Quantifying Dataflow Analysis with Gradients in LLVM

Gabriel Ryan\textsuperscript{1}, Abhishek Shah\textsuperscript{1}, Dongdong She\textsuperscript{1}, Koustubha Bhat\textsuperscript{2}, Suman Jana\textsuperscript{1}

1: Columbia University
2: Vrije Universiteit
Dataflow Analysis

Sample Program

```plaintext
int x = ...

int z;

z = x + x;

syscall(z);
```
Dataflow Analysis

Is there a dataflow between variables x and z?

Sample Program

```plaintext
int x = ...

int z;

z = x + x;

syscall(z);
```
Dataflow Analysis

Is there a dataflow between variables x and z?

Vulnerability Analysis

Sample Program

```c
int x = ... // user_input()
int z;

z = x + x;
syscall(z);
```
Dataflow Analysis

Common building block for program analysis
Dynamic Taint Analysis (DTA)

Sample Program

```c
int x = ... // taint source

int z; // taint sink

z = x + x;

syscall(z);
```
Dynamic Taint Analysis (DTA)

Dataflow Encoding
- Boolean labels represent absence or presence of taint

Sample Program
```c
int x = ... // taint source
int z; // taint sink

z = x + x;
syscall(z);
```
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
```plaintext
int x = ... // taint source
int z; // taint sink

z = x + x;

syscall(z);
```
Limitation 1: Imprecise Rules

Sample Program

```java
int x = ... // taint source

int z; // taint sink

z = x - x;

syscall(z);
```
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:
```c
int x = ... // taint source
int z; // taint sink

z = x - x;
sySCALL(z);
```
Limitation 2: Boolean Taint Labels

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

Sample Program

```java
int x = ... // taint source

int z; // taint sink

z = x - x;

syscall(z);
```
Gradients
Key Insight
- Gradients track influence of inputs on outputs

Sample Program

```plaintext
int x = ... // taint source

int z; // taint sink

z = x - x;

syscall(z);
```
Key Insight
- Gradients track influence of inputs on outputs

Why gradients?
- Gradients quantify dataflows
- Precise composition and rules over differentiable operations due to chain rule of calculus
Problem: Nondifferentiable Operator

Programs contain nondifferentiable operators
- Bitwise And

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

![Diagram showing the behavior of the function `f(x)` for `x=6`. Points on the curve indicate the function values at specific `x` values.](image-url)
Problem: Nondifferentiable Operator

Programs contain nondifferentiable operators
- Bitwise And

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

Local gradient is flat!
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
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
Implementation

Proximal Gradient Analysis implemented in LLVM
- Based on DataFlowSanitizer, LLVM’s state-of-the-art DTA tool
Proximal Gradient Analysis implemented in LLVM
- Based on DataFlowSanitizer, LLVM’s state-of-the-art DTA tool

Application Code

\[ y = 2 \times x \]

Instrumentation Code

\[ y\textunderscore\text{shad} = \text{alloc\_shadow}() \]
\[ y\textunderscore\text{grad} = \text{gradient}(2 \times x) \]

Application Memory

Shadow Memory

Gradient Table
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
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;
Example LLVM IR

/* variable allocation */

%0 = alloca i16 // x_shadow
%1 = alloca i16 // z_shadow

int x;
int z;

z = x + x;
Example LLVM IR

/* variable allocation */

%0 = alloca i16 // x_shadow

%1 = alloca i16 // z_shadow

/* 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

int x;
int z;

z = x + x;
Example LLVM IR

/* variable allocation */

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

/* load operations */

%2 = load i16, i16* %0
%2 = load i32, i32* %x, align 4
%3 = load i16, i16* %0
%3 = 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;
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

Application Code

\[ y = 2 \times x \]

Instrumentation Code

\[ y_{\text{shad}} = \text{alloc\_shadow}() \]
\[ y_{\text{grad}} = \text{gradient}(2 \times x) \]
Gradient Storage: Shadow Memory

y = 2 * x

y_shad = alloc_shadow()
y_grad = gradient(2 * x)

Instrumentation Code

Application Code

Application Memory

Shadow Memory

Gradient Table
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)

<table>
<thead>
<tr>
<th></th>
<th>dfsan</th>
<th></th>
<th>grsan</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Prec.</td>
<td>Rec.</td>
<td>F1</td>
</tr>
<tr>
<td>minigzip</td>
<td>0.29</td>
<td>0.60</td>
<td>0.39</td>
</tr>
<tr>
<td>djjpeg</td>
<td>0.22</td>
<td>1.00</td>
<td>0.37</td>
</tr>
<tr>
<td>mutool</td>
<td>0.63</td>
<td>0.61</td>
<td>0.62</td>
</tr>
<tr>
<td>xmlllint</td>
<td>0.62</td>
<td>0.99</td>
<td>0.76</td>
</tr>
<tr>
<td>objdump</td>
<td>0.37</td>
<td>0.93</td>
<td>0.52</td>
</tr>
<tr>
<td>strip</td>
<td>0.20</td>
<td>0.96</td>
<td>0.33</td>
</tr>
<tr>
<td>size</td>
<td>0.37</td>
<td>0.95</td>
<td>0.53</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Library</th>
<th>Test Program</th>
<th>Integer Overflow</th>
<th>Memory Corruption</th>
</tr>
</thead>
<tbody>
<tr>
<td>libjpeg-9c</td>
<td>djpeg</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>mupdf-1.14.0</td>
<td>mutool show</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>binutils-2.30</td>
<td>size</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>objdump -xD</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>strip</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>
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 Ryan¹, Abhishek Shah¹, Dongdong She¹, Koustubha Bhat², Suman Jana¹

¹: Columbia University
²: Vrije Universiteit