Static-Analysis Technology for Security

Thomas Reps
University of Wisconsin
Goals of the Briefing

- Relevance of static analysis to security issues
- Tutorial on some static-analysis issues
- Some forward pointers to other presentations
The Role of Static Analysis

• De-obfuscation to detect . . .
  – Undesirable information flows
  – Buffer-overrun attacks
  – Actions of a virus

• More precise = less freedom for attacker
Dependence Analysis and Analysis of Malicious Code

• Post-mortem analysis
  – What affects what?

• Information Flow
  – Data dependences + control dependences

• Program Differencing
  – Compare slices to identify changes

• CodeSurfer
int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
        sum = sum + i;
        i = i + 1;
    }  
    printf("%d\n",sum);
    printf("%d\n",i);
}

Backward slice with respect to “printf("%d\n",i)"
int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
        sum = sum + i;
        i = i + 1;
    }
    printf("%d\n", sum);
    printf("%d\n", i);
}

Backward slice with respect to “printf("%d\n",i)”
int main() {
    int i = 1;
    while (i < 11) {
        i = i + 1;
    }
    printf("%d\n", i);
}

Backward slice with respect to “printf("%d\n", i)”
int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
        sum = sum + i;
        i = i + 1;
    }
    printf("%d\n",sum);
    printf("%d\n",i);
}
int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
        sum = sum + i;
        i = i + 1;
    }
    printf("%d\n", sum);
    printf("%d\n", i);
}

**Forward slice** with respect to \("\text{sum} = 0\)"
int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
        sum = sum + i;
        i = i + 1;
    }
    printf("%d\n", sum);
    printf("%d\n", i);
}
Control Dependence Graph

```c
int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
        sum = sum + i;
        i = i + 1;
    }
    printf("%d\n", sum);
    printf("%d\n", i);
}
```

**Control dependence**

- If condition $p$ is true ($T$), not otherwise.
- Similar for false ($F$).
int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
        sum = sum + i;
        i = i + 1;
    }
    printf("%d\n",sum);
    printf("%d\n",i);
}

Flow dependence

\[ p \rightarrow q \]
Value of variable assigned at \( p \) may be used at \( q \).

```
sum = 0
i = 1
while (i < 11)
    sum = sum + i
    i = i + 1;
printf("%d\n",sum);
printf("%d\n",i);
```
int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
        sum = sum + i;
        i = i + 1;
    }
    printf("%d\n", sum);
    printf("%d\n", i);
}
int main() {
    int i = 1;
    int sum = 0;
    while (i < 11) {
        sum = sum + i;
        i = i + 1;
    }
    printf("%d\n",sum);
    printf("%d\n",i);
}

Program Dependence Graph

Opposite Order
Same PDG
```c
int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
        sum = sum + i;
        i = i + 1;
    }
    printf("%d\n",sum);
    printf("%d\n",i);
}
```

The diagram shows the control flow of the `main` function, with nodes for `sum = 0`, `i = 1`, `while(i < 11)`, `printf(sum)`, and `printf(i)`. The edges represent the flow of control, with `T` indicating a true condition and `F` indicating a false condition. The diagram visually illustrates how the variables change and the flow of execution through the `while` loop.
int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
        sum = sum + i;
        i = i + 1;
    }
    printf("%d\n", sum);
    printf("%d\n", i);
}

t = 0  i = 1  while(i < 11)  printf(sum)  printf(i)
int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
        sum = sum + i;
        i = i + 1;
    }
    printf("%d\n", sum);
    printf("%d\n", i);
}
```c
int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
        sum = sum + i;
        i = i + 1;
    }
    printf("%d\n", sum);
    printf("%d\n", i);
}
```

**Backward Slice**

- **sum = 0**
- **i = 1**
- **while(i < 11)**
  - **sum = sum + i**
  - **i = i + 1**
- **printf(sum)**
- **printf(i)**
int main() {
    int i = 1;
    while (i < 11) {
        i = i + 1;
    }
    printf("%d\n", i);
}
Static-Analysis Issues

- Context-sensitive vs. context-insensitive
- Flow-sensitive vs. flow-insensitive
- Coping with pointers
Static-Analysis Issues

- Context-sensitive vs. context-insensitive
- Flow-sensitive vs. flow-insensitive
- Coping with pointers
**Interprocedural Slice**

```c
int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
        sum = add(sum, i);
        i = add(i, 1);
    }
    printf("%d\n", sum);
    printf("%d\n", i);
}
int add(int x, int y) {
    return x + y;
}
```

