Vulnerability Analysis (IV): Program Verification

Slide credit: Vijay D’Silva
Interval Analysis: Example

define foo(unsigned input) {
    if (input < UINT_MAX - 2) {
        unsigned len, s;
        char* buf;
        len = input + 3;
        if (len < 10)
            s = len;
        else if (len & 2 == 0)
            s = len;
        else {
            assert(len < UINT_MAX - 1);
            s = len + 2;
        }
        buf = malloc(s);
        read(fd, buf, len);
        ....
    }
}
Interval Analysis: Example

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

#include "stdint.h"

uint16_t input;

int main() {
  input = 10;

  unsigned len, s;
  char* buf;
  len = input + 3;
  if (len < 10) {
    s = len;
  }
  else {
    s = len + 2;
  }

  buf = malloc(s);
  read(fd, buf, len);
  free(buf);
  return 0;
}
```
unsigned len, s;
char* buf;
len = input + 3;
if (len < 10)
    s = len;
else
    s = len + 2;
assert(len < UINT_MAX - 1);
buf = malloc(s);
read(fd, buf, len);
A transformer (or transfer function) is
- a function on a lattice
- that respects the order (monotone)

Transformers
- abstract the effect of program statements
- may lose precision
Quiz: Sign Analysis Transformers

Which of the following is the right transformer for $x = x - 1$?

C

Answer:
Quiz: Sign Analysis Transformers

Which of the statements below is best represented by this transformer?

- $x = x - 2$
- if $(x < -4)$
- $x = -4$
- if $(x > -4)$

Answer: $x = -4$
<table>
<thead>
<tr>
<th></th>
<th>Analysis Frameworks</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Lattices</td>
</tr>
<tr>
<td>b</td>
<td>Transformers</td>
</tr>
<tr>
<td>c</td>
<td>Systems of Equations</td>
</tr>
<tr>
<td>d</td>
<td>Solving Equations</td>
</tr>
</tbody>
</table>
Dawn Song

Programs to Equations

Programs
• convenient to write
• difficult to analyze: datatypes, loops, branches, etc.

Systems of equations
• well-studied in mathematics
• simple compared to programs: expressions and equalities
Example Static Analysis Equations

\[
\begin{align*}
  d_{\text{out}1} &= f_1(d_{\text{Entry}}) \\
  d_{\text{out}2-f} &= f_2-f(d_{\text{out}1} \sqcup d_{\text{out}6}) \\
  d_{\text{out}2-t} &= f_2-f(d_{\text{out}1} \sqcup d_{\text{out}6}) \\
  d_{\text{out}3} &= f_3(d_{\text{out}2-f}) \\
  d_{\text{out}4} &= f_4(d_{\text{out}2-t}) \\
  d_{\text{Exit}} &= f_5-f(d_{\text{out}3} \sqcup d_{\text{out}4}) \\
  d_{\text{out}5-t} &= f_5-f(d_{\text{out}3} \sqcup d_{\text{out}4}) \\
  d_{\text{Err}} &= f_6-t(d_{\text{out}5-t}) \\
  d_{\text{out}6-f} &= f_6-f(d_{\text{out}5-t}) 
\end{align*}
\]
Example Static Analysis Equations

Variables represent facts at different program points

\[
\begin{align*}
d_{\text{out1}} & = f_1(d_{\text{Entry}}) \\
d_{\text{out2-f}} & = f_{2-f}(d_{\text{out1}} \sqcup d_{\text{out6}}) \\
d_{\text{out2-t}} & = f_{2-t}(d_{\text{out1}} \sqcup d_{\text{out6}}) \\
