# Tracelet-Based Code Search in Executables

Yaniv David & Eran Yahav
Technion,Israel

# Finding vulnerable apps

We can find identical or patched code

```
int foo() {
  …
 // buffer
 // overflow
  …
  printf(…)
  …
}
```

```
int alsoFoo() {
  …
  // buffer
  // overflow
  …
  printf(…)
  …
}
```

```
int patchedFoo()
{
  …
  // buffer
  // overflow
  …
  if (…) {}
  printf(…)
  …
}
```

Where else does this vulnerable function exist?

# Finding vulnerable apps

We can find identical or patched code

```
int foo() {
}
```

```
int alsoFoo() {
}
```

```
int patchedFoo()
…
}
```

What if we don't have the source code?

Where else does this vulnerable function exist?

# Boogle

binary functions

```
...
mov [esp+18h+var_18],offset aD1
mov ecx,1
mov [esp+18h+var_14], ecx
call _printf
...
```

Search in Binaries

Function 1 - wc

Coreutils 6.12