**Backward slice** with respect to “printf("%d\n", i)”
Interprocedural Slice

```c
int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
        sum = add(sum, i);
        i = add(i, 1);
    }
    printf("%d\n", sum);
    printf("%d\n", i);
}

int add(int x, int y) {
    return x + y;
}
```

Backward slice with respect to “printf("%d
",i)"
```c
int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
        sum = add(sum, i);
        i = add(i, 1);
    }
    printf("%d\n", sum);
    printf("%d\n", i);
}

int add(int x, int y) {
    return x + y;
}
```

Superfluous components included by Weiser’s slicing algorithm [TSE 84]
Left out by algorithm of Horwitz, Reps, & Binkley [PLDI 88; TOPLAS 90]
How is an SDG Created

- Each PDG has nodes for
  - entry point
  - procedure parameters and function result
- Each call site has nodes for
  - call
  - arguments and function result
- Appropriate edges
  - entry node to parameters
  - call node to arguments
  - call node to entry node
  - arguments to parameters
System Dependence Graph (SDG)
sum = 0
i = 1
while(i < 11)
printf(sum)
printf(i)
Call add
Call add
xin = sum
yin = i
sum = xout
xin = i
yin = 1
i = xout
Enter add
x = xin
y = yin
x = x + y
xout = x
Interprocedural Backward Slice
Interprocedural Backward Slice (2)
Interprocedural Backward Slice (3)
Interprocedural Backward Slice (4)
Interprocedural Backward Slice (5)
Interprocedural Backward Slice (6)
Matched-Parenthesis Path
Interprocedural Backward Slice (6)
Interprocedural Backward Slice (7)
Slice Extraction

[Diagram showing a process flow with nodes labeled "Enter main", "Call p", and "Enter p".]
Slice of the Sum Program

Enter main

\[ i = 1 \]
while \((i < 11)\)

printf \((i)\)

Call add

\[ x_{in} = i \]
\[ y_{in} = 1 \]
\[ i = x_{out} \]

Enter add

\[ x = x_{in} \]
\[ y = y_{in} \]
\[ x = x + y \]
\[ x_{out} = x \]
CFL-Reachability

[Yannakakis 90]

- $G$: Graph ($N$ nodes, $E$ edges)
- $L$: A context-free language
- $L$-path from $s$ to $t$ iff $s \xrightarrow{\alpha}^* t$, $\alpha \in L$
- Running time: $O(N^3)$
Interprocedural Slicing via CFL-Reachability

• Graph: System dependence graph
• $L: L(matched)$ [roughly]
• Node $m$ is in the slice w.r.t. $n$ iff there is an $L(matched)$-path from $m$ to $n$
Asymptotic Running Time
[Reps, Horwitz, Sagiv, & Rosay 94]

• CFL-reachability
  – System dependence graph: \( N \) nodes, \( E \) edges
  – Running time: \( O(N^3) \)

• System dependence graph \( \rightarrow \) Special structure

Running time: \( O(E + \text{CallSites} \oplus \text{MaxParams}^3) \)
matched → ε

( e [ e ] e [ e [ e ] ] e )

( matched )

matched matched

Ordinary Graph Reachability

CFL Graph Reachability
CFL-Reachability via Dynamic Programming

Graph

Grammar

A → B C
Static Analysis for Mandatory Access Control

Flow of information from high' to low?

Program Chopping: Chop(high', low) = Ø?
Program Chopping

Given source $S$ and target $T$, what program points transmit effects from $S$ to $T$?

Intersect forward slice from $S$ with backward slice from $T$, right?
Dynamic Transitive Closure ?!

• Aiken et al.
  – Set-constraint solvers
  – Points-to analysis

• Henglein et al.
  – type inference

• But a CFL captures a non-transitive reachability relation [Valiant 75]
int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
        sum = add(sum,i);
        i = add(i,1);
    }
    printf("%d\n",sum);
    printf("%d\n",i);
}

int add(int x, int y) {
    return x + y;
}

Forward slice with respect to “sum = 0”
int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
        sum = add(sum, i);
        i = add(i, 1);
    }
    printf("%d\n", sum);
    printf("%d\n", i);
}

int add(int x, int y) {
    return x + y;
}
Context-Sensitivity and Chopping

```c
int main() {    int add(int x, int y) {
    int sum = 0;    return x + y;
    int i = 1;    }
    while (i < 11) {
        sum = add(sum, i);
        i = add(i, 1);
    }
    printf("%d\n", sum);
    printf("%d\n", i);
}
```

