Automatically Identifying Critical Input and Code Regions in Applications

Michael Carbin and Martin Rinard

Massachusetts Institute of Technology

Computer Science and Artificial Intelligence Laboratory

Open Challenges in Program Analysis

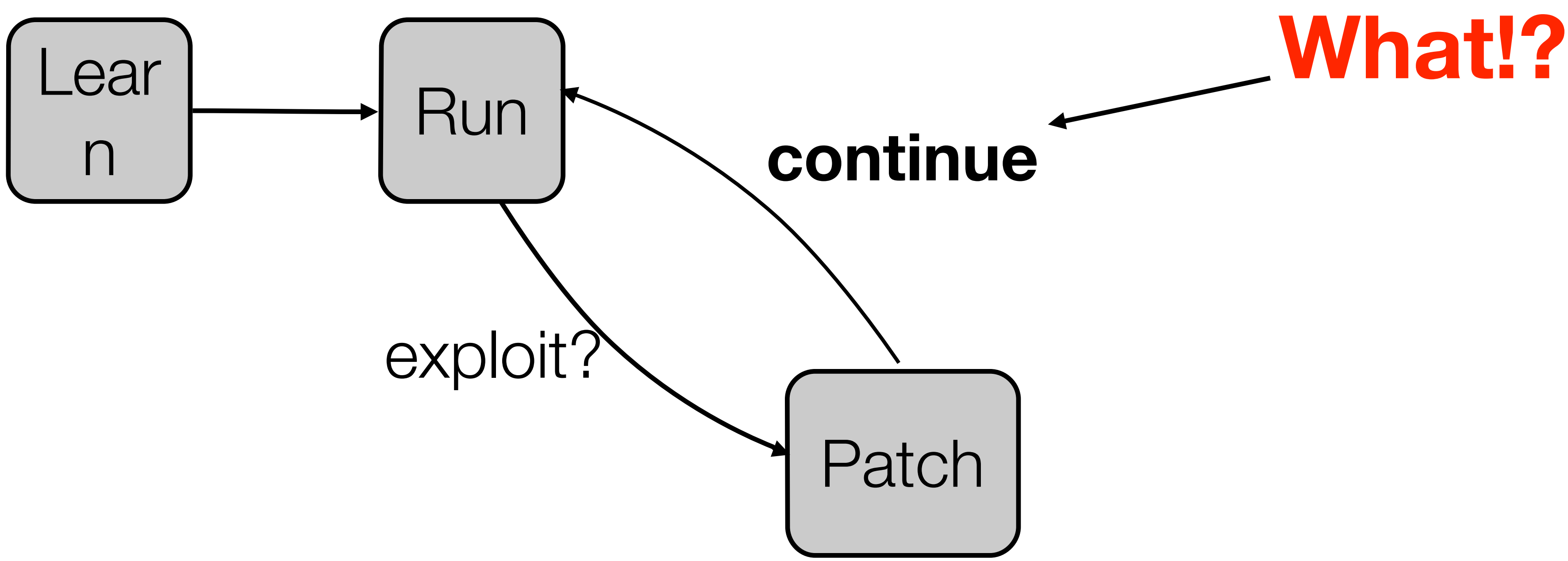
- **Reliability**
  - Memory Safety, Memory Leaks, Data Structure Corruption, Error Recovery
- **Performance**
  - Excess Running Times, Excess Power Consumption
- **Security**
  - Code Injection Attacks

New Class of Solutions

- **Memory Safety** (OSDI'04)
- **Data Structure Repair** (ICSE'05, ISSTA'06)
- **Bounded Memory Consumption** (ISMM'07)
- **Automatic Error Recovery** (ASPLOS'09)
- **Automatic Patching** (SOSP'09)
- **Performance Profiling** (ICSE'10)
- **Reduced Power Consumption** (PLDI10, sub OSDI'10)