d_{\text{out3}} & = f_3(d_{\text{out2-f}}) \\
d_{\text{out4}} & = f_4(d_{\text{out2-t}}) \\
d_{\text{Exit}} & = f_{5-f}(d_{\text{out3}} \sqcup d_{\text{out4}}) \\
d_{\text{out5-t}} & = f_{5-t}(d_{\text{out3}} \sqcup d_{\text{out4}}) \\
d_{\text{Err}} & = f_{6-t}(d_{\text{out5-t}}) \\
d_{\text{out6-f}} & = f_{6-f}(d_{\text{out5-t}})
\end{align*}
\]
Example Static Analysis Equations

\[
\begin{align*}
\text{d}_{\text{out}1} &= f_1(d_{\text{Entry}}) \\
\text{d}_{\text{out}2-f} &= f_2-f(d_{\text{out}1} \sqcup d_{\text{out}6}) \\
\text{d}_{\text{out}2-t} &= f_2-f(d_{\text{out}1} \sqcup d_{\text{out}6}) \\
\text{d}_{\text{out}3} &= f_3(d_{\text{out}2-f}) \\
\text{d}_{\text{out}4} &= f_4(d_{\text{out}2-t}) \\
\text{d}_{\text{Exit}} &= f_5-f(d_{\text{out}3} \sqcup d_{\text{out}4}) \\
\text{d}_{\text{out}5-t} &= f_5-t(d_{\text{out}3} \sqcup d_{\text{out}4}) \\
\text{d}_{\text{Err}} &= f_6-t(d_{\text{out}5-t}) \\
\text{d}_{\text{out}6-f} &= f_6-f(d_{\text{out}5-t})
\end{align*}
\]

Variables represent facts at different program points

- \text{Err} is an analysis domain (true, true)
Example Static Analysis Equations

Expressions represent how data is transformed.
Example Static Analysis Equations

An equation relates the facts flowing in and out of a basic block.
A static analysis equation is a set of equalities of the form

\[
\begin{align*}
    d_1 & = \text{exp}_1(d_1, \ldots, d_k) \\
    \ldots & = \ldots \\
    d_k & = \text{exp}_k(d_1, \ldots, d_k)
\end{align*}
\]

- **variables** $d_i$ represent facts flowing in and out of basic blocks
- **expressions** $\text{exp}_i(d_1, \ldots, d_k)$
  - describe how data is transformed
  - are composed of variables, transfer functions, meet, join
  - define how the data flows between software components
Equations for a Single Statement

The relationship between facts that are true at different points in a program can be encoded as an equation.

\[ x \geq 0 \]

\[ x = x + 1; \]

\[ x > 0 \]

\[ d_{\text{in}} \rightarrow f \rightarrow d_{\text{out}} \]

\[ d_{\text{out}} = f(d_{\text{in}}) \]
Sequential composition applies the function in one equation to the result of a previous equation.

\[
\begin{align*}
\text{if } (x > 5) \\
\quad x &= x - 1; \\
\text{if } x \geq 0 \\
\quad x &= x > 0 \\
\text{if } (x > 5) \\
\quad x &= x > 0
\end{align*}
\]
The relationship between facts that are true at different points in a program can be encoded as an equation.

\[ \begin{align*}
    d_{out1} &= f_1(d_{in1}) \\
    d_{out2} &= f_2(d_{in2}) \\
    d_{join} &= d_{out1} \cup d_{out2} \\
    d_{out3} &= f_3(d_{join})
\end{align*} \]
Equations at Join Points

The relationship between facts that are true at different points in a program can be encoded as an equation.

\[
\begin{align*}
  d_{\text{out1}} &= f_1(d_{\text{in1}}) \\
  d_{\text{out2}} &= f_2(d_{\text{in2}}) \\
  d_{\text{join}} &= d_{\text{out1}} \sqcup d_{\text{out2}} \\