Backward slice with respect to “printf("%d\n",i)”
Context-Sensitivity and Chopping

```c
int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
        sum = add(sum, i);
        i = add(i, 1);
    }
    printf("%d\n", sum);
    printf("%d\n", i);
}

int add(int x, int y) {
    return x + y;
}
```

Backward slice with respect to "printf("%d\n",i)"
Context-Sensitivity and Chopping

```c
int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
        sum = add(sum, i);
        i = add(i, 1);
    }
    printf("%d\n", sum);
    printf("%d\n", i);
}

int add(int x, int y) {
    return x + y;
}
```

**Forward slice** with respect to “sum = 0”

**Backward slice** with respect to “printf("%d\n", i)”
int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
        sum = add(sum, i);
        i = add(i, 1);
    }
    printf("%d\n", sum);
    printf("%d\n", i);
}

int add(int x, int y) {
    return x + y;
}

Forward slice with respect to “sum = 0”

Backward slice with respect to “printf("%d\n", i)”
int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
        sum = add(sum, i);
        i = add(i, 1);
    }
    printf("%d\n", sum);
    printf("%d\n", i);
}

int add(int x, int y) {
    return x + y;
}

Chop with respect to “sum = 0” and “printf("%d\n", i)”
Context-Sensitivity and Chopping

Enter main

\(\text{sum} = 0\)
\(\text{i} = 1\)
while \(\text{i} < 11\)
printf(sum)
printf(i)

Call add

\(\text{x}_{\text{in}} = \text{sum}\)
\(\text{y}_{\text{in}} = \text{i}\)
\(\text{sum} = \text{x}_{\text{out}}\)
\(\text{x}_{\text{in}} = \text{i}\)
\(\text{y}_{\text{in}} = 1\)
\(\text{i} = \text{x}_{\text{out}}\)

Enter add

\(\text{x} = \text{x}_{\text{in}}\)
\(\text{y} = \text{y}_{\text{in}}\)
\(\text{x} = \text{x} + \text{y}\)
\(\text{x}_{\text{out}} = \text{x}\)
Program Chopping

Given source $S$ and target $T$, what program points transmit effects from $S$ to $T$?

“Precise interprocedural chopping”
[Reps & Rosay FSE 95]
More Expressive Memory-Safety Policies

Opportunity: “Checking System Rules” [Engler]

\*v.unknown:\*

\[ (v = malloc(_)) == 0 \] \quad \rightarrow_t \quad v.null

\[ v = malloc(_) \] \quad \rightarrow_f \quad v.notNull

\*v.unknown, v.null, v.notNull:\*

\[ free(v) \] \quad \rightarrow \quad v.freed

\*v.freed:\*

\[ free(v) \] \quad \rightarrow \quad “double free!”

\[ v \] \quad \rightarrow \quad “use after free!”
More Expressive Access Policies

“No send after file read”

\[
\begin{align*}
\text{f.hasNotBeenRead:} & \\
[\text{read}(f)] & \rightarrow \text{f.hasBeenRead} \\
[\text{send}] & \rightarrow \text{f.hasNotBeenRead}
\end{align*}
\]

\[
\begin{align*}
\text{f.hasBeenRead:} & \\
[\text{read}(f)] & \rightarrow \text{f.has BeenRead} \\
[\text{send}] & \rightarrow \text{“send after file f was read!”}
\end{align*}
\]

A model-checking problem:
Control-flow graph © Security automaton?
The Need for Context Sensitivity

double free!
false alarm: invalid path!
The Need for Context Sensitivity

v = malloc()
call q
ret q
call q
ret q
enter p
exit p
free(v)
call q
ret q
enter q
exit q

OK!
A Richer Setting

[Engler]

- C code
- C compiler
- CFG + call graph
- model checker
  - OK
  - error report

[Our objective]

- Binary code
- typestate analyzer
- CFG + call graph + typestate info
- context-sensitive model checker
  - OK
  - error report
The Need for Context Sensitivity

- Information-flow analysis
- Virus scanning via model checking [Mihai]
- Buffer-overflow detection [Vinod]
- Dynamic traces of system calls [Jon]
Reps et al: Context Sensitivity

- Program slicing and chopping
  - Horwitz, Reps, & Binkley, TOPLAS 90
  - Reps, Horwitz, Sagiv, & Rosay, FSE 94

- Dataflow analysis
  - Reps, Horwitz, & Sagiv, POPL 95

- Survey of a collection of analyses
  - Reps, IST 98

- Context-sensitive model checking
  - Benedikt, Godefroid, & Reps, ICALP 01
Outline

• Slicing and Dependence Graphs
• CodeSurfer
• Interprocedural Slicing and Chopping
• Points-to Analysis
• Demos
Static-Analysis Issues