UNSOUND

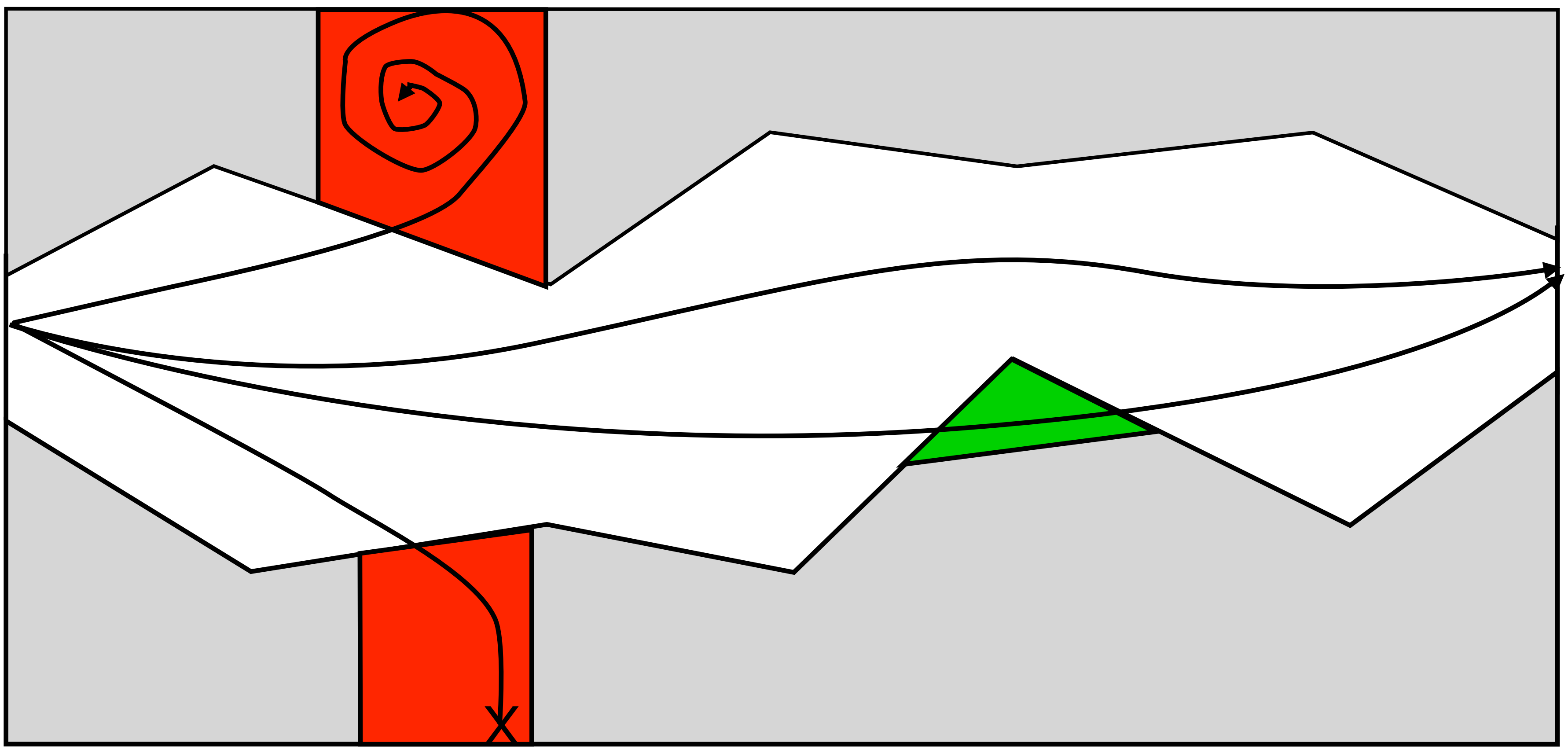
Automatic Patching Against Exploits (SOSP '09)



- **Patch** restores learned invariants.
  - Skip call to a never before seen function.
  - Set variable to a previously seen value.

Visualizing Execution

- Program may transition to previously unreachable program states.



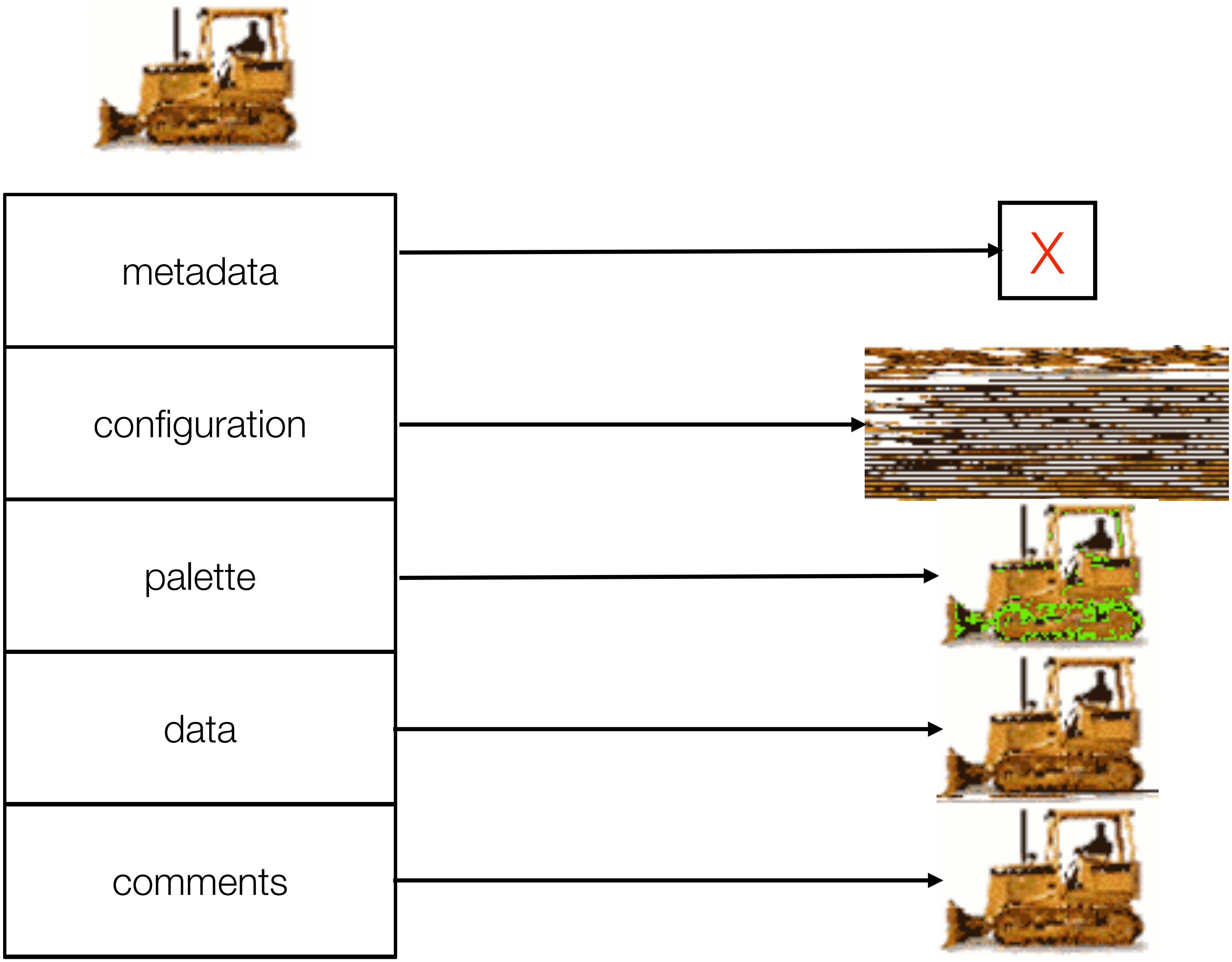
- Solution is to identify regions that won't get us into **trouble**.

