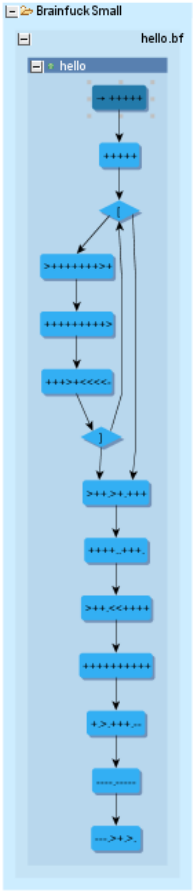


hello.bf

```
1 +++++
2 +++++
3 [>++++++++>++++++++++>+++>+<<<<-]
4 >++.>+.+++++++..+++.>++.<<+++++++++++++++.>.+++.------.--------.>+.>.
```

- Brainf*ck Hello World Program
- Graph contains information necessary to execute program
- This language should be simple to analyze right???
  - No variables, just tape cells
  - How many behaviors could there be?

Graph 2



atlas

Error Log | Search | Problems | Atlas Shell | Console | Progress | Analysis Keys | Git Staging

```
Operation completed successfully...

show(cfg(universe.nodes(XCSG.Function)))

com.benjholla.atlas.brainfuck.interpreter.BrainfuckGraphInterpreter.execute(selected)

res1: String =
"Hello World!
"
```

Evaluate:

atlas

Elemental: A Brainf*ck Derivative

- github.com/benjholla/Elemental
  - Goal is to be basic, not to be tiny
  - Separates looping and branching
  - New features to explore impacts of modern language features

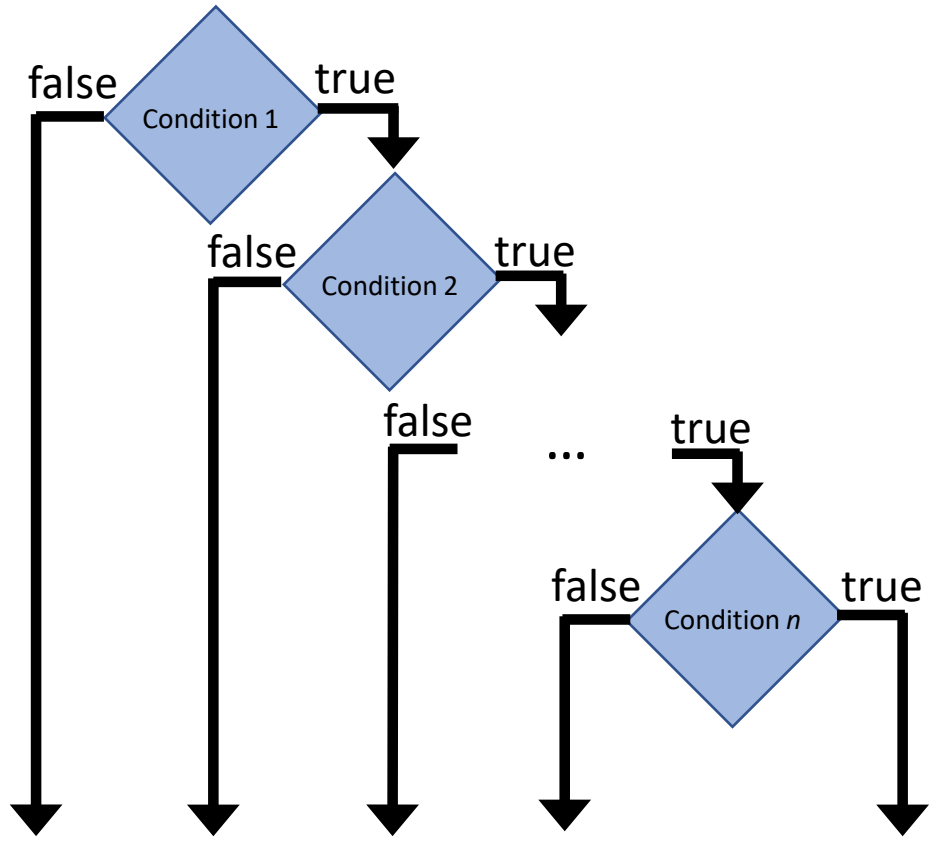
Instruction	Description
+	Increment the byte at the current tape cell by 1
-	Decrement the byte at the current tape cell by 1
<	Move the tape one cell to the left
>	Move the tape one cell to the right
,	(Store) Read byte value from input into current tape cell
.	(Recall) Write byte value to output from current tape cell
(	(Branch) If the byte value at the current cell is 0 then jump to the instruction following the matching), else execute the next instruction
[	(While Loop) If the byte value at the current cell is 0 then jump to the instruction following the matching], else execute instructions until the matching] and then unconditionally return to the [
[0-9]+:	(Function) Declares a uniquely named function (named [0-9]+ within range 0-255)
{*[0-9]+*}	(Static Dispatch) Jump to a named function
?	(Dynamic Dispatch/Function Pointer) Jumps to a named function with the value of the current cell
"*[0-9]+*"	(Label) Sets a unique label (named [0-9]+ within range 0-255) within a function
'*[0-9]+*'	(GOTO) Jumps to a named label within the current function
&	(Computed GOTO) Jumps to the named label within the current function with the value of the current cell
#	A one line comment

Counting Program Paths: Branching

- How many paths are there for *n* nested branches?

```
if(condition_1){
   if(condition_2){
      if(condition_3){
         …
         if(condition_n){
            // conditions 1 through n
            // must all be true to reach here
         }
      }
   }
}
```

n+1 paths!

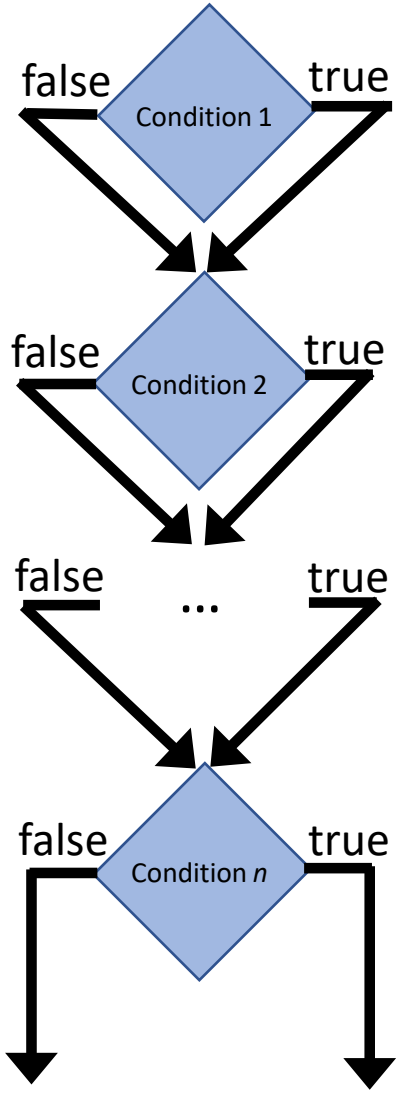


Counting Program Paths: Branching

- How many paths are there for *n* non-nested branches?

```
if(condition_1){
   // code block 1
}
if(condition_2){
   // code block 2
}
if(condition_3){
   // code block 3
}
…
if(condition_n){
   // code block n
}
```

2^n paths!



Elemental: A Brainf*ck Derivative

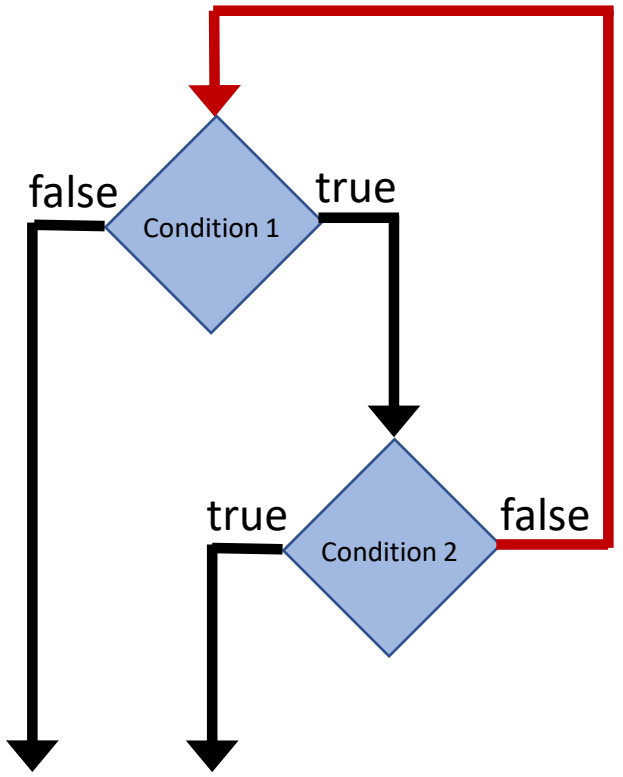
- github.com/benjholla/Elemental
  - Goal is to be basic, not to be tiny
  - Separates looping and branching
  - New features to explore impacts of modern language features

Instruction	Description
+	Increment the byte at the current tape cell by 1
-	Decrement the byte at the current tape cell by 1
<	Move the tape one cell to the left
>	Move the tape one cell to the right
,	(Store) Read byte value from input into current tape cell
.	(Recall) Write byte value to output from current tape cell
(	(Branch) If the byte value at the current cell is 0 then jump to the instruction following the matching), else execute the next instruction
[	(While Loop) If the byte value at the current cell is 0 then jump to the instruction following the matching], else execute instructions until the matching] and then unconditionally return to the [
[0-9]+:	(Function) Declares a uniquely named function (named [0-9]+ within range 0-255)
{*[0-9]+*}	(Static Dispatch) Jump to a named function
?	(Dynamic Dispatch/Function Pointer) Jumps to a named function with the value of the current cell
"*[0-9]+*"	(Label) Sets a unique label (named [0-9]+ within range 0-255) within a function
'*[0-9]+*'	(GOTO) Jumps to a named label within the current function
&	(Computed GOTO) Jumps to the named label within the current function with the value of the current cell
#	A one line comment

Considering Loops

- Programs may have loops
  - How many paths does this program have?
  - Can we say if this program halts?

```
while(condition_1){
    if(condition_2){
        break;
    }
}
```



