Dynamic Analysis

Nick Sumner
wsumner@sfu.ca
Dynamic Analysis

- Sometimes we want to study or adapt the behavior of executions of a program
Dynamic Analysis

- Sometimes we want to study or adapt the behavior of *executions* of a program
  - Did my program ever ...?
Dynamic Analysis

- Sometimes we want to study or adapt the behavior of *executions* of a program
  - Did my program ever ...?
  - Why/how did ... happen?
Dynamic Analysis

- Sometimes we want to study or adapt the behavior of executions of a program
  - Did my program ever ...?
  - Why/how did ... happen?
  - Where am I spending time?
Dynamic Analysis

- Sometimes we want to study or adapt the behavior of *executions* of a program
  - Did my program ever ...? 
  - Why/how did ... happen? 
  - Where am I spending time? 
  - Where might I parallelize?
Dynamic Analysis

- Sometimes we want to study or adapt the behavior of executions of a program
  - Did my program ever ...?  
  - Why/how did ... happen?  
  - Where am I spending time?  
  - Where might I parallelize?  
  - Tolerate errors
Dynamic Analysis

- Sometimes we want to study or adapt the behavior of executions of a program
  - Did my program ever ...?
  - Why/how did ... happen?
  - Where am I spending time?
  - Where might I parallelize?
  - Tolerate errors
  - Manage memory / resources.
e.g. Reverse Engineering

Static CFG (from e.g. Apple Fairplay):

This is the result of a control flow flattening obfuscation.
[http://tigress.cs.arizona.edu/transformPage/docs/flatten/]
e.g. Reverse Engineering

Static CFG (from e.g. Apple Fairplay):

This is the result of a control flow flattening obfuscation.  
[http://tigress.cs.arizona.edu/transformPage/docs/flatten/]

Dynamically Simplified CFG:
How?

- Can record the execution
How?

• Can record the execution
  – Record to a trace
  – Analyze post mortem / offline
  – Scalability issues: need enough space to store it
How?

- **Can record the execution**
  - Record to a trace
  - Analyze post mortem / offline
  - Scalability issues: need enough space to store it

- **Can perform analysis online**
How?

- **Can record the execution**
  - Record to a trace
  - Analyze post mortem / offline
  - Scalability issues: need enough space to store it

- **Can perform analysis online**
  - *Instrument* the program to collect useful facts
  - Modified program invokes code to 'analyze' itself
How?

- **Can record the execution**
  - Record to a trace
  - Analyze post mortem / offline
  - Scalability issues: need enough space to store it

- **Can perform analysis online**
  - *Instrument* the program to collect useful facts
  - Modified program invokes code to 'analyze' itself

- **Can do both!**
  - Lightweight *recording*
  - Instrument a *replayed* instance of the execution
How?

- Can record the execution
  - Record to a trace
  - Analyze post mortem / offline
  - Scalability issues: need enough space to store it

- Can perform analysis online
  - *Instrument* the program to collect useful facts
  - Modified program invokes code to 'analyze' itself

- Can do both!
  - Lightweight recording
  - Instrument a *replayed* instance of the execution

Some analyses only make sense online. Why?
Simple Idea: Basic Block Profiling

Knowing where we are spending time is useful:

- **Goal:** Which basic blocks execute most frequently?
Simple Idea: Basic Block Profiling

Knowing where we are spending time is useful:

- **Goal:** Which *basic blocks* execute most frequently?

Profiling is a common dynamic analysis!
Simple Idea: Basic Block Profiling

Knowing where we are spending time is useful:

- **Goal:** *Which basic blocks execute most frequently?*
- How can we modify our program to find this?
Simple Idea: Basic Block Profiling

Knowing where we are spending time is useful:

- **Goal:** Which basic blocks execute most frequently?
- How can we modify our program to find this?
Knowing where we are spending time is useful:

- **Goal:** Which basic blocks execute most frequently?
- How can we modify our program to find this?

```plaintext
BB:0
BB:1
BB:2
BB:3

count[2] += 1

x = foo()
y = bar()
...
Simple Idea: Basic Block Profiling

Knowing where we are spending time is useful:

- **Goal:** *Which basic blocks execute most frequently?*
- How can we modify our program to find this?

```python
for i in BBs:
    count[i] = 0
for i in BBs:
    print(count[i])
```

Start:
```
for i in BBs:
    count[i] = 0
```

End:
```
for i in BBs:
    print(count[i])
```
Simple Idea: Basic Block Profiling

Knowing where we are spending time is useful:

- **Goal:** Which basic blocks execute most frequently?
- How can we modify our program to find this?

```python
for i in BBs:
    count[i] = 0
for i in BBs:
    count[i] += 1
x = foo()
y = bar()
...

for i in BBs:
    print(count[i])
```

Start:
```python
for i in BBs:
    count[i] = 0
```

End:
```python
for i in BBs:
    print(count[i])
```
Simple Idea: Basic Block Profiling

- Big concern: How efficient is it?
  - The more overhead added, the less practical the tool
Simple Idea: Basic Block Profiling

- Big concern: How efficient is it?
  - The more overhead added, the less practical the tool

```c
count[0] += 1
...

count[1] += 1
...

count[2] += 1
...

count[3] += 1
...

count[4] += 1
...

count[5] += 1
...

count[6] += 1
...
```
Simple Idea: Basic Block Profiling

- Big concern: How efficient is it?
  - The more overhead added, the less practical the tool

- Can we do better?
Simple Idea: Basic Block Profiling

- **Big concern: How efficient is it?**
  - The more overhead added, the less practical the tool

```
```

- Can we do better?
Simple Idea: Basic Block Profiling

- Big concern: How efficient is it?
  - The more overhead added, the less practical the tool

```plaintext
```

- Can we do better?
Simple Idea: Basic Block Profiling

- Big concern: How efficient is it?
  - The more overhead added, the less practical the tool

\[
\begin{align*}
\text{count}[1] &= \text{count}[4] = \text{count}[2] + \text{count}[3] \\
\text{count}[0] &= \text{count}[6] = \text{count}[1] + \text{count}[5]
\end{align*}
\]

- Can we do better?
Efficiency Tactics

- Abstraction
Efficiency Tactics

- Abstraction
- Identify & avoid redundant information
Efficiency Tactics

- Abstraction
- Identify & avoid redundant information
- Sampling
Efficiency Tactics

- Abstraction
- Identify & avoid redundant information
- Sampling
- Compression / encoding
Efficiency Tactics

- Abstraction
- Identify & avoid redundant information
- Sampling
- Compression / encoding
- Profile guided instrumentation
Efficiency Tactics

- Abstraction
- Identify & avoid redundant information
- Sampling
- Compression / encoding
- Profile guided instrumentation
- Thread local analysis
Efficiency Tactics

- Abstraction
- Identify & avoid redundant information
- Sampling
- Compression / encoding
- Profile guided instrumentation
- Thread local analysis
- Inference
How / When to Instrument

- **Source / IR Instrumentation**
  - LLVM, CIL, Soot, Wala
  - During (re)compilation
  - Requires an analysis dedicated build
How / When to Instrument

- **Source / IR Instrumentation**
  - LLVM, CIL, Soot, Wala
  - During (re)compilation
  - Requires an analysis dedicated build

- **Static Binary Rewriting**
  - Diablo, DynamoRIO, SecondWrite,
  - Applies to arbitrary binaries
  - Imprecise IR info, but more complete *binary* behavior
How / When to Instrument

- **Source / IR Instrumentation**
  - LLVM, CIL, Soot, Wala
  - During (re)compilation
  - Requires an analysis dedicated build

- **Static Binary Rewriting**
  - Diablo, DynamoRIO, SecondWrite,
  - Applies to arbitrary binaries
  - Imprecise IR info, but more complete *binary* behavior

- **Dynamic Binary Instrumentation**
  - Valgrind, Pin, Qemu (& other Vms)
  - Can adapt at runtime, but less info than IR
How / When to Instrument

- **Source / IR Instrumentation**
  - LLVM, CIL, Soot, Wala
  - During (re)compilation
  - Requires an analysis dedicated build

- **Static Binary Rewriting**
  - Diablo, DynamoRIO, SecondWrite,
  - Applies to arbitrary binaries
  - Imprecise IR info, but more complete binary behavior

- **Dynamic Binary Instrumentation**
  - Valgrind, Pin, Qemu (& other Vms)
  - Can adapt at runtime, but less info than IR

- **Black Box Dynamic Analysis**
Phases of Dynamic Analysis

In general, 2-3 phases occur:

1) **Instrumentation**
   - Add code to the program for data collection/analysis
Phases of Dynamic Analysis

In general, 2-3 phases occur:

1) **Instrumentation**
   - Add code to the program for data collection/analysis

2) **Execution**
   - Run the program and analyze its actual behavior
Phases of Dynamic Analysis

In general, 2-3 phases occur:

1) **Instrumentation**
   - Add code to the program for data collection/analysis

2) **Execution**
   - Run the program and analyze its actual behavior

3) **(Optional) Postmortem Analysis**
   - Perform any analysis that can be deferred after termination
Phases of Dynamic Analysis

In general, 2-3 phases occur:

1) **Instrumentation**
   - Add code to the program for data collection/analysis

2) **Execution**
   - Run the program and analyze its actual behavior

3) **(Optional) Postmortem Analysis**
   - Perform any analysis that can be deferred after termination

Very, very common mistake to mix 1 & 2.
1) Compile whole program to IR
Static Instrumentation

1) Compile whole program to IR
2) Instrument / add code directly to the IR
Static Instrumentation

1) Compile whole program to IR
2) Instrument / add code directly to the IR
3) Generate new program that performs analysis
Static Instrumentation

1) Compile whole program to IR
2) Instrument / add code directly to the IR
3) Generate new program that performs analysis
4) Execute
Dynamic Binary Instrumentation (DBI)

1) Compile program as usual
2) Run program under analysis framework
   (Valgrind, PIN, Qemu, etc)

valgrind --tool=memcheck ./myBuggyProgram
Dynamic Binary Instrumentation (DBI)

1) Compile program as usual

2) Run program under analysis framework
   (Valgrind, PIN, Qemu, etc)

3) Instrument & execute in same command:
   - Fetch & instrument each basic block individually
   - Execute each basic block

\texttt{valgrind --tool=memcheck ./myBuggyProgram}
Example: Test Case Reduction
Testing and Dynamic Analysis

- In some cases, just running a program with different inputs is enough
Testing and Dynamic Analysis

- In some cases, just running a program with different inputs is enough
  - Carefully selected inputs can target the analysis
  - The result of running the program reveals coarse information about its behavior
Testing and Dynamic Analysis

- In some cases, just running a program with different inputs is enough
  - Carefully selected inputs can target the analysis
  - The result of running the program reveals coarse information about its behavior

- Intuitively, even just testing is a dynamic analysis
  - It requires no transformation
  - The result is just the success or failure of tests
Testing and Dynamic Analysis

- In some cases, just running a program with different inputs is enough
  - Carefully selected inputs can target the analysis
  - The result of running the program reveals coarse information about its behavior

- Intuitively, even just testing is a dynamic analysis
  - It requires no transformation
  - The result is just the success or failure of tests

- But even that is interesting to consider....
Bug reports are problematic

- Failing inputs can be large and complex

MB? GB?
Bug reports are problematic

- Failing inputs can be large and complex

MB? GB? What is relevant and essential to the bug?
Bug reports are problematic

- Failing inputs can be large and complex