The Takeaway

- **Critical** Input and Code Regions
  - **Hard** functional correctness requirements - must.
- **Forgiving** Input and Code Regions
  - **Soft** functional correctness requirements - may.
- **Regions are characterized by application's response to change.**
  - **Critical** - intolerant to change.
  - **Forgiving** - tolerant to change.
- **We can automatically determine regions by modeling application response.**

Critical and Forgiving Regions

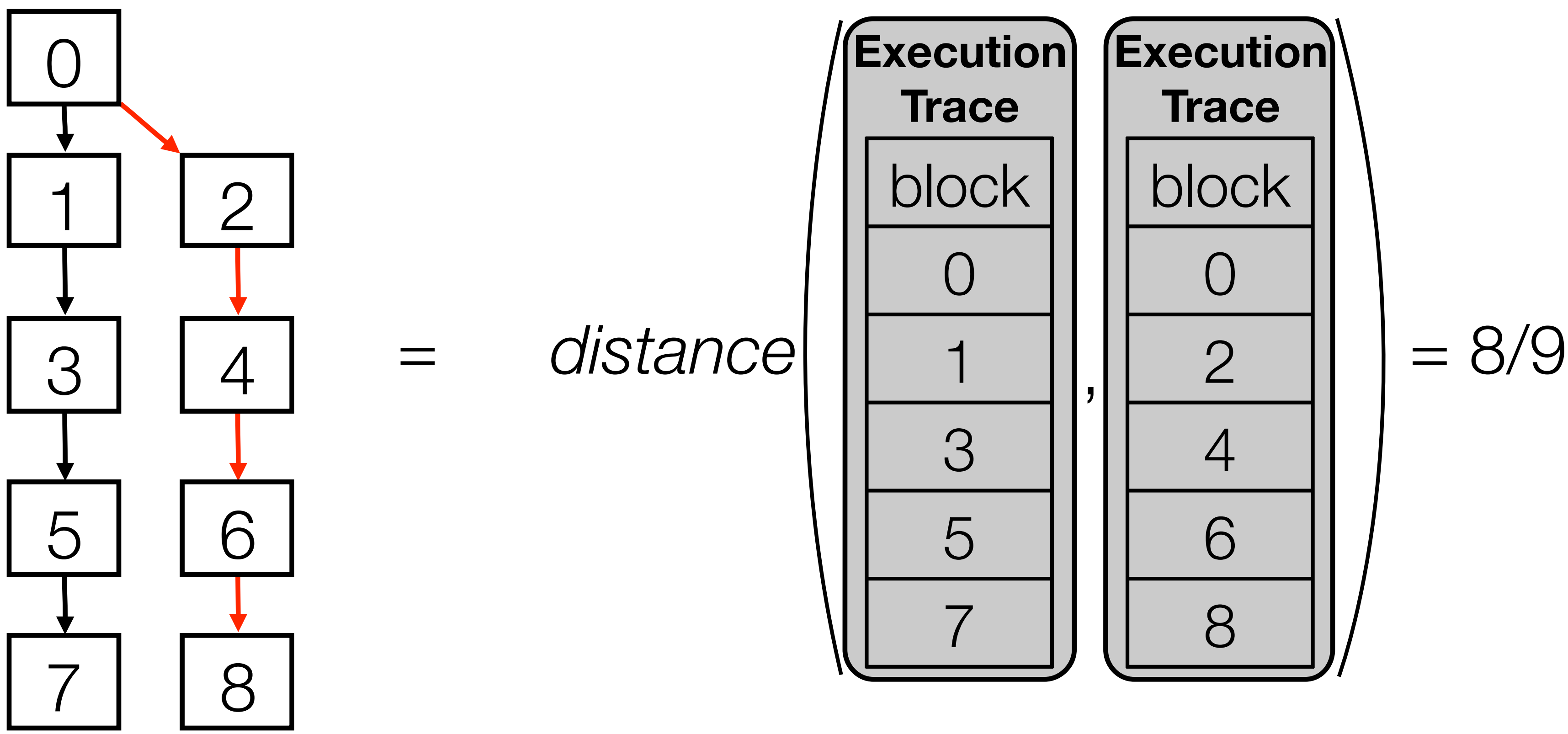
Critical Input Regions in GIF Image Conversion



Modeling Application Response

Modeling Response

- **Behavioral Distance**
    - Change in input region causes change in the behavior of the application.



