Quantifying Dataflow Analysis with Gradients in LLVM

Gabriel Ryan[1], *Abhishek Shah*[1], Dongdong She[1],

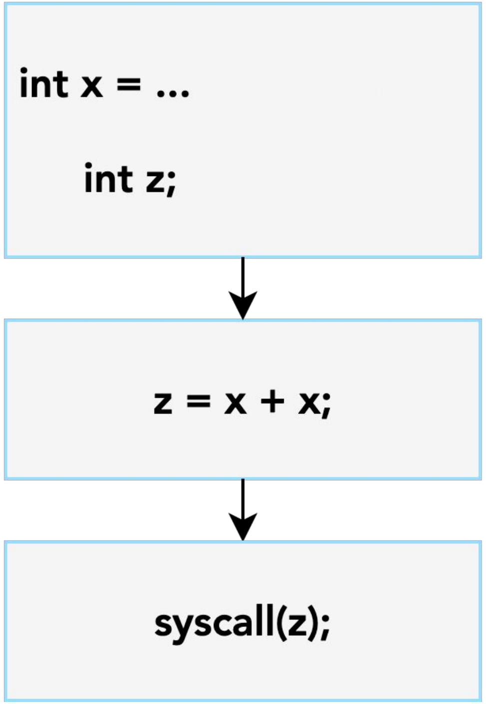
Koustubha Bhat[2], Suman Jana[1]

1: Columbia University

2: Vrije Universiteit

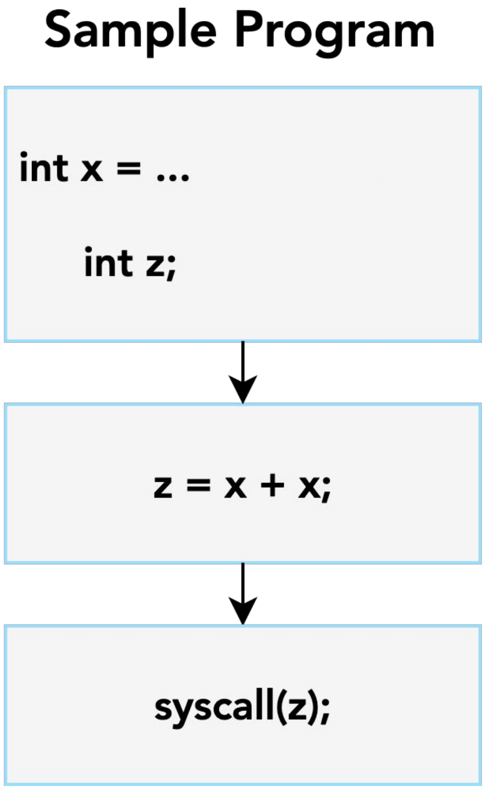
Dataflow Analysis

Sample Program



Dataflow Analysis

Is there a dataflow between variables x and z?

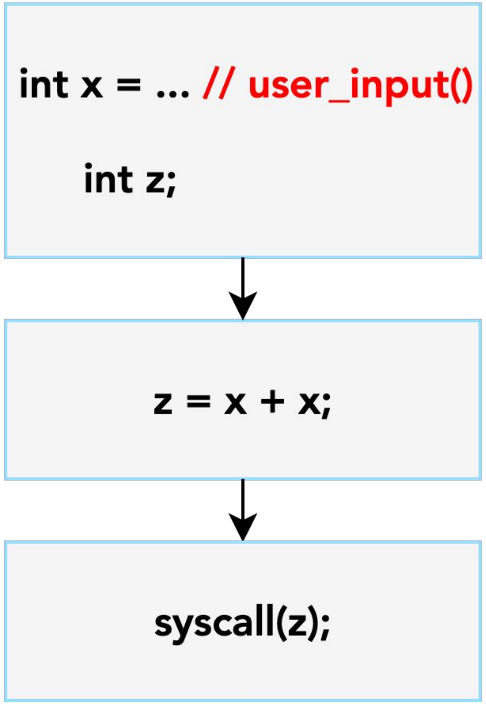


Dataflow Analysis

Is there a dataflow between variables x and z?

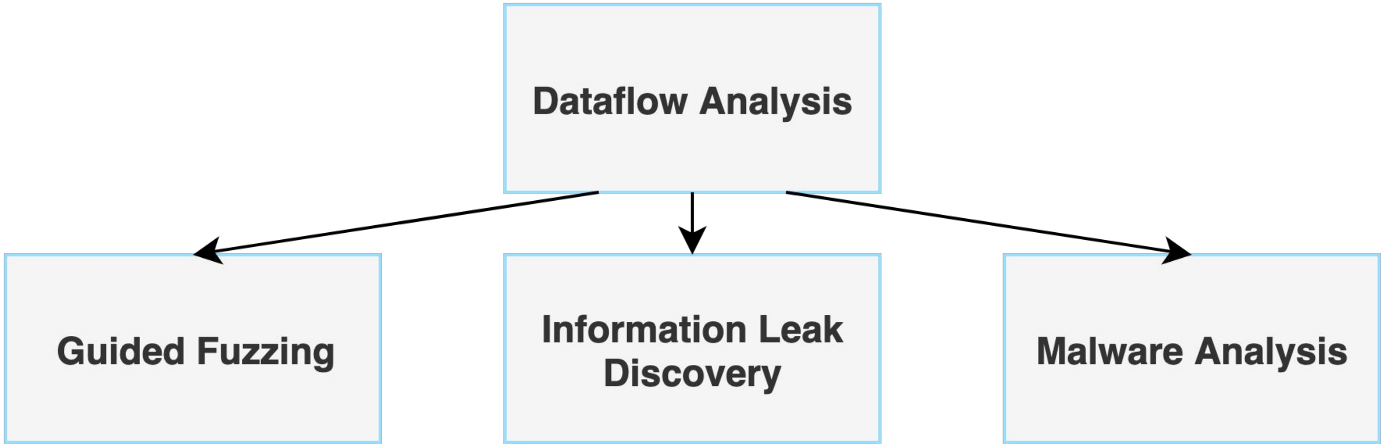
Vulnerability Analysis

Sample Program



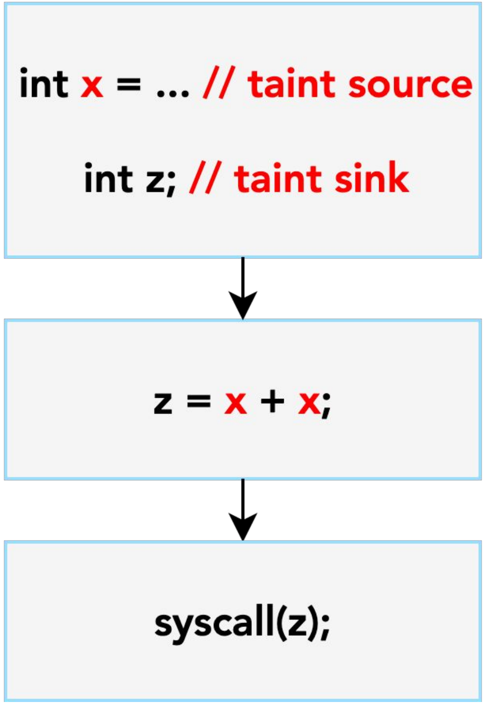
Dataflow Analysis

Common building block for program analysis



Dynamic Taint Analysis (DTA)

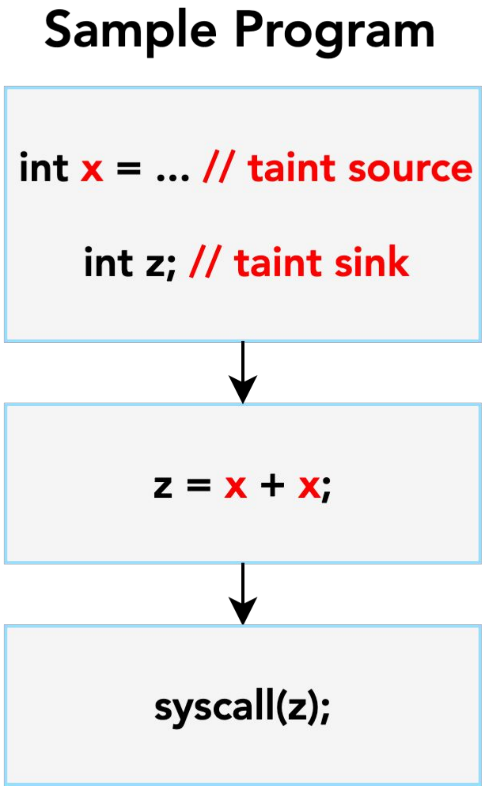
Sample Program



Dynamic Taint Analysis (DTA)

Dataflow Encoding

- Boolean labels represent absence or presence of taint



Dynamic Taint Analysis (DTA)

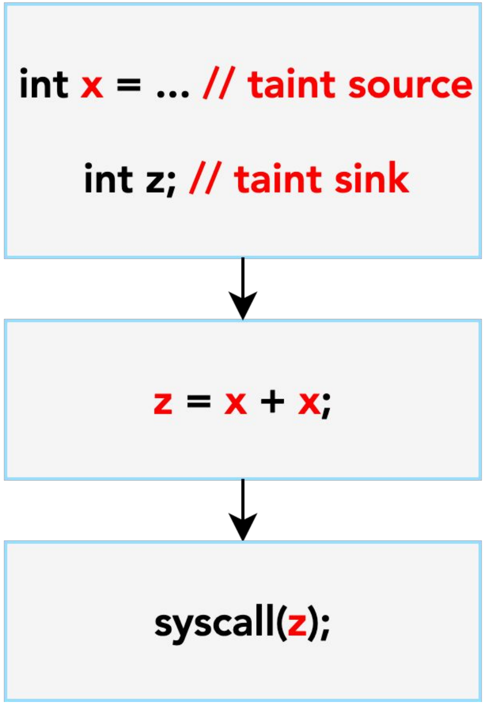
Dataflow Encoding

- Boolean labels represent absence or presence of taint

Per-operation rules propagate taint

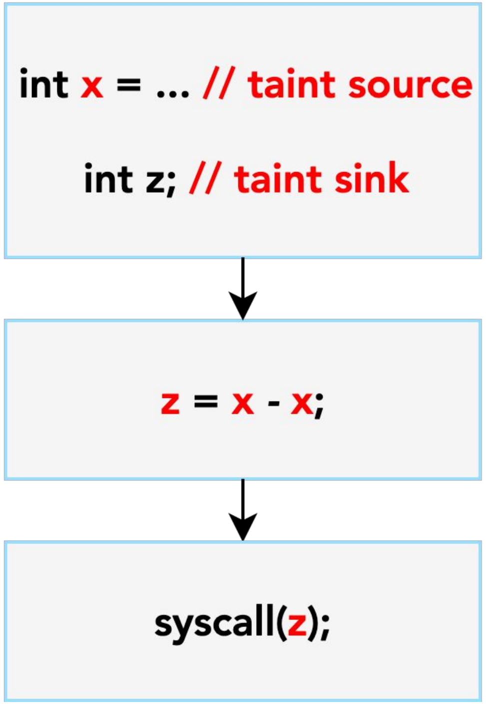
- Example Rule for Add/Subtract operation:
  - If input operands carry taint, output operand carries taint too

Sample Program



Limitation 1: Imprecise Rules

Sample Program

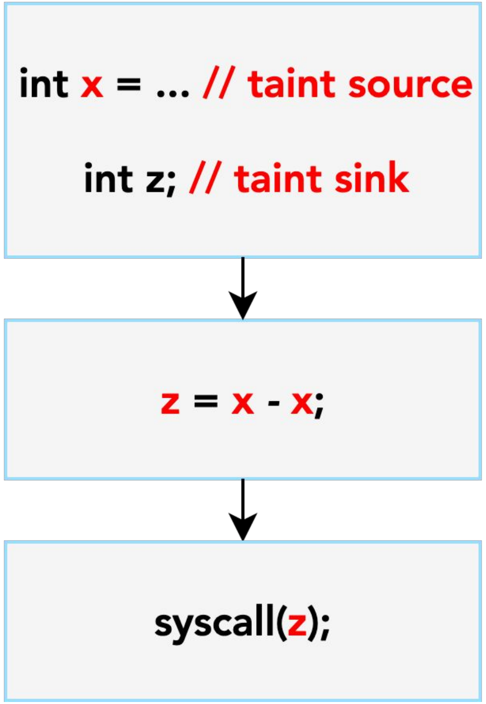


Limitation 1: Imprecise Rules

Subtraction rule introduces false positives

- z is incorrectly tainted as x - x is zero (i.e. no dataflow from x to z)

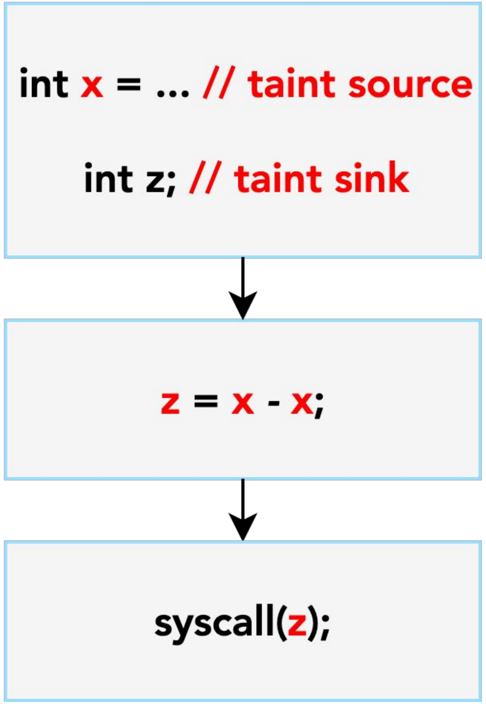
Sample Program



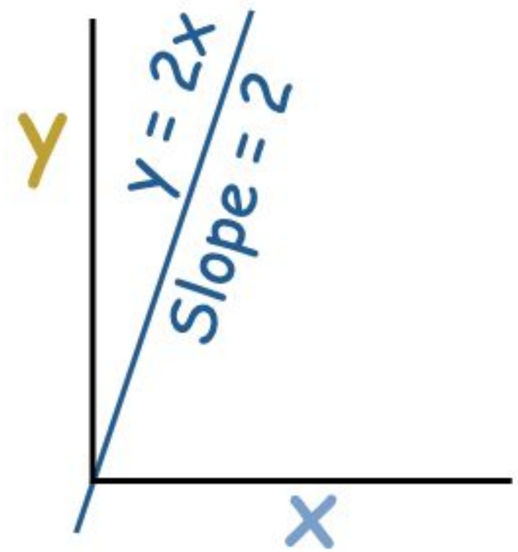
Limitation 2: Boolean Taint Labels

Boolean taint labels cannot
- Quantify dataflows between x and z
- Order amount of influence of each dataflow

Sample Program



Gradients

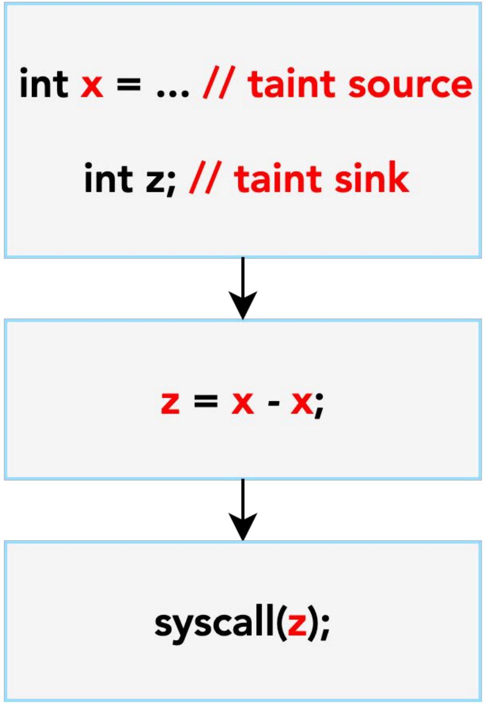


New Approach to Dataflow Analysis

Key Insight

- Gradients track influence of inputs on outputs

Sample Program



New Approach to Dataflow Analysis

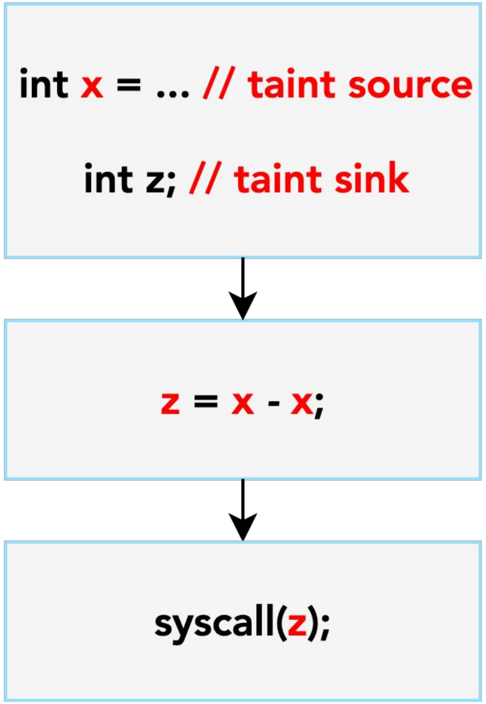
Key Insight

- Gradients track influence of inputs on outputs

Why gradients?

- Gradients quantify dataflows
- Precise composition and rules <u>over differentiable operations</u> due to chain rule of calculus

Sample Program

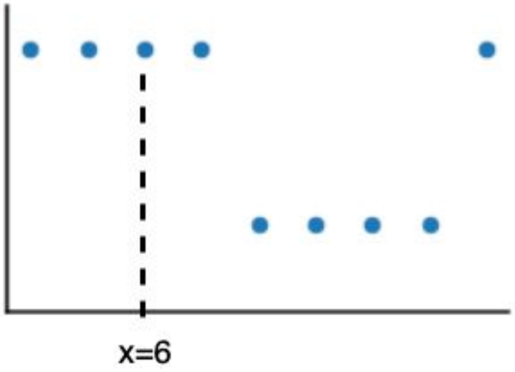


Problem: Nondifferentiable Operator

Programs contain nondifferentiable operators

- Bitwise And

```
int f(int x) {
    return x & 4
}
```

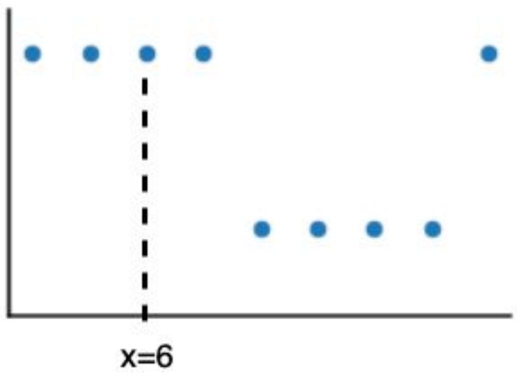


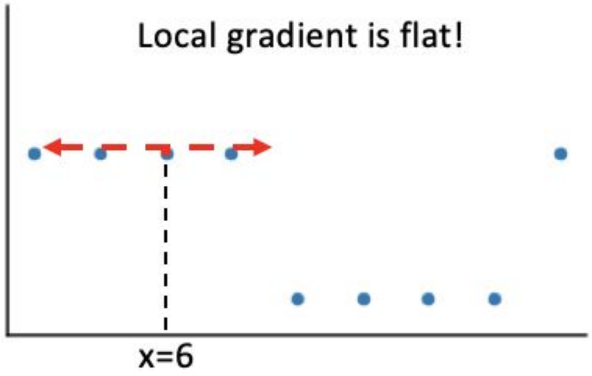
Problem: Nondifferentiable Operator

Programs contain nondifferentiable operators

- Bitwise And

```
int f(int x) {
    return x & 4
}
```





Solution: Proximal Gradients

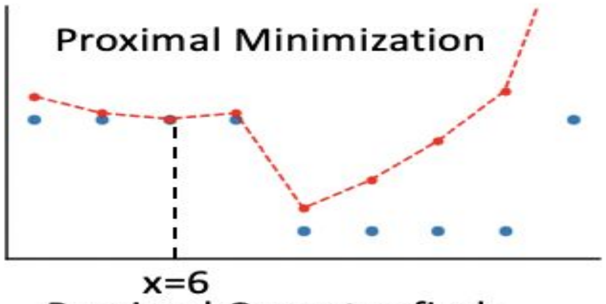
How to compute gradient of nondifferentiable operator?

- Proximal gradients find local minima in region to approximate the gradient

Solution: Proximal Gradients

How to compute gradient of nondifferentiable operator?

- Proximal gradients find local minima in region to approximate the gradient

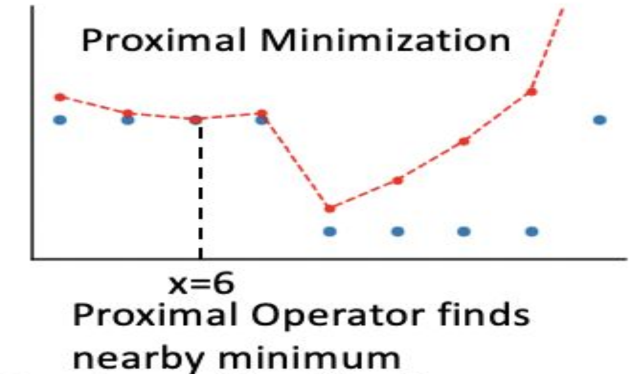


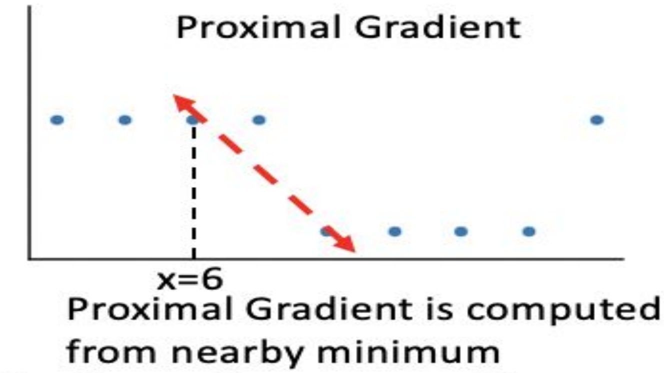
Proximal Operator finds nearby minimum

Solution: Proximal Gradients

How to compute gradient of nondifferentiable operator?

- Proximal gradients find local minima in region to approximate the gradient





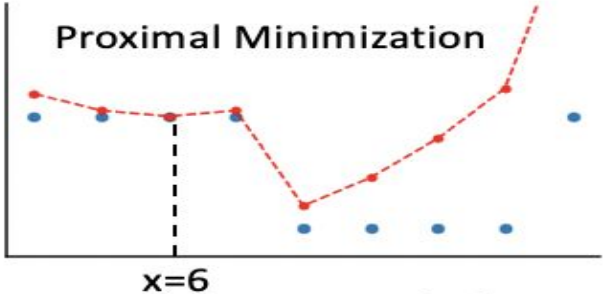
Solution: Proximal Gradients

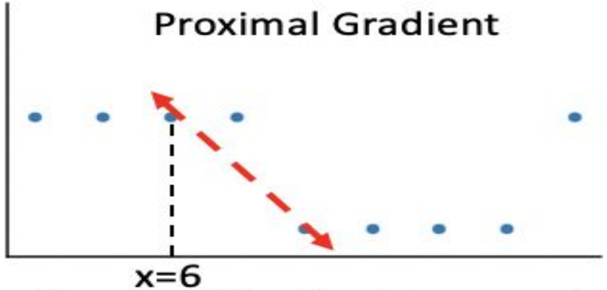
How to compute gradient of nondifferentiable operator?

- Proximal gradients find local minima in region to approximate the gradient

Why Proximal Gradients?

- Region can be bounded to make computation tractable





COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

Implementation

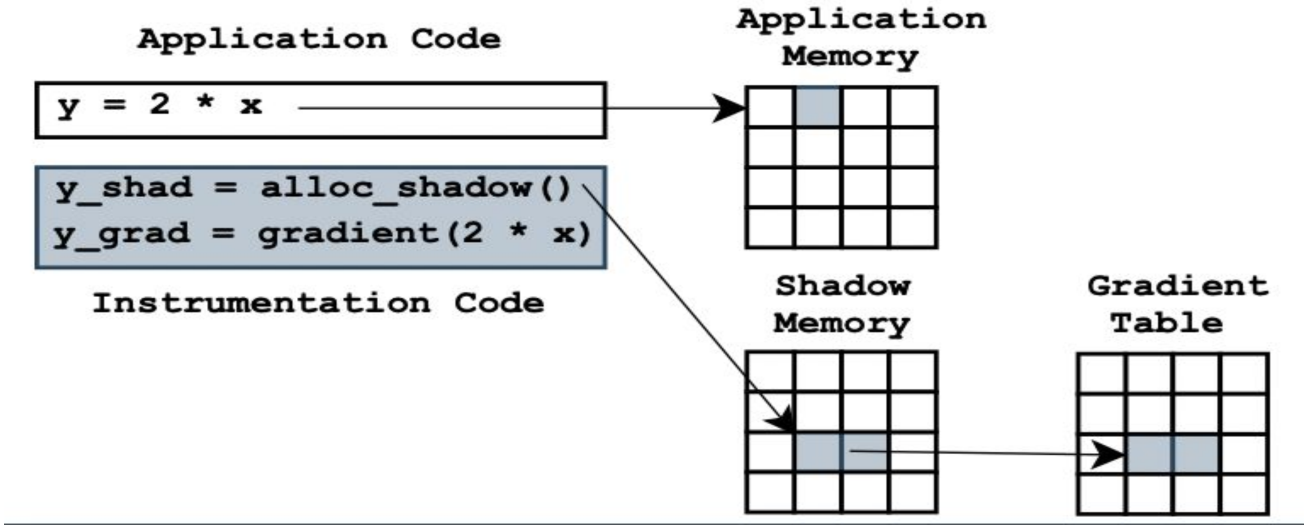
Proximal Gradient Analysis implemented in LLVM

- Based on DataFlowSanitizer, LLVM's state-of-the-art DTA tool

Implementation

Proximal Gradient Analysis implemented in LLVM

- Based on DataFlowSanitizer, LLVM's state-of-the-art DTA tool



Implementation

Proximal Gradient Analysis implemented in LLVM

- Based on DataFlowSanitizer, LLVM's state-of-the-art DTA tool

Main idea 1 (instrumentation)

  - Instrument operations to propagate gradients

Implementation

Proximal Gradient Analysis implemented in LLVM

- Based on DataFlowSanitizer, LLVM's state-of-the-art DTA tool

Main idea 1 (instrumentation)

- Instrument operations to propagate gradients

Main idea 2 (gradient storage)

- Store gradients for each variable in shadow memory

Example LLVM IR

```
int x;
int z;

z = x + x;
```

COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

Example LLVM IR

```
/* variable allocation */

 %x = alloca i32, align 4 // int x;

 %z = alloca i32, align 4 // int z;
```

```
int x;
int z;

z = x + x;
```

Example LLVM IR

```
/* variable allocation */

%x = alloca i32, align 4 // int x;

%z = alloca i32, align 4 // int z;

/* load operations */

%3 = load i32, i32* %x, align 4

%5 = load i32, i32* %x, align 4
```

```
int x;
int z;

z = x + x;
```

Example LLVM IR

```
/* variable allocation */

%x = alloca i32, align 4 // int x;

%z = alloca i32, align 4 // int z;

/* load operations */

%3 = load i32, i32* %x, align 4

%5 = load i32, i32* %x, align 4

/* add instruction */

%add = add nsw i32 %3, %5 // z = x + x;

store i32 %add, i32* %z, align 4
```

```
int x;
int z;

z = x + x;
```

Example LLVM IR

```
/* variable allocation */
%0 = alloca i16 // x_shadow
%x = alloca i32, align 4 // int x;
%1 = alloca i16 // z_shadow
%z = alloca i32, align 4 // int z;

/* load operations */

%3 = load i32, i32* %x, align 4

%5 = load i32, i32* %x, align 4

/* add instruction */

%add = add nsw i32 %3, %5 // z = x + x;

store i32 %add, i32* %z, align 4
```

```
int x;
int z;

z = x + x;
```

Example LLVM IR

```
/* variable allocation */
 %0 = alloca i16 // x_shadow
 %x = alloca i32, align 4 // int x;
 %1 = alloca i16 // z_shadow
 %z = alloca i32, align 4 // int z;

 /* load operations */
 %2 = load i16, i16* %0
 %3 = load i32, i32* %x, align 4
 %4 = load i16, i16* %0
 %5 = load i32, i32* %x, align 4

 /* add instruction */

 %add = add nsw i32 %3, %5 // z = x + x;

 store i32 %add, i32* %z, align 4
```

```
int x;
int z;

z = x + x;
```

Example LLVM IR

```
/* variable allocation */
%0 = alloca i16 // x_shadow
%x = alloca i32, align 4 // int x;
%1 = alloca i16 // z_shadow
%z = alloca i32, align 4 // int z;

/* load operations */
%2 = load i16, i16* %0
%3 = load i32, i32* %x, align 4
%4 = load i16, i16* %0
%5 = load i32, i32* %x, align 4

/* add instruction */
%6 = call zeroext i16 @__dfsan_union(…%2, %3, %4, %5…)
%add = add nsw i32 %3, %5 // z = x + x;
store i16 %6, i16* %1
store i32 %add, i32* %z, align 4
```

```
int x;
int z;

z = x + x;
```

Instrumentation: Compile-time

Instrument operations with InstVisitor class

- For example, *visitBinaryOperator*() inserts a call to runtime library that computes gradient dynamically based on opcode

Instrumentation: Compile-time

Instrument operations with InstVisitor class

- For example, *visitBinaryOperator*() inserts a call to runtime library that computes gradient dynamically based on opcode

What if operations cannot be instrumented?

- Create wrapper for original function that propagates dataflow
- Instrumentation inserts a call to wrapper instead of original function

Instrumentation: Compile-time

Instrument operations with InstVisitor class

- For example, *visitBinaryOperator*() inserts a call to runtime library that computes gradient dynamically based on opcode

What if operations cannot be instrumented?

- Create wrapper for original function that propagates dataflow
- Instrumentation inserts a call to wrapper instead of original function

Similarly instrument functions and their arguments

Instrumentation: Runtime

Dynamically propagate dataflow

- Bitwise And operation instrumentation finds proximal gradient with concrete values

Instrumentation: Runtime

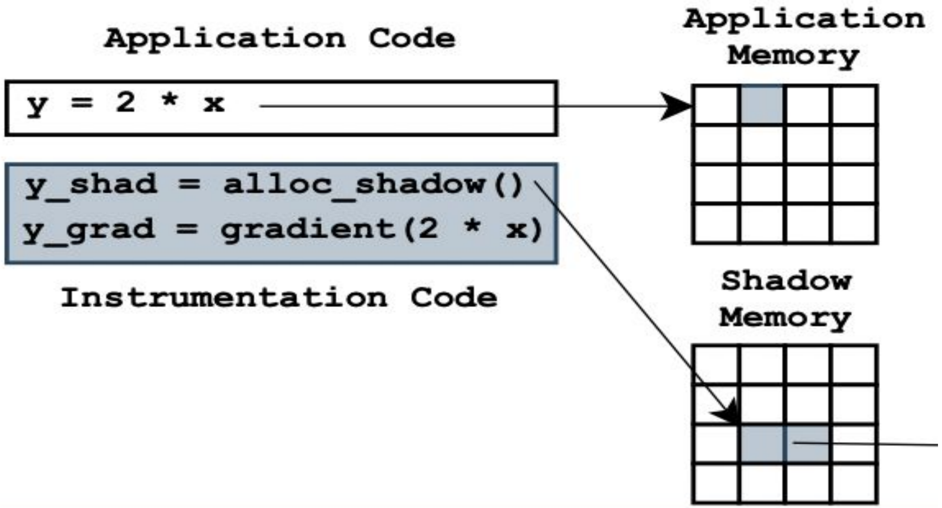
Dynamically propagate dataflow

- Bitwise And operation instrumentation finds proximal gradient with concrete values

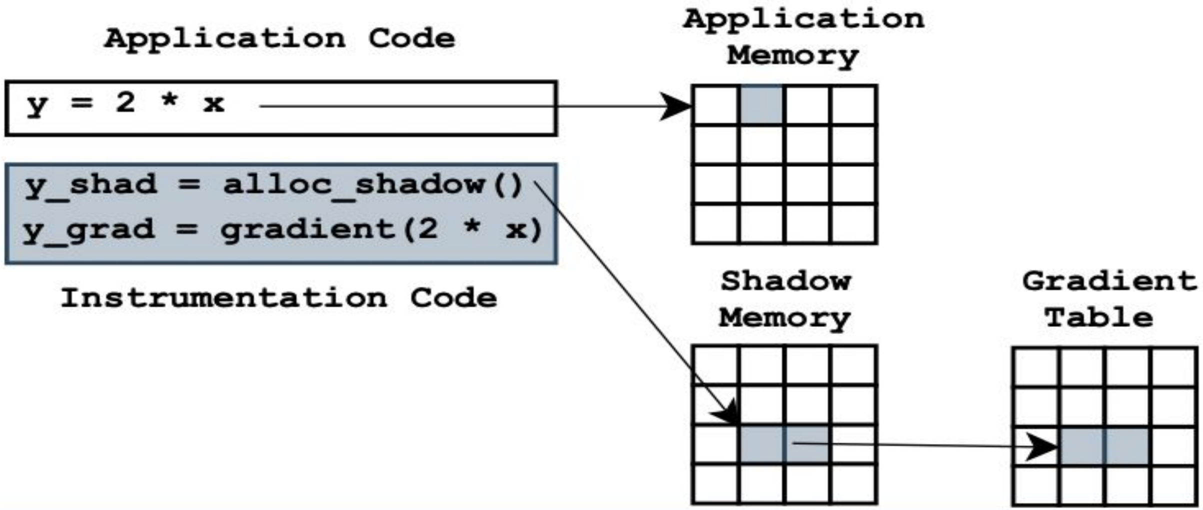
Minimal runtime overhead

- Based on compile-time instrumentation vs runtime instrumentation

Gradient Storage: Shadow Memory



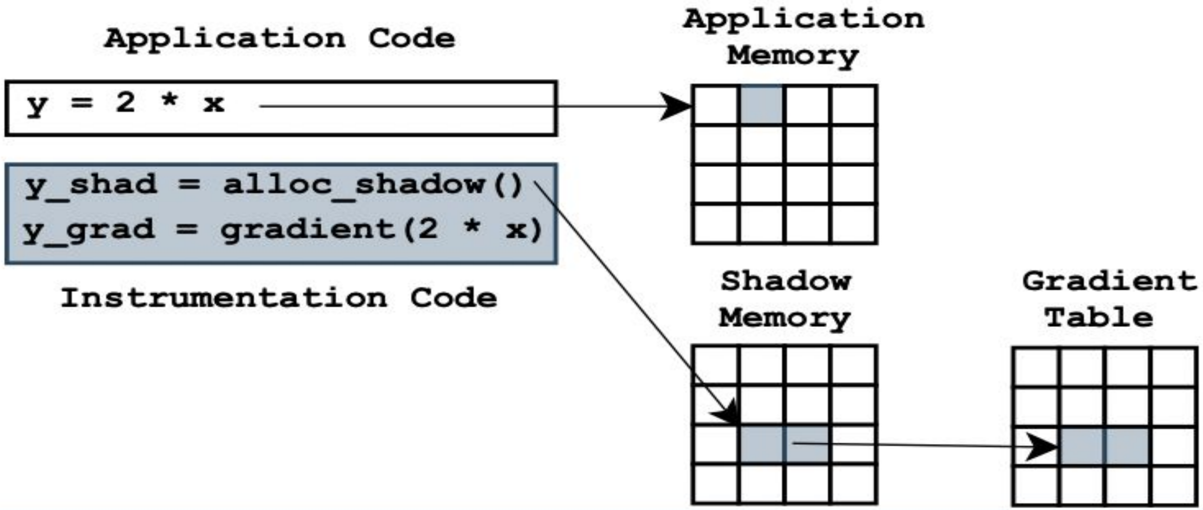
Gradient Storage: Shadow Memory



Gradient Storage: Shadow Memory

Gradient sharing with indirection

- Every variable has associated shadow memory with label
- Label indexes into a table holding data structure
- Enables sharing gradients across multiple variables



Evaluation: Accuracy

Better accuracy on 7 real-world parser programs

- Our tool (grsan) achieves up to 33% better dataflow accuracy than DataFlowSanitizer (dfsan)

	dfsan			grsan		
	Prec.	Rec.	F1	Prec.	Rec.	F1
minigzip	0.29	0.60	0.39	0.63	0.51	**0.57**
djpeg	0.22	1.00	0.37	0.60	0.83	**0.69**
mutool	0.63	0.61	0.62	0.86	0.51	**0.64**
xmllint	0.62	0.99	0.76	0.94	0.91	**0.92**
objdump	0.37	0.93	0.52	0.66	0.77	**0.71**
strip	0.20	0.96	0.33	0.50	0.86	**0.63**
size	0.37	0.95	0.53	0.62	0.91	**0.74**

Evaluation: Bug Finding

We find 23 previously undiscovered bugs

- Track gradients for arguments to known vulnerable operations such as bitwise and memory copy operators
- As an example, we altered an input byte with high gradient to a shift operator to trigger an overflow

Library	Test Program	Integer Overflow	Memory Corruption
libjpeg-9c	djpeg	2	3
mupdf-1.14.0	mutool show	1	0
binutils-2.30	size	0	1
	objdump -xD	0	9
	strip	0	7

Key Takeaways

DataflowSanitizer enables many dynamic analyses

- Our dynamic analysis propagates gradients with minimal changes

Nonsmooth optimization and program analysis connections

Quantifying Dataflow Analysis with Gradients in LLVM

Gabriel Ryan[1], *Abhishek Shah*[1], Dongdong She[1],

Koustubha Bhat[2], Suman Jana[1]

1: Columbia University

2: Vrije Universiteit

Columbia University in the City of New York