# Static Analysis and Software Verification
# Introduction



EISTI

# Classical Bugs with Integer Operations

## Factorial program

```
#include <stdio.h>
int fact(int n) {
    int r, i;
    r = 1;
    for (i=2; i<=n; i++) {
        r = r*i;
    }
    return r;
}
int main() {
    int n;
    scanf("%d", &n);
    printf("%d ! = %d",n,fact(n));
 }
```

# Classical Bugs with Integer Operations

## Factorial program

```
% gcc fact.c -o fact.exe
% ./fact.exe
4
4 ! = 24
% ./fact.exe
100
100 ! = 0
% ./fact.exe
20
20 ! = -2102132736
```

## Questions :

```
fact(-1) ?
Different results with respect to programming languages ?
```

# Classical Bugs with Floating-point Operations

## Example with rounding error (1)

```
#include <stdio.h>
int main() {
    double x, a;
    float y, z;
    x = 1125899973951488.0;
    a = 1.0;
    y = x + a;
    z = x - a;
    printf("%f", y-z);
}
```

# Classical Bugs with Floating-point Operations

## Example with rounding error (1)

```
% gcc arrondi1.c -o arrondi1.exe
% ./arrondi1.exe
134217728.000000
```

# Classical Bugs with Floating-point Operations

## Example with rounding error (2)

```
#include <stdio.h>
int main() {
    double x, a;
    float y, z;
    x = 1125899973951487.0;
    a = 1.0;
    y = x + a;
    z = x - a;
    printf("%f", y-z);
}
```

# Classical Bugs with Floating-point Operations

## Example with rounding error (2)

```
% gcc arrondi2.c -o arrondi2.exe
% ./arrondi2.exe
0.000000
```

# Some Arithmetic Precision Errors

- **Ariane 5's failure** : data conversion from a 64-bit floating point to 16-bit signed integer value caused an arithmetic overflow.
- **Patriot's failure** : software error in the system's clock $((0,1)_{10} = (0,00011001100110011...)_2)$.

# Some Arithmetic Precision Errors

- **Excel 2007 bug** : 77,1 x 850 = 65535 but 100000 is displayed (round-off error while converting IEEE 754 64-bit floating point to Unicode string)
- **Yorktown bug** : division by zero

# Common Memory-related Bugs in C Programs (Computer Systems : A Programmer's Perspective, Bryant and O'Hallaron)

- Dereferencing bad pointers
- Reading uninitialized memory
- Overwriting memory
- Referencing nonexistent variables
- Freeing blocks multiple times
- Referencing freed blocks
- Failing to free blocks
- Buffer overflow

# Dereferencing Bad Pointers

## The classic **scanf** bug

```
int val;

...

scanf("%d", val);
```

Will cause **scanf** to interpret contents of **val** as an address !

- Best case : program terminates immediately due to segmentation fault
- Worst case : contents of **val** correspond to some valid read/write area of virtual memory, causing **scanf** to overwrite that memory, with disastrous and baffling consequences much later in program execution

# Reading Uninitialized Memory

## Assuming that heap data is initialized to zero

```
/* return y = Ax */
int *matvec(int **A, int *x) {
   int *y = (int *)malloc(N * sizeof(int));
   int i, j;

   for (i=0; i<N; i++) {
      for (j=0; j<N; j++) {
         y[i] += A[i][j] * x[j];
      }
   }
   return y;
}
```

# Overwriting Memory

## Allocating the (possibly) wrong sized object

```
int **p;

p = (int **)malloc(N * sizeof(int));

for (i=0; i<N; i++) {
   p[i] = (int *)malloc(M * sizeof(int));
}
```

# Overwriting Memory

## Off-by-one error

```
int **p;

p = (int *)malloc(N * sizeof(int *));

for (i=0; i<=N; i++) {
   p[i] = (int *)malloc(M * sizeof(int));
}
```

# Overwriting Memory

## Misunderstanding pointer arithmetic

```
int *search(int *p, int val) {

   while (p && *p != val)
      p += sizeof(int);

   return p;
}
```