```plaintext
arhwlnyueumgkowh>`p
```

Bug 1
```
abca
```

Bug 2
```
abca
```

Bug 3
```
abca
```
Bug reports are problematic

- Failing inputs can be large and complex

1) Are these reports the same bug?
2) Can we make it easier to reproduce?
Bug reports are problematic

- Failing inputs can be large and complex
Bug reports are problematic

- Failing inputs can be large and complex

1) Same? Yes!
2) Easier? Yes! And easier to understand!
Bug reports are problematic

- Failing inputs can be large and complex

```
arahtwilnyeumgkowh>`p
```

Bug 1: a b c

Bug 2: a b c

Bug 3: a b c

Test Case Reduction: finding smaller test cases that reproduce a failure
Classically – Delta Debugging

<br />
<textarea>
<SELECT NAME="priority" MULTIPLE SIZE=7>
</textarea>

Classically – Delta Debugging

<SELECT NAME="priority" MULTIPLE SIZE=7>


print

N E T S C A P E

Classically – Delta Debugging

<SELECT NAME="priority" MULTIPLE SIZE=7>
  Intuition: trial and error
</SELECT>
Classically – Delta Debugging

Intuition: trial and error
1) Start w/ a failing text configuration c
Classically – Delta Debugging

<SELECT NAME="priority" MULTIPLE SIZE=7>

Intuition: trial and error
1) Start w/ a failing text configuration \( c \)
2) Try removing subsets (\( \Delta \)) of input elements (\( \{\delta\} \))
Classically – Delta Debugging

<SELECT NAME="priority" MULTIPLE SIZE=7>

Intuition: trial and error
1) Start w/ a failing text configuration c
2) Try removing subsets (Δ) of input elements (δ)
3) Failure still exists → new input is “better”
Classically – Delta Debugging

Intuition: trial and error
1) Start w/ a failing text configuration $c$
2) Try removing subsets ($\Delta$) of input elements ($\{\delta\}$)
3) Failure still exists $\rightarrow$ new input is “better”
4) Repeat on the new input
Classically – Delta Debugging

<SELECT NAME="priority" MULTIPLE SIZE=7>
Intuition: trial and error
1) Start w/ a failing text configuration c
2) Try removing subsets (Δ) of input elements ({{δ}})
3) Failure still exists → new input is “better”
4) Repeat on the new input

When do we stop? / What is our goal?
• Global Minimum: c : ∀ |c'|<|c|, c'
Intuition: trial and error
1) Start with a failing text configuration $c$
2) Try removing subsets ($\Delta$) of input elements ($\{\delta\}$)
3) Failure still exists $\rightarrow$ new input is “better”
4) Repeat on the new input

When do we stop? / What is our goal?
• **Global Minimum:** $c : \forall c' |c'| < |c|$, $c' \rightarrow$ Smallest subset of the original input reproducing the failure
Classically – Delta Debugging

Intuition: trial and error
1) Start w/ a failing text configuration \( c \)
2) Try removing subsets (\( \Delta \)) of input elements (\( \{\delta\} \))
3) Failure still exists \( \rightarrow \) new input is “better”
4) Repeat on the new input

When do we stop? / What is our goal?
- **Global Minimum:** \( c : \forall |c'| < |c|, c' \)

Completely impractical! Why?
- Smallest subset of the original input reproducing the failure
Classically – Delta Debugging

Intuition: trial and error
1) Start w/ a failing text configuration \( c \)
2) Try removing subsets (\( \Delta \)) of input elements (\( \{\delta\} \))
3) Failure still exists \( \rightarrow \) new input is “better”
4) Repeat on the new input

When do we stop? / What is our goal?
- Global Minimum: \( c : \forall |c'| < |c|, c' \)
- Local Minimum: \( c : \forall c' \subset c, c' \)
Classically – Delta Debugging

<SELECT NAME="priority" MULTIPLE SIZE=7>
Intuition: trial and error
1) Start w/ a failing text configuration $c$
2) Try removing subsets ($\Delta$) of input elements ($\{\delta\}$)
3) Failure still exists $\rightarrow$ new input is “better”
4) Repeat on the new input

When do we stop? / What is our goal?
• Global Minimum: $c : \forall |c'| < |c|, c'$
• Local Minimum: $c : \forall c' \subset c, c'$

No subset of the result can reproduce the failure.
Classically – Delta Debugging

Intuition: trial and error
1) Start w/ a failing text configuration \( c \)
2) Try removing subsets (\( \Delta \)) of input elements (\( \{\delta\} \))
3) Failure still exists \( \rightarrow \) new input is “better”
4) Repeat on the new input

When do we stop? / What is our goal?
- Global Minimum: \( c : \forall \ |c'| < |c|, c' \)
- Local Minimum: \( c : \forall c' \subset c, c' \)

How does this differ from a global minimum? Is it still problematic?

No subset of the result can reproduce the failure.
Classically – Delta Debugging

Intuition: trial and error
1) Start w/ a failing text configuration \( c \)
2) Try removing subsets \( \Delta \) of input elements \( \{\delta\} \)
3) Failure still exists \( \rightarrow \) new input is “better”
4) Repeat on the new input

When do we stop? / What is our goal?
- Global Minimum: \( c : \forall |c'| < |c|, c' \)
- Local Minimum: \( c : \forall c' \subset c, c' \)
- 1-Minimal: \( c: \forall \delta \in c, (c-\{\delta\}) \)
Classically – Delta Debugging

Intuition: trial and error
1) Start w/ a failing text configuration $c$
2) Try removing subsets ($\Delta$) of input elements ($\{\delta\}$)
3) Failure still exists $\rightarrow$ new input is "better"
4) Repeat on the new input

When do we stop? / What is our goal?

- **Global Minimum**: $c : \forall \ |c'| < |c|, c' \\
- **Local Minimum**: $c : \forall c' \subset c, c'$
- **1-Minimal**: $c: \forall \delta \in c, (c-\{\delta\})$

No one element can be removed and still reproduce the failure
Classically – Delta Debugging

Intuition: trial and error
1) Start w/ a failing text configuration \( c \)
2) Try removing subsets (\( \Delta \)) of input elements (\( \{\delta\} \))
3) Failure still exists \( \rightarrow \) new input is “better”
4) Repeat on the new input

When do we stop? / What is our goal?
- Global Minimum: \( c : \forall |c'| < |c|, c' \)
- Local Minimum: \( c : \forall c' \subset c, c' \)
- \( 1 \)-Minimal: \( c: \forall \delta \in c, (c-\{\delta\}) \)

This is the classic goal.
In practice, the formalism may not pay for itself in terms of quality or efficiency! (Be pragmatic!)
Classically – Delta Debugging

Does binary search work?
Classically – Delta Debugging
Classically – Delta Debugging

So what should we do?
So what should we do?

We refine the granularity
Classically – Delta Debugging
Classically – Delta Debugging
Classically – Delta Debugging
Classically – Delta Debugging
Classically – Delta Debugging

And now check complements
Classically – Delta Debugging

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
Classically – Delta Debugging
Classically – Delta Debugging
Classically – Delta Debugging
Classically – Delta Debugging

What's clever about how we recurse?
Classically – Delta Debugging
Classically – Delta Debugging

So close! How many more?
Classically – Delta Debugging

Done?
## Classically – Delta Debugging

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

![Diagram](image)

**Done?**
Classically – Delta Debugging

\[ \text{ddmin}(c) = \text{ddmin}_2(c, 2) \]

Defined over
- \( c \) - the input / configuration
- \( n \) - the # of partitions
Classically – Delta Debugging

\[ \text{ddmin}(c) = \text{ddmin}_2(c, 2) \]

\[ c = \begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array} \]
Classically – Delta Debugging

ddmin(c) = ddmin_2(c, 2)

c = \Delta_1 \Delta_2 \Delta_3 \Delta_4

n = 4
Classically – Delta Debugging

$ddmin(c) = ddmin_2(c, 2)$
Classically – Delta Debugging

$$\text{ddmin}(c) = \text{ddmin}_2(c, 2)$$

$$\text{ddmin}_2(c, n)=$$
Classically – Delta Debugging

\[ \text{ddmin}(c) = \text{ddmin}_2(c, 2) \]

\[ \text{ddmin}_2(c, n) = \begin{cases} 
\text{ddmin}_2(\Delta_i, 2) & \text{If } \ldots (a) \\
\text{Try each partition} & 
\end{cases} \]

\[ \Delta_i = \{3, 4\} \]
Classically – Delta Debugging

\[ \text{ddmin}(c) = \text{ddmin}_2(c, 2) \]

\[ \text{ddmin}_2(c, n) = \begin{cases} 
\text{ddmin}_2(\Delta_i, 2) & \text{if } \ldots \text{ (a)} \\
\text{ddmin}_2(\nabla i, \max(n-1,2)) & \text{if } \ldots \text{ (b)} 
\end{cases} \]

Try each complement

\[ \Delta_i = \{3,4\} \]

(a) 1 2 3 4 5 6 7 8
(b) 1 2 3 4 5 6 7 8

103
Classically – Delta Debugging

$$ddmin(c) = ddmin_2(c, 2)$$

$$ddmin_2(c, n) = \begin{cases} 
ddmin_2(\Delta_i, 2) & \text{If ... (a)} \\
ddmin_2(\nabla_i, \max(n-1, 2)) & \text{If ... (b)} 
\end{cases}$$

$$\Delta_i = \{3, 4\}$$

(a) $\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}$

(b) $\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}$
Classically – Delta Debugging

\[ \text{ddmin}(c) = \text{ddmin}_2(c, 2) \]

\[
\text{ddmin}_2(c, n) = \begin{cases} 
\text{ddmin}_2(\Delta_i, 2) & \text{If } ... \text{ (a)} \\
\text{ddmin}_2(\nabla i, \max(n-1, 2)) & \text{If } ... \text{ (b)} \\
\text{ddmin}_2(c, \min(|c|, 2n)) & \text{If } ... \text{ (c)} 
\end{cases}
\]

Refine the granularity

\[ \Delta_i = \{3,4\} \]

(a) \[\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}\]  

(b) \[\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}\]  

(c) \(n < |c|\)
Classically – Delta Debugging

\[ \text{ddmin}(c) = \text{ddmin}_2(c, 2) \]

\[ \text{ddmin}_2(c, n) = \begin{cases} 
\text{ddmin}_2(\Delta_i, 2) & \text{If } \ldots \ (a) \\
\text{ddmin}_2(\nabla i, \max(n-1, 2)) & \text{If } \ldots \ (b) \\
\text{ddmin}_2(c, \min(|c|, 2n)) & \text{If } \ldots \ (c) 
\end{cases} \]

\[ \Delta_i = \{3, 4\} \]

(a) \[1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8\]

(b) \[1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8\]

(c) \[n < |c|\]
Classically – Delta Debugging

\[ ddmin(c) = ddmin_2(c, 2) \]

\[ ddmin_2(c, n) = \begin{cases} 
ddmin_2(\Delta_i, 2) & \text{If ... (a)} \\
ddmin_2(\nabla i, \max(n-1,2)) & \text{If ... (b)} \\
ddmin_2(c, \min(|c|, 2n)) & \text{If ... (c)} \\
c & \text{otherwise}
\end{cases} \]

\[ \Delta_i = \{3,4\} \]

1. 1 2 3 4 5 6 7 8
2. 1 2 3 4 5 6 7 8
3. n < |c|
Classically – Delta Debugging

- Worst Case: $|c|^2 + 3|c|$ tests
  - All tests unresolved until maximum granularity
  - Testing complement succeeds
Classically – Delta Debugging

- **Worst Case**: $|c|^2 + 3|c|$ tests
  - All tests unresolved until maximum granularity
  - Testing complement succeeds

- **Best Case**: # tests $\leq 2\log_2(|c|)$
  - Falling back to binary search!
Classically – Delta Debugging

- **Worst Case:** $|c|^2 + 3|c|$ tests
  - All tests unresolved until maximum granularity
  - Testing complement succeeds

- **Best Case:** # tests $\leq 2\log_2(|c|)$
  - Falling back to binary search!

- **Minimality**
  - When will it only be locally minimal?
  - When will it only be 1-minimal?
Classically – Delta Debugging

- **Worst Case**: $|c|^2 + 3|c|$ tests
  - All tests unresolved until maximum granularity
  - Testing complement succeeds

- **Best Case**: # tests $\leq 2\log_2(|c|)$
  - Falling back to binary search!

- **Minimality**
  - When will it only be locally minimal?
  - When will it only be 1-minimal?
  - *Does formal minimality even matter?*
Classically – Delta Debugging

- Observation: 
  *In practice* DD may revisit elements in order to guarantee minimality
Classically – Delta Debugging

- Observation: In practice DD may revisit elements in order to guarantee minimality

\[ \text{ddmin}_2(\nabla_i, \max(n-1,2)) \]
Classically – Delta Debugging

- Observation:
  *In practice* DD may revisit elements in order to guarantee minimality

\[ \text{ddmin}_2(\forall i, \max(n-1,2)) \]
Classically – Delta Debugging

- Observation: 
  *In practice* DD may revisit elements in order to guarantee minimality

\[ \text{ddmin}_2(\nabla_i, \max(n-1,2)) \]

- If guaranteeing 1-minimality does not matter the algorithm can drop to linear time!
  - In practice this can be effective for what developers may care about
Classically – Delta Debugging

- Observation: 
  In practice DD may revisit elements in order to guarantee minimality

$$\text{ddmin}_2(\bigvee_i, \max(n-1,2))$$

- If guaranteeing 1-minimality does not matter the algorithm can drop to linear time!
  - In practice this can be effective for what developers may care about

Don’t get bogged down by formalism when it doesn’t serve you!
Test Case Reduction in Practice

- Most problems do not use DD directly for TCR.
  - It provides inspiration, but frequently behaves poorly
Test Case Reduction in Practice

- Most problems *do not* use DD directly for TCR.
  - It provides inspiration, but frequently behaves poorly

- **What are the possible causes of problems?**

  - **Monotonicity matters**
Test Case Reduction in Practice

- Most problems *do not* use DD directly for TCR.
  - It provides inspiration, but frequently behaves poorly

- **What are the possible causes of problems?**

  - Monotonicity matters
  - Determinism matters
Test Case Reduction in Practice

- Most problems *do not* use DD directly for TCR.
  - It provides inspiration, but frequently behaves poorly

- What are the possible causes of problems?

