CMPSC 497: Static Analysis

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University
Our Goal

• In this course, we want to develop techniques to detect vulnerabilities before they are exploited automatically
  ‣ What’s a vulnerability?
  ‣ How to find them?
Dynamic Analysis Limits

• Major advantage
  ‣ When we produce a crash, it is a real crash

• Issue
  ‣ However, may not be exploitable (i.e., not a vulnerability)
  ‣ On the other hand, want to fix memory errors
    • But often not assertion failures

• Major limitation
  ‣ We cannot find all vulnerabilities in a program

• Why not?
Goal

• Can we build a technique that identifies *all* vulnerabilities?
Goal

- Can we build a technique that identifies *all* vulnerabilities?
  - Turns out that we can: static analysis
  - But, it has its own major limitation
    - Can identify many false positives (not actual vulnerabilities)
  - Can be effective when used carefully
Static Analysis

- Explore all possible executions of a program
  - All possible inputs
  - All possible states
A Form of Testing

• Static analysis is an alternative to dynamic testing

• Dynamic
  ‣ Select concrete inputs
  ‣ Obtain a sequence of states given those inputs
  ‣ Apply many concrete inputs (i.e., run many tests)

• Static
  ‣ Select abstract inputs with common properties
  ‣ Obtain sets of states created by executing abstract inputs
  ‣ One run
Static Analysis

- Provides an approximation of behavior
- “Run in the aggregate”
  - Rather than executing on ordinary states
  - Finite-sized descriptors representing a collection of states
- “Run in non-standard way”
  - Run in fragments
  - Stitch them together to cover all paths
- Runtime testing is inherently incomplete, but static analysis can cover all paths
Static Analysis

• Various properties of programs can be tracked
  ‣ Control flow
  ‣ Constants
  ‣ Data flow
  ‣ Types
  ‣ Values

• Which ones will expose which vulnerabilities accurately (not too many false positives) requires some finesse
No Return Check

• Can we detect code with no return check?
• From original Miller fuzzing paper

```c
format.c (line 276):
...
while (lastc != '\n') {
    rdc();
}
...

input.c (line 27):
rdc()
{ do { readchar(); } 
    while (lastc == '' || lastc == '\t'); return (lastc);
}
Control Flow Analysis

• Compute the control flow of a program
  ‣ I.e., possible execution paths

• To find an execution path that does not check the return value of a function
  ‣ That is actually run by the program
  ‣ How do we do this?
Intraprocedural CFG

• Statements
  ‣ Nodes
  ‣ One successor and one predecessor

• Basic Blocks
  ‣ Multiple successors (multiple predecessors)

• Unique Enter and Exit
  ‣ All start nodes are successors of enter
  ‣ All return nodes are predecessors of exit
Control Flow Analysis

- Compute Control Flow
- Function by function — “intraprocedural”
- Program statements of interest
  - Sequences – basic blocks
  - Conditionals – transitions between basic blocks in function
  - Loops – transitions that connect to prior basic blocks
  - Calls – transition to another function
  - Return – transition that completes the function
Control Flow Analysis

• Compute Intraprocedural Control Flow

From Last Time: BB and CFG

- Basic block – a sequence of consecutive operations in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end

- Control Flow Graph – Directed graph, G = (V,E) where each vertex V is a basic block and there is an edge E, v1 (BB1) → v2 (BB2) if BB2 can immediately follow BB1 in some execution sequence

```
x = y+1;
if (c)
    x++;
else
    x--;
```
```
y = z + 1;
if (a)
    y++;
else
    y--;
z++;
```
No Return Check

- Can we detect code with no return check?
  - Detect function calls in CFG reachable that have no return assignment

```c
format.c (line 276):
... 
while (lastc != '\n') {
    rdc();
}
...

input.c (line 27):
rdc()
{ do { readchar(); } 
    while (lastc == ' ' || lastc == '\t'); return (lastc);
}
```
Double Free

• Can we detect double frees?
  ‣ CFG shows a flow from free(buf2R1) to free(buf2R1)

```c
main(int argc, char **argv)
{
    ...
    buf1R1 = (char *) malloc(BUFSIZE2);
    buf2R1 = (char *) malloc(BUFSIZE2);
    free(buf1R1);
    free(buf2R1);
    buf1R2 = (char *) malloc(BUFSIZE1);
    strncpy(buf1R2, argv[1], BUFSIZE1-1);
    free(buf2R1);
    free(buf1R2);
}
```
Double Free

• Can we detect double frees?
  ‣ CFG shows a flow from free(buf2R1) to free(buf2R1)

• More complex if…
  ‣ Free occurs in a different function
    • Interprocedural CFG
  ‣ Free is performed on a different variable
    • Track assignments and aliases
Partial Evaluation

• Can we detect double frees?
  ‣ One advantage of static analysis is we can start our analysis from anywhere – just analyze this function