Modeling Response (cont.)

- **Computation Influence**
  - Contribution of input region to computation's intermediate results.

input

1	2	3	4	5	6	7	8

operations

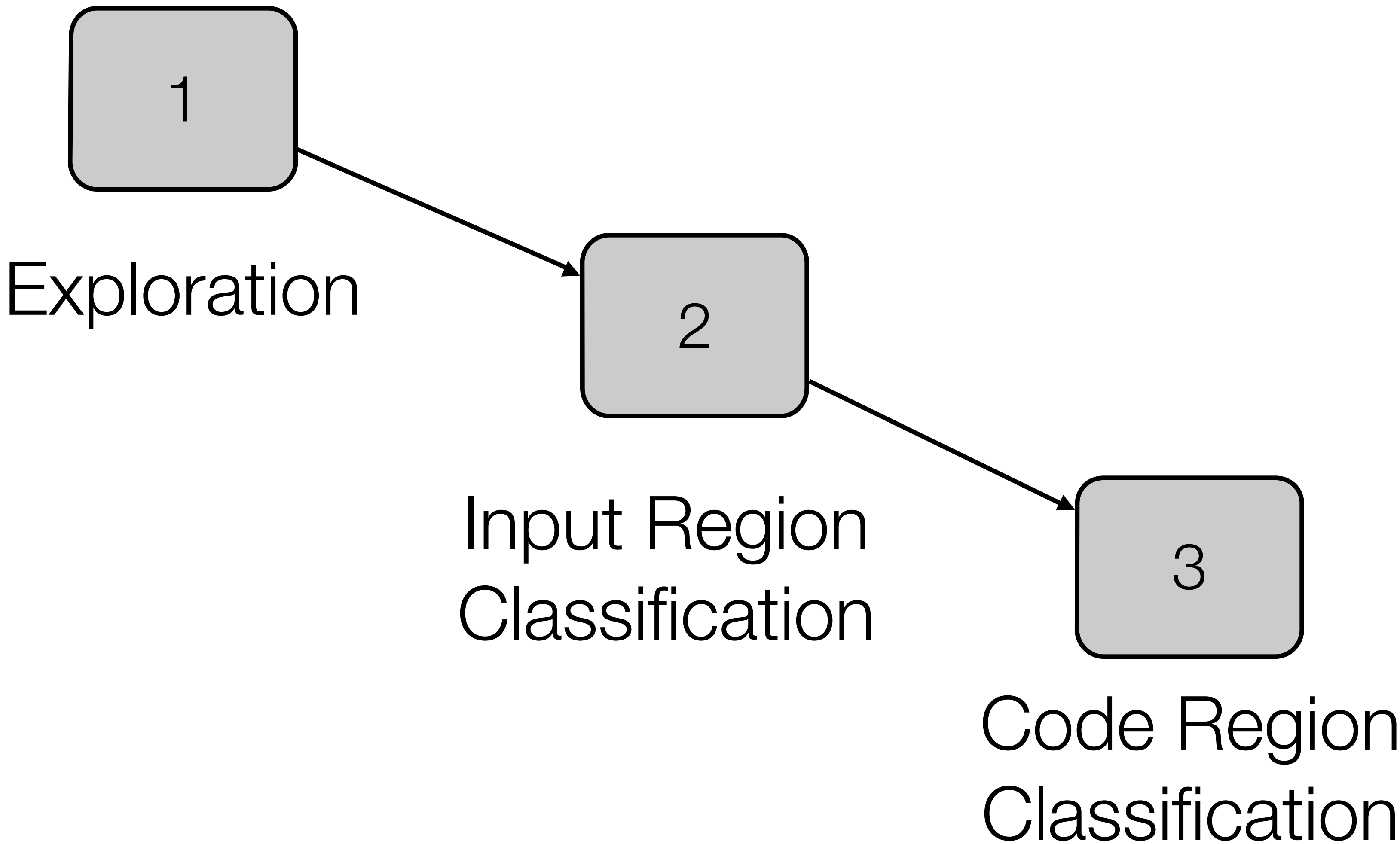
x = add ({3,4}, {2})
y = add (x, {1})
z = neg({5, 6, 7, 8})
a = sub{{3,4}, z}
b = mult({1}, {2})