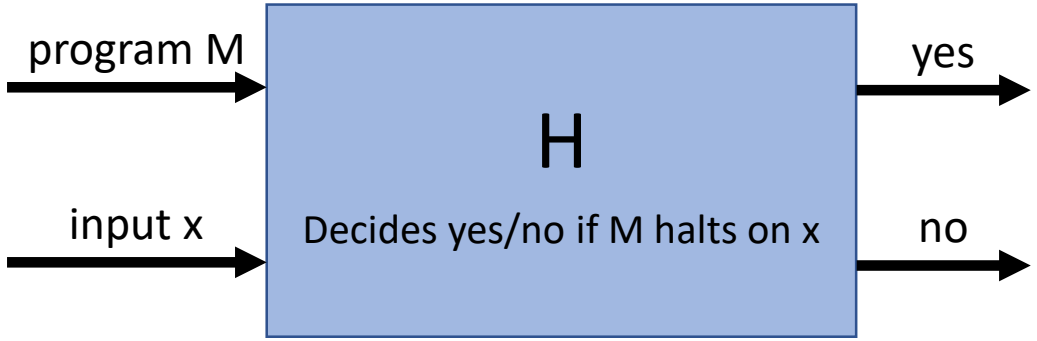
The Halting Problem

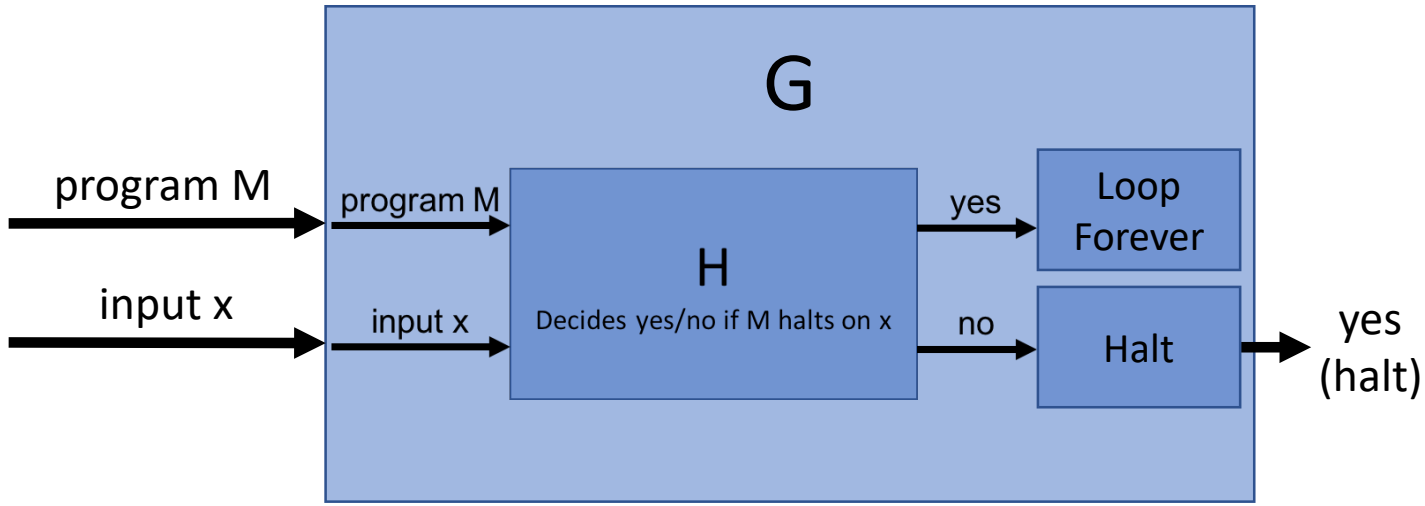
Suppose, we could construct:
H(M, x) := if *M* halts on *x* then return true else return false

Then we could construct:
G(M, x) := if G(M*, x)* is false then return true else loop forever

But if we then pass *G* to itself, that is G(G,G), we get a contradiction between what *G* does and what *H* says that *G* does. If *H* says that *G* halts, then *G* does not halt. If *H* says that *G* does not halt, then it does halt.

H cannot exist.





Elemental: A Brainf*ck Derivative

- github.com/benjholla/Elemental
  - Goal is to be basic, not to be tiny
  - Separates looping and branching
  - New features to explore impacts of modern language features

- '?' could pass control to any function!
- '&' could jump to any line!
- Goto labels with '?' or '&' could be simulated with branching or loops
- These blur control flow with data

Instruction	Description
+	Increment the byte at the current tape cell by 1
-	Decrement the byte at the current tape cell by 1
<	Move the tape one cell to the left
>	Move the tape one cell to the right
,	(Store) Read byte value from input into current tape cell
.	(Recall) Write byte value to output from current tape cell
(	(Branch) If the byte value at the current cell is 0 then jump to the instruction following the matching), else execute the next instruction
[	(While Loop) If the byte value at the current cell is 0 then jump to the instruction following the matching], else execute instructions until the matching] and then unconditionally return to the [
[0-9]+:	(Function) Declares a uniquely named function (named [0-9]+ within range 0-255)
{*[0-9]+*}	(Static Dispatch) Jump to a named function
?	(Dynamic Dispatch/Function Pointer) Jumps to a named function with the value of the current cell
"*[0-9]+*"	(Label) Sets a unique label (named [0-9]+ within range 0-255) within a function
'*[0-9]+*'	(GOTO) Jumps to a named label within the current function
&	(Computed GOTO) Jumps to the named label within the current function with the value of the current cell
#	A one line comment

Positive Trend 1 – Address the Languages

- Data drives execution
  - Data is half of the program!
  - “The illusion that your program is manipulating its data is powerful. But it is an illusion: The data is controlling your program.”
- Crema: A LangSec-Inspired Programming Language
  - Giving a developer a Turning complete language for every task is like giving a 16 year old a formula one car (something bad is bound to happen soon)

Trend 1 – Address the Languages

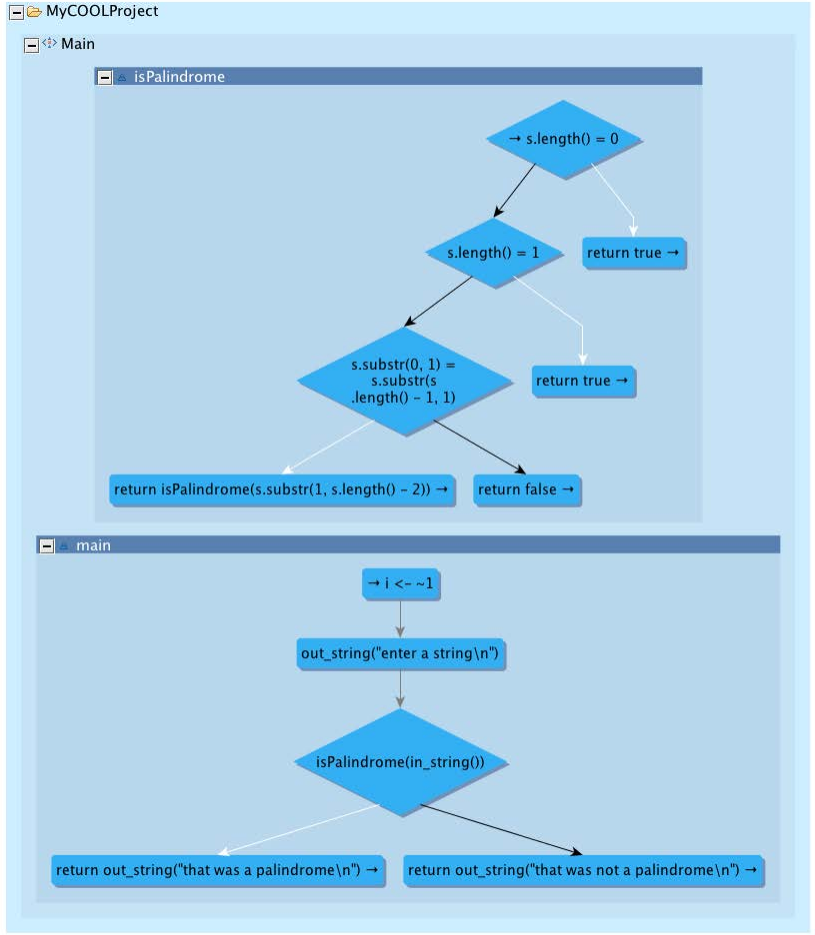
- Data drives execution
  - Data is half of the program!
  - “The illusion that your program is manipulating its data is powerful. But it is an illusion: The data is controlling your program.”

- Crema: A LangSec-Inspired Programming Language (DARPA Pilot Study)
  - Giving a developer a Turing complete language for every task is like giving a 16 year old a formula one car (something bad is bound to happen soon)
  - Apply principle of least privilege to computation (least computation principle)
    - Computational power exposed to attacker ***is*** privilege. Minimize it.
    - Try copy-pasting the XML billion-laughs attack from Notepad into MS Word if you want to see why…

Scaling Up: Program Analysis for COOL

- Classroom Object Oriented Language (COOL)
  - https://en.wikipedia.org/wiki/Cool_(programming_language)
  - http://openclassroom.stanford.edu/MainFolder/CoursePage.php?course=Compilers

- COOL Program Graph Indexer
  - Type hierarchy
  - Containment relationships
  - Function / Global variable signatures
  - Function Control Flow Graph
  - Data Flow Graph (in progress)
  - Inter-procedural relationships:
    - Call Graph (implemented via compliance to XCSG!)
  - https://github.com/benjholla/AtlasCOOL (currently private)



Program Analysis for Contemporary Languages

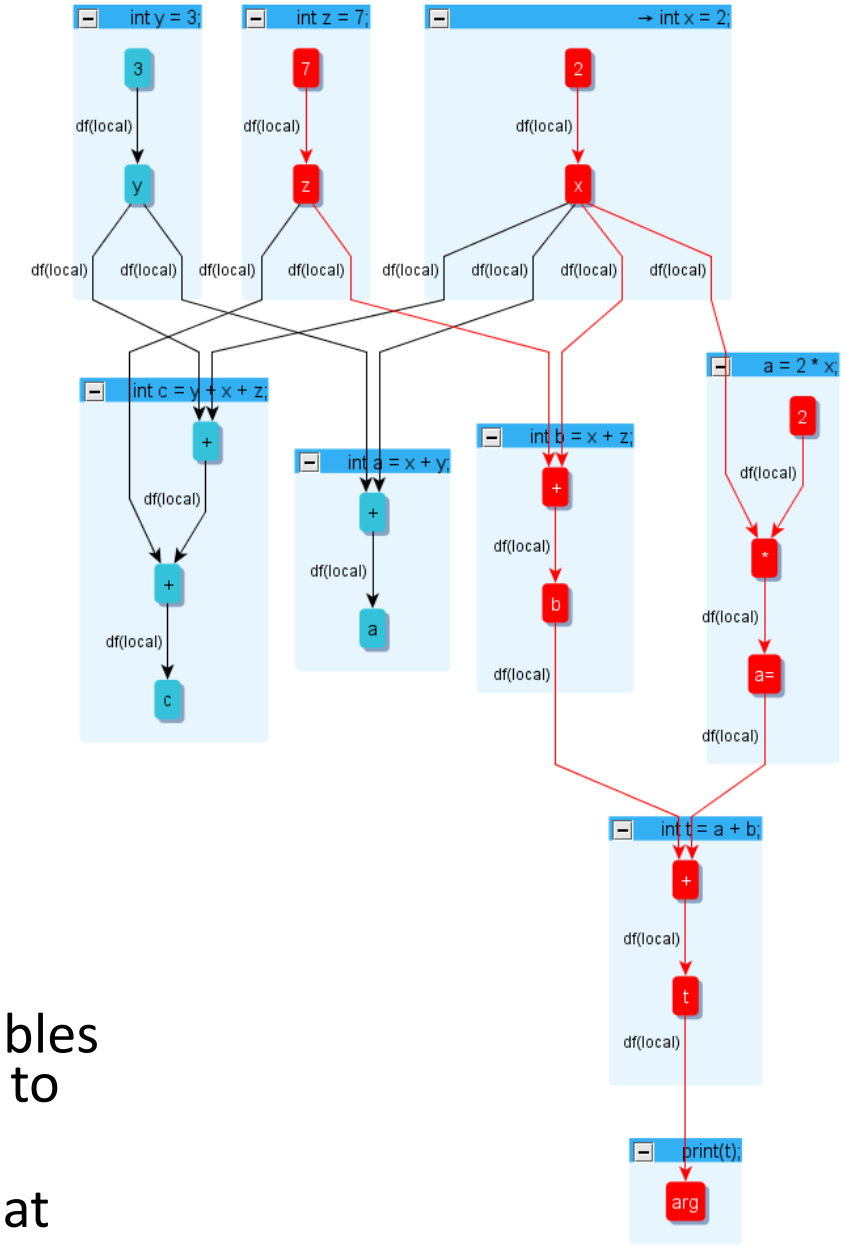
- http://www.ensoftcorp.com/atlas (Atlas)
  - C, C++, Java Source, Java Bytecode, *and now Brainfuck/COOL!*
- https://scitools.com (Understand)
  - C, C++ Source
- http://mlsec.org/joern (Joern)
  - C, C++, PHP Source
- https://www.hex-rays.com/products/ida (IDA)
- https://binary.ninja (Binary Ninja)
- https://www.radare.org (Radare)

Data Flow Graph (DFG)

Example:

1. x = 2;
2. y = 3;
3. z = 7;
4. a = x + y;
5. b = x + z;
6. a = 2 * x;
7. c = y + x + z;
8. t = a + b;
9. print(t); ← detected failure

