Buffer Overflow Analyzer using LLVM

Jonathan Lam and Daniel Tsarev

Motivation

Lab 1
BUFFER OVERFLOW BAD

Threat Model

Buffer overflow compromise the whole system,

S ✓
T ✓
R ✓
I ✓
D ✓
E ✓

Risk Analysis

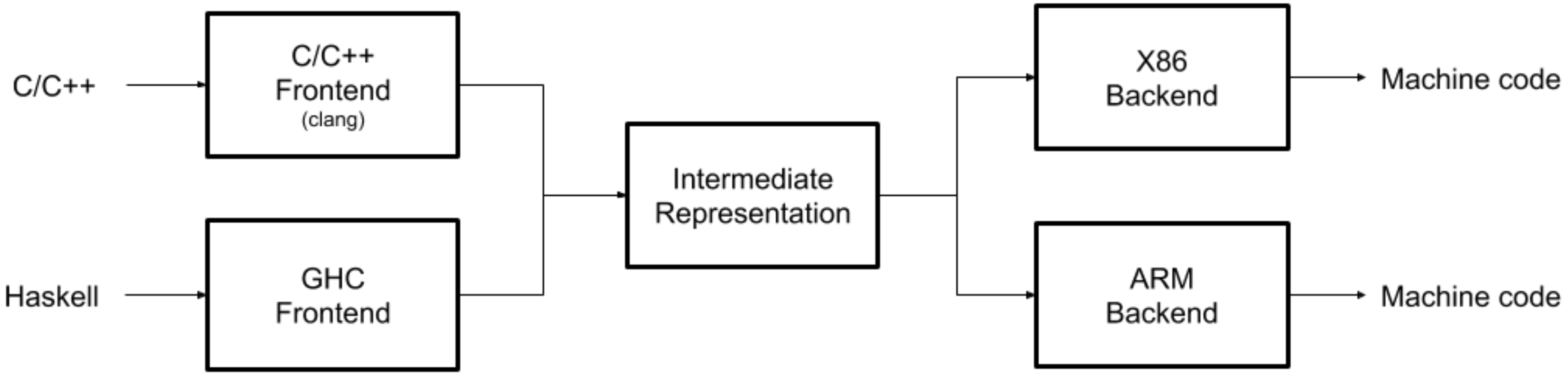
Downtime costs **$$$**

- Gartner, estimates a $5,600 cost per minute of downtime of an IT system

Data leaks can be crippling for a business

- LOTS OF $$$$

What is LLVM?



C/C++
Haskell
C/C++
Frontend
(clang)
GHC
Frontend
Intermediate
Representation
X86
Backend
ARM
Backend
Machine code
Machine code

Intermediate Representation (IR)

- ❑ Instructions
- ❑ Basic Blocks
- ❑ Control Flow Graphs

Code

```
W := 0;
X  := W + 1;
Y  := 2;
If (X > Z) {
    Y = X;
    X++;
} else {
    Y := Z;
}
W := X + Z;
```

B1

```
W = 0;
X = W + 1;
Y = 2;
If (X> Z)
```

B2

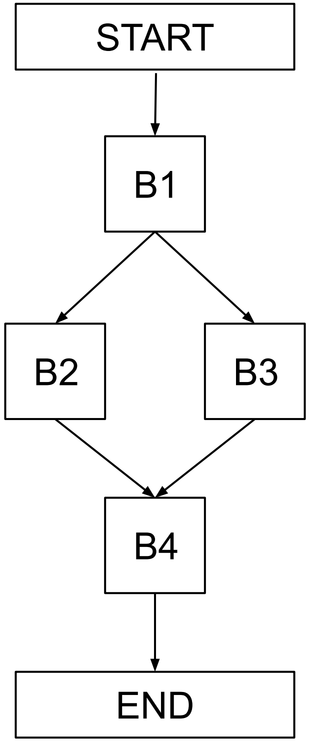
```
Y = X;
X++;
```

B3

```
Y = Z;
```

B4

```
W = X + Z;
```



Intro. to Data-flow Analysis (DFA)