```
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4
```

- **Monotonicity matters**
- **Determinism matters**

```
for i in [5, 10]:
a[i] = i * 5
```

- **Structure matters**
Test Case Reduction for Compilers

- Programs are highly structured, so TCR for compilers faces challenges
Test Case Reduction for Compilers

- Programs are highly structured, so TCR for compilers faces challenges
- What structures could we use to guide the process?
Test Case Reduction for Compilers

- Programs are highly structured, so TCR for compilers faces challenges
- What structures could we use to guide the process?

```
for i in range(5, 10):
    a[i] = i * 5
```
Test Case Reduction for Compilers

- Programs are highly structured, so TCR for compilers faces challenges.
- What structures could we use to guide the process?
- What challenges still remain?
Generalizing Further

- What else could we think of as test case reduction?
Generalizing Further

- What else could we think of as test case reduction?
  - Failing traces of a program?
  - "" in a distributed system?
  - "" microservice application?
Generalizing Further

- What else could we think of as test case reduction?
  - Failing traces of a program?
  - “ ” in a distributed system?
  - “ ” microservice application?
  - Automatically generated test cases?
What else could we think of as test case reduction?
- Failing traces of a program?
- “” in a distributed system?
- “” microservice application?
- Automatically generated test cases?
- ...
Generalizing Further

• What else could we think of as test case reduction?
  – Failing traces of a program?
  – “” in a distributed system?
  – “” microservice application?
  – Automatically generated test cases?

• The ability to treat the program \textit{as an oracle} is also very powerful
Generalizing Further

• What else could we think of as test case reduction?
  – Failing traces of a program?
  – “” in a distributed system?
  – “” microservice application?
  – Automatically generated test cases?

• The ability to treat the program *as an oracle* is also very powerful
  – We can get new data by running the program
  – This can be combined with reinforcement learning to accomplish hard tasks
Generalizing Further

- What else could we think of as test case reduction?
  - Failing traces of a program?
  - “” in a distributed system?
  - “” microservice application?
  - Automatically generated test cases?

- The ability to treat the program as an oracle is also very powerful
  - We can get new data by running the program
  - This can be combined with reinforcement learning to accomplish hard tasks
  - We saw this before when discussing test suites!
Example: Memory Safety Bugs
Example: Finding memory safety bugs

- Memory safety bugs are one of the most common sources of security vulnerabilities
Example: Finding memory safety bugs

- Memory safety bugs are one of the most common sources of security vulnerabilities

- Effects may only be visible long after invalid behavior
  - This complicates comprehension & debugging
Example: Finding memory safety bugs

- Memory safety bugs are one of the most common sources of security vulnerabilities

- Effects may only be visible long after invalid behavior
  - This complicates comprehension & debugging

- Two main types of issues:
  - Spatial – Out of bounds stack/heap/global accesses
  - Temporal – Use after free
Example: Finding memory safety bugs

- Memory safety bugs are one of the most common sources of security vulnerabilities

- Effects may only be visible long after invalid behavior
  - This complicates comprehension & debugging

- Two main types of issues:
  - Spatial – Out of bounds stack/heap/global accesses
  - Temporal – Use after free

- We would like to automatically identify & provide assistance with high precision and low overhead
  - Suitable for testing & sometimes maybe deployment!
Example: Finding memory safety bugs

- Most common approach – track which regions of memory are valid
  - During execution!
  - Updated when new memory is allocated
  - Checked when pointers are accessed
  - With low overhead
Example: Finding memory safety bugs

- Most common approach – track which regions of memory are valid
  - During execution!
  - Updated when new memory is allocated
  - Checked when pointers are accessed
  - With low overhead

- Common implementations
  - Valgrind – DBI based
  - AddressSanitizer – static instrumentation based
Example: Finding memory safety bugs

- Most common approach – track which regions of memory are valid
  - During execution!
  - Updated when new memory is allocated
  - Checked when pointers are accessed
  - With low overhead

- Common implementations
  - Valgrind – DBI based
  - AddressSanitizer – static instrumentation based

Note, the implementation style affects which bugs can be recognized!

Why?
AddressSanitizer

- Need to track which memory is valid & check efficiently...
- Big Picture:
  - Replace calls to `malloc & free`
AddressSanitizer

• Need to track which memory is valid & check efficiently...

• Big Picture:
  – Replace calls to malloc & free
  – Poison memory: (create red zones)
AddressSanitizer

- Need to track which memory is valid & check efficiently...

- Big Picture:
  - Replace calls to `malloc` & `free`
  - Poison memory: (create red zones)
    1) around `malloced` chunks

```c
ptr = malloc(sizeof(MyStruct));
```
AddressSanitizer

- Need to track which memory is valid & check efficiently...

- Big Picture:
  - Replace calls to malloc & free
  - Poison memory: (create red zones)
    1) around malloced chunks
    2) when it is freed

```c
free(ptr);
```
AddressSanitizer

- Need to track which memory is valid & check efficiently...

- Big Picture:
  - Replace calls to malloc & free
  - **Poison memory:** (create red zones)
    1) around malloced chunks
    2) when it is freed
    3) around buffers and local variables

```c
void foo() {
    int buffer[5];
    ...
}
```
AddressSanitizer

• Need to track which memory is valid & check efficiently...

• Big Picture:
  – Replace calls to malloc & free
  – Poison memory: (create red zones)
    1) around malloced chunks
    2) when it is freed
    3) around buffers and local variables
  – Access of poisoned memory causes an error
AddressSanitizer

• Need to track which memory is valid & check efficiently...

• Big Picture:
  – Replace calls to malloc & free
  – Poison memory: (create red zones)
    1) around malloced chunks
    2) when it is freed
    3) around buffers and local variables
  – Access of poisoned memory causes an error

*address = ...;
AddressSanitizer

- Need to track which memory is valid & check efficiently...

- Big Picture:
  - Replace calls to malloc & free
  - Poison memory: (create red zones)
    1) around malloced chunks
    2) when it is freed
    3) around buffers and local variables
  - Access of poisoned memory causes an error