- Context-sensitive vs. context-insensitive
- Flow-sensitive vs. flow-insensitive
- Coping with pointers
Static-Analysis Issues

- Context-sensitive vs. context-insensitive
- Flow-sensitive vs. flow-insensitive
- Coping with pointers
  - CodeSurfer: flow-insensitive points-to analysis
  - TVLA: flow-sensitive analysis of heap-allocated storage
The Need for Pointer Analysis

```c
int main() {
    int sum = 0;
    int i = 1;
    int *p = &sum;
    int *q = &i;
    int (*f)(int, int) = add;
    while (*q < 11) {
        *p = (*f)(*p, *q);
        *q = (*f)(*q, 1);
    }
    printf("%d\n", *p);
    printf("%d\n", *q);
}

int add(int x, int y) {
    return x + y;
}
```

int main() {
    int sum = 0;
    int i = 1;
    int *p = &sum;
    int *q = &i;
    int (*f)(int, int) = add;
    while (*q < 11) {
        *p = (*f)(*p, *q);
        *q = (*f)(*q, 1);
    }
    printf("%d\n", *p);
    printf("%d\n", *q);
}

int add(int x, int y) {
    return x + y;
}
The Need for Pointer Analysis

```c
int main() {
    int sum = 0;
    int i = 1;
    int *p = &sum;
    int *q = &i;
    int (*f)(int, int) = add;
    while (i < 11) {
        sum = add(sum, i);
        i = add(i, 1);
    }
    printf("%d\n", sum);
    printf("%d\n", i);
}

int add(int x, int y) {
    return x + y;
}
```
Flow-Sensitive Dataflow Analysis

\[
\begin{align*}
V[n_1] &= \tau_1(V[\text{main}]) \otimes \tau_4(V[n_3]) \\
V[n_2] &= \tau_2(V[n_1]) \\
V[n_3] &= \tau_3(V[n_2]) \\
V[\text{exit}] &= \tau_5(V[n_1]) \\
V[\text{main}] &= \emptyset
\end{align*}
\]
Flow-Sensitive Points-To Analysis

\[ p \rightarrow q \]

\[ p = &q; \]

\[ *p = q; \]

\[ p = q; \]

\[ p = *q; \]

\[ *p = q; \]
Flow-Sensitive $\rightarrow$ Flow-Insensitive
Flow-Insensitive Points-To Analysis
[Andersen 94, Shapiro & Horwitz 97]

\[
p = \& q; \quad p \rightarrow q
\]

\[
p = q; \quad p \rightarrow r_1 \rightarrow q
\]

\[
p = * q; \quad p \rightarrow s_1 \rightarrow r_1 \rightarrow q
\]

\[
*p = q; \quad p \rightarrow r_1 \rightarrow s_1 \rightarrow q
\]
Flow-Insensitive Points-To Analysis

```plaintext
a = &e;
b = a;
c = &f;
*b = c;
d = *a;
```

Diagram:
```
a
  
  b
  
  c
  
  *b
  
  d
  
  e

  f
```
Shape Analysis: Formalizing “…”

A list:

\[ x \rightarrow \{p[x]\} \rightarrow \{r[x]\} \rightarrow \{p[y], r[x]\} \rightarrow \{r[y], r[x]\} \]

Informal:

Formal:
Shape Analysis: Formalizing “. . .”

\[ z = x \rightarrow \text{next} \]
The Need for Pointer/Shape Analysis

- Indirect function calls
  - (*fp)(e1,e2,e3);
  - What does fp point to?
  - [Mihai, Vinod, Jon]

- “Shape” of heap-allocated data structures
  - Information flow through data structures [Reps]
  - Relating behavior patterns [Mulhern]
Limitations of Static Analysis

- Undecidable questions (yes/no)
- Safe
  - yes/maybe
  - no/maybe
  - yes/no/maybe
- Time/space tradeoffs
  - context-sensitive/context-insensitive
  - flow-sensitive/flow-insensitive
  - Find a sweet spot
Static Analysis of Binaries