  d_{\text{out3}} &= f_3(d_{\text{join}})
\end{align*}
\]
Simplifying Equations

It is common to simplify equations by eliminating variables related by equalities.

\[
\begin{align*}
\text{d}_{\text{out1}} &= f_1(\text{d}_{\text{in1}}) \\
\text{d}_{\text{out2}} &= f_2(\text{d}_{\text{in2}}) \\
\text{d}_{\text{out3}} &= f_3(\text{d}_{\text{join}}) \\
\text{d}_{\text{join}} &= \text{d}_{\text{out2}} \\
\text{d}_{\text{out1}} &= f_1(\text{d}_{\text{in1}}) \\
\text{d}_{\text{out2}} &= f_2(\text{d}_{\text{in2}}) \\
\text{d}_{\text{out3}} &= f_3(\text{d}_{\text{out1}} \sqcup \text{d}_{\text{out2}})
\end{align*}
\]
Why Equations?

Several properties of equations are well studied

• Existence of solutions
• How to compute solutions when they exist
• How to approximate solutions if finding exact solutions is too difficult

By using equations, program analysis reduces to a well known problem and existing intuition and techniques can be applied
<table>
<thead>
<tr>
<th></th>
<th>Analysis Frameworks</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Lattices</td>
</tr>
<tr>
<td>b</td>
<td>Transformers</td>
</tr>
<tr>
<td>c</td>
<td>Systems of Equations</td>
</tr>
<tr>
<td>d</td>
<td>Solving Equations</td>
</tr>
</tbody>
</table>
A solution to the equations is a mapping of variables to lattice elements such that the equations are satisfied.

- Does a solution exist?
- If it exists, how can we find it?
A solution to the equations is a mapping of variables to lattice elements such that the equations are satisfied.

- Does a solution exist?
- If it exists, how can we find it?
A fixed point of a function is an element satisfying
\[ x = \exp(x) \]
This is an equation and a fixed point is a solution to an equation.

A solution to the equations is a mapping of variables to lattice elements such that the equations are satisfied.

- Does a solution exist?
- If it exists, how can we find it?
How to Solve Equations

Solving equations by iteration:
• Start from least element
• Apply transformers once: exp(x)
• Update all variables
• Apply transformers again: exp(exp(x))
• Repeat until no variables change

Issues
• wasteful updates to variables
• termination of the iteration
• termination in reasonable time
Iteration Strategies

- **Round robin**: Update equations in an apriori fixed order.
- **Topological order**: Update equations following the structure of the CFG.
- **Chaotic Iteration**: Update equations in arbitrary order making sure all are eventually updated.

Many more advanced strategies exist.
Consider this program. Some questions that we can ask a program analyzer are:

- Is it possible to violate the assertion?
- What sequence of inputs leads to an assertion violation?
Control Flow Graphs are representations of programs used in program analyzers. The graph structure makes control flow in a program explicit.