```c
*address = ...;
If (IsPoisoned(address, size)) {
    ReportError(address, size, isWrite);
}
*address = ...;
```

instrumentation
AddressSanitizer

- Need to track which memory is valid & check efficiently...
- Big Picture:
  - Replace calls to malloc & free
  - Poison memory: (create red zones)
    1) around malloced chunks
    2) when it is freed
    3) around buffers and local variables
  - Access of poisoned memory causes an error
- The tricky part is tracking & efficiently checking redzones.
AddressSanitizer

- Need to track which memory is valid & check efficiently...

- Big Picture:
  - Replace calls to malloc & free
  - Poison memory: (create red zones)
    1) around malloced chunks
    2) when it is freed
    3) around buffers and local variables
  - Access of poisoned memory causes an error

- The tricky part is tracking & efficiently checking redzones.
  - Instrumenting every memory access is costly!
AddressSanitizer

- Need to track which memory is valid & check efficiently...

- Big Picture:
  - Replace calls to malloc & free
  - Poison memory: (create red zones)
    1) around malloced chunks
    2) when it is freed
    3) around buffers and local variables
  - Access of poisoned memory causes an error

- The tricky part is tracking & efficiently checking redzones.
  - Instrumenting every memory access is costly!
  - We must track all memory ... inside that same memory!
AddressSanitizer

• Need to track which memory is valid & check efficiently...

• Big Picture:
  – Replace calls to malloc & free
  – Poison memory: (create red zones)
    1) around malloced chunks
    2) when it is freed
    3) around buffers and local variables
  – Access of poisoned memory causes an error

• The tricky part is tracking & efficiently checking redzones.
  – Instrumenting every memory access is costly!
  – We must track all memory ... inside that same memory!

This kind of issue is common in dynamic analyses.
AddressSanitizer – Shadow Memory

Need to know whether *any byte* of application memory is poisoned.
AddressSanitizer – Shadow Memory

- We maintain 2 views on memory
We maintain 2 views on memory.
AddressSanitizer – Shadow Memory

- We maintain 2 views on memory
- Shadow Memory is pervasive in dynamic analysis
  - Can be thought of as a map containing analysis data
AddressSanitizer – Shadow Memory

- We maintain 2 views on memory
- **Shadow Memory is pervasive in dynamic analysis**
  - Can be thought of as a map containing analysis data
  - For every bit/byte/word/chunk/allocation/page, maintain information in a *compact* table
AddressSanitizer – Shadow Memory

- We maintain 2 views on memory
- **Shadow Memory is pervasive in dynamic analysis**
  - Can be thought of as a map containing analysis data
  - For every bit/byte/word/chunk/allocation/page, maintain information in a *compact* table

Where have you encountered this before?
AddressSanitizer – Shadow Memory

- We maintain 2 views on memory
- **Shadow Memory is pervasive in dynamic analysis**
  - Can be thought of as a map containing analysis data
  - For every bit/byte/word/chunk/allocation/page, maintain information in a compact table
  - Common in runtime support, e.g. page tables
AddressSanitizer – Shadow Memory

- Designing efficient analyses (& shadow memory) often requires a careful domain insight

Encoding & abstraction efficiency strategies
AddressSanitizer – Shadow Memory

- Designing efficient analyses (& shadow memory) often requires a careful domain insight
- NOTE: Heap allocated regions are N byte aligned (N usually 8)
AddressSanitizer – Shadow Memory

- Designing efficient analyses (& shadow memory) often requires a careful domain insight

- **NOTE**: Heap allocated regions are N byte aligned (N usually 8)
  - In an N byte region, only the first k may be addressable
AddressSanitizer – Shadow Memory

- Designing efficient analyses (& shadow memory) often requires a careful domain insight

- **NOTE:** Heap allocated regions are N byte aligned (N usually 8)
  - In an N byte region, only the first k may be addressable
  - Every N bytes has only N+1 possible states
AddressSanitizer – Shadow Memory

- Designing efficient analyses (& shadow memory) often requires a careful domain insight

- **NOTE: Heap allocated regions are N byte aligned (N usually 8)**
  - In an N byte region, only the first k may be addressable
  - Every N bytes has only N+1 possible states
  - Map every N bytes to 1 shadow byte encoding state as a number

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
AddressSanitizer – Shadow Memory

- Designing efficient analyses (& shadow memory) often requires a careful domain insight

- **NOTE: Heap allocated regions are N byte aligned (N usually 8)**
  - In an N byte region, only the first k may be addressable
  - Every N bytes has only N+1 possible states
  - Map every N bytes to 1 shadow byte encoding state as a number
    - All good = 0    All bad = -1    Partly good = # good

![Diagram showing shadow memory states with binary representation](image-url)
AddressSanitizer – Shadow Memory

- Designing efficient analyses (& shadow memory) often requires a careful domain insight

- NOTE: Heap allocated regions are N byte aligned (N usually 8)
  - In an N byte region, only the first k may be addressable
  - Every N bytes has only N+1 possible states
  - Map every N bytes to 1 shadow byte encoding state as a number

- What does accessing shadow memory for an address look like? (N=8)
AddressSanitizer – Shadow Memory

- Designing efficient analyses (& shadow memory) often requires a careful domain insight

- **NOTE: Heap allocated regions are N byte aligned (N usually 8)**
  - In an N byte region, only the first k may be addressable
  - Every N bytes has only N+1 possible states
  - Map every N bytes to 1 shadow byte encoding state as a number

- **What does accessing shadow memory for an address look like? (N=8)**
  - Preallocate a large table
AddressSanitizer – Shadow Memory

- Designing efficient analyses (& shadow memory) often requires a careful domain insight

- NOTE: Heap allocated regions are N byte aligned (N usually 8)
  - In an N byte region, only the first k may be addressable
  - Every N bytes has only N+1 possible states
  - Map every N bytes to 1 shadow byte encoding state as a number

- What does accessing shadow memory for an address look like? (N=8)
  - Preallocate a large table
  - Shadow = (address >> 3) + Offset
AddressSanitizer – Shadow Memory

- Designing efficient analyses (& shadow memory) often requires a careful domain insight

- NOTE: Heap allocated regions are N byte aligned (N usually 8)
  - In an N byte region, only the first k may be addressable
  - Every N bytes has only N+1 possible states
  - Map every N bytes to 1 shadow byte encoding state as a number

- What does accessing shadow memory for an address look like? (N=8)
  - Preallocate a large table
  - Shadow = (address >> 3) + Offset
  - With PIE, Shadow = (address >> 3)
AddressSanitizer – Shadow Memory

- Designing efficient analyses (& shadow memory) often requires a careful domain insight

- NOTE: Heap allocated regions are N byte aligned (N usually 8)
  - In an N byte region, only the first k may be addressable
  - Every N bytes has only N+1 possible states
  - Map every N bytes to 1 shadow byte encoding state as a number

- What does accessing shadow memory for an address look like? (N=8)
  - Preallocate a large table
  - Shadow = (address >> 3) + Offset
  - With PIE, Shadow = (address >> 3)

```c
if (*(address>>3)) {
    ReportError(...);
}
*address = ...
```
Designing efficient analyses (& shadow memory) often requires a careful domain insight.

**NOTE:** Heap allocated regions are N byte aligned (N usually 8)
- In an N byte region, only the first k may be addressable
- Every N bytes has only N+1 possible states
- Map every N bytes to 1 shadow byte encoding state as a number

What does accessing shadow memory for an address look like? (N=8)
- Preallocate a large table
- Shadow = (address >> 3) + Offset
- With PIE, Shadow = (address >> 3)

```
if (*((address)>>3)) {
    ReportError(...);
}
*address = ...
```

Now you can also see the reason for the numerical encoding....
AddressSanitizer – Shadow Memory

- Handling accesses of size < N (N=8)

```c
shadow = address >> 3
state = *shadow
if (state != 0 && (state < (address & 7) + size)) {
    ReportError(...);
}
*address = ...
```
AddressSanitizer – Shadow Memory

- Handling accesses of size < N (N=8)

```c
shadow = address >> 3
state = *shadow
if (state != 0 && (state < (address & 7) + size)) {
    ReportError(...);
}
*address = ...
```

Careful construction of states can make runtime checks efficient
AddressSanitizer - Evaluating

- In dynamic analyses, we care about both overheads & result quality
AddressSanitizer - Evaluating

- In dynamic analyses, we care about both overheads & result quality
- Overheads
  - Need to determine what resources are being consumed
AddressSanitizer - Evaluating

- In dynamic analyses, we care about both overheads & result quality

- **Overheads**
  - Need to determine what resources are being consumed
  - Memory – Shadow memory *capacity* is cheap, but accessed shadows matter
AddressSanitizer - Evaluating

- In dynamic analyses, we care about both overheads & result quality
- **Overheads**
  - Need to determine what resources are being consumed
  - Memory – Shadow memory capacity is cheap, but accessed shadows matter
  - Running time – Can effectively be free for I/O bound projects
    Up to 25x overheads on some benchmarks
AddressSanitizer - Evaluating

- In dynamic analyses, we care about both overheads & result quality

- Overheads
  - Need to determine what resources are being consumed
  - Memory – Shadow memory capacity is cheap, but accessed shadows matter
  - Running time – Can effectively be free for I/O bound projects
    Up to 25x overheads on some benchmarks

- Quality
  - Precision & recall matter

Where will it miss bugs? Where will it raise false alarms?
AddressSanitizer - Evaluating

- False negatives
  - Computed pointers that are accidentally in bounds
AddressSanitizer - Evaluating

- **False negatives**
  - Computed pointers that are accidentally in bounds
  - Unaligned accesses that are partially out of bounds
Address Sanitizer - Evaluating

- False negatives
  - Computed pointers that are accidentally in bounds
  - Unaligned accesses that are partially out of bounds
  - Use after frees with significant churn
Example: Comparing Executions
Why compare traces or executions?

- Understanding the differences between two executions (and how some differences cause others) can help explain program behavior.
Why compare traces or executions?

- Understanding the differences between two executions (& how some differences cause others) can help explain program behavior
- Several tasks could be made simpler by trace comparison
Why compare traces or executions?

- Understanding the differences between two executions (& how some differences cause others) can help explain program behavior.

- Several tasks could be made simpler by trace comparison
  - Debugging regressions – old vs new
Why compare traces or executions?

- Understanding the differences between two executions (and how some differences cause others) can help explain program behavior

- Several tasks could be made simpler by trace comparison
  - Debugging regressions – old vs new
  - Validating patches – old vs new
Why compare traces or executions?

- Understanding the differences between two executions (and how some differences cause others) can help explain program behavior.

- Several tasks could be made simpler by trace comparison:
  - Debugging regressions – old vs new
  - Validating patches – old vs new
  - Understanding automated repair – old vs new
Why compare traces or executions?

- Understanding the differences between two executions (& how some differences cause others) can help explain program behavior.

- **Several tasks could be made simpler by trace comparison**
  - Debugging regressions – old vs new
  - Validating patches – old vs new
  - Understanding automated repair – old vs new
  - **Debugging with concurrency** – buggy vs nonbuggy schedules
Why compare traces or executions?

- Understanding the differences between two executions (and how some differences cause others) can help explain program behavior.

- Several tasks could be made simpler by trace comparison:
  - Debugging regressions – old vs new
  - Validating patches – old vs new
  - Understanding automated repair – old vs new
  - Debugging with concurrency – buggy vs nonbuggy schedules
  - Malware analysis – malicious vs nonmalicious run
Why compare traces or executions?

- Understanding the differences between two executions (& how some differences cause others) can help explain program behavior

- **Several tasks could be made simpler by trace comparison**
  - Debugging regressions – old vs new
  - Validating patches – old vs new
  - Understanding automated repair – old vs new
  - Debugging with concurrency – buggy vs nonbuggy schedules
  - Malware analysis – malicious vs nonmalicious run
  - Reverse engineering – desired behavior vs undesirable
How it might look
How it might look

Correct  Buggy

x was 5 instead of 3
How it might look

Correct

Buggy

x was 5 instead of 3

So y was 2 instead of 7
How it might look

Correct

Buggy

x was 5 instead of 3

So y was 2 instead of 7

So the TRUE branch executed instead of the FALSE branch
How it might look

Correct

Buggy

x was 5 instead of 3
So y was 2 instead of 7
So the TRUE branch executed instead of the FALSE branch
So the update of z was skipped
How it might look

Correct

Buggy

x was 5 instead of 3
So y was 2 instead of 7
So the TRUE branch executed instead of the FALSE branch
So the update of z was skipped
So the incorrect value of z was printed
How it might look

Correct

Buggy

• x was 5 instead of 3
• So y was 2 instead of 7
• So the TRUE branch executed instead of the FALSE branch
• So the update of z was skipped
• So the incorrect value of z was printed

What do we need?

• locations
• state
• flow
• causation
How it might look

What do we need?
- locations
- state
- flow
- causation

x was 5 instead of 3
So y was 2 instead of 7
So the TRUE branch executed instead of the FALSE branch
So the update of z was skipped
So the incorrect value of z was printed
How it might look

Correct

Buggy

So y was 2 instead of 7
So the TRUE branch executed instead of the FALSE branch
So the update of z was skipped
So the incorrect value of z was printed

What do we need?

• locations
• state
• flow
• causation
How it might look

How it might look

What do we need?
- locations
- state
- flow
- causation

Correct

Buggy

x was 5 instead of 3
So y was 2 instead of 7
So the TRUE branch executed instead of the FALSE branch
So the update of z was skipped
So the incorrect value of z was printed
How it might look

- x was 5 instead of 3
- So y was 2 instead of 7
- So the TRUE branch executed instead of the FALSE branch
- So the update of z was skipped
- So the incorrect value of z was printed

What do we need?
- locations
- state
- flow
- causation
How it might look

What do we need?
- locations
- state
- flow
- causation

We can construct this backward from a point of failure/difference

Correct

Buggy

x was 5 instead of 3
So y was 2 instead of 7
So the TRUE branch executed instead of the FALSE branch
So the update of z was skipped
So the incorrect value of z was printed
So why not just...

- Traces can be viewed as sequences....
  - Why not just do LCS based sequence alignment?
So why not just...

- Traces can be viewed as sequences....
  - Why not just do LCS based sequence alignment?