# Overwriting Memory

## Referencing a pointer instead of the object it points to

```
int *getPacket(int **packets, int *size) {
   int *packet = packets[0];
   packets[0] = packets[*size - 1];
   *size--;
   reorderPackets(packets, *size);
   return(packet);
}
```

# Overwriting Memory

## Referencing a pointer instead of the object it points to

```
int *getPacket(int **packets, int *size) {
    int *packet = packets[0];
    packets[0] = packets[*size - 1];
    *size--;                    // what is happening here?
    reorderPackets(packets, *size);
    return(packet);
}
```

- −− and * operators have same precedence and associate from right-to-left, so −− happens first!
- gcc will raise a warning for this line of code only if -Wall is used (value computed is not used [-Wunused-value])

# Referencing Nonexistent Variables

**Forgetting that local variables disappear when a function returns**

```
int *foo () {
   int val;

   return &val;
}
```

# Freeing Blocks Multiple Times

## Nasty !

```
x = (int *)malloc(N * sizeof(int));
        <manipulate x>
free(x);
...

y = (int *)malloc(M * sizeof(int));
free(x);
        <manipulate y>
```

# Referencing Freed Blocks

## Evil !

```
x = (int *)malloc(N * sizeof(int));
  <manipulate x>
free(x);
   ...
y = (int *)malloc(M * sizeof(int));
for (i=0; i<M; i++)
   y[i] = x[i]++;
```

# Failing to Free Blocks (Memory Leaks)

## Slow, silent, long-term killer !

```
foo() {
   int *x = (int *)malloc(N*sizeof(int));
   ...
   return;
}
```

# Failing to Free Blocks (Memory Leaks)

## Freeing only part of a data structure

```
struct list {
   int val;
   struct list *next;
};

foo() {
   struct list *head =
      (struct list *)malloc( sizeof(struct list) );
   head->val = 0;
   head->next = NULL;
   <create and manipulate the rest of the list>
    ...
   free(head);
   return;
}
```