```c
int max = getchar();
if (max == EOF)
  exit(0);
c = getchar();
while (c != EOF)
{
  assert(c < max);
c = getchar();
}
```
Control Flow Unwinding

Unwinding of a control flow graph is a possibly infinite tree containing every path in the graph.
An execution corresponds to a path in the tree unwinding.

Multiple executions can traverse the same path.

A path is feasible if there is an execution that traverses it.
The question of whether an assertion violation exists is equivalent to asking if one of the paths to an error location is feasible.
The question of whether an assertion violation exists is equivalent to asking if one of the paths to an error location is feasible.

Vulnerability detection techniques attempt to find if one such feasible path exists.
Fuzzing techniques feed inputs to the system and try to trigger a crash.
Fuzzing

Fuzzing techniques feed inputs to the system and try to trigger a crash. Main questions in fuzzing
• How to generate inputs?
• How to feed inputs to the system?

```c
int max = getchar();
if (max == EOF)
    c = getchar();
if (c != EOF)
    if (c < max)
        c = getchar();
    Err
Exit
```

```c
if (c != EOF)
    if (c < max)
        c = getchar();
    Err
Exit
```

```c
if (c != EOF)
    if (c < max)
        c = getchar();
    Err
Exit
```

```c
if (c != EOF)
    if (c < max)
        c = getchar();
    Err
Exit
```
Fuzzing

Fuzzing techniques feed inputs to the system and try to trigger a crash. Main questions in fuzzing
• How to generate inputs?
• How to feed inputs to the system?
Fuzzing techniques feed inputs to the system and try to trigger a crash. Main questions in fuzzing:

- How to generate inputs?
- How to feed inputs to the system?

Goal: Maximize the likelihood that a set of inputs trigger an error.
Symbolic execution uses techniques from logic to avoid exploring the same path multiple times.
Symbolic execution uses techniques from logic to avoid exploring the same path multiple times.
Symbolic Execution

Symbolic execution uses techniques from logic to avoid exploring the same path multiple times.

The highlighted path is feasible exactly if a certain formula is satisfiable.

\[
\text{max} == \text{getchar()} \\
\&\& \text{max} != \text{EOF} \\
\&\& \text{c} == \text{getchar()} \\
\&\& \text{c} != \text{EOF} \\
\&\& \text{c} >= \text{max}
\]
Static analysis techniques do not execute the program. They use approximations to explore multiple paths simultaneously.

```c
int max = getchar();
if (max == EOF)
  c = getchar();
if (c != EOF)
  if (c < max)
    c = getchar();
  Exit
Err
if (c != EOF)
  if (c < max)
    c = getchar();
  Exit
Err
if (c != EOF)
  if (c < max)
    c = getchar();
  Exit
Err
if (c != EOF)
  if (c < max)
    c = getchar();
  Exit
Err
```
Static Analysis

Static analysis techniques do not execute the program. They use approximations to explore multiple paths simultaneously.

```c
int max = getchar();
if (max == EOF)
    c = getchar();
if (c != EOF)
    if (c < max)
        c = getchar();
```
Static Analysis

Static analysis techniques do not execute the program. They use approximations to explore multiple paths simultaneously.

```c
int max = getchar();
if (max == EOF)
    c = getchar();
if (c != EOF)
    if (c < max)
        c = getchar();
```
Static analysis techniques do not execute the program. They use approximations to explore multiple paths simultaneously.
Sequence of States vs. Executions

```c
int a[5];
for (int i=0; i<5; ++i)
a[i] = 0;
```
All Sequences of States vs. All Executions
All Sequences of States vs. All Executions

Execution leading to an error

Program

1 2 1 3
1 2 1 2 1 3
1 2 1 2 1
1 2 1
3 3 2
1 2 1
3

All Executions

All State Sequences
An underapproximation contains some but not all executions. Underapproximate analysis may conclude there is no error when an error exists: a false negative.
An *underapproximation* contains some but not all executions. Underapproximate analysis may conclude there is no error when an error exists: a *false negative*. A better underapproximation considers *more* executions.
An overapproximation contains sequences that are not executions.
An overapproximation contains sequences that are not executions. Overapproximate analysis may conclude there is an error when no error exists: a false positive or false alarm.
An overapproximation contains sequences that are not executions. Overapproximate analysis may conclude there is an error when no error exists: a false positive or false alarm. A more precise overapproximation considers fewer sequences that are not executions.
## Soundness andCompleteness

<table>
<thead>
<tr>
<th>Property</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Soundness</td>
<td>If the program contains an error, the analysis will report a warning.</td>
</tr>
<tr>
<td></td>
<td>“Sound for reporting correctness”</td>
</tr>
<tr>
<td>Completeness</td>
<td>If the analysis reports an error, the program will contain an error.</td>
</tr>
<tr>
<td></td>
<td>“Complete for reporting correctness”</td>
</tr>
</tbody>
</table>

Note: these terms have different meaning in other contexts
**Complete**
- Reports all errors
- Reports no false alarms

**Incomplete**
- Reports all errors
- May report false alarms

**Sound**
- Undecidable

(Ex: Manual Program Verification)

(Ex: Abstract Interpretation)

**Unsound**
- May not report all errors
- Reports no false alarms

(Ex: Symbolic Execution)

(Ex: Syntactic Analysis)
Program Verification
Program Verification

• How to prove a program free of buffer overflows?
  – Precondition
  – Postcondition
  – Loop invariants
Precondition

- Precondition for $f()$ is an assertion (a logical proposition) that must hold at input to $f()$
  - If any precondition is not met, $f()$ may not behave correctly
  - Callee may freely assume obligation has been met
- The concept similarly holds for any statement or block of statements
Precondition Example

• Precondition:
  – fp points to a valid location in memory
  – fp points to a file
  – the file that fp points to contains at least 4 characters
  – ...