Relevant lines:
1,3,5,6,8

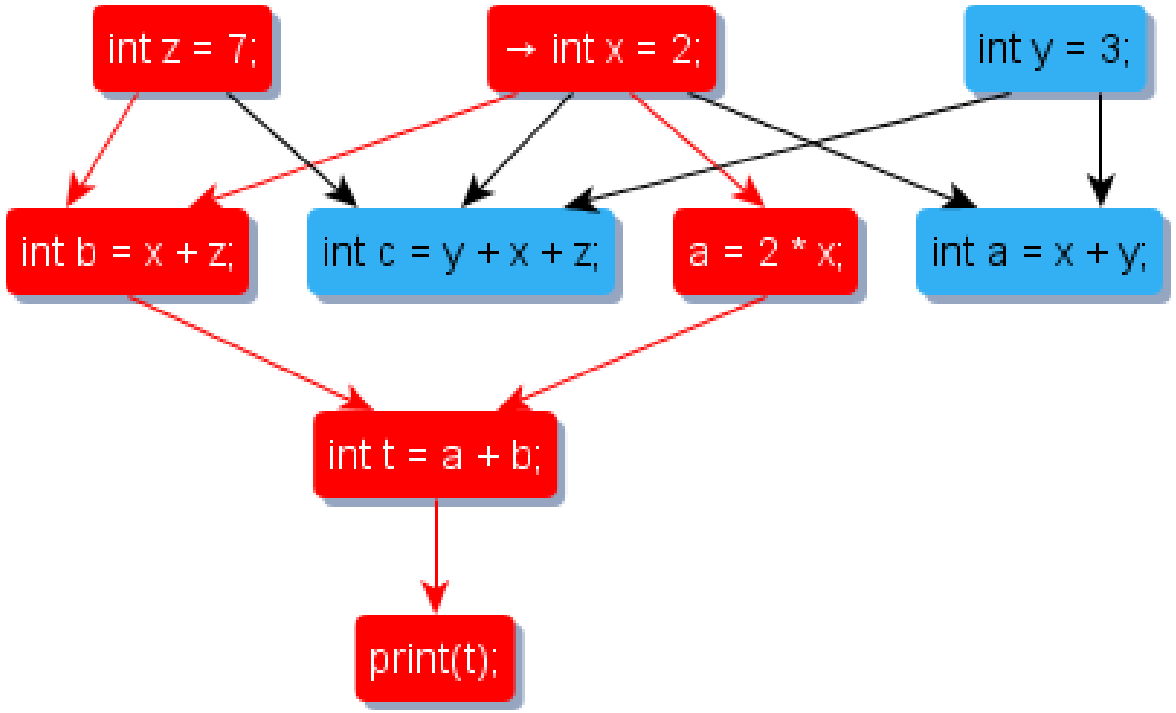


What lines must we consider if the value of *t* printed is incorrect?

- A *Data Flow Graph* creates a graph of primitives and variables where each assignment represents an edge from the RHS to the LHS of the assignment
- The *Data Flow Graph* represents global data dependence at the operator level (the atomic level) [FOW87]

Data Dependence Graph (DDG)

- Note that we could summarize data flow on a per statement level
- This graph is called a *Data Dependence Graph* (DDG)
- DDG dependences represent only the *relevant* data flow relationships of a program [FOW87]



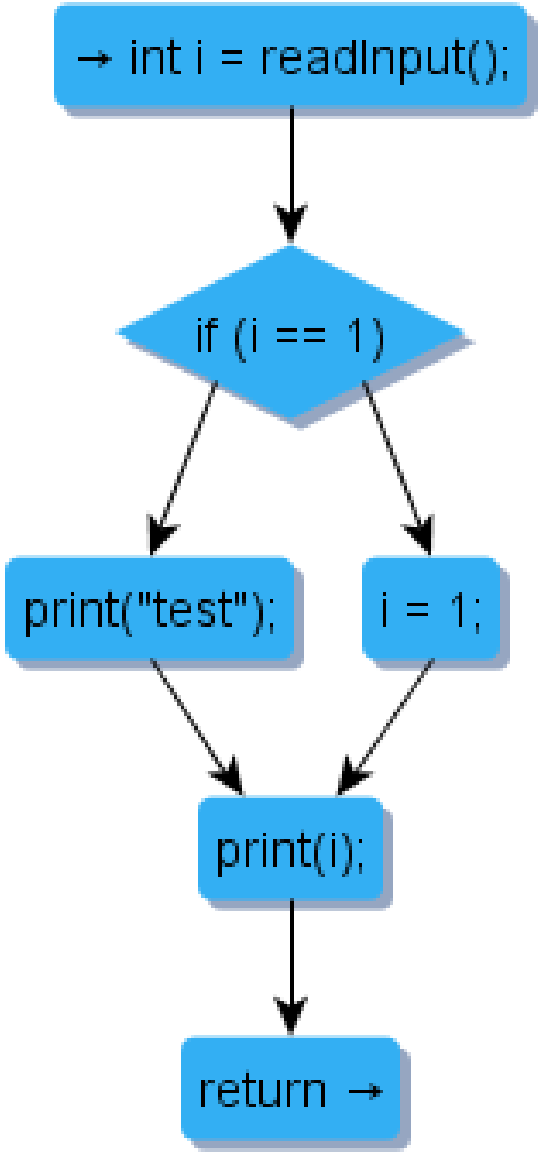
Control Flow Graph (CFG)

Example:

```
1. i = readInput();
2. if(i == 1)
3.     print(“test”);
   else
4.     i = 1;
5. print(i);
6. return; // terminate
```

Relevant lines:
1,2,4

detected failure

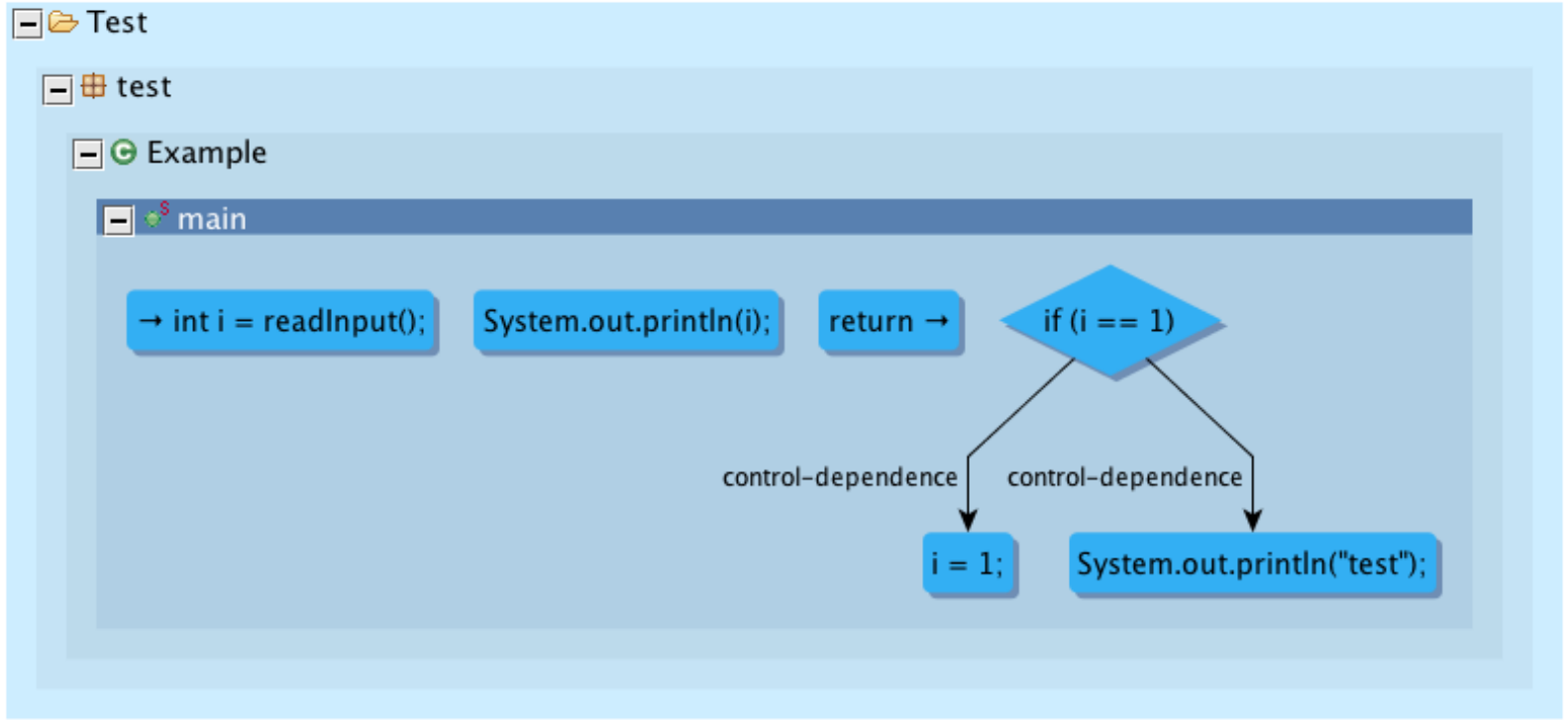


What lines must we consider if the value of *i* printed is incorrect?

- A *Control Flow Graph* (CFG) represents the possible sequential execution orderings of each statement in a program
- Data flow influences control flow, so this graph is not enough

Control Dependence Graph (CDG)

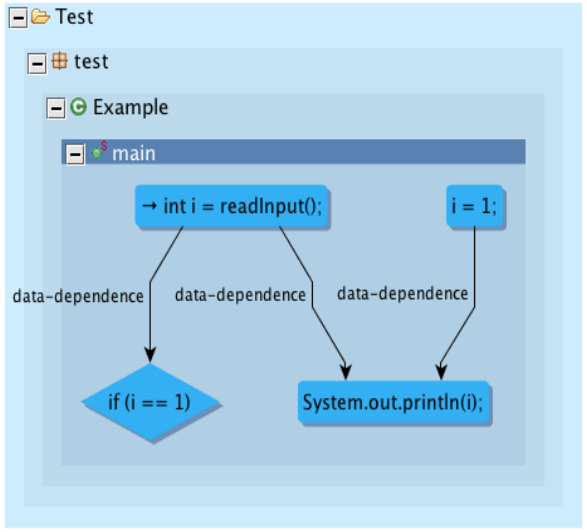
- If a statement X determines whether a statement Y can be executed then statement Y is *control dependent* on X
- Control dependence exists between two statements, if a statement directly controls the execution of the other statement [FOW87]



Program Dependence Graph (PDG)

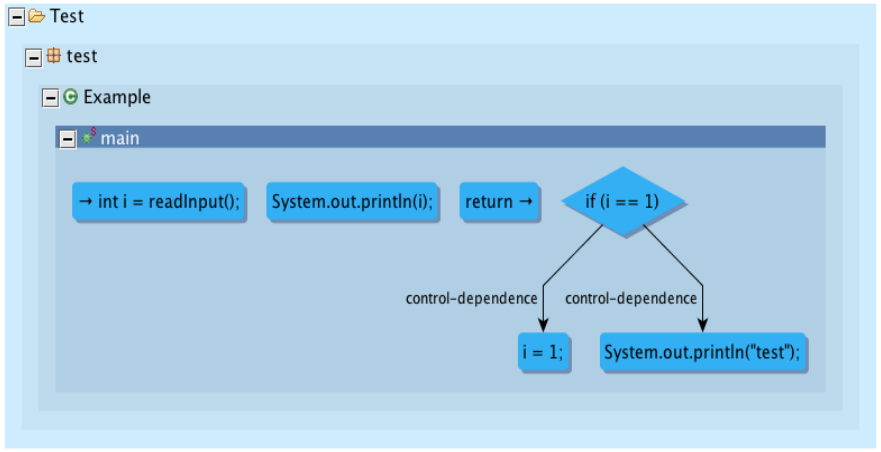
- Both DDG and CDG nodes are statements
- The union of a DDG and the CDG is a PDG

DDG



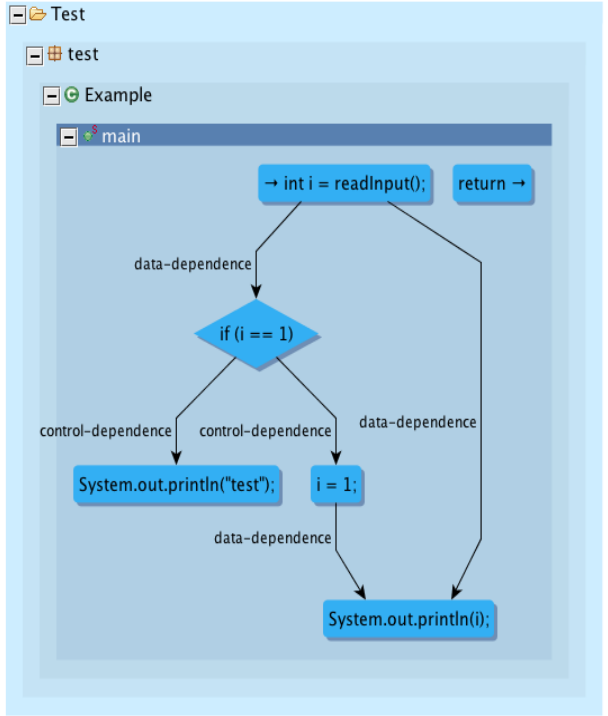
+

CDG



=

PDG



Program Slicing (Impact Analysis)

- Reverse Program Slice

  Answers: What statements does this statement's execution depend on?