```python
def foo(int c):
    if c:
        while bar():
            ...
    foo(...)
    baz(...)
    foo(...)
```
So why not just...

- Traces can be viewed as sequences....
  - Why not just do LCS based sequence alignment?

```python
def foo(int c):
    if c:
        while bar():
            ...
    foo(...)  
    baz(...)  
    foo(...)
```
Traces can be viewed as sequences....
- Why not just do LCS based sequence alignment?

def foo(int c):
    if c:
        while bar():
            ...

foo(...) baz() foo(...)
foo() baz() foo()
So why not just...

- Traces can be viewed as sequences....
  - Why not just do LCS based sequence alignment?

```python
def foo(int c):
    if c:
        while bar():
            ...
    foo(...)
    baz(...)
    foo(...)
```

What is marked as different?
So why not just...

- Traces can be viewed as sequences....
  - Why not just do LCS based sequence alignment?

```python
def foo(int c):
    if c:
        while bar():
            ...
    foo(...)  # This function call is different
    baz(...)  # This function call is different
    foo(...)  # This function call is different
```

What is marked as *different*?
So why not just...

- Traces can be viewed as sequences....
  - Why not just do LCS based sequence alignment?

```python
def foo(int c):
    if c:
        while bar():
            ...

foo(...) baz() foo(...)
```

What is marked as *different*?

What is *intuitively* different?
So why not just...

- Traces can be viewed as sequences....
  - Why not just do LCS based sequence alignment?

```python
def foo(int c):
    if c:
        while bar():
            ...
    foo(...)
    baz(...)
    foo(...)
```

What is marked as *different*?
What is *intuitively* different?
So why not just...

- Traces can be viewed as sequences....
  - Why not just do LCS based sequence alignment?

```python
def foo(int c):
    if c:
        while bar():
            ...
    foo(...)
    baz(...)
    foo(...)
```

What is marked as different?
What is *intuitively* different?

Execution comparison must account for what a program *means* and *does*!
The big picture

- Fundamentally, execution comparison needs to account for
The big picture

- Fundamentally, execution comparison needs to account for
  - Structure      - How is a program organized?
The big picture

Fundamentally, execution comparison needs to account for

- Structure – How is a program organized?
- Value – What are the values in the different executions?
The big picture

- Fundamentally, execution comparison needs to account for
  - Structure  - How is a program organized?
  - Value      - What are the values in the different executions?
  - Semantics  - How is the meaning of the program affected by the differences?
The big picture

- Fundamentally, execution comparison needs to account for
  - Structure
  - Value
  - Semantics

- We can attack these through
The big picture

- Fundamentally, execution comparison needs to account for
  - Structure
  - Value
  - Semantics

- We can attack these through
  - Temporal alignment
    - What parts of the trace correspond?
The big picture

- Fundamentally, execution comparison needs to account for
  - Structure
  - Value
  - Semantics

- We can attack these through
  - Temporal alignment
    - What parts of the trace correspond?
  - Spatial alignment
    - What variables & values correspond across traces?
The big picture

- Fundamentally, execution comparison needs to account for
  - Structure
  - Value
  - Semantics

- We can attack these through
  - Temporal alignment
    - What parts of the trace correspond?
  - Spatial alignment
    - What variables & values correspond across traces?
  - Slicing
    - How do differences transitively flow through a program?
The big picture

- Fundamentally, execution comparison needs to account for
  - Structure
  - Value
  - Semantics

- We can attack these through
  - Temporal alignment
    - What parts of the trace correspond?
  - Spatial alignment
    - What variables & values correspond across traces?
  - Slicing
    - How do differences transitively flow through a program?
  - Causality testing
    - Which differences actually induce difference behavior?
Temporal Alignment

- Given $i_1$ in $T_1$ and $i_2$ in $T_2$,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?
Temporal Alignment

- Given $i_1$ in $T_1$ and $i_2$ in $T_2$,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?

- In the simplest case $T_1$ and $T_2$ may follow the same path
  [Mellor-Crummey, ASPLOS 1989]
Temporal Alignment

- Given $i_1$ in $T_1$ and $i_2$ in $T_2$,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?

- In the simplest case $T_1$ and $T_2$ may follow the same path
  [Mellor-Crummey, ASPLOS 1989]

```
foo()
```
Temporal Alignment

- Given $i_1$ in $T_1$ and $i_2$ in $T_2$,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?

- In the simplest case $T_1$ and $T_2$ may follow the same path
  [Mellor-Crummey, ASPLOS 1989]
Temporal Alignment

- Given $i_1$ in $T_1$ and $i_2$ in $T_2$,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?

- In the simplest case $T_1$ and $T_2$ may follow the same path
  [Mellor-Crummey, ASPLOS 1989]
Temporal Alignment

- Given $i_1$ in $T_1$ and $i_2$ in $T_2$,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?

- In the simplest case $T_1$ and $T_2$ may follow the same path
  [Mellor-Crummey, ASPLOS 1989]
Temporal Alignment

- Given $i_1$ in $T_1$ and $i_2$ in $T_2$,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?

- In the simplest case $T_1$ and $T_2$ may follow the same path
  [Mellor-Crummey, ASPLOS 1989]
Temporal Alignment

- Given $i_1$ in $T_1$ and $i_2$ in $T_2$,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?

- In the simplest case $T_1$ and $T_2$ may follow the same path
  [Mellor-Crummey, ASPLOS 1989]
Temporal Alignment

- Given $i_1$ in $T_1$ and $i_2$ in $T_2$,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?

- In the simplest case $T_1$ and $T_2$ may follow the same path
  [Mellor-Crummey, ASPLOS 1989]
Temporal Alignment

- Given $i_1$ in $T_1$ and $i_2$ in $T_2$,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?

- In the simplest case $T_1$ and $T_2$ may follow the same path
  [Mellor-Crummey, ASPLOS 1989]
Temporal Alignment

- Given $i_1$ in $T_1$ and $i_2$ in $T_2$,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?

- In the simplest case $T_1$ and $T_2$ may follow the same path
  [Mellor-Crummey, ASPLOS 1989]

\[
\text{foo()}
\]

Position along a path can be maintained via a counter
Temporal Alignment

- Given $i_1$ in $T_1$ and $i_2$ in $T_2$,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?

- In the simplest case $T_1$ and $T_2$ may follow the same path
  [Mellor-Crummey, ASPLOS 1989]

Position along a path can be maintained via a counter

Only need to increment along
1) back edges
2) function calls
Temporal Alignment

- Given \( i_1 \) in \( T_1 \) and \( i_2 \) in \( T_2 \),
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?

- In the simplest case \( T_1 \) and \( T_2 \) may follow the same path
  [Mellor-Crummey, ASPLOS 1989]

- Suppose that we know the programs are acyclic?
Temporal Alignment

- Given $i_1$ in $T_1$ and $i_2$ in $T_2$,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?

- In the simplest case $T_1$ and $T_2$ may follow the same path
  [Mellor-Crummey, ASPLOS 1989]

- Suppose that we know the programs are acyclic?
Temporal Alignment

- Given $i_1$ in $T_1$ and $i_2$ in $T_2$,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?

- In the simplest case $T_1$ and $T_2$ may follow the same path
  [Mellor-Crummey, ASPLOS 1989]

- Suppose that we know the programs are acyclic?

The position in the DAG relates the paths
Temporal Alignment

- Given $i_1$ in $T_1$ and $i_2$ in $T_2$,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?

- In the simplest case $T_1$ and $T_2$ may follow the same path
  [Mellor-Crummey, ASPLOS 1989]

- Suppose that we know the programs are acyclic?

- Now consider the case where cycles can occur... [Sumner, ASE 2013]

How can we extend the acyclic case?
Temporal Alignment

- Given $i_1$ in $T_1$ and $i_2$ in $T_2$, when should we say that they correspond? [Xin, PLDI 2008] [Sumner, ASE 2013]
- How can we compute such relations?

- In the simplest case $T_1$ and $T_2$ may follow the same path [Mellor-Crummey, ASPLOS 1989]

- Suppose that we know the programs are acyclic?

- **Now consider the case where cycles can occur...** [Sumner, ASE 2013]
  - How can we extend the acyclic case?

We can unwind the loop to make it logically acyclic.
Temporal Alignment

- Given $i_1$ in $T_1$ and $i_2$ in $T_2$, where do we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
- How can we compute such relations?

- In the simplest case, $T_1$ and $T_2$ may follow the same path [Mellor-Crummey, ASPLOS 1989]

- Suppose that we know the programs are acyclic?

- Now consider the case where cycles can occur... [Sumner, ASE 2013]

How can we extend the acyclic case?
Temporal Alignment

- Given $i_1$ in $T_1$ and $i_2$ in $T_2$, when should we say that they correspond? [Xin, PLDI 2008] [Sumner, ASE 2013]

  - How can we compute such relations?

- In the simplest case, $T_1$ and $T_2$ may follow the same path [Mellor-Crummey, ASPLOS 1989]

- Suppose that we know the programs are acyclic?

- Now consider the case where cycles can occur... [Sumner, ASE 2013]

  How can we extend the acyclic case?
Temporal Alignment

- Given $i_1$ in $T_1$ and $i_2$ in $T_2$, when should we say that they correspond? [Xin, PLDI 2008] [Sumner, ASE 2013]
- How can we compute such relations?

- In the simplest case, $T_1$ and $T_2$ may follow the same path. [Mellor-Crummey, ASPLOS 1989]

- Suppose that we know the programs are acyclic?

- **Now consider the case where cycles can occur...** [Sumner, ASE 2013]

  How can we extend the acyclic case?
Temporal Alignment

- Given \( i_1 \) in \( T_1 \) and \( i_2 \) in \( T_2 \), when should we say that they correspond?
- How can we compute such relations?
- In the simplest case, \( T_1 \) and \( T_2 \) may follow the same path.
- Suppose that we know the programs are acyclic.
- Now consider the case where cycles can occur...

These are different iterations of one loop. A counter for each active loop suffices (mostly).

How can we extend the acyclic case?
Temporal Alignment

- Given $i_1$ in $T_1$ and $i_2$ in $T_2$, when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]

- How can we compute such relations?

- In the simplest case $T_1$ and $T_2$ may follow the same path. [Mellor-Crummey, ASPLOS 1989]

- Suppose that we know the programs are acyclic?

- Now consider the case where cycles can occur... [Sumner, ASE 2013]

  How can we extend the acyclic case?

  1 counter per active loop + the call stack disambiguates!
Temporal Alignment

- Given $i_1$ in $T_1$ and $i_2$ in $T_2$,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?

- In the simplest case $T_1$ and $T_2$ may follow the same path
  [Mellor-Crummey, ASPLOS 1989]

- Suppose that we know the programs are acyclic?

- Now consider the case where cycles can occur... [Sumner, ASE 2013]
  - Can we efficiently represent this?

\[
\text{? } i_1 \leftarrow i_2
\]
Temporal Alignment

- Given $i_1$ in $T_1$ and $i_2$ in $T_2$, when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
- how can we compute such relations?

- In the simplest case $T_1$ and $T_2$ may follow the same path [Mellor-Crummey, ASPLOS 1989]

- Suppose that we know the programs are acyclic?

- Now consider the case where cycles can occur... [Sumner, ASE 2013]
  - Can we efficiently represent this?
Temporal Alignment

- Given $i_1$ in $T_1$ and $i_2$ in $T_2$,
  - when should we say that they correspond? [Xin, PLDI 2008][Sumner, ASE 2013]
  - how can we compute such relations?

- In the simplest case $T_1$ and $T_2$ may follow the same path
  [Mellor-Crummey, ASPLOS 1989]

- Suppose that we know the programs are acyclic?

- Now consider the case where cycles can occur... [Sumner, ASE 2013]
  - Can we efficiently represent this?

![Diagram showing call stack/context, iteration stack, and instruction ID, with indication that some can be encoded/inferred and others can be inferred.]
Spatial Alignment

- We must also ask what it means to compare program state across executions
Spatial Alignment

- We must also ask what it means to compare program state across executions
  - How can we compare two integers $X$ and $Y$?

  $3 \neq 5$
Spatial Alignment

- We must also ask what it means to compare program state across executions
  - How can we compare two integers X and Y?
  - How can we compare two pointers A and B?

0xdeadbeef in T1  ≠  0xcafe00d in T2
Spatial Alignment

- We must also ask what it means to compare program state across executions
  - How can we compare two integers X and Y?
  - How can we compare two pointers A and B?

0xdeadbeef in T1 ≠ 0xc0ffee00d in T2

If you allocated other stuff in only one run, this can be true even without ASLR!
Spatial Alignment

- We must also ask what it means to compare program state across executions
  - How can we compare two integers X and Y?
  - How can we compare two pointers A and B?
  - How can we compare allocated regions on the heap? Should they even be compared?!
Spatial Alignment

- We must also ask what it means to compare program state across executions
  - How can we compare two integers X and Y?
  - How can we compare two pointers A and B?
  - How can we compare allocated regions on the heap? Should they even be compared?!

- In practice, comparing state across executions requires comparing memory graphs
  - We need a way to identify corresponding nodes (state elements)
Spatial Alignment

- We must also ask what it means to compare program state across executions:
  - How can we compare two integers X and Y?
  - How can we compare two pointers A and B?
  - How can we compare allocated regions on the heap? Should they even be compared?!!

- In practice, comparing state across executions requires comparing memory graphs:
  - We need a way to identify corresponding nodes (state elements)
Spatial Alignment

- We must also ask what it means to compare program state across executions.
  - How can we compare two integers X and Y?
  - How can we compare two pointers A and B?
  - How can we compare allocated regions on the heap? Should they even be compared?!!?

In practice, comparing state across executions requires comparing memory graphs.
  - We need a way to identify corresponding nodes (state elements).
We must also ask what it means to compare program state across executions:

- How can we compare two integers X and Y?
- How can we compare two pointers A and B?
- How can we compare allocated regions on the heap? Should they even be compared?!

In practice, comparing state across executions requires comparing memory graphs:
- We need a way to identify corresponding nodes (state elements)
Spatial Alignment

- We must also ask what it means to compare program state across executions:
  - How can we compare two integers \( X \) and \( Y \)?
  - How can we compare two pointers \( A \) and \( B \)?
  - How can we compare allocated regions on the heap? Should they even be compared?!?

- In practice, comparing state across executions requires comparing memory graphs:
  - We need a way to identify corresponding nodes (state elements)

What are the differences?

1) \( \text{list.append(value++)} \)
2) if \( c \):
3) \( \text{list.append(value++)} \)
4) \( \text{list.append(value++)} \)
Spatial Alignment

- We must also ask what it means to compare program state across executions –
  - How can we compare two integers X and Y?
  - How can we compare two pointers A and B?
  - How can we compare allocated regions on the heap?
  - Should they even be compared?!

- In practice, comparing state across executions requires comparing memory graphs
  - We need a way to identify corresponding nodes (state elements)

What are the differences?
1) `list.append(value++)`
2) `if c:`
3) `list.append(value++)`
4) `list.append(value++)`
• We must also ask what it means to compare program state across executions – How can we compare two integers X and Y? Should they even be compared?!?

• In practice, comparing state across executions requires comparing memory graphs – We need a way to identify corresponding nodes (state elements)

What are the differences?
1) `list.append(value++)`
2) `if c:`
3) `list.append(value++)`
4) `list.append(value++)`
We must also ask what it means to compare program state across executions—

- How can we compare two integers $X$ and $Y$?
- How can we compare two pointers $A$ and $B$?
- How can we compare allocated regions on the heap?
  Should they even be compared?!?

In practice, comparing state across executions requires comparing memory graphs—

- We need a way to identify corresponding nodes (state elements)
We must also ask what it means to compare program state across executions—
- How can we compare two integers $X$ and $Y$?
- How can we compare two pointers $A$ and $B$?
- How can we compare allocated regions on the heap? Should they even be compared?!?

In practice, comparing state across executions requires comparing memory graphs—
- We need a way to identify corresponding nodes (state elements)
We must also ask what it means to compare program state across executions –
- How can we compare two integers $X$ and $Y$?
- How can we compare two pointers $A$ and $B$?
- How can we compare allocated regions on the heap? Should they even be compared?!?

In practice, comparing state across executions requires comparing memory graphs –
- We need a way to identify corresponding nodes (state elements)

Again, the semantics of the program dictate the solution –
- Identify heap allocations by the aligned time of allocation
Dual Slicing

- Now we can
  - Identify corresponding times across executions
  - Identify & compare corresponding state at those times
Dual Slicing

- Now we can
  - Identify corresponding times across executions
  - Identify & compare corresponding state at those times

- We can use these to enhance dynamic slicing by being aware of differences! (called dual slicing)
Dual Slicing

• Now we can
  – Identify corresponding times across executions
  – Identify & compare corresponding state at those times

• We can use these to enhance dynamic slicing by being aware of differences! (called dual slicing)
  – Based on classic dynamic slicing
  – Include transitive dependencies that differ or exist in only 1 execution
Dual Slicing

• Now we can
  – Identify corresponding times across executions
  – Identify & compare corresponding state at those times

• We can use these to enhance dynamic slicing by being aware of differences! (called dual slicing)
  – Based on classic dynamic slicing
  – Include transitive dependencies that differ or exist in only 1 execution
Dual Slicing

- Now we can
  - Identify corresponding times across executions
  - Identify & compare corresponding state at those times

- We can use these to enhance dynamic slicing by being aware of differences! (called dual slicing)
  - Based on classic dynamic slicing
  - Include transitive dependencies that differ or exist in only 1 execution