```c
1: int parse(FILE *fp) {
2:   char cmd[256], *url, buf[5];
3:   fread(cmd, 1, 256, fp);
4:   int i, header_ok = 0;
5:   if (cmd[0] == 'G')
6:     if (cmd[1] == 'E')
7:       if (cmd[2] == 'T')
8:         if (cmd[3] == ' ')
9:           header_ok = 1;
10:  if (!header_ok) return -1;
11:  url = cmd + 4;
12:  i = 0;
13:  while (i<5 && url[i] != '\0' && url[i] != '\n') {
14:    buf[i] = tolower(url[i]);
15:    i++;
16:  }
17:  buf[i] = '\0';
18:  printf("Location is %s\n", buf);
19:  return 0; }
```
Postcondition

• **Postcondition for f( )**
  – An assertion that holds when f() returns
  – f() has obligation of ensuring condition is true when it returns
  – Caller may assume postcondition has been established by f()
Postcondition Example

• Postcondition:
  – \( \text{buf} \) contains no uppercase letters
  – \((\text{return } 0) \Rightarrow (\text{cmd}[0..3] \text{ == “GET “)}\)
Proving Precondition $\Rightarrow$ Postcondition

- Given preconditions and postconditions
  - Specifying what obligations caller has and what caller is entitled to rely upon

- Verify: No matter how function is called,
  - if precondition is met at function’s entrance,
  - then postcondition is guaranteed to hold upon function’s return
Proving Precondition ⇒ Postcondition

• Basic idea:
  – Write down a precondition and postcondition for every line of code
  – Use logical reasoning

• Requirement:
  – Each statement’s postcondition must match (imply) precondition of any following statement
  – At every point between two statements, write down invariant that must be true at that point
    • Invariant is postcondition for preceding statement, and precondition for next one
We’ll take our example, fix the bug, and show that we can successfully prove that the bug no longer exists.

1: int parse(FILE *fp) {
2:     char cmd[256], *url, buf[5];
3:     fread(cmd, 1, 256, fp);
4:     int i, header_ok = 0;
5:     if (cmd[0] == 'G')
6:         if (cmd[1] == 'E')
7:             if (cmd[2] == 'T')
8:                 if (cmd[3] == ' ')
9:                     header_ok = 1;
10:     if (!header_ok) return -1;
11:     url = cmd + 4;
12:     i = 0;
13:     while (i<5 && url[i]!='\0' && url[i]!='n') {
14:         buf[i] = tolower(url[i]);
15:         i++;
16:     }
17:     assert(i>=0 && i<5);
18:     buf[i] = '\0';
19:     printf("Location is %s\n", buf);
20:     return 0; }

\[
\begin{align*}
\varphi(x) &= f(x) \\
\psi &= \text{CRASH!}
\end{align*}
\]

\[
\begin{align*}
\text{is}(i<5 \land \text{url}[i]!='0' \land \text{url}[i]!='n')? \\
\text{assert}(i>0 \land i<5); \\
\text{buf}[i] &= '{0}'
\end{align*}
\]
We’ll take our example, fix the bug, and show that we can successfully prove that the bug no longer exists.