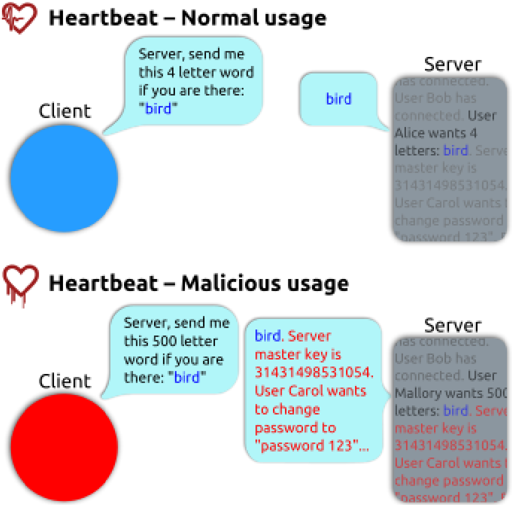
# Buffer Overflow

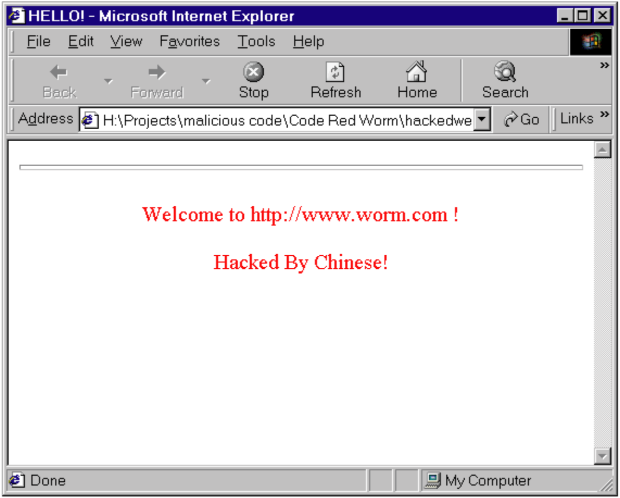
**Definition (wikipedia)**

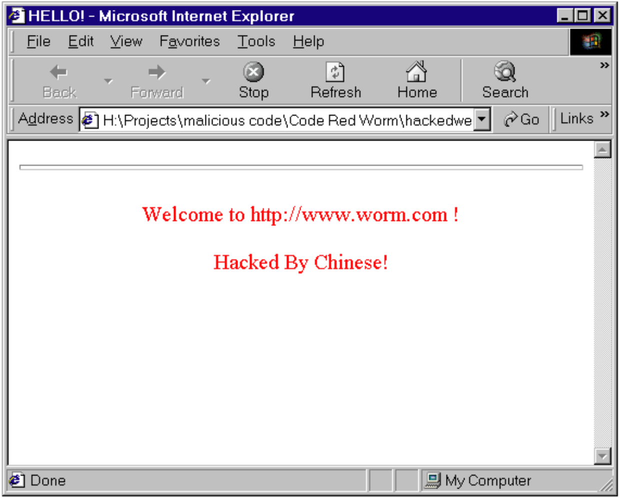
A **buffer overflow** (**buffer overrun**) is an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory.

- 50% of software errors
- erratic program behaviors : memory access errors, incorrect results, crashes, security holes, ...
- sources of problem in C, C++ : gets, strcpy, memcpy, ...

*Heartbleed is a security bug disclosed in April 2014 in the OpenSSL cryptography library, which is a widely used implementation of the Transport Layer Security (TLS) protocol. It results from improper input validation (due to a missing bounds check) in the implementation of the TLS heartbeat extension, thus the bug's name derives from "heartbeat". The vulnerability is classified as a buffer over-read, a situation where software allows more data to be read than should be allowed.*


Heartbeat – Normal usage
Client
Server, send me this 4 letter word if you are there: "bird"
bird
Server
User Bob has connected. User Alice wants 4 letters: bird. Serve master key is 31431498531054. User Carol wants t change password
Heartbeat – Malicious usage
Client
Server, send me this 500 letter word if you are there: "bird"
bird. Server master key is 31431498531054. User Carol wants to change password to "password 123"...
Server
User Bob has connected. User Mallory wants 50C letters: bird. Serv master key is 31431498531054. User Carol wants t change password


HELLO! - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search
Address H:\Projects\malicious code\Code Red Worm\hackedwe
Go
Links
Welcome to http://www.worm.com !
Hacked By Chinese!
Done
My Computer




Code Red Exploit Code, attack www.whitehouse.gov (denial of service attack)

# Verification and Validation

# Verification and Validation

- Validation : Are we building the right system ?
- Verification : Are we building the system right ?

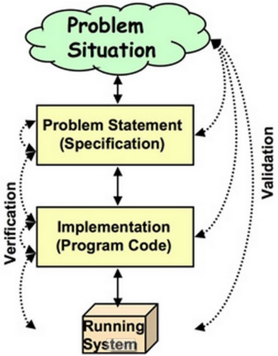



FIGURE – www.easterbrook.ca/steve

# Verification and Validation

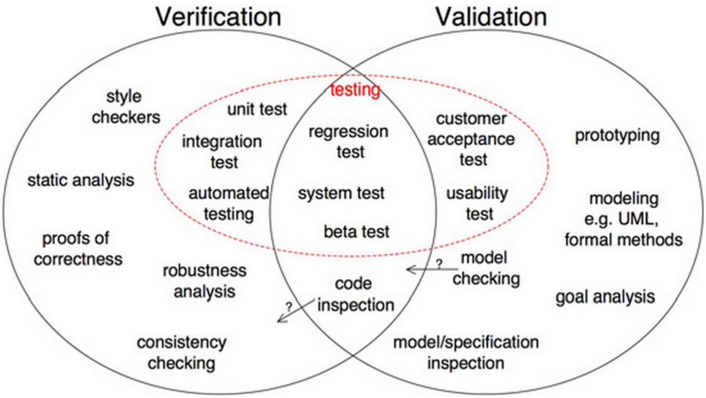



FIGURE – Verification and Validation Toolbox (www.easterbrook.ca/steve)