```
1) x ← 0
2) y ← 1
3) print(x+y)
```

```
1) x ← 1
2) y ← 1
3) print(x+y)
```
Now we can
- Identify corresponding times across executions
- Identify & compare corresponding state at those times

We can use these to enhance dynamic slicing by being aware of differences! (called dual slicing)
- Based on classic dynamic slicing
- Include transitive dependencies that differ or exist in only 1 execution
Dual Slicing

- The differences in dependencies capture multiple kinds of information
Dual Slicing

- The differences in dependencies capture multiple kinds of information
  - Value-only differences
Dual Slicing

- The differences in dependencies capture multiple kinds of information
  - Value-only differences
  - Provenance/Source differences
Dual Slicing

- The differences in dependencies capture multiple kinds of information
  - Value-only differences
  - Provenance/Source differences
  - Missing/Extra behavior
Dual Slicing

- The differences in dependencies capture multiple kinds of information
  - Value-only differences
  - Provenance/Source differences
  - Missing/Extra behavior

- Recall: Dynamic slicing could not handle execution omission, but dual slicing can!
Dual Slicing

- The differences in dependencies capture multiple kinds of information
  - Value-only differences
  - Provenance/Source differences
  - Missing/Extra behavior
- Recall: Dynamic slicing could not handle execution omission, but dual slicing can!
- **Dual slices can be effective for concurrent debugging & exploit analysis**
  [Weeratunge, ISSTA 2010][Johnson, S&P 2011]
Adding Causation

- Now we can produce explanations exactly like our example!
  - Can answer “Why” and “Why not” questions about behavior & differences
    [Ko, ICSE 2008]
Adding Causation

- Now we can produce explanations exactly like our example!
  - Can answer “Why” and “Why not” questions about behavior & differences [Ko, ICSE 2008]
  - But they may still contain extra information/noise...
Adding Causation

- Now we can produce explanations exactly like our example!
  - Can answer “Why” and “Why not” questions about behavior & differences [Ko, ICSE 2008]
  - But they may still contain extra information/noise...

```python
1) x = ...
2) y = ...
3) if x + y > 0:
   4)     z = 0
5) else:
6)     z = 1
7) print(z)
```
Adding Causation

- Now we can produce explanations exactly like our example!
  - Can answer “Why” and “Why not” questions about behavior & differences [Ko, ICSE 2008]
  - But they may still contain extra information/noise...

```python
1) x = ...
2) y = ...
3) if x + y > 0:
   4)     z = 0
   5) else:
   6)     z = 1
7) print(z)
```

Correct

```
Correct
x = 10
y = -1
True
z = 0
```

```
"0"
```
Adding Causation

- Now we can produce explanations exactly like our example!
  - Can answer “Why” and “Why not” questions about behavior & differences [Ko, ICSE 2008]
  - But they may still contain extra information/noise...

```
1) x = ...
2) y = ...
3) if x + y > 0:
   4)     z = 0
   5) else:
   6)     z = 1
7) print(z)
```

<table>
<thead>
<tr>
<th>Correct</th>
<th>Buggy</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 10</td>
<td>x = 0</td>
</tr>
<tr>
<td>y = -1</td>
<td>y = -2</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>z = 0</td>
<td>z = 0</td>
</tr>
</tbody>
</table>

```
"0"
"1"
```
Adding Causation

- Now we can produce explanations exactly like our example!
  - Can answer “Why” and “Why not” questions about behavior & differences [Ko, ICSE 2008]
  - But they may still contain extra information/noise...

```
1) x = ...
2) y = ...
3) if x + y > 0:
   True
   z = 0
   "0"
3) else:
   z = 1
   "1"
7) print(z)
```

```
Correct  Buggy
x = 10   x = 0
y = -1   y = -2
True     False
z = 0     z = 1
"0"      "1"
```
Adding Causation

- Now we can produce explanations exactly like our example!
  - Can answer “Why” and “Why not” questions about behavior & differences [Ko, ICSE 2008]
  - But they may still contain extra information/noise...