- Forward Program Slice

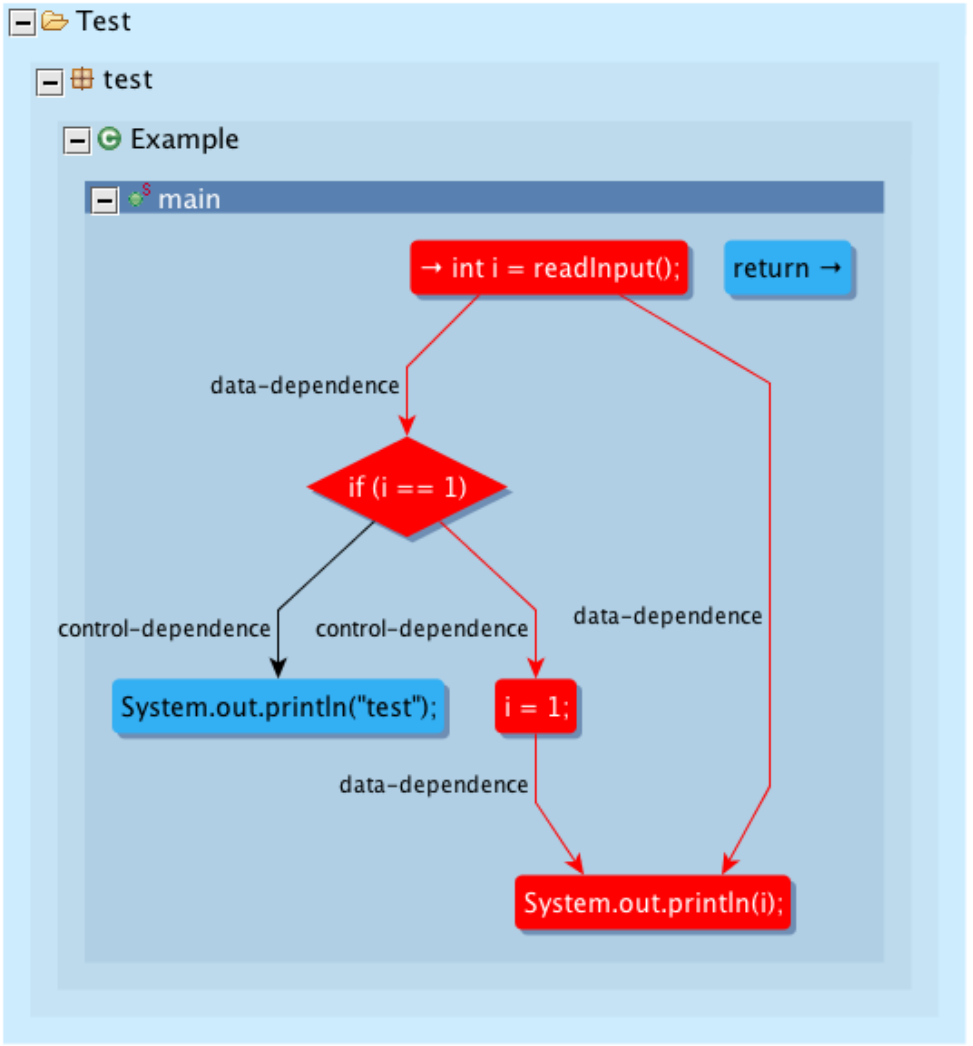
  Answers: What statements could execute as a result of this statement?

Example:

1. i = readInput();
2. if(i == 1)
3. print("test");

   else
4. i = 1;
5. print(i); ← detected failure
6. return; // terminate

Relevant lines:
1,2,4



Taint Analysis

How can we track the flow of data from the source (*x*) to the sink (*y*)?

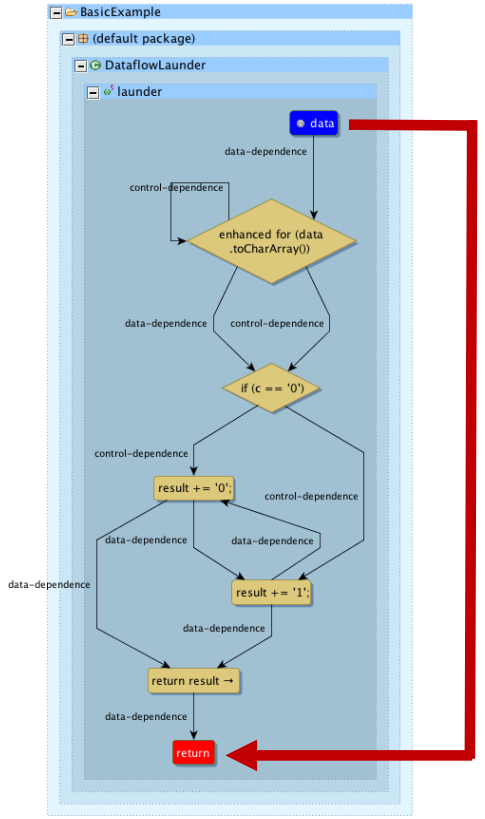
- Neither DFG/DDG nor CFG/CDG alone are enough to answer whether *x* flows to *y*
- Taint = (forward slice of *source)* intersection (reverse slice of *sink)*

```
public class DataflowLaunder {

        public static void main(String[] args) {
                String x = "1010";
                String y = launder(x);
                System.out.println(y + " is a laundered version of " + x);
        }

        public static String launder(String data){
           String result = "";
           for(char c : data.toCharArray()){
             if(c == '0')
                result += '0';
             else
                result += '1';
           }
           return result;
        }

}
```

DataflowLaunder.java

```java
/**
 * A toy example of laundering data through "implicit dataflow paths"
 * The launder method uses the input data to reconstruct a new result
 * with the same value as the original input.
 *
 * @author Ben Holland
 */
public class DataflowLaunder {

    public static void main(String[] args) {
        String x = "1010";
        String y = launder(x);
        System.out.println(y + " is a laundered version of " + x);
    }

    public static String launder(String data){
       String result = "";
       for(char c : data.toCharArray()){
         if(c == '0')
            result += '0';
         else
            result += '1';
       }
       return result;
    }

}
```

Taint Graph



Problems | Javadoc | Declaration | Console | Progress | Atlas Shell (shell) | Error Log | Analysis Keys | Element Detail View | Call Hierarchy

Atlas Shell (Project: shell)

```
var taint = new com.ensoftcorp.open.slice.analysis.TaintGraph(source, sink)

taint: com.ensoftcorp.open.slice.analysis.TaintGraph = com.ensoftcorp.open.slice.analysis.TaintGraph@13df7ce4

show(taint.getGraph(), taint.getHighlighter(), title="Taint Graph")
```

https://github.com/EnSoftCorp/slicing-toolbox

Evaluate: <type an expression or enter ":help" for more information>

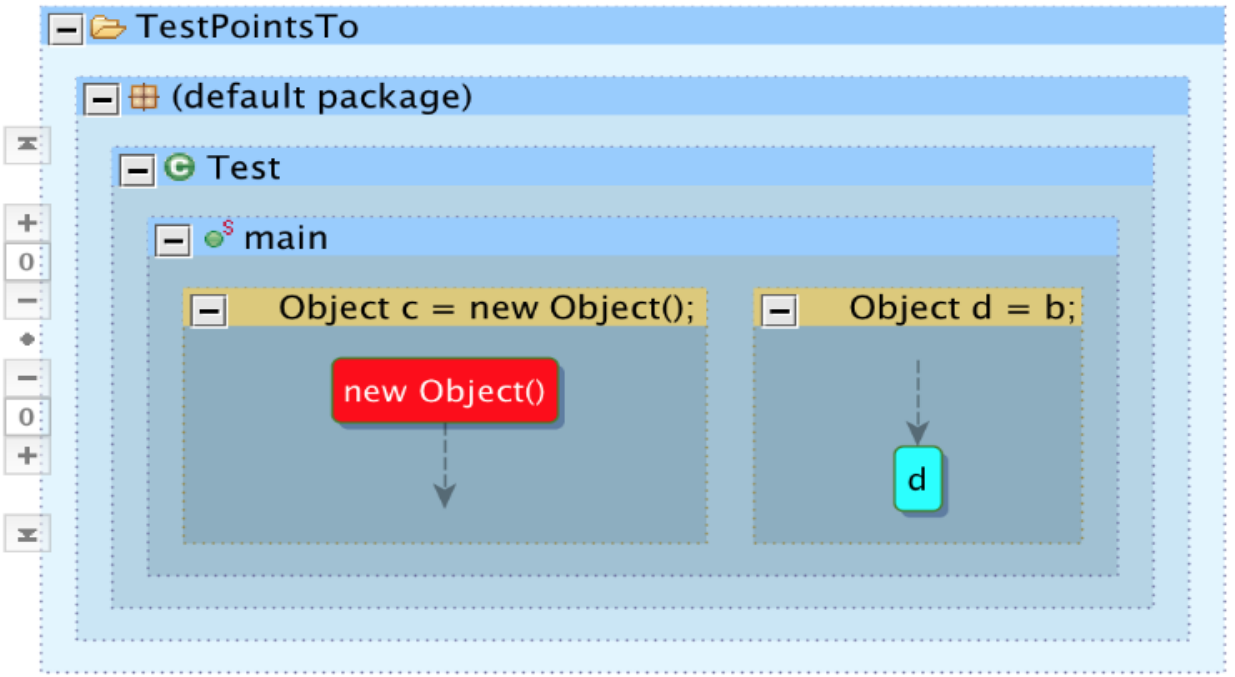
Analysis Woes: Going Inter-procedural

- Function pointers and dynamic dispatches force us to solve data flow problems to precisely identify inter-procedural control flows
- Class Hierarchy Analysis / Rapid Type Analysis
  - Cheap but *very* conservative results
- Points-to Analysis
  - A variable *v* points-to what data in memory? Knowing this we can more precisely resolve
  - Obtaining a perfect solution has been proven to reduce to solving the halting problem…
  - Expensive even for conservative results!!!
    - Each level of precision adds an exponent
    - Not really a lot to be gained (in my opinion)

Test.java

```
public class Test {

    public static Object evil;

    public static Object[] arr = new Object[2];

    public static void main(String[] args) {
        new Object();
        Object a = new Object();
        Object b = a;
        Object c = new Object();
        b = c;
        Object d = b;
        foo(c);
        evil =
        foo(a)

        ar
    }

    public s                    Object x){
        Syste                   (x);
        Object                  1];
    }

}
```