# Why Verification and Validation ?

- The most expensive bug : explosion of Ariane 5 (10 year project costing 7$ billions)
- Verification and validation are critical to guarantee reliability, robustness and quality of software systems
- But it is an expensive process : 80% of development cost in safety-critical systems

# Verification and Validation Methods

- Software testing
  - development, validation, in-exhaustive verification
- Theorem proving
  - mathematical foundations, human experts
- Model checking
  - exhaustive enumeration, state explosion problem
- Dynamic analysis
  - run-time checking
- **Static analysis**

# Static Analysis

## Definition

Methods try to discover properties of a program **without running** it :

- Code optimization : improving time, space, energy ; compilation for special architectures (multicore, ...)
- Software and reverse engineering : program comprehension, code review, documentation, maintenance, ...
- Verification and validation

# Static Analysis

## Verification and validation

- Finding run-time errors at compilation-time
- Proving the absence of errors in the source code, i.e the program will :
  - never divide a number by zero
  - never dereference a NULL pointer
  - close all opened files, all opened socket connections
  - not allow buffer overflow security violation
  - ...

# Some Static Analysis Tools

- CodeSonar (Grammatech) : C, C++, Java
- PolySpace (MathWorks) : C, C++, Ada
- Frama-C (CEA + INRIA) : C
- CodePeer (AdaCore) : SPARK
- Coverity : C, C++, Java, C#
- Splint / LCLint (University of Virginia) : C ; PC-Lint (Grimpel) : C, C++
- Microsoft/analyze : C, C++ (Visual Studio)
- Fortify (HP)
- PVS-Studio : C, C++
- Astree : C embedded and real-time systems
- Clang Static Analyzer (LLVM) : C, C++, Objective-C
- Java : programming rules, dead code, optimization (Checkstyle, FindBugs, PMD, ...)

# Polyspace Code Prover

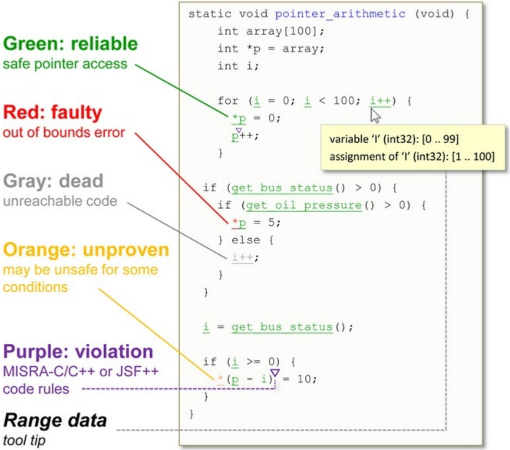



FIGURE – Polyspace Coded Color

# Static Analysis

## Program analysis approaches

- Abstract interpretation
- Program flow analysis
- ...

# Abstract Interpretation (P. and R. Cousot)

Abstraction theory :

- concrete semantics of a program : undecidable (i.e termination)
  - P = while termination(P) do skip od (K. Godel)
- abstract semantics : safe approximations of program semantics
- safe approximations must be :
  - **simple** enough to be computable by computer
  - **precise** enough to avoid false alarms (errors do not correspond to a real execution)
  - **sound** so that no possible error can be forgotten

# Program Semantics

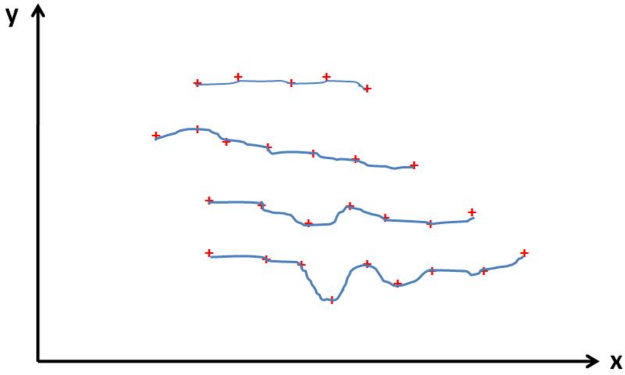



FIGURE – Set of traces (finite ou infinite)