Better with an example

DFA Example: Zero Analysis

```
X := 0
Y := 1
Z := Y
Y := Z + X
X := Y - Z

X / Y
Y / X
```

DFA EXAMPLE: ZERO ANALYSIS

X := 0
Y := 1
Z := Y
Y := Z + X
X := Y - Z

X / Y
Y / X

1. Indeterminate values

DFA EXAMPLE: ZERO ANALYSIS

```
X := 0
Y := 1
Z := Y
Y := Z + X
X := Y - Z

X / Y
Y / X
```

1. Indeterminate values

2. Loss of precision -- conservative

DFA EXAMPLE: ZERO ANALYSIS

```
X := 0
Y := 1
Z := Y
Y := Z + X
X := Y - Z

X / Y
Y / X
```

1. Indeterminate values
2. Loss of precision -- conservative
3. Define invariant and transition fn

INTRO TO DATAFLOW ANALYSIS

Trace some **invariant/property/thing to analyze** throughout the program

We track the analysis over **variables**; the analysis is expressed as a **abstract value**

Abstract values form a **lattice** (partial order); dataflow analysis generalizes all possible abstract values

Decide how each instruction affects the property -- define a **transition function**

Other Data-flow Analyses

Zero analysis: detect divide by zero

Constant propagation: optimization, convenient compile-time computations

Live variables: optimization of registers

Reaching definitions: detect uninitialized variables

Buffer origin (our analysis): detect some cases of buffer overflows

BUFFER ORIGIN ANALYSIS

Invariant: set of static buffer variables that a buffer can refer to

Transition function: pointer assignment, load/store operations

BUFFER ORIGIN DATAFLOW ANALYSIS (BODA)

Invariant: set of static buffer variables that a buffer can refer to

Transition function: pointer assignment, load/store operations

Assumptions:

- Only analyzing attention to stack-allocated fixed-size buffers
- No pointer arithmetic
- Function calls do not affect dataflow analysis (transition fn is identity map)
    - Probably an OK assumption

BUFFER ORIGIN DATAFLOW ANALYSIS (BODA)

```
char *p, d[512], s[512];
p = d;
strcpy(p, s);
```

WORKLIST ALGORITHM

function BODAWORKLIST(f)
    $B \leftarrow$ basic blocks in f
    mark B_0 dirty
    $B_d \leftarrow B[0]$
    while B_d is not empty **do**
        $b \leftarrow$ pop from B_d
        if b is dirty **then**
            analyze(b)
            mark b clean
            if b not at fixpoint **then**
                for all $b_s \in B : b_s$ succeeds b **do**
                    mark b_s dirty
                    insert b_s into B_d
                end for
            end if
        end if
    end while
end function

INTRO TO INTERPROCEDURAL ANALYSIS

```
int f(char *d, char *s) {
	strcpy(d, s);
}

int main() {
	char d1[512], s1[512], d2[1024], s2[1024];
	f(d1, s1);
	f(d2, s2);
}
```

INTRO TO INTERPROCEDURAL ANALYSIS

Loss of precision is common in dataflow analysis -- analysis captures global information and doesn't consider **context** due to only traveling down one branch

Analyzing branches separately is **expensive** but we can provide better context-sensitivity

Dataflow analysis within functions (many instructions, efficiency) but interprocedural call-graph tracing (few fncalls, context-sensitive)

TRACING THE CALL GRAPH

```
function TraceCallGraphRec(f, O)
    for all function invocations i_g() ∈ f do
        p_g ← [O_{i_g}/O]p_g
        if g is dangerous then
            warn about dangerous usage
        else if g is user-defined then
            TraceCallGraphRec(g, p_g)
        end if
    end for
end function

function TraceCallGraph(M)
    f_m ← main function of M
    TraceCallGraphRec(f_m, {})
end function
```

Handling (mutual) recursion: convergence along call stack

DEMO

FUTURE WORK

- Examine more complicated sample programs
- Improve presentation of analysis
- Finish interprocedural analysis implementation
- Handle missing assumptions
- Other dataflow analyses to augment BODA (e.g., constant propagation)
- Study implementation of existing static analyzers

QUESTIONS?

GitHub

Program Analysis textbook