Points-to Aliases: Atlas Smart View



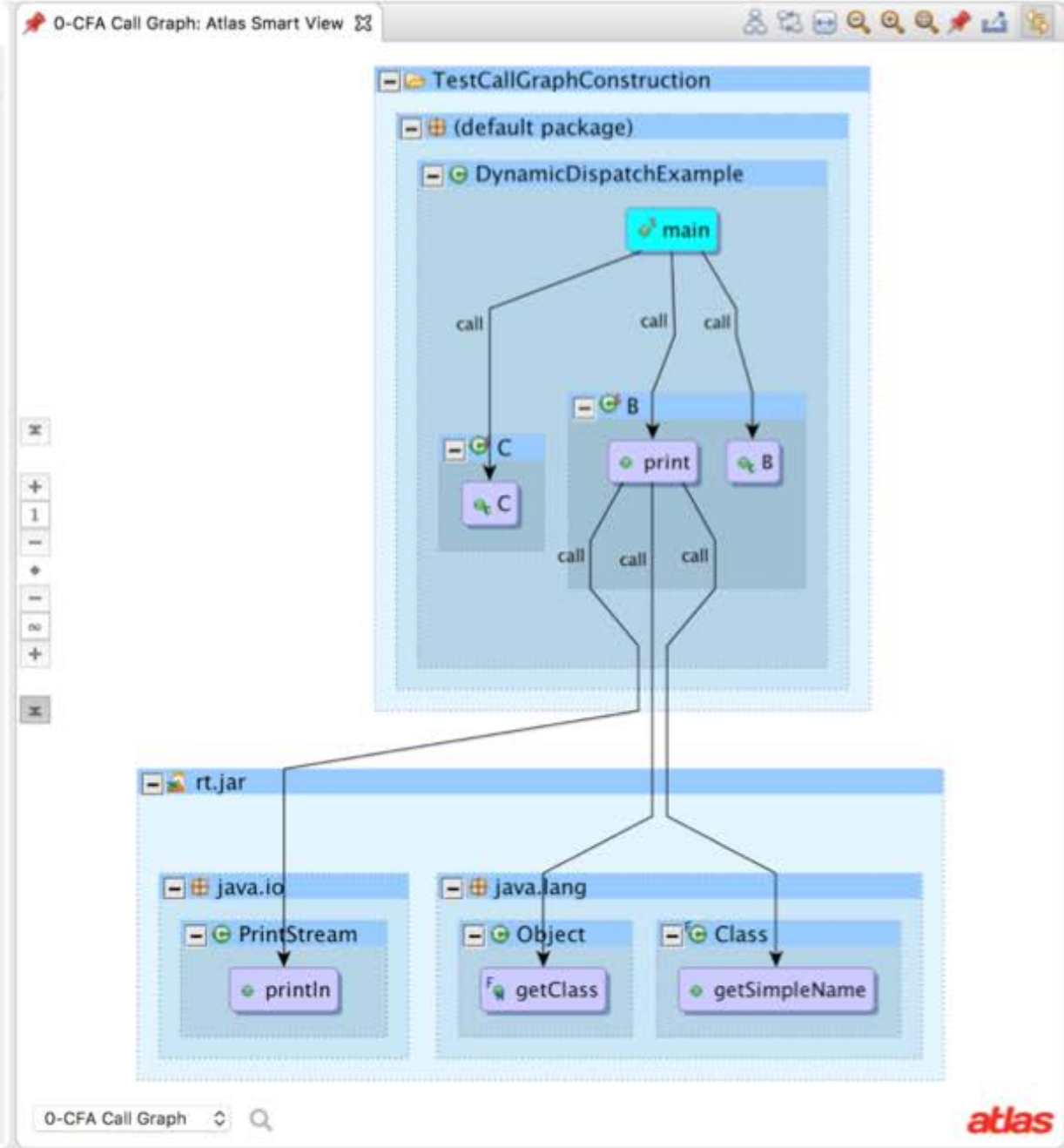
Points-to Aliases

atlas

https://github.com/EnSoftCorp/points-to-toolbox

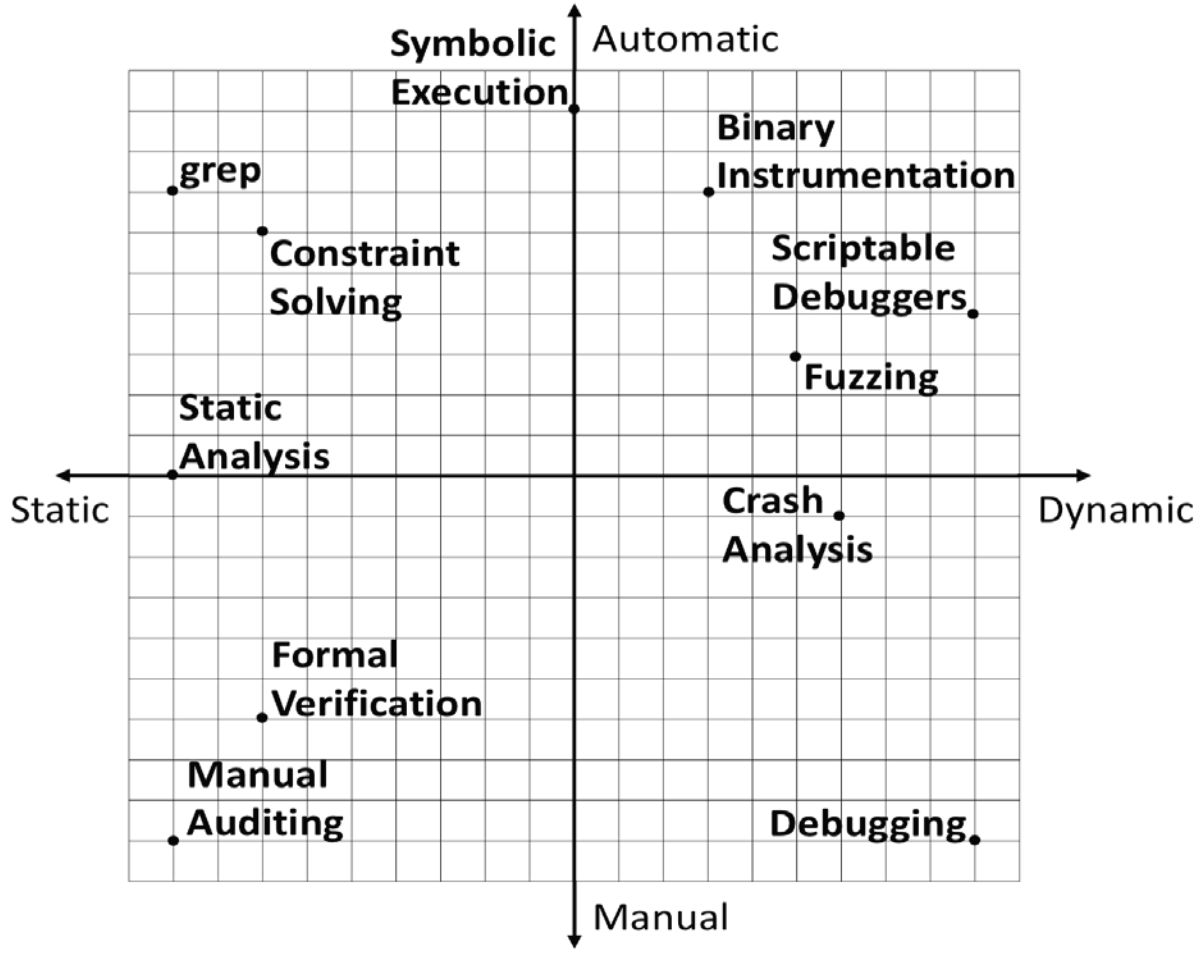
DynamicDispatchExample.java

```
public class DynamicDispatchExample {

    public static void main(String[] args){
        A b1 = new B();
        A c1 = new C();

        A b2 = b1;
        A c2 = c1;

        // what will ge
        b2.print(c2);
    }

    public static class A extends Object {
        public void print(A object) {
            System.out.println("An instance of " + object.getClass().getSimpleN
                    + " was passed to A's print(A object)");
        }
    }

    public static class B extends A {
        public void print(A object) {
            System.out.println("An instance of " + object.getClass().getSimpleN
                    + " was passed to B's print(A object)");
        }
    }

    public static class C extends B {
        public void print(A object) {
            System.out.println("An instance of " + object.getClass().getSimpleN
                    + " was passed to C's print(A object)");
        }
    }

    public static class D extends A {
        public void print(A object) {
            System.out.println("An instance of " + object.getClass().getSimpleN
                    + " was passed to D's print(A object)");
        }
    }
```



https://ensoftcorp.github.io/call-graph-toolbox

A Spectrum of Program Analysis Techniques



Source: Contemporary Automatic Program Analysis,
Julian Cohen, Blackhat 2014

Symbolic Execution

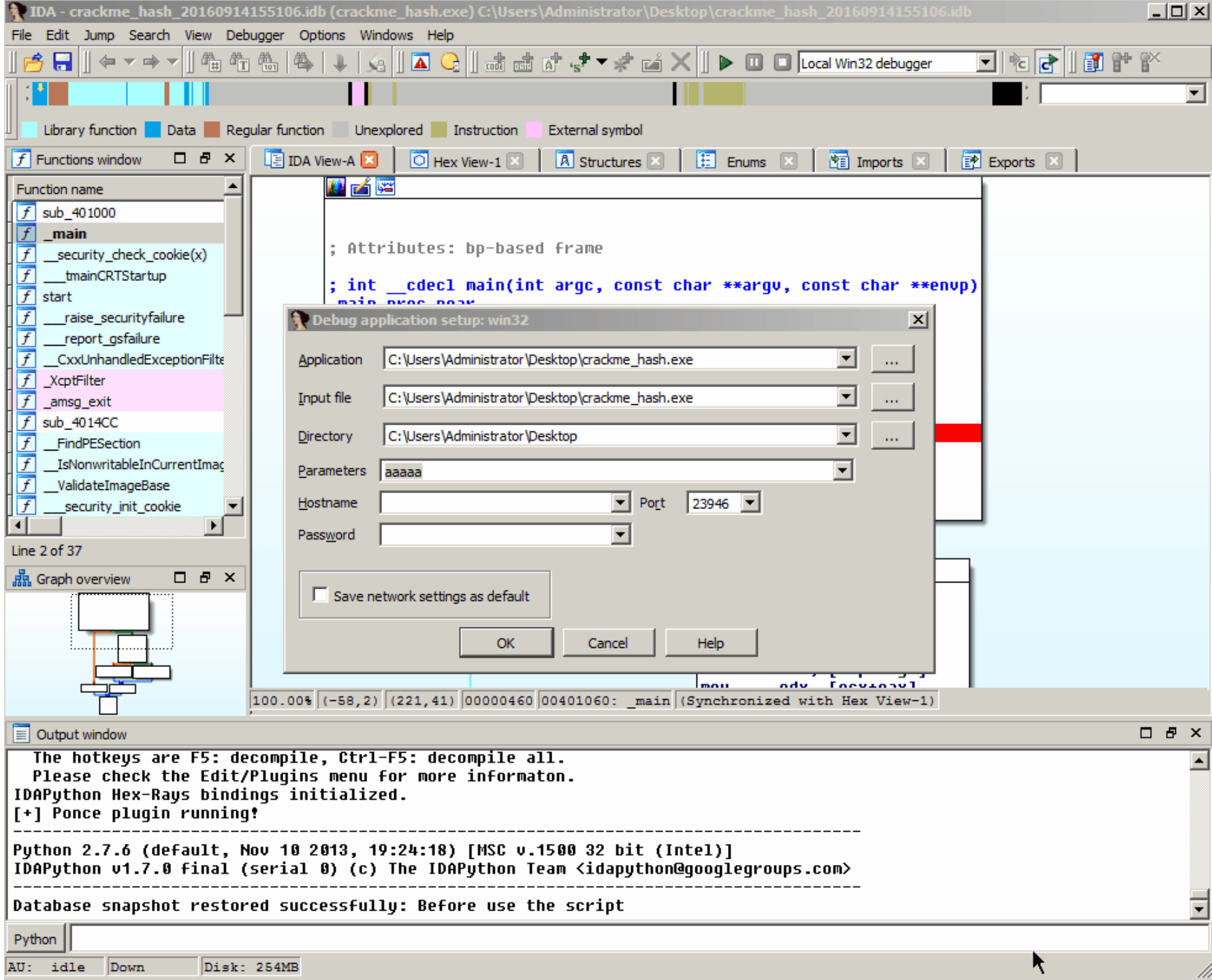
- Replace concrete assignment values with symbolic values
- Perform operations on symbolic values abstractly
- At each branch fork the abstracted logic
  - Dealing with path explosion problem is a challenge
- Utilize SAT/SMT solvers to determine if the constraints are satisfiable for a path of interest
  - Example: fail occurs if y * 2 = z = 12 is satisfiable
    - Solve(y * 2 = 12, y), y = 6 satisfies the constraint
    - Program crash occurs when read() returns 6

```
int f() {
  ...
  y = read();
  z = y * 2;
  if (z == 12) {
    crash();
  } else {
    printf("OK");
  }
}
```

On what inputs does the code crash?