# Program Semantics

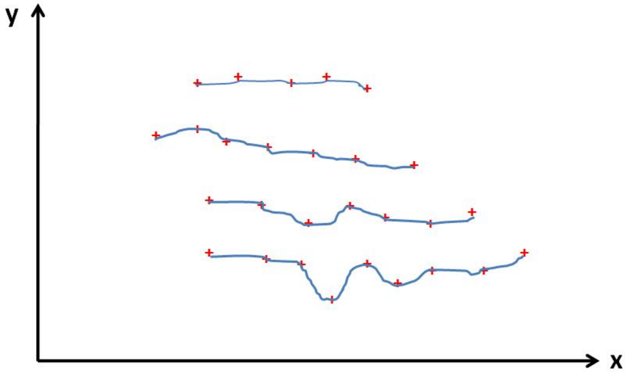



FIGURE – Set of points $\{(x_i, y_i)\}$

# Abstract Semantics : Abstraction by Signs

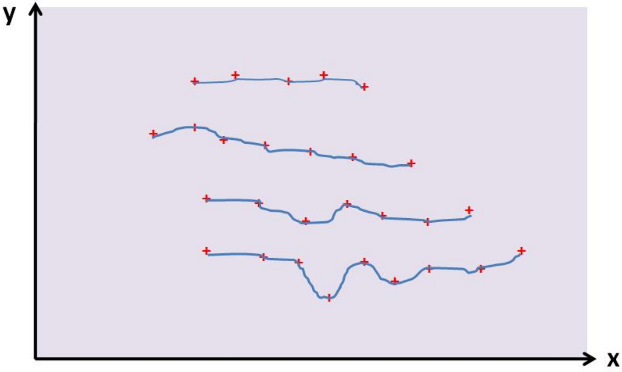



FIGURE – Signs $x \geq 0, y \geq 0$

# Abstract Semantics : Abstraction by Intervals

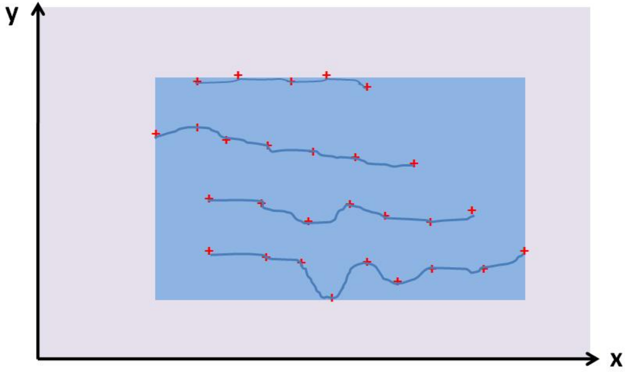



FIGURE – Intervals $a \leq x \leq b, c \leq y \leq d$

# Abstract Semantics : Abstraction by Octagons

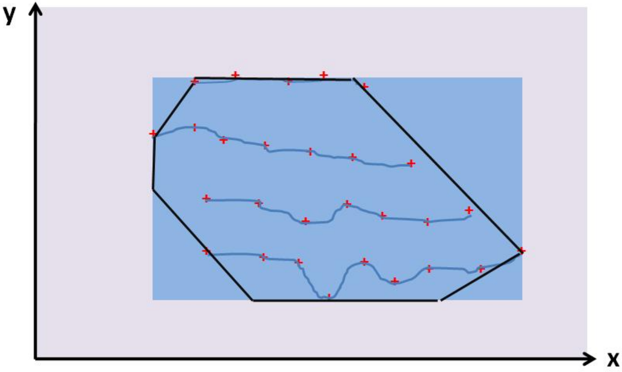



FIGURE – Octagons $x - y \leq a, x + y \leq b$

# Abstract Semantics : Abstraction by Polyhedrons

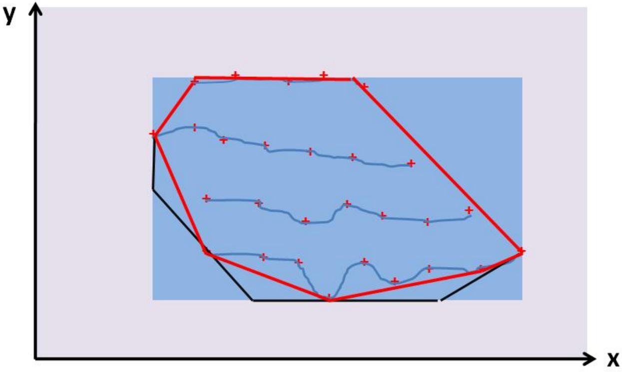



FIGURE – Polyhedrons $a.x + b.y \leq c$

# Abstract Interpretation

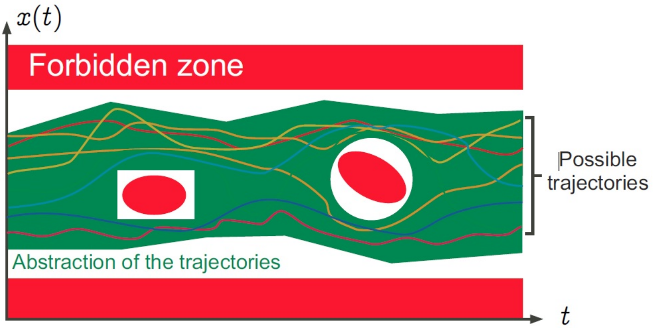



FIGURE – Graphic Example (Cousot_MIT_2005_Course)

If the abstract semantics is safe (does not intersect the forbidden zone), then so is the concrete semantics.