```c
foo(int x, char **y) // need not be "main"
{
  ...
  buf1R1 = (char *) malloc(BUFSIZE2);
  buf2R1 = (char *) malloc(BUFSIZE2);
  free(buf1R1);
  free(buf2R1);
  buf1R2 = (char *) malloc(BUFSIZE1);
  strncpy(buf1R2, argv[1], BUFSIZE1-1);
  free(buf2R1);
  free(buf1R2);
}
```
Interprocedural CFG

• The basics are easy
  ‣ **Call** – connect to CFG of callee function
  ‣ **Return** – create an edge back to the caller function

• Challenges
  ‣ May not know targets of CFG edges
    • Function pointers, indirect jumps, and returns without call
  ‣ The same function may be called from multiple places
    • `Printk` is called 18,000 times in FreeBSD – copy it?
  ‣ Exponential growth – caused by branching
Constant Propagation

• Substitute the values of known constants in expressions
• Propagate the values among variables assigned those constants
• Example assignments resulting from propagation to detect problems
Detect Buffer Overflow

• What are the constant values below?

```c
1  char text[] = "Foo           Bar";
2  char buffer1[4], buffer2[4];
3
4  int i, n = sizeof(text);
5  for(i=0; i<n; ++i)
6      buffer2[i] = text[i];
7  printf("Last char of text is: %c", text[n]);
```
Detect Buffer Overflow

• Where can they be propagated?

1   char    text[] = "Foo            Bar";
2   char buffer1[4], buffer2[4];
3
4  int i, n = sizeof(text);
5  for(i=0;i<n;++i)
6     buffer2[i] = text[i];
7 printf("Last char of text is: %c", text[n]);
Detect Buffer Overflow

• Where are the memory errors?

1 char text[] = "Foo Bar";
2 char buffer1[4], buffer2[4];
3
4 int i, n = 20;
5 for(i=0;i<20;++i)
6 buffer2[i] = text[i];
7 printf("Last char of text is: %c", text[20]);
Detect Buffer Overflow

• Where are the memory errors?

```c
1 char text[] = "Foo           Bar";
2 char buffer1[4], buffer2[4];
3
4 int i, n = 20;
5 for(i=0;i<20;++i)
6     buffer2[i] = text[i];
7 printf("Last char of text is: %c", text[20]);
```
Constant Propagation

• Typically, constant propagation is a start, but need more to detect an error.

• For the buffer overflow we need to know that access to buffer2[4-19] and text[20] are memory errors.
Type-based Analysis

• Maybe we want to check for certain properties about variables in our program

• Can use type information associated with variables to perform such checks
Type-based Analysis

• Maybe we want to check for certain properties about variables in our program

• Suppose we want to know if a variable’s value has been “checked” – such as for input validation

• We can use type-based analysis to do that
Type-based Analysis

- Maybe we want to check for certain properties about variables in our program
- Suppose we want to know if a variable’s value has been “checked” – such as for input validation
- We can use type-based analysis to do that

Figure 2: The complete mediation problem.
Type-based Analysis

• Maybe we want to check for certain properties about variables in our program

• Suppose we want to know if a variable’s value has been “checked” – such as for input validation

• Using type qualifiers, can extend basic types

```c
void func_a(struct file * $checked filp);

void func_b( void )
{
    struct file * $unchecked filp;
    ...
    func_a(filp);
    ...
}
```

Security Check

C: Checked
U: Unchecked
Type-based Analysis

• Maybe we want to check for certain properties about variables in our program

• Suppose we want to know if a variable’s value has been “checked” – such as for input validation

• To find missing mediation (e.g., input validation)
  ‣ Initialize untrusted inputs to “unchecked”
  ‣ Initialize security-sensitive operation to use “checked”
  ‣ Identify mediation (create “checked” version)
  ‣ Detect type error – from “unchecked” to “checked”
Type-based Analysis

- Vulnerability in the code to the right
  - Can you see it?
Type-based Analysis

- Vulnerability in the code to the right
  - fd is unchecked as is filp initially in sys_fnctl
  - However, filp would be reassigned to a checked variable after security_op
- So what’s the problem?
Type-based Analysis

- Vulnerability in the code to the right
  - \( fd \) is unchecked as is \( filp \) initially in \( sys\_fnctl \)
  - However, \( filp \) would be reassigned to a checked variable after \( security\_op \)
- \( fd \), not the checked \( filp \) is passed to \( do\_fcntl \) and then to \( fcntl\_getlk \)
Data Flow

• Track dependency among variable values

• Given a statement “\( y = a \times b \)” a data flow dependence is created from “\( a \)” and “\( b \)” to “\( y \)”

• Data flow can be used for a lot of purposes

• For security, an important property to compute using data flow is “taint”
  ‣ Is a public statement/variable tainted by secret data?
  ‣ Is a high integrity statement/variable tainted by low integrity inputs?
Data Flow

- Taints from “text” input