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

char *serial = "\x31\x3e\x3d\x26\x31";

int check(char *ptr)
{
	int i;
	int hash = 0xABCD;

	for (i = 0; ptr[i]; i++)
		hash += ptr[i] ^ serial[i % 5];

	return hash;
}

int main(int ac, char **av)
{
	int ret;

	if (ac != 2)
		return -1;

	ret = check(av[1]);
	if (ret == 0xad6d)
		printf("Win\n");
	else
		printf("fail\n");

	return 0;
}
```

https://github.com/illera88/Ponce

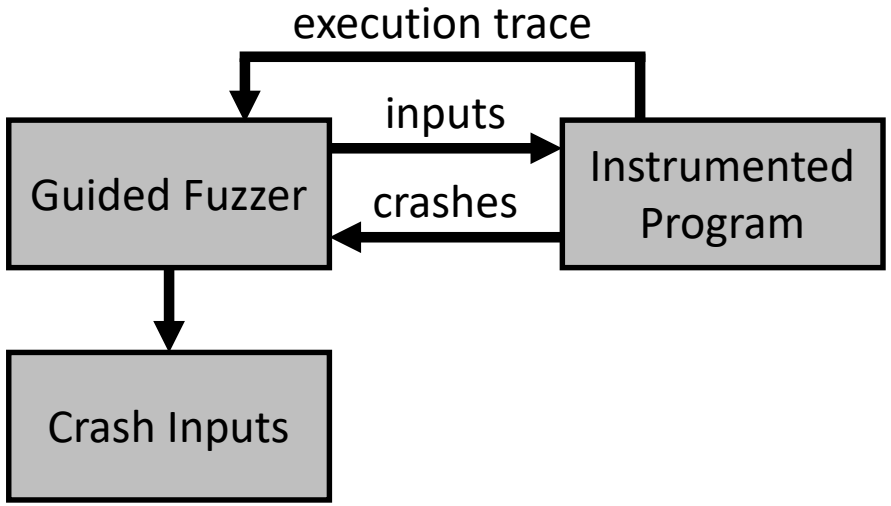


Concolic Execution

- A hybrid of dynamic analysis and symbolic execution
  - Perform symbolic execution on some variables and concrete execution on other variables
  - Symbolic variables could be made concrete in order to:
    - Move past symbolic limitations such as challenges in modeling the program environment (example network interaction)
    - Deal with path explosion problem and satisfiability problem by replacing difficult symbolic values with concrete values to simplify analysis
  - Pays cost in time for symbolic computations and execution time of program
- Several well known tools:
  - Angr - http://angr.io
  - KLEE - https://klee.github.io
  - DART - https://dl.acm.org/citation.cfm?id=1065036
  - CREST (formerly CUTE) - https://code.google.com/archive/p/crest
  - Microsoft SAGE - https://patricegodefroid.github.io/public_psfiles/ndss2008.pdf

Smart (Guided) Fuzzing

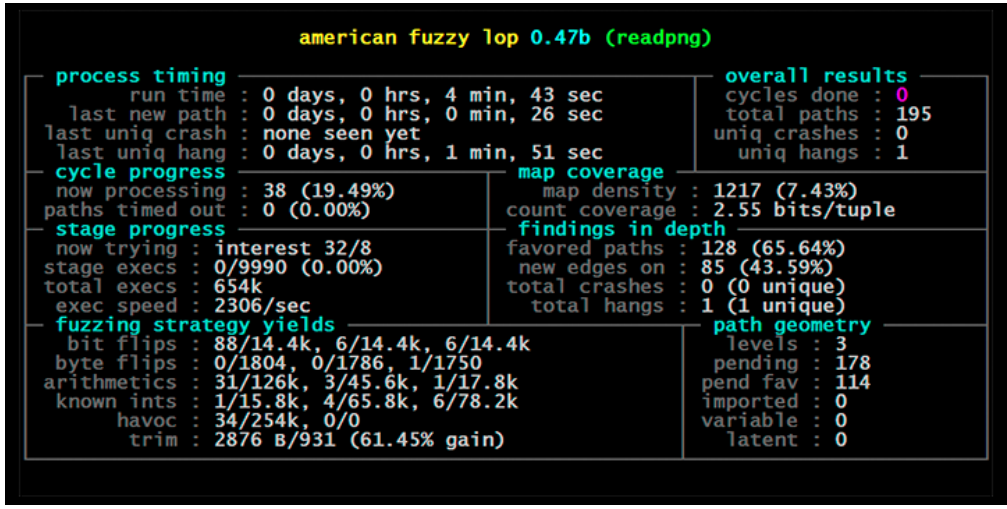
- Start with a test corpus of well formed program inputs
- Apply random or systematic mutations to program inputs
- Instrument the program branch points
- Run the instrumented program with mutated inputs and 1) observe whether or not the program crashes and 2) record the program execution path coverage
- If the input results in new program paths being explored then prioritize mutations of the tested input
- Repeat until the program crashes



Heuristics guide genetic algorithm to generate program inputs that push the fuzzer deeper into the program control flow, avoiding the common pitfalls of fuzzers to only test “shallow” code regions.

AFL (American Fuzzy Lop) Fuzzer

- Recognized as the current state of art implementation of guided fuzzing
  - Effective mutation strategy to generate new inputs from initial test corpus
  - Lightweight instrumentation at branch points
  - Genetic algorithm promotes mutations of inputs that discover new branch edges
    - Aims to explore all code paths
  - Huge trophy case of bugs found in wild
    - 371+ reported bugs in 161 different programs as of March 2018
    - http://lcamtuf.coredump.cx/afl/
- A game of economics. AFL tends to “guess” the correct input faster than a smart tool “computes” the correct input.



DARPA's Cyber Grand Challenge (CGC)

- "Cyber Grand Challenge (CGC) is a contest to build high-performance computers capable of playing in a Capture-the-Flag style cyber-security competition."

https://www.darpa.mil/program/cyber-grand-challenge

DARPA's Cyber Grand Challenge (CGC)

- Fully automatic reasoning to:
  - Detect program vulnerabilities
  - Patch programs to prevent exploitation
  - Develop and execute vulnerability exploits against competitors
- No human players!

CGC Results (Reading Between the Lines)

- All teams published the same essential combination of strategies
  - Guided fuzzing (nearly every team modified AFL)
  - Symbolic execution to assist fuzzer sometimes aided by classical program analyses (points-to, reachability, slicing, etc.)
  - Some state space pruning and prioritization scheme catered to expected vulnerability types
- Effective patches were more often generic patches which addressed the class of vulnerabilities not the one-off vulnerability that was given
  - Example: Adding stack guards for memory protection
- Competitor scores were close!
  - The difference between 1st and 7th place was not substantial
- Classes of vulnerabilities were known *a priori*

Context Matters!

- Head problems vs. hand problems
  - Design vs. implementation errors