# Program Flow Analysis

**Definition :**

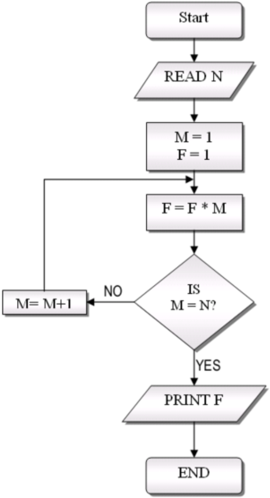
Method for describing what a program does to its data :

- Control-flow analysis
- Data-flow analysis

# Control-flow Analysis

- Discover the hierarchical control flow within each procedure
- Control flow graph (flowchart) : block, test, loop

# Example of an Informal Control-flow Graph

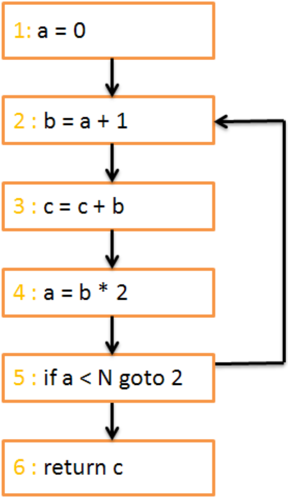


# Control-flow Graph

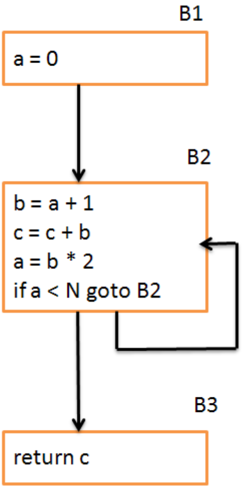
## Definitions

- basic block : maximal sequence of instructions that can be entered only at the first instruction and exited only from the last one
- entry node, exit node
- split node : a node that has more than one successor
- join node : a node that has more than one predecessor

# Control-flow Graph



# Control-flow Graph with Blocks



# Data-flow Analysis

- next course !