```c
1:int parse(FILE *fp) {
2:  char cmd[256], *url, buf[5];
3:  fread(cmd, 1, 256, fp);
4:  int i, header_ok = 0;
5:  if (cmd[0] == ‘G’)
6:    if (cmd[1] == ‘E’)
7:      if (cmd[2] == ‘T’)
8:        if (cmd[3] == ‘ ‘)
9:          header_ok = 1;
10:  if (!header_ok) return -1;
11:  url = cmd + 4;
12:  i=0;
13:  while (i<4 && url[i]! = ‘\0’ && url[i]! = ‘\n’)
14:    { buf[i] = tolower(url[i]);
15:      i++;
16:    }
17:  assert(i>=0 && i<5);
18:  buf[i] = ‘\0’;
19:  printf("Location is %s
", buf);
20:  return 0; }
```
We’ll take our example, fix the bug, and show that we can successfully prove that the bug no longer exists...

...So assuming fp points to a file that begins with "GET ", we want to show that `parse` never goes down the false assertion path.

But first, we will need the concept of loop invariant.

```
1: int parse(FILE *fp) {
2:   char cmd[256], *url, buf[5];
3:   fread(cmd, 1, 256, fp);
4:   int i, header_ok = 0;
5:   if (cmd[0] == 'G')
6:     if (cmd[1] == 'E')
7:       if (cmd[2] == 'T')
8:         if (cmd[3] == ' ')
9:           header_ok = 1;
10:   if (!header_ok) return -1;
11:   url = cmd + 4;
12:   i=0;
13:   while (i<4 && url[i]!='\0' && url[i]!='n') {
14:     buf[i] = tolower(url[i]);
15:     i++;
16:   }  // buf[i] = '\0';
17:   printf("Location is %s\n", buf);
18:   return 0; }
```
Loop Invariant and Induction

• An assertion that is true at entrance to the loop, on any path through the code
  – Must be true before every loop iteration
    • Both a pre- and post-condition for the loop body
• To verify:
  – Base Case: Prove true for first iteration: \( \phi(0) \)
  – Inductive step: Assume \( \phi(i) \) at the beginning of the loop. Prove \( \phi(i+1) \) at the start of the next iteration.
Try with our familiar example, proving that \((0 \leq i < 5)\) after the loop terminates:

**Loop Invariant:**

\[\varphi(i) = (0 \leq i < 5)\]

**Base Case:**

\[\varphi(0) = (0 \leq 0 < 5)\]

**Inductive Step:**

\[\begin{align*}
\text{assume } & (0 \leq i < 5) \text{ at the beginning of the loop} \\
\text{for } & (0 \leq i < 4), \text{ clearly } (0 \leq i+1 < 5) \\
\text{no } & (i=5) \text{ is not a possible case since} \\
\text{that would fail the looping predicate} \\
\Rightarrow & (0 \leq i+1 < 5) \text{ at the end of the loop} \\
\Rightarrow & \text{parse never fails the assertion}
\end{align*}\]
Function Post-/Pre-Conditions

- For every function call, we have to verify that its precondition will be met
  - Then we can conclude its postcondition holds and use this fact in our reasoning
- Annotating every function with pre- and post-conditions enables *modular reasoning*
  - Can verify function $f()$ by looking at only its code and the annotations on every function $f()$ calls
    - Can ignore code of all other functions and functions called transitively
  - Makes reasoning about $f()$ an almost purely local activity
Dafny

• A programming language with built-in specification constructs.
• A static program verifier to verify the functional correctness of programs.
• Powered by Boogie and Z3.
• Available here: http://rise4fun.com/dafny/
Documentation

• Pre-/post-conditions serve as useful documentation
  – To invoke Bob’s code, Alice only has to look at pre- and post-conditions – she doesn’t need to look at or understand his code

• Useful way to coordinate activity between multiple programmers:
  – Each module assigned to one programmer, and pre-/post-conditions are a contract between caller and callee
  – Alice and Bob can negotiate the interface (and responsibilities) between their code at design time
Preventing Security Vulnerabilities

• Identify implicit requirements code must meet
  – Must not make out-of-bounds memory accesses, dereference null pointers, etc.

• Prove that code meets these requirements
  – Ex: when a pointer is dereferenced, there is an implicit precondition that pointer is non-null and in-bounds
Preventing Security Vulnerabilities

• How easy it is to prove a certain property of code depends on how code is written
  – Structure your code to make it easy to prove