+ Single language to deal with
- Some “inherent” obfuscation
  - operations on registers (vs. variables)
  - very weak types (8-bit, 16-bit, 32-bit quantities)
  - jump tables
  - . . .
No text content available from the image.
InBuff = (unsigned char *)from_buf;
outBuff = (unsigned char *)to_buf;
do_decomp = action;

if (do_decomp == 0) {
    compress();
    #ifdef DEBUG
    if (verbose)
        dump_tab();
    #endif /* DEBUG */
    } else {
    /* Check the magic number */
    if (nomagic == 0) {
        if ((getbyte() != (magic_header[0] & 0xFF))
            || (getbyte() != (magic_header[1] & 0xFF))) {
            fprintf(stderr, "stdin: not in compressed format\n");
            exit(1);
        }
        maxbits = getbyte(); /* set -b from file */
        block_compress = maxbits & BLOCK_MASK;
        maxbits &= BIT_MASK;
        maxmaxcode = 1 << maxbits;
        ssize = 100000;
        /* assume stdin large for USERMEM */
        if (maxbits > BITS) {
            fprintf(stderr, "stdin: compressed with %d bits, can only handle %d bits
maxbits, BITS\n");
            exit(1);
        }
    }#ifndef DEBUG
decompress();
#else
}
Browsing a Dependence Graph

Pretend this is your favorite browser

What does clicking on a link do?

You get a new page

Or you move to an internal tag
Program point: `if (do_decomp == 0) { compress(); #ifdef ...`  

Program point kind: control-point  

Function: `spec_select_action`  

File: `/afs/cs.wisc.edu/p/wpis/imports/slicing-tools/CodeSurfer/codes`  

Data Predecessors:  

- `[expression] do_decomp = action`  

Data Successors: none  

Control Predecessors:  

- `[entry] spec_select_action entry point`  

Control Successors:  

- `[call-site] compress()`  
- `[control-point] if (nomagic == 0)`  
- `[call-site] decompress()`  

Variables:  

- `(Global) do_decomp`
InBuff = (unsigned char *)from_buf;
outBuff = (unsigned char *)to_buf;
do_decomp = action;

if (do_decomp == 0) {
    compress();
    #ifdef DEBUG
      if (verbose)
          dump_tab();
    #endif /* DEBUG */
} else {
    /* Check the magic number */
    if (nomagic == 0) {
        if ((getbyte() != (magic_header[0] & 0xFF))
            || (getbyte() != (magic_header[1] & 0xFF))) {
            fprintf(stderr, "stdin: not in compressed format
");
            exit(1);
        }
        maxbits = getbyte(); /* set -b from file */
        block_compress = maxbits & BLOCK_MASK;
        maxbits &= BIT_MASK;
        maxmaxcode = 1 << maxbits;
        fsize = 1000000; /* assume stdin large for USERMEM */
        if (maxbits > BITS) {
            fprintf(stderr,
                "stdin: compressed with %d bits, can only handle %d bits
" maxbits, BITS);
            exit(1);
        }
    }
    #ifndef DEBUG
    decompress();
    #else
    
    #endif  /* DEBUG */
}
```c
int main(int argc, char *argv[])
{
    int count, i, oper;
    int comp_count, new_count;
    char start_char;
    int N;
    char C;

    printf("SPEC 129.compress harness\n");
    scanf("%i %c %i", &count, &start_char, &seedi);
    printf("Initial File Size:%i Start character:%c\n", count, start_char);
    fill_text_buffer(count, start_char, orig_text_buffer);
    for (i = 1; i <= 25; i++)
    {
        new_count=add_line(orig_text_buffer, count, i, start_char);
        count=new_count;
        oper=CMPRESS;
        printf("The starting size is: %d\n", count);
        comp_count=spec_select_action(orig_text_buffer, count, oper, comp_text_buffer);
        printf("The compressed size is: %d\n", comp_count);
        oper=UNCOMPRESSION;
        new_count=spec_select_action(comp_text_buffer, comp_count, oper, new_text_buffer);
        printf("The compressed/uncorpressed size is: %d\n", new_count);
        compare_buffer(orig_text_buffer, count, new_text_buffer, new_count);
    }
}```
CodeSurfer System Architecture

Front Ends
- FE1
- FE2
- FE3
- FE4
- FE5

EDG
- ANSI C
- C++
- x86 binary code

IdaPro
- Java
- UML (Rose/RT)
- Verilog
- VHDL
- Jovial

Pre-IR
- ASTs
- symbol table
- per-statement variable usage
- pointer uses
- CFGs

Builder

IR

Analysis operations
- points-to graph
- call multi-graph
- GMOD / GREF
- program-wide variable usage
- PDGs / SDG

Synthesis operations
- Precise interprocedural
  - predecessors and successors
  - slicing and chopping
- Boolean operations on sets of PDG nodes
- model checking
- constraint generation

API

C client code

Scheme scripts

GUI

Other infrastructure: command-line, preprocessor, include-file instances, library, and loader support
Demos

wc

x86

sum

information flow

hello

script example