```c
1 char text[] = "Foo           Bar";
2 char buffer1[4], buffer2[4];
3
4 int i, n = 20;
5 for(i=0;i<20;++i)
6    buffer2[i] = text[i];
7 printf("Last char of text is: %c", text[20]);
```
Data Flow

• Tells us that buffer2 may be affected by untrusted input in text as well as printf output

1  char  text[] = "Foo Bar";
2  char  buffer1[4], buffer2[4];
3
4  int  i, n = 20;
5  for(i=0;i<20;++i)
6      buffer2[i] = text[i];
7  printf("Last char of text is: %c", text[20]);
Data Flow

• Statement
  ‣ Create node for output variable
  ‣ Connect inputs to output

• Sequence (Basic Block)
  ‣ Process data flows for each statement

• Inter-block
  ‣ Connect data flows from any predecessor

• Call and return
  ‣ Connect data flows from input parameters and to outputs
Data Flow

• This simple view of data flow mainly tells us about reachability
  ‣ More detailed analyses later
Take Away

• Static analysis evaluates all the ways that a program may execute in one pass
  ‣ Can be sound (no false negatives)
  ‣ But, often produces false positives

• Examined some building blocks of static analysis and how they could be used
  ‣ Constant propagation, control flow, type analysis, data flow

• We’ll examine static analysis for more complex programs next time – try to track values
Abstract Interpretation

• Descriptors represent the sign of a value
  ‣ Positive, negative, zero, unknown

• For an expression, \( c = a \times b \)
  ‣ If \( a \) has a descriptor \( \text{pos} \)
  ‣ And \( b \) has a descriptor \( \text{neg} \)

• What is the descriptor for \( c \) after that instruction?
• How might this help?
Abstract Interpretation

• Descriptors represent the sign of a value
  ‣ Positive, negative, zero, unknown

• For an expression, \( c = a \times b \)
  ‣ If \( a \) has a descriptor \( pos \)
  ‣ And \( b \) has a descriptor \( neg \)

• What is the descriptor for \( c \) after that instruction?
• How might this help?
Descriptors

• Choose a set of descriptors that
  ‣ Abstracts away details to make analysis tractable
  ‣ Preserves enough information that key properties hold
    • Can determine interesting results

• Using sign as a descriptor
  ‣ Abstracts away specific integer values (billions to four)
  ‣ Guarantees when \( a \times b = 0 \) it will be zero in all executions

• Choosing descriptors is one key step in static analysis
Precision

• Abstraction loses some precision

• Enables run in aggregate, but may result in executions that are not possible in the program
  ‣ \((a \leq b)\) when both are \textit{pos}
  ‣ If \(b\) is equal to \(a\) at that point, then false branch is never possible in concrete executions

• Results in \textit{false positives}
Soundness

- The use of descriptors “over-approximates” a program’s possible executions
- Abstraction must include all possible legal values
  - May include some values that are not actually possible
- The run-in-aggregate must preserve such abstractions
  - Thus, must propagate values that are not really possible
Implications of Soundness

• Enables proof that a class of vulnerabilities are completely absent
  ‣ No false negatives in a sound analysis

• Comes at a price
  ‣ Ensuring soundness can be complex, expensive, cautious

• Thus, unsound analyses have gained in popularity
  ‣ Find bugs quickly and simply
  ‣ Such analyses have both false positives and false negatives
What Is Static Analysis?

- **Abstract Interpretation**
  - Execute the system on a simpler data domain
    - Descriptors of the *abstract domain*
  - Rather than the *concrete domain*
- Elements in an abstract domain represent sets of concrete states
  - Execution mimics all concrete states at once
- Abstract domain provides an over-approximation of the concrete domain
Abstract Domain Example

• Use interval as abstract domain
  ‣ $b = [40, 41]$

• $a = 2 \times b$
  ‣ $a = [x, y]$?

• What are the possible concrete values represented?
  ‣ Which concrete states are possible?
Joins

• A join combines states from multiple paths
  ‣ Approximates set-union as either path is possible

• Use Interval as abstract domain
  ‣ \(a = [36, 39], b = [40, 41]\)

• If \((a >= 38) \ a=2*b; /* join */\)
  ‣ \(a = [x, y], b=[40, 41]\) – what are \(x\) and \(y\)?

• What’s the impact of over-approximation?
Impact of Abstract Domain

• The choice of abstract domain must preserve the over-approximation to be sound (no false negatives)

• Integer arithmetic vs 2’s-complement arithmetic

• $a = [253, 254], b = [10, 12]$
  - What is $c = a + b$ in an 32-bit machine?
  - What is $c = a + b$ in an 8-bit machine?
Use AI to find bug

• Find bug from Parfait?
  ‣ Constant propagation
  ‣ Partial execution
  ‣ Example for buffer overflow

• What do you need in general – Data Flow
  ‣ To do

• Challenges
  ‣ Illegal paths – 1st paper
  ‣ CFG – contexts
Intraprocedural CFG

• Statements
  ‣ Nodes
  ‣ One successor and one predecessor

• Basic Blocks
  ‣ Multiple successors to the join (multiple predecessors)
  ‣ Examples?

• Unique Enter and Exit
  ‣ All start nodes are successors of enter
  ‣ All return nodes are predecessors of exit