= *influence* ({3,4}, Influence Trace) = 3

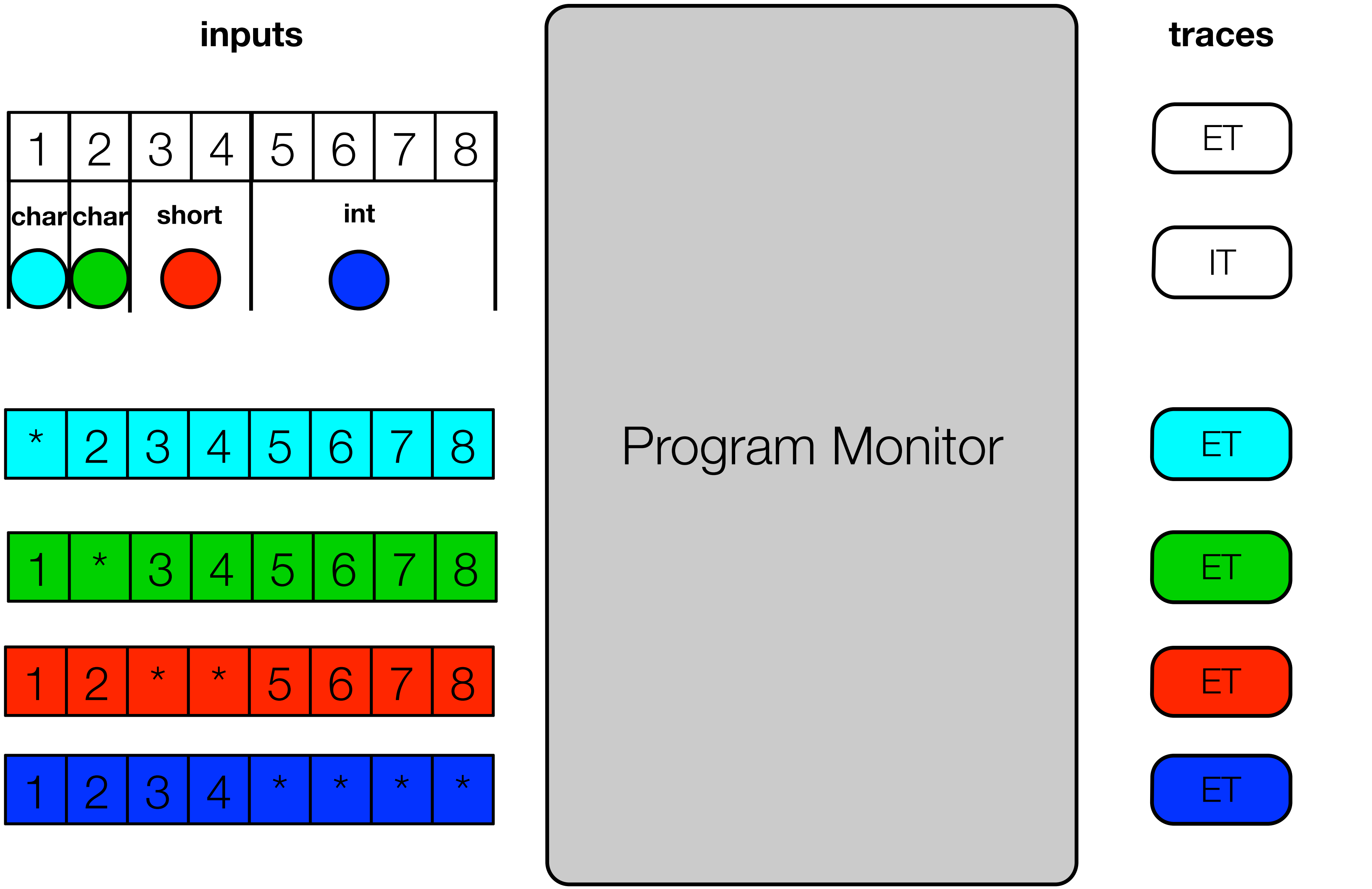
Influence Trace

op	bytes
1	{2,3,4}
2	{1,2,3,4}
3	{5,6,7,8}
4	{3,4,5,6,7,8}
5	{1,2}

Classification Scheme



Exploration Phase



Input Region Classification

- For each region:

determined by clustering

if *distance*(ET , ET) is large :

return “critical”

parameterized threshold

if *influence*(, IT) is large :

return “critical”

else

return “forgiving”

Code Region Classification

- Given input region classifications
  - For each basic block, identify accessed input regions and aggregate input region classifications.
    - Majority are critical => critical code region
    - Majority are forgiving => forgiving code region.
    - No majority => **mixed** code region.

Evaluation Methodology

- Input Region Classification
  - Compare automatic classifications to golden test oracle.
- Code Region Classification
  - Manually determine if code classifications are sensible.

Benchmarks

- Three image processing libraries
  - **gif** (5KLOC), **png** (36KLOC), **jpeg** (35KLOC)
- One Task
  - Convert a image (gif, png, or jpeg) to a bitmap file.
- Five inputs per benchmark
  - Each input exercises different functionality.

Constructing Golden Classifications

- Given input of length n. Run the program to produce the **de facto** output.
- For each of the n bytes of the given input, generate m **fuzzed inputs** by replacing the value of the byte with a random value.
- For each of the **n*m** fuzzed inputs, run the program to produce **n*m fuzzed outputs**.
- Compute the **dissimilarity** between the de facto output and each of the n*m fuzzed outputs.
- For each byte, if one of the m fuzzed outputs is more than 10% dissimilar, classify as **critical**. Otherwise, classify as **forgiving**.

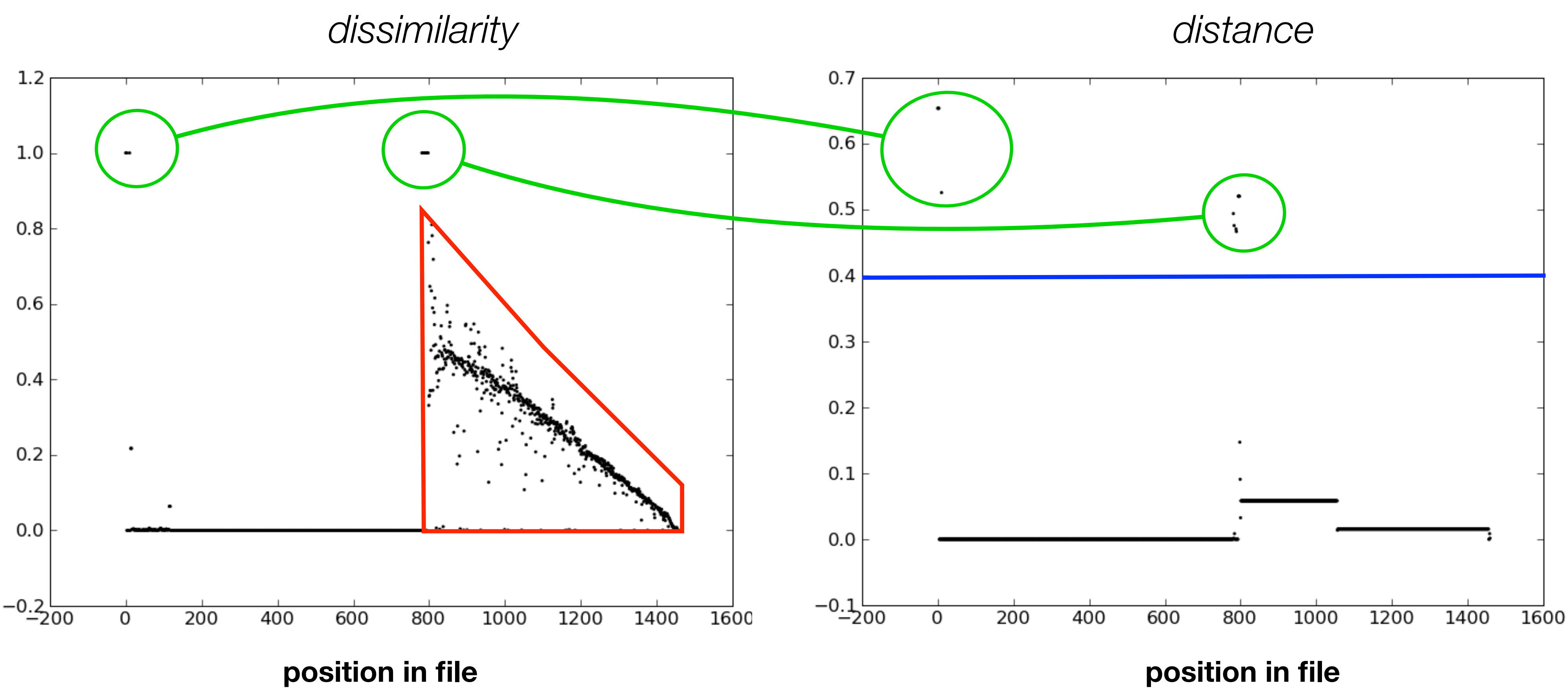
Input Region Classification Results

- Precision: % of critical classification's that were correct.
- Recall: % of critical classifications that were identified.

benchmark	**CC***	**IC****	**CF***	**IF****	**Precision**	**Recall**
png	9580	5	451	18	99%	99%
gif	6951	23	2149	1412	99%	83%
jpeg	5123	27	542	1831	99%	73%

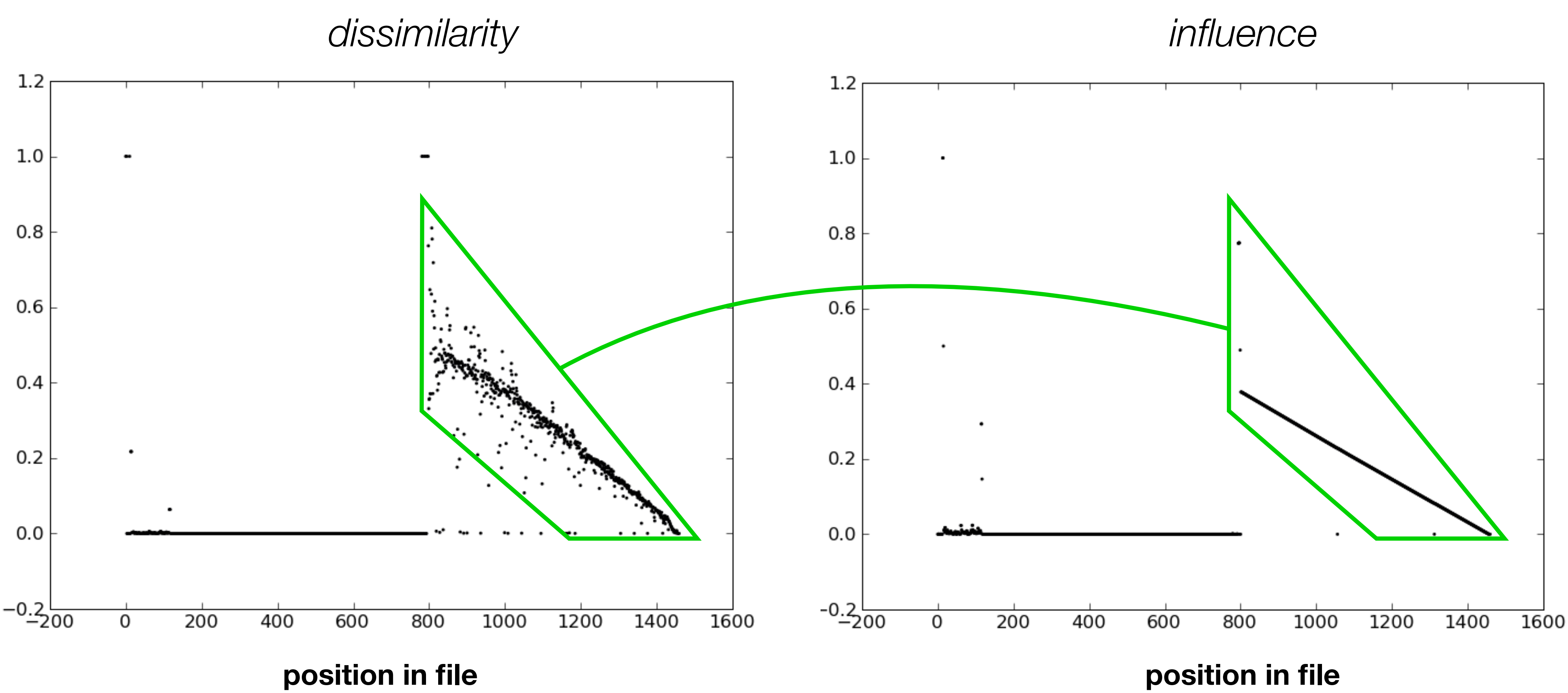
*****higher is better ******lower is better

Role of Behavioral Distance (GIF)



- Clean Separation
- Large behavioral distance implies large dissimilarity
- Behavioral distance does not account for all dissimilarity.

Role of Computation Influence (GIF)



- Computation influence correlates strongly with dissimilarity.

PNG Code Classification Results

B-Critical

- png_handle_IHDR
- png_memcpy_check
- png_handle_tRNS
- png_do_read_transformations
- png_read_start_row

C-Critical

- inflate_table
- inflate_fast
- inflate_table
- png_read_row
- png_read_finish_row
- updatewindow

Forgiving

- png_handle_tIME
- png_handle_gAMA
- png_handle_IEND
- png_handle_pHYs

Mixed

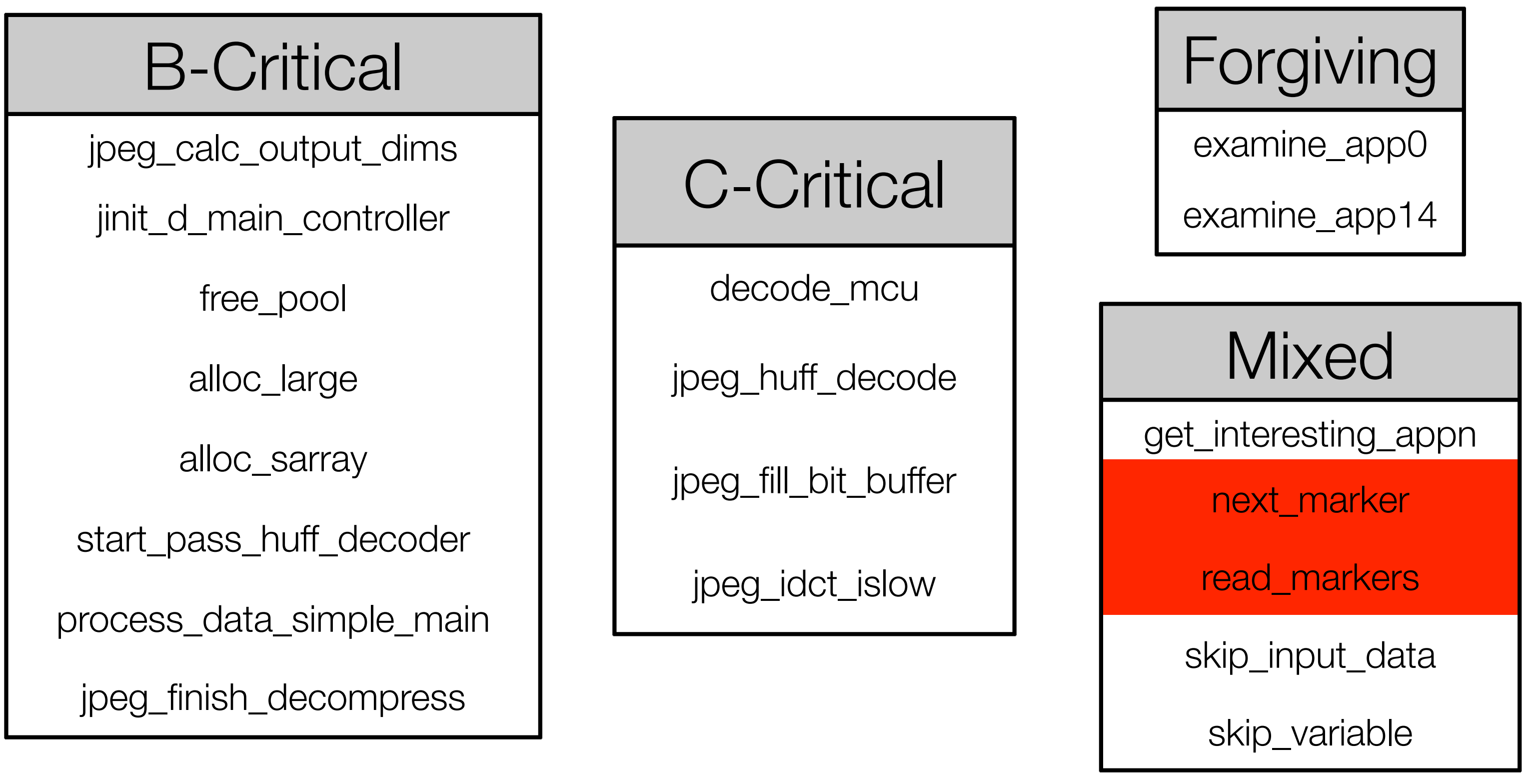
- png_crc_read
- png_crc_error
- png_get_int_31
- png_read_data

GIF Code Classification Results

B-Critical	C-Critical	Mixed
DGifGetLine	DGifDecompressLine	DGifGetWord
DGifGetImageDesc	DGifDecompressInput	DGifDecompressInput

- Few functions because of small size of the benchmark

JPEG Code Classification Results



- Misclassifications in Mixed category due to lower recall.

Limitations and Future Work

- Benchmarks
  - all image conversion
- Behavioral Influence
  - Does not capture all behaviors of interest.
- Computation Influence
  - Does not track indirect (pointer arithmetic) influence.

Conclusion

- New approaches to program analysis are enabled by the distinction between:
  - **Critical** Input and Code Regions - must.
  - **Forgiving** Input and Code Regions - may.
- Input and Code Regions are determined by application's response to change.
  - **Critical** - intolerant to change.
  - **Forgiving** - tolerant to change.
- We can automatically determine regions by modeling application response.

Thanks

Related Work

- Perturbation Analysis (Voas ’92)
- Critical and Forgiving (Rinard ’05)
  - Definition and manual exploration.
- Critical Memory (Pattabiraman ’08)
  - Programmers manually allocate memory in a critical heap that provides probabilistic memory safety
- Continuity (Chaudhuri ’10)

Implementation

- LLVM-based static bitcode instrumentor and dynamic runtime.
- Currently requires source code.
    - C, C++, Java, Ada, MSIL
    - x86 -> LLVM would eliminate need for source.
- Runtime tracks influence (like taint tracing) of input bytes on each operand and memory location.
    - Shadow Execution (registers, stack, memory, filesystem).
    - External library model

Input Specification Generator

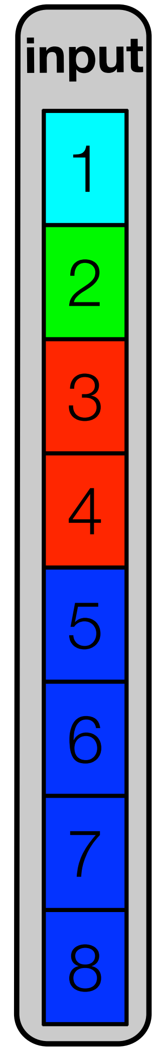
- Groups input bytes by *affinity*: #together/#total

Influence Trace

Op	Bytes
1	{1,2}
2	{3,4}
3	{1}
4	{2}
5	{1}
6	{5,6,7,8}
7	{5,6,7,8}
8	{2}
9	{3,4}
10	{3,4}

affinity

A(1,2)	1/5 = .2	N
A(2,3)	0	N
A(3,4)	3/3 = 1	Y
A(4,5)	0	N
A(5,6)...A(7,8)	2/2 = 1	Y



Evaluating Input Region Classifications

- Precision-Recall:
  - **True Positive**: **C**orrect **C**ritical (**CC**)
  - **False Positive**: **I**ncorrect **C**ritical (**IC**)
  - **True Negatives**: **C**orrect **F**orgiving (**CF**)
  - **False Negatives**: **I**ncorrect **F**orgiving (**IF**)