- Implementation errors may be eventually largely eliminated with advances in programming languages, compilers, theorem provers, etc.
- Security design problems tend to be closely tied to failures in threat modeling, which is largely a human task
  - Many security problems arise due to reuse of software in changing contexts
    - Example: SCADA devices designed for isolated networks being placed on the internet

DARPA's High-Assurance Cyber Military Systems (HACMS) Program

- "formal methods-based approach to enable semi-automated code synthesis from executable, formal specifications"
- Creation of an "unhackable" drone!

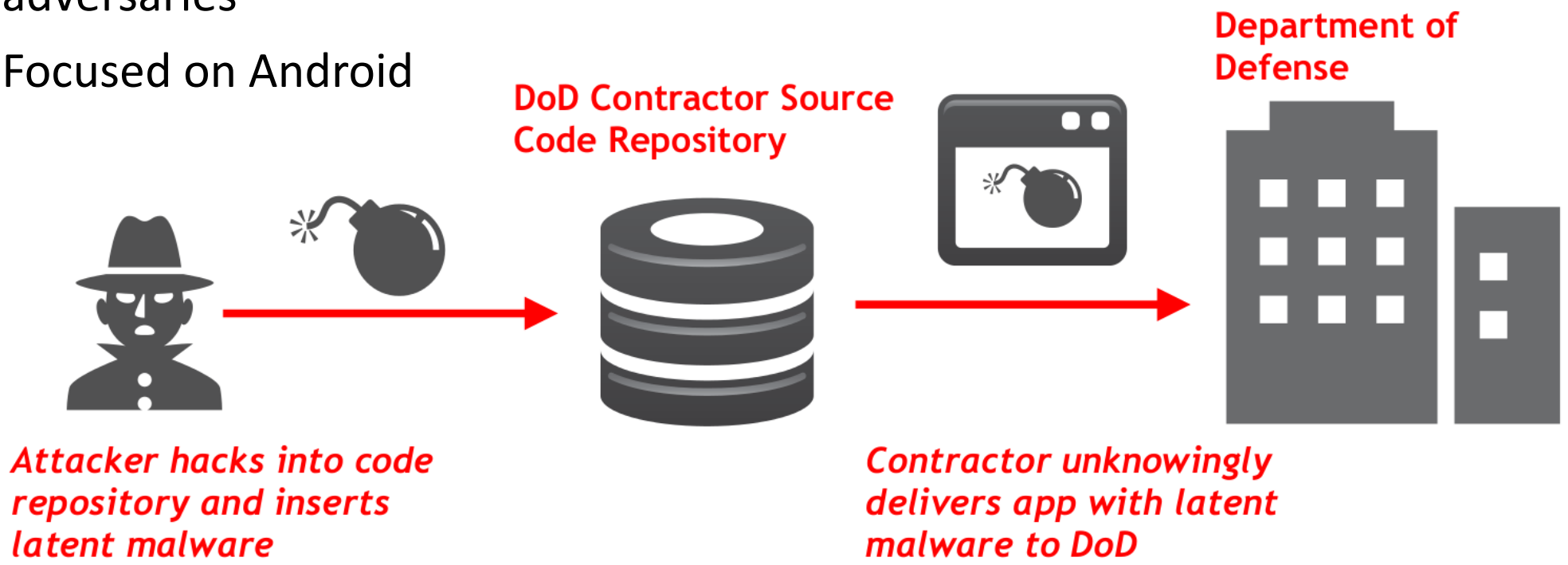
https://www.darpa.mil/program/high-assurance-cyber-military-systems

HACMS Results (Reading Between the Lines)

- Failures tended to stem back to human failures to fully account for the attack model
  - Example: Red Team debrief noted that Red team sent a radio reboot command that dropped that shut off drone engines for 3 seconds (enough to crash the drone). Blue's response was "Oh! We didn't think of that!"
- The "unhackable" drone produced by the program was not protected from the later discovered Meltdown and Spectre exploits

DARPA's APAC Program

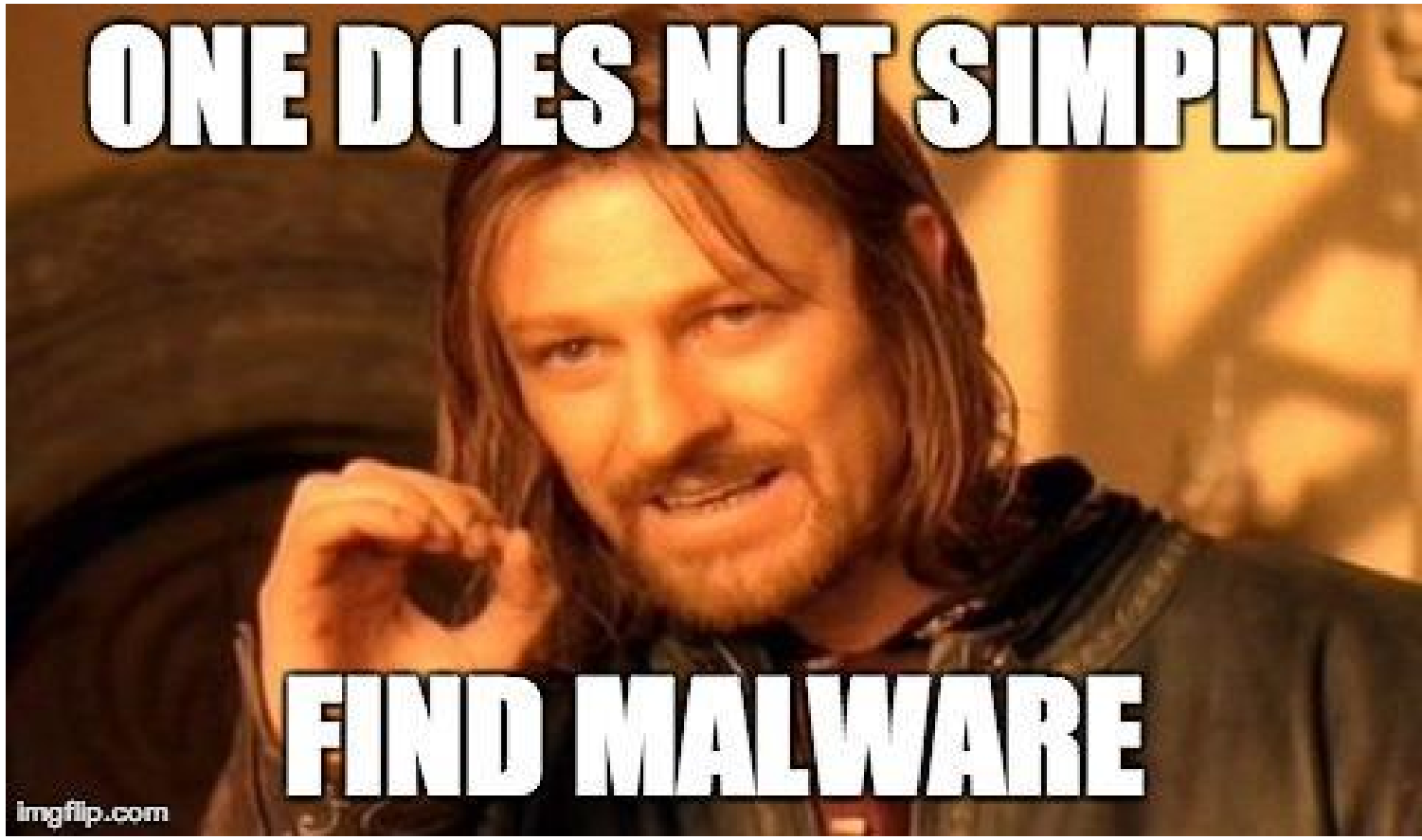
- Automated Program Analysis For Cybersecurity (APAC)
- Scenario: Hardened devices, internal app store, untrusted contractors, expert adversaries
- Focused on Android



https://www.darpa.mil/program/automated-program-analysis-for-cybersecurity

APAC Results (Reading Between the Lines)

- Is a bug malware? What if its planted intentionally? We know bug finding is hard…
- Bugs have plausible deniability and malicious intent cannot be determined from code.
- Novel attacks have escaped previous threat models.
- Need precision tools to detected **novel** and **sophisticated** malware in advance!



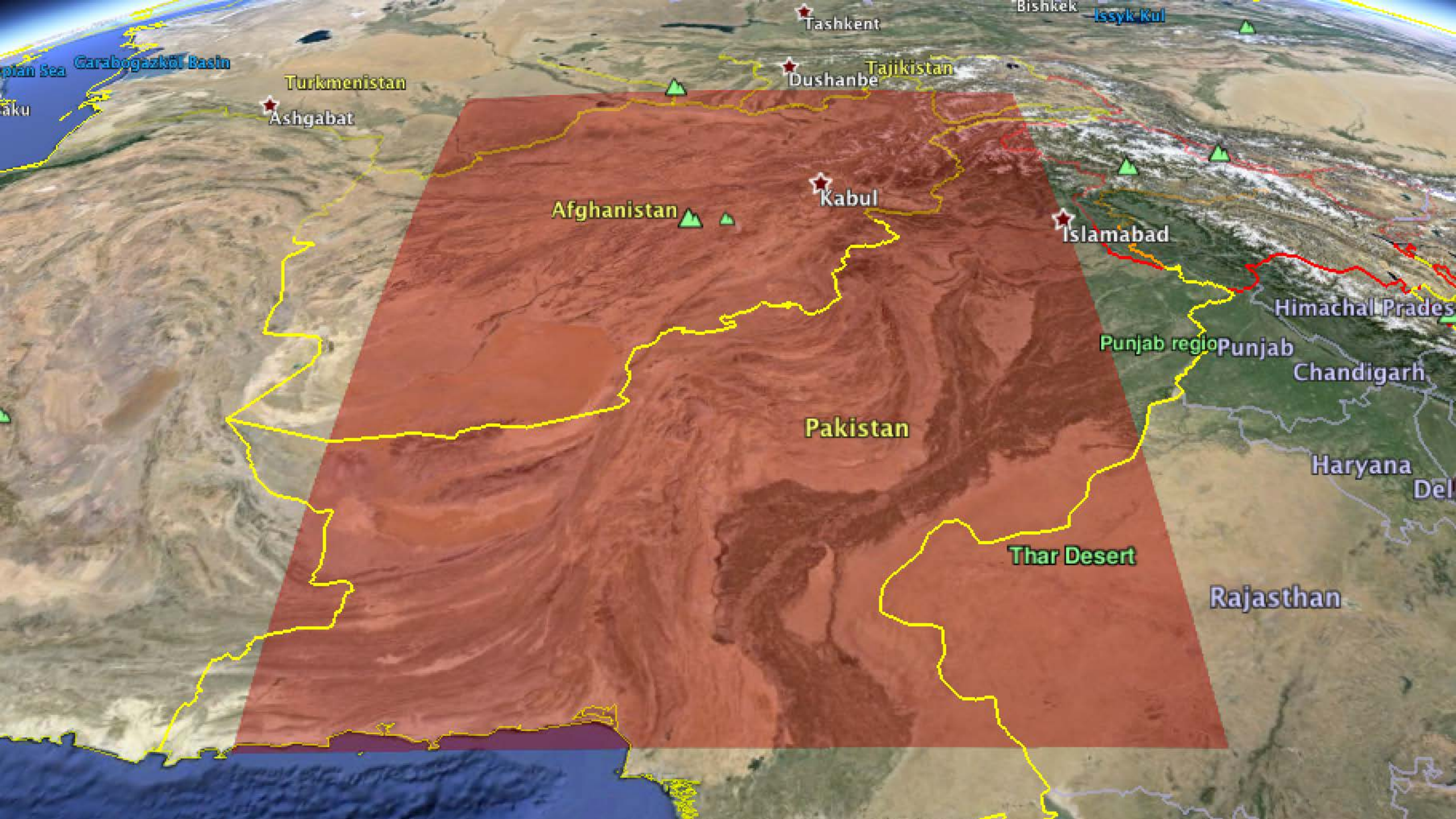
APAC Example: DARPA's Transformative Apps

- 55K lines of code
- Data gathering and relaying tool for military
  - Strategic mission planning/review
  - Audio and video recording
  - Geo-tagged camera snapshots
  - Real-time map updates based on GPS

APAC Example: DARPA's Transformative Apps

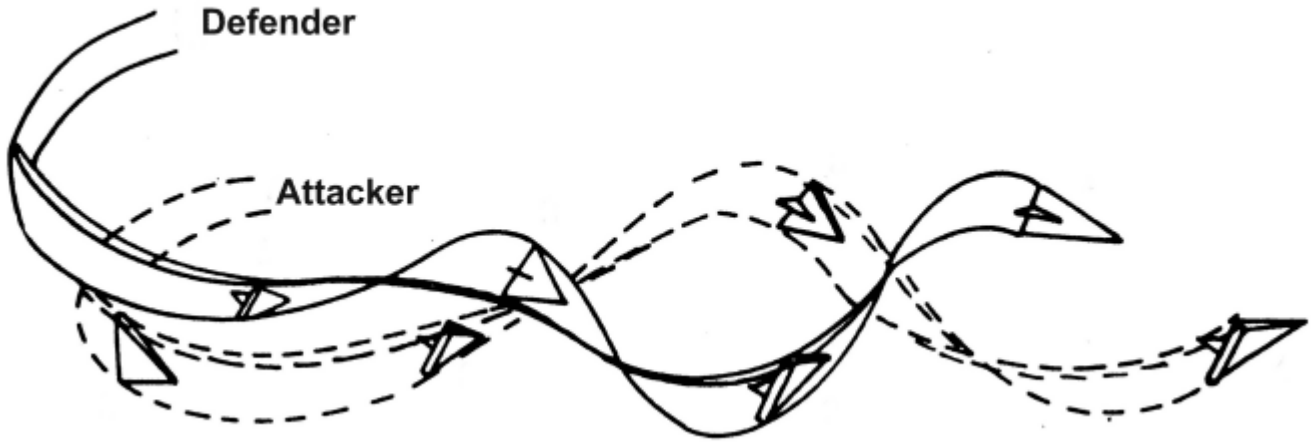
```
@Override
public void onLocationChanged(Location tmpLoc) {
    location = tmpLoc;
   double latitude = location.getLatitude();
   double longitude = location.getLongitude();
   if((longitude >= 62.45 && longitude <= 73.10) &&
       (latitude >= 25.14 && latitude <= 37.88)) {
        location.setLongitude(location.getLongitude() + 9.252);
        location.setLatitude(location.getLatitude() + 5.173);
   }
        …
 }
```

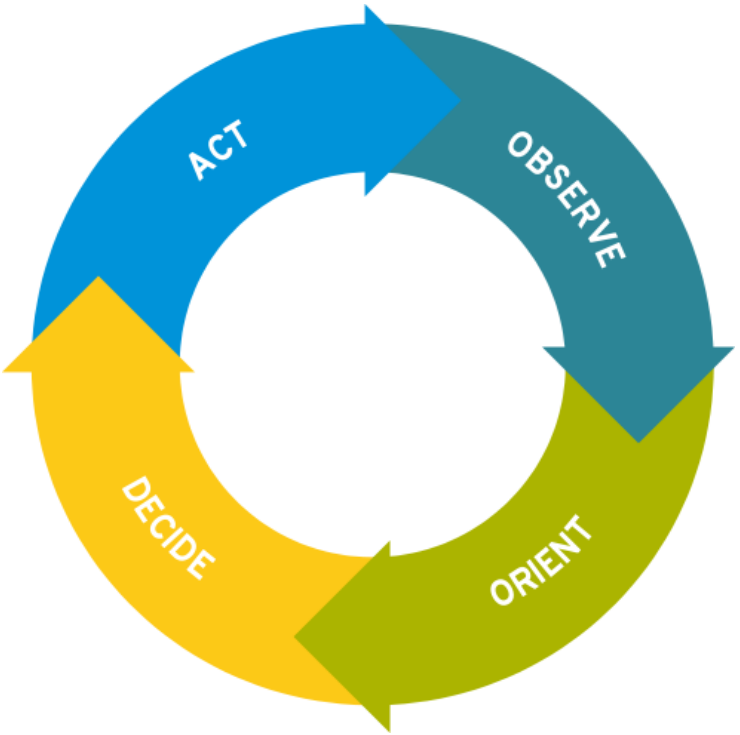
One of several locations were GPS coordinates were modified.
This corrupts GPS coordinates if user is in Afghanistan/Pakistan!


Tashkent
Bishkek
Issyk Kul
Garabogazkol Basin
Turkmenistan
Dushanbe
Tajikistan
Ashgabat
Kabul
Afghanistan
Islamabad
Himachal Prades
Punjab regio
Punjab
Chandigarh
Pakistan
Haryana
Thar Desert
Rajasthan

Program Analysis, OODA, and YOU

- “Security is a process, not a product” – Bruce Schneier





Program Analysis, OODA, and YOU

Our opponent

- Time
- Evolution of malware

"...IA > AI, that is, that intelligence amplifying systems can, at any given level of available systems technology, beat AI systems. That is, a machine and a mind can beat a mind-imitating machine working by itself." – Fred Brooks

Trend 2 – Employ Human Reasoning

- DARPA Programs on the verge
  - Computers and Humans Exploring Software Security (CHESS)
  - Explainable Artificial Intelligence (XAI)

- Let human’s do what humans are good at
- Let machines do what machines are good at
- Enable collaboration between the two

Your job in security is not going away soon…

Your job in security is not going away soon…

…probably.

(but it is an exciting field)

Activity: Does this program contain a vulnerability?

```
#include <stdio.h>
int main(int argc, char *argv) {
    char buf[64];
    strcpy(buf, argv[1]);
    return 0;
}
```

Activity: Does this program contain a vulnerability?

```
#include <stdio.h>
int main(int argc, char *argv) {
    char buf[64];
    a = read_user_input();
    b = read_user_input();
    …
    if(a * b == c){
      strcpy(buf, argv[1]);
    }
    return 0;
}
```

What if *c* is the product of two large primes?

Then yes, but only if you know the prime factorization…

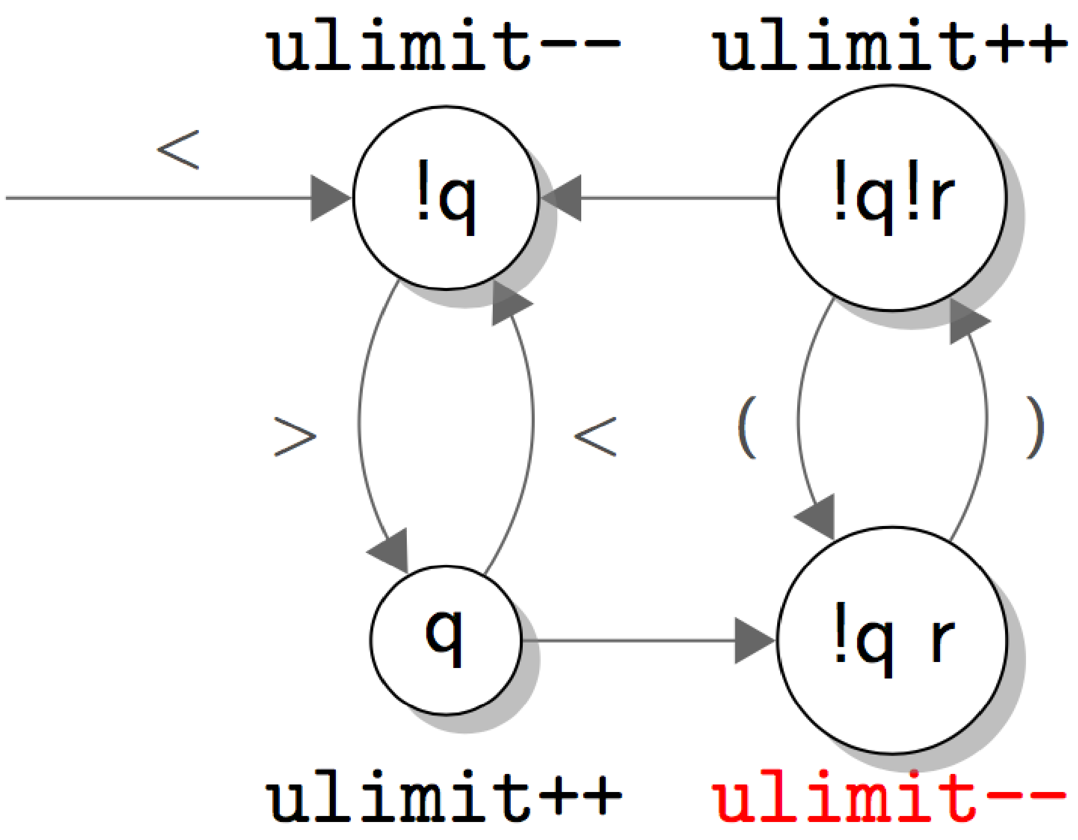
Activity: Does this program contain a vulnerability?

```
#define BUFFERSIZE 200
int copy_it (char* input , unsigned int length){
   char c, localbuf[BUFFERSIZE];
   unsigned int upperlimit = BUFFERSIZE - 10;
   unsigned int quotation = roundquote = FALSE;
   unsigned int input_index = output_index = 0;
   while (input_index < length){ c = input[input_index++];
      if((c == '<') && (!quotation)){ quotation = true; upperlimit--; }
      if((c == '>') && (quotation)){ quotation = false; upperlimit++; }
      if((c == '(') && (!quotation) && (!roundquote)){ roundquote = true; }
      if((c == ')') && (!quotation) && (roundquote)){ roundquote = false; upperlimit++; }
      // if there is sufficient space in the buffer, write the character
      if(output_index < upperlimit){ localbuf[output_index] = c; output_index++; }
   }
   if(roundquote){ localbuf[output_index] = ')'; output_index++; }
   if(quotation){ localbuf[output_index] = '>'; output_index++; }
   return output_index;
}
```

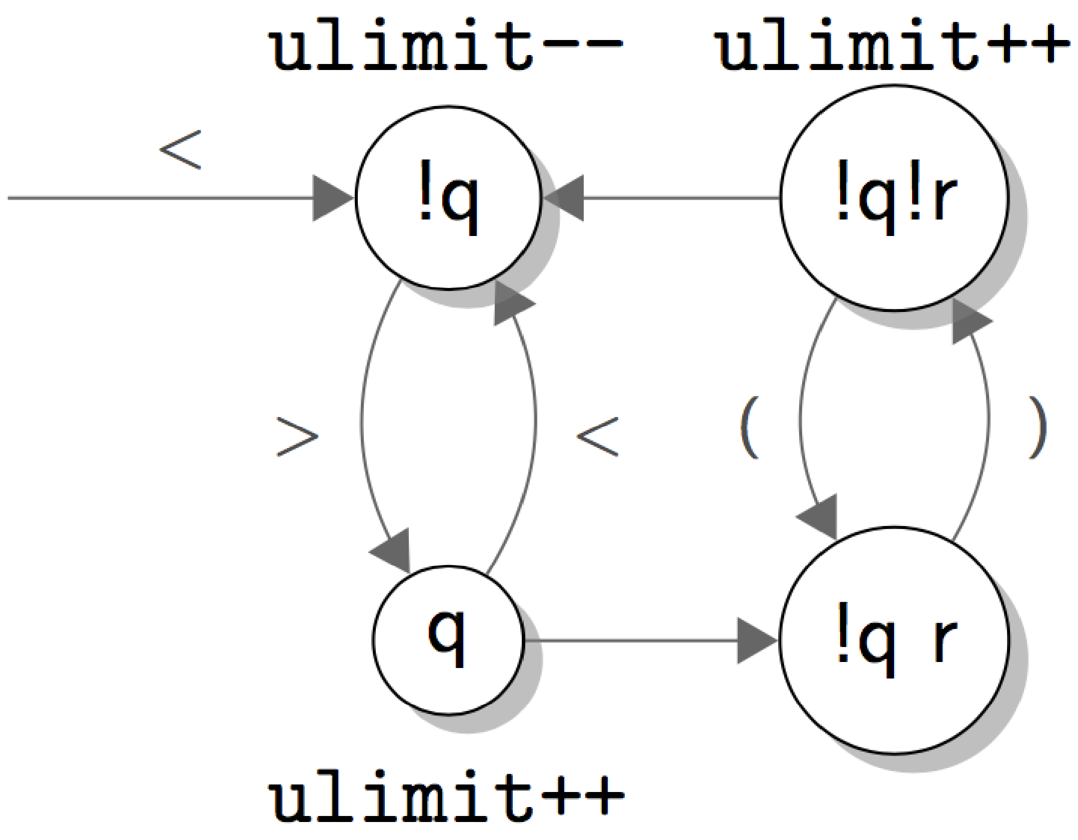
Activity: Does this program contain a vulnerability?

```
#define BUFFERSIZE 200
int copy_it (char* input , unsigned int length){
   char c, localbuf[BUFFERSIZE];
   unsigned int upperlimit = BUFFERSIZE - 10;
   unsigned int quotation = roundquote = FALSE;
   unsigned int input_index = output_index = 0;
   while (input_index < length){ c = input[input_index++];
      if((c == '<') && (!quotation)){ quotation = true; upperlimit--; }
      if((c == '>') && (quotation)){ quotation = false; upperlimit++; }
      if((c == '(') && (!quotation) && (!roundquote)){ roundquote = true; /* (missing) upperlimit--; */ }
      if((c == ')') && (!quotation) && (roundquote)){ roundquote = false; upperlimit++; }
      // if there is sufficient space in the buffer, write the character
      if(output_index < upperlimit){ localbuf[output_index] = c; output_index++; }
   }
   if(roundquote){ localbuf[output_index] = ')'; output_index++; }
   if(quotation){ localbuf[output_index] = '>'; output_index++; }
   return output_index;
}
```

Good:



Bad:



input = "Name Lastname < name@mail.org >
()
()"

Source: Cracking Sendmail crackaddr Still a challenge for automated program analysis?

Activity: Does this program contain a vulnerability?

```
int main(int argc, char *argv[]) {
  int64_t x = strtoll(argv[1], NULL, 10);
  char buf[64];
  if (x <= 2 || (x & 1) != 0)
    return 1;
  int64_t i;
  for (i = x; i > 0; i--)
    if (foo(i) && foo(x - i))
      return 1;
  strcpy(buf, argv[2]); // reachable?
}
```

```
int foo(int64_t x) {
  int64_t i, s;
  for (i = x – 1; i >= 2; i--)
    for (s = x; s >= 0; s -= i)
      if (s == 0)
        return FALSE;
  return TRUE;
}
```

Activity: Does this program contain a vulnerability?

```
int main(int argc, char *argv[]) {
  int64_t x = strtoll(argv[1], NULL, 10);
  char buf[64];
  // x is an even number that is greater than 2
  if (x <= 2 || (x & 1) != 0)
    return 1;
  // x can be expressed as the sum of 2 primes
  int64_t i;
  for (i = x; i > 0; i--)
    if (is_prime(i) && is_prime(x - i))
      return 1;
  strcpy(buf, argv[2]); // reachable?
}
```

```
int is_prime(int64_t x) {
  int64_t i, s;
  for (i = x – 1; i >= 2; i--)
    for (s = x; s >= 0; s -= i)
      if (s == 0)
        return FALSE;
  return TRUE;
}
```

Answer: No for all 32 bit integers, but unknown for all 64bit integers…

Goldbach's conjecture:
275+ year old unsolved math problem

Thanks!

- Questions?

ben-holland.com