```
1) x = ...
2) y = ...
3) if x + y > 0:
   True
   z = 0
   else:
   z = 1
4)     z = 0
5)     If
6)     z = 1
7) print(z)
```

**Correct**

1) x = 10
2) y = -1
3) if x + y > 0: True
   z = 0
4)     z = 0
5)     If
6)     z = 1
7) print(z)

**Buggy**

1) x = 0
2) y = -2
3) if x + y > 0: False
   z = 1
4)     z = 1
5)     If
6)     z = 1
7) print(z)
Adding Causation

• Now we can produce explanations exactly like our example!
  – Can answer “Why” and “Why not” questions about behavior & differences
    [Ko, ICSE 2008]
  – But they may still contain extra information/noise...

1) \( x = \ldots \)
2) \( y = \ldots \)
3) if \( x + y > 0 \):
   True
   \( z = 0 \)
5) else:
6) \( z = 1 \)
7) print(\( z \))

<table>
<thead>
<tr>
<th>Correct</th>
<th>Buggy</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = 10 )</td>
<td>( x = 0 )</td>
</tr>
<tr>
<td>( y = -1 )</td>
<td>( y = -2 )</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>( z = 0 )</td>
<td>( z = 1 )</td>
</tr>
<tr>
<td>“0”</td>
<td>“1”</td>
</tr>
</tbody>
</table>

Correct

1
2
3
4
5
6
7

Buggy

1
2
3
6
7

Correct

Buggy
Adding Causation

• Now we can produce explanations exactly like our example!
  – Can answer “Why” and “Why not” questions about behavior & differences [Ko, ICSE 2008]
  – But they may still contain extra information/noise...

1) \(x = \ldots\)
2) \(y = \ldots\)
3) if \(x + y > 0\):
   True
   \(z = 0\)
4) \(z = 0\)
5) else:
6) \(z = 1\)
7) print(z)

Correct

Buggy

1) \(x = 10\)
2) \(y = -1\)
3) \(True\)
4) \(z = 0\)

Correct

1) \(x = 0\)
2) \(y = -2\)
3) \(False\)
4) \(z = 1\)

Buggy

"0"
"1"
Adding Causation

- Now we can produce explanations exactly like our example!
  - Can answer “Why” and “Why not” questions about behavior & differences [Ko, ICSE 2008]
  - But they may still contain extra information/noise...

Correct

1) \( x = \ldots \)
2) \( y = \ldots \)
3) if \( x + y > 0 \):
   True
   z = 0
4) z = 0
5) else:
6) z = 1
7) print(z)

Buggy

1) \( x = \ldots \)
2) \( y = \ldots \)
3) if \( x + y > 0 \):
   False
   z = 1
4) z = 0
5) else:
6) z = 1
7) print(z)
Adding Causation

Now we can produce explanations exactly like our example!
- Can answer “Why” and “Why not” questions about behavior & differences [Ko, ICSE 2008]
- But they may still contain extra information/noise...

Dual slicing captures differences, not causes. What does that mean here?

1) \( x = \ldots \)  
2) \( y = \ldots \)  
3) if \( x + y > 0 \):  
   True \( \Rightarrow z = 0 \)  
   False \( \Rightarrow z = 0 \)  
4) \( z = 0 \)  
5) else:  
6) \( z = 1 \)  
7) print(z)

Correct:  
- \( x = 10 \)  
- \( y = -1 \)  
- True \( \Rightarrow z = 0 \)  
- “0”

Buggy:  
- \( x = 0 \)  
- \( y = -2 \)  
- False \( \Rightarrow z = 0 \)  
- “1”

Correct

Buggy
Now we can produce explanations exactly like our example!

- Can answer “Why” and “Why not” questions about behavior & differences [Ko, ICSE 2008]
- But they may still contain extra information/noise...

```
1) x = ...
2) y = ...
3) if x + y > 0:
    True
    z = 0
    4)     z = 0
    5) else:
    6)     z = 1
7) print(z)
```

**Correct**
- x = 10
- y = -1
- True
- z = 0
- "0"

**Buggy**
- x = 0
- y = -2
- False
- z = 1
- "1"
Adding Causation

- Now we can produce explanations exactly like our example!
  - Can answer “Why” and “Why not” questions about behavior & differences.
  - But they may still contain extra information/noise...

```
1) x = ...
2) y = ...
3) if x + y > 0:
   4)     z = 0
   5) else:
   6)     z = 1
7) print(z)
```

Correct

```
x = 10
y = -1
```

Buggy

```
x = 0
y = -2
```

Correct

```
True
z = 0
```

Buggy

```
False
```

```
0
```

```
1
```

The cost of these extra edges is high in practice! All transitive dependencies...

1) x = ...
2) y = ...
3) if x + y > 0:
   4)     z = 0
   5) else:
   6)     z = 1
7) print(z)
Adding Causation

- So what would we want an explanation to contain?
  - This is an area with unsolved problems & open research
Adding Causation

- So what would we want an explanation to contain?
  - This is an area with unsolved problems & open research
  - What does it mean for one explanation to be better than another?
Adding Causation

- So what would we want an explanation to contain?
  - This is an area with unsolved problems & open research
  - What does it mean for one explanation to be better than another?

- There are several things we could consider
  - In general, simpler explanations are preferred
Adding Causation

- So what would we want an explanation to contain?
  - This is an area with unsolved problems & open research
  - What does it mean for one explanation to be better than another?

- There are several things we could consider
  - In general, simpler explanations are preferred
  - Minimize the “# steps”? 

![Diagram](image_url)
Adding Causation

- So what would we want an explanation to contain?
  - This is an area with unsolved problems & open research
  - What does it mean for one explanation to be better than another?

- There are several things we could consider
  - In general, simpler explanations are preferred
  - Minimize the “# steps”?
  - Minimize the “# dependencies”?
Adding Causation

- So what would we want an explanation to contain?
  - This is an area with unsolved problems & open research
  - What does it mean for one explanation to be better than another?

- There are several things we could consider
  - In general, simpler explanations are preferred
  - Minimize the “# steps”?
  - Minimize the “# dependencies”?

What big challenges do you see with these 2 approaches?
Adding Causation

- So what would we want an explanation to contain?
  - This is an area with unsolved problems & open research
  - What does it mean for one explanation to be better than another?

- **There are several things we could consider**
  - In general, simpler explanations are preferred
  - Minimize the “# steps”? 
  - Minimize the “# dependencies”? 
  - Minimize the “# local dependencies”?
Adding Causation

- So what would we want an explanation to contain?
  - This is an area with unsolved problems & open research
  - What does it mean for one explanation to be better than another?

- There are several things we could consider
  - In general, simpler explanations are preferred
  - Minimize the “# steps”?
  - Minimize the “# dependencies”?
  - Minimize the “# local dependencies”?

\[
\arg\min_{s \in sd} |sd| \\
\land E_1[sd(E_2)_i] \rightarrow sd(E_2)_{i+1} \\
\land E_2[sd(E_1)_i] \rightarrow sd(E_1)_{i+1}
\]
Adding Causation

- So what would we want an explanation to contain?
  - This is an area with unsolved problems & open research
  - What does it mean for one explanation to be better than another?

- There are several things we could consider
  - In general, simpler explanations are preferred
  - Minimize the “# steps”?
  - Minimize the “# dependencies”?
  - Minimize the “# local dependencies”?

\[
\arg\min_{s \in sd, |sd|} \quad \begin{align*}
\land E &1[sd(E2)_i] \rightarrow sd(E2)_{i+1} \\
\land E &2[sd(E1)_i] \rightarrow sd(E1)_{i+1}
\end{align*}
\]
Adding Causation

- So what would we want an explanation to contain?
  - This is an area with unsolved problems & open research
  - What does it mean for one explanation to be better than another?

- There are several things we could consider
  - In general, simpler explanations are preferred
  - Minimize the “# steps”?
  - Minimize the “# dependencies”?
  - Minimize the “# local dependencies”?
Adding Causation

- So what would we want an explanation to contain?
  - This is an area with unsolved problems & open research
  - What does it mean for one explanation to be better than another?

- There are several things we could consider
  - In general, simpler explanations are preferred
  - Minimize the “# steps”? 
  - Minimize the “# dependencies”?
  - Minimize the “# local dependencies”? 

Diagram: Two explanations, E1 and E2, with numbered nodes connected by arrows.
Adding Causation

- So what would we want an explanation to contain?
  - This is an area with unsolved problems & open research
  - What does it mean for one explanation to be better than another?

- There are several things we could consider
  - In general, simpler explanations are preferred
  - Minimize the “# steps”?
  - Minimize the “# dependencies”?
  - Minimize the “# local dependencies”?
Adding Causation

- So what would we want an explanation to contain?
  - This is an area with unsolved problems & open research
  - What does it mean for one explanation to be better than another?

- There are several things we could consider
  - In general, simpler explanations are preferred
  - Minimize the “# steps”?
  - Minimize the “# dependencies”?
  - Minimize the “# local dependencies”?
Adding Causation

• So what would we want an explanation to contain?
  – This is an area with unsolved problems & open research
  – What does it mean for one explanation to be better than another?

• There are several things we could consider
  – In general, simpler explanations are preferred
  – Minimize the “# steps”?
  – Minimize the “# dependencies”?
  – Minimize the “# local dependencies”?

• There are currently unknown trade offs between tractability, intuitiveness, and correctness
Adding Causation

- So what would we want an explanation to contain?
  - This is an area with unsolved problems & open research
  - What does it mean for one explanation to be better than another?

- There are several things we could consider
  - In general, simpler explanations are preferred
  - Minimize the “# steps”?
  - Minimize the “# dependencies”?
  - Minimize the “# local dependencies”?

- There are currently unknown trade offs between tractability, intuitiveness, and correctness

Even local blame is actually challenging
Adding Causation

- Causation is often framed via “alternate worlds” & “what if” questions...
  - We can answer these causality questions by running experiments!
What Should We Blame?
What Should We Blame?

Trial

?
What Should We Blame?

\[
\begin{array}{c}
x = 5 \\
y = 4 \\
z = 3 \\
\end{array}
\quad \text{Trial} \quad \begin{array}{c}
x = 5 \\
y = 4 \\
z = 1 \\
\end{array}
\]

?
What Should We Blame?

\[
\begin{array}{c}
x = 5 \\
y = 4 \\
z = 3 \\
\end{array}
\quad \text{Trial} \quad \begin{array}{c}
x = 5 \\
y = 4 \\
z = 1 \\
\end{array}
\]
What Should We Blame?

Trial

What does this patched run even mean?
Example – Altered Meaning

**Buggy**

1) $x \leftarrow \text{input}()$
2) $y \leftarrow \text{input}()$
3) $z \leftarrow \text{input}()$
4) if $y+z > 10$:
5) $y \leftarrow 5$
6) else: $y \leftarrow y+1$
7) \text{print}(y)

**Correct**

1) $x \leftarrow 0$
2) $y \leftarrow 7$
3) $z \leftarrow 3$
4) if False:
5) $y \leftarrow 5$
6) else: $y \leftarrow 8$
7) \text{print}(8)
What should we blame here?

**Buggy**

1) $x \leftarrow \text{input()}$
2) $y \leftarrow \text{input()}$
3) $z \leftarrow \text{input()}$
4) if $y+z > 10$
5)  $y \leftarrow 5$
6) else: $y \leftarrow y+1$
7) print($y$)

**Correct**

1) $x \leftarrow 0$
2) $y \leftarrow 7$
3) $z \leftarrow 3$
4) if False
5) $y \leftarrow 8$
6) else: $y \leftarrow 4$
7) print($8$)
Example – Altered Meaning

Buggy
1) x ← input()
2) y ← input()
3) z ← input()
4) if y+z > 10:
   5) y ← 5
   6) else: y ← y+1
7) print(y)

Trial
x ← 0
y ← 7
z ← 3
if False:
   else: y ← 8
print(8)

Correct
x ← 1
y ← 3
z ← 6
if False:
   else: y ← 4
print(4)
**Example – Altered Meaning**

<table>
<thead>
<tr>
<th>Buggy</th>
<th>Trial</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) (x \leftarrow \text{input}())</td>
<td>(x \leftarrow 0)</td>
<td>(x \leftarrow 1)</td>
</tr>
<tr>
<td>2) (y \leftarrow \text{input}())</td>
<td>(y \leftarrow 7)</td>
<td>(y \leftarrow 3)</td>
</tr>
<tr>
<td>3) (z \leftarrow \text{input}())</td>
<td>(z \leftarrow 3)</td>
<td>(z \leftarrow 6)</td>
</tr>
<tr>
<td>4) if (y+z &gt; 10):</td>
<td>if False:</td>
<td>if False:</td>
</tr>
<tr>
<td>5) (y \leftarrow 5)</td>
<td>else: (y \leftarrow 8)</td>
<td>else: (y \leftarrow 4)</td>
</tr>
<tr>
<td>6) else: (y \leftarrow y+1)</td>
<td>print(8)</td>
<td>print(4)</td>
</tr>
<tr>
<td>7) print(y)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example – Altered Meaning

<table>
<thead>
<tr>
<th>Buggy</th>
<th>Trial</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>1)x ← input()</td>
<td>x ← 0</td>
<td>x ← 1</td>
</tr>
<tr>
<td>2)y ← input()</td>
<td>y ← 7</td>
<td>y ← 3</td>
</tr>
<tr>
<td>3)z ← input()</td>
<td>z ← 3</td>
<td>z ← 6</td>
</tr>
<tr>
<td>4)if y+z &gt; 10:</td>
<td>if False:</td>
<td>if False:</td>
</tr>
<tr>
<td>5) y ← 5</td>
<td>else: y ← 8</td>
<td>else: y ← 4</td>
</tr>
<tr>
<td>6)else: y ← y+1</td>
<td>print(8)</td>
<td>print(8)</td>
</tr>
<tr>
<td>7)print(y)</td>
<td></td>
<td>print(4)</td>
</tr>
</tbody>
</table>
### Example – Altered Meaning

**Buggy**

1) \(x \leftarrow \text{input}()\)
2) \(y \leftarrow \text{input}()\)
3) \(z \leftarrow \text{input}()\)
4) if \(y + z > 10\):  
   5) \(y \leftarrow 5\)
   6) else: \(y \leftarrow y + 1\)
7) \(\text{print}(y)\)

**Trial**

1) \(x \leftarrow 0\)
2) \(y \leftarrow 7\)
3) \(z \leftarrow 3\)
4) if False:  
   5) \(y \leftarrow 8\)
7) \(\text{print}(8)\)

**Correct**

1) \(x \leftarrow 1\)
2) \(y \leftarrow 3\)
3) \(z \leftarrow 6\)
4) if False:  
   5) \(y \leftarrow 4\)
7) \(\text{print}(4)\)
### Example – Altered Meaning

<table>
<thead>
<tr>
<th>Buggy</th>
<th>Trial</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) <code>x ← input()</code></td>
<td><code>x ← 0</code></td>
<td><code>x ← 1</code></td>
</tr>
<tr>
<td>2) <code>y ← input()</code></td>
<td><code>y ← 7</code></td>
<td><code>y ← 7</code></td>
</tr>
<tr>
<td>3) <code>z ← input()</code></td>
<td><code>z ← 3</code></td>
<td><code>z ← 6</code></td>
</tr>
<tr>
<td>4) <code>if y+z &gt; 10:</code></td>
<td><code>if False:</code></td>
<td><code>if False:</code></td>
</tr>
<tr>
<td>5)   y ← 5</td>
<td>else: <code>y ← y+1</code></td>
<td>else: <code>y ← 4</code></td>
</tr>
<tr>
<td>6) else: <code>y ← y+1</code></td>
<td>print(8)</td>
<td>print(4)</td>
</tr>
<tr>
<td>7) print(y)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Example – Altered Meaning

<table>
<thead>
<tr>
<th></th>
<th>Buggy</th>
<th>Trial</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>x ← input()</code></td>
<td><code>x ← 0</code></td>
<td><code>x ← 1</code></td>
</tr>
<tr>
<td>2</td>
<td><code>y ← input()</code></td>
<td><code>y ← 7</code></td>
<td><code>y ← 7</code></td>
</tr>
<tr>
<td>3</td>
<td><code>z ← input()</code></td>
<td><code>z ← 3</code></td>
<td><code>z ← 6</code></td>
</tr>
<tr>
<td>4</td>
<td><code>if y+z &gt; 10:</code></td>
<td><code>if False:</code></td>
<td><code>if True:</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>y ← 5</code></td>
<td><code>y ← 5</code></td>
</tr>
<tr>
<td>5</td>
<td><code>else: y ← y+1</code></td>
<td><code>else: y ← 8</code></td>
<td><code>else: y ← 4</code></td>
</tr>
<tr>
<td>6</td>
<td><strong>print(y)</strong></td>
<td><strong>print(8)</strong></td>
<td><strong>print(3)</strong></td>
</tr>
</tbody>
</table>

Corrected code:

```python
1)x ← 1
2)y ← 7
3)z ← 6
4)if True:
   y ← 5
5)print(y)
```

Incorrect code:

```python
1)x ← 0
2)y ← 7
3)z ← 3
4)if False:
   y ← 8
5)print(y)
```
Example – Altered Meaning

<table>
<thead>
<tr>
<th>Buggy</th>
<th>Trial</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1)</strong> (x \leftarrow \text{input}())</td>
<td>(x \leftarrow 0)</td>
<td>(x \leftarrow 1)</td>
</tr>
<tr>
<td><strong>2)</strong> (y \leftarrow \text{input}())</td>
<td>(y \leftarrow 7)</td>
<td>(y \leftarrow 7)</td>
</tr>
<tr>
<td><strong>3)</strong> (z \leftarrow \text{input}())</td>
<td>(z \leftarrow 3)</td>
<td>(z \leftarrow 6)</td>
</tr>
<tr>
<td><strong>4)</strong> if (y+z &gt; 10):</td>
<td>if False:</td>
<td>if True:</td>
</tr>
<tr>
<td><strong>5)</strong> (y \leftarrow 5)</td>
<td></td>
<td>(y \leftarrow 5)</td>
</tr>
<tr>
<td><strong>6)</strong> else: (y \leftarrow y+1)</td>
<td>else: (y \leftarrow 8)</td>
<td>else: (y \leftarrow 4)</td>
</tr>
<tr>
<td><strong>7)</strong> print(y)</td>
<td>print(8)</td>
<td>print(5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>print(4)</td>
</tr>
</tbody>
</table>

- New control flow unlike original runs
- Occurs in large portion of real bugs
Dual Slicing

Buggy

1) x ← input()
2) y ← input()
3) z ← input()
4) if y+z > 10:
   5) y ← 5
   6) else: y ← y+1
7) print(y)

Correct

2) y ← input()
6) y ← y+1
7) print(y)
Example – Extracted Meaning

2) $y \leftarrow \text{input}()$
6) $y \leftarrow y+1$
7) $\text{print}(y)$

Buggy

$y \leftarrow 7$
$y \leftarrow 8$
$\text{print}(8)$

Trial

Correct

$y \leftarrow 3$
$y \leftarrow 4$
$\text{print}(4)$
Example – Extracted Meaning

2) \( y \leftarrow \text{input()} \)
6) \( y \leftarrow y+1 \)
7) print(y)

<table>
<thead>
<tr>
<th>Buggy</th>
<th>Trial</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y \leftarrow 7 )</td>
<td>( y \leftarrow 7 )</td>
<td>( y \leftarrow 7 )</td>
</tr>
<tr>
<td>( y \leftarrow 8 )</td>
<td>( y \leftarrow 4 )</td>
<td>( y \leftarrow 4 )</td>
</tr>
<tr>
<td>print(8)</td>
<td>print(4)</td>
<td></td>
</tr>
</tbody>
</table>
### Example – Extracted Meaning

<table>
<thead>
<tr>
<th>Buggy</th>
<th>Trial</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>2) y ← input()</td>
<td>y ← 7</td>
<td>y ← 7</td>
</tr>
<tr>
<td>6) y ← y+1</td>
<td>y ← 8</td>
<td>y ← 8</td>
</tr>
<tr>
<td>7) print(8)</td>
<td>print(8)</td>
<td>print(4)</td>
</tr>
</tbody>
</table>

Trial can now correctly blame y
Adding Causation

• Causation is often framed via “alternate worlds” & “what if” questions...
  – We can answer these causality questions by running experiments!

• We perform these causality tests in both directions in order to collect symmetric information
Adding Causation

- Causation is often framed via “alternate worlds” & “what if” questions...
  - We can answer these causality questions by running experiments!

- We perform these causality tests in both directions in order to collect symmetric information
  - How did the buggy run behave differently than the correct one?
  - How did the correct run behave differently than the buggy one?
  - These questions do not have the same answer!
Adding Causation

• Causation is often framed via “alternate worlds” & “what if” questions...
  – We can answer these causality questions by running experiments!

• We perform these causality tests in both directions in order to collect symmetric information
  – How did the buggy run behave differently than the correct one?
  – How did the correct run behave differently than the buggy one?
  – These questions do not have the same answer!

• In practice, there are additional issues, and even defining causation in this context needs further research.
Adding Causation

- Causation is often framed via “alternate worlds” & “what if” questions...
  - We can answer these causality questions by running experiments!

- We perform these causality tests in both directions in order to collect symmetric information
  - How did the buggy run behave differently than the correct one?
  - How did the correct run behave differently than the buggy one?
  - These questions do not have the same answer!

- In practice, there are additional issues, and even defining causation in this context needs further research.
  - Did we want to blame only $y$ in the example?
  - Pruning blame on $y$ is necessary in many real cases, can they be refined?
  - Can it be done without execution? With a stronger statistical basis?
Summing Up
Key Challenges

- Identifying the information you care about
  - Dynamic dependence? Valid memory? Just the execution outcome?
Key Challenges

- Identifying the information you care about
  - Dynamic dependence? Valid memory? Just the execution outcome?

- Collecting that information efficiently
  - abstraction, encoding, compression, sampling, ...
Key Challenges

- Identifying the information you care about
  - Dynamic dependence? Valid memory? Just the execution outcome?

- Collecting that information efficiently
  - abstraction, encoding, compression, sampling, ...

- Selecting which executions to analyze
  - Existing test suite? Always on runtime? Directed test generation?
  - How does underapproximation affect your conclusions?
  - Can you still achieve your objective in spite of it?
Key Challenges

- Identifying the information you care about
  - Dynamic dependence? Valid memory? Just the execution outcome?

- Collecting that information efficiently
  - abstraction, encoding, compression, sampling, ...

- Selecting which executions to analyze
  - Existing test suite? Always on runtime? Directed test generation?
  - How does underapproximation affect your conclusions?
  - Can you still achieve your objective in spite of it?

- Doing some of the work ahead of time
  - What can you precompute to improve all of the above?
Summary

- Analyze the actual/observed behaviors of a program
Summary

- Analyze the actual/observed behaviors of a program
- Modify or use the program’s behavior to collect information
Summary

- Analyze the actual/observed behaviors of a program
- Modify or use the program’s behavior to collect information
- Analyze the information online or offline
Summary

- Analyze the actual/observed behaviors of a program
- Modify or use the program’s behavior to collect information
- Analyze the information online or offline
- **The precise configuration must be tailored to the objectives & insights**
  - Compiled vs DBI
  - Online vs Postmortem
  - Compressed, Encoded, Samples, ...
  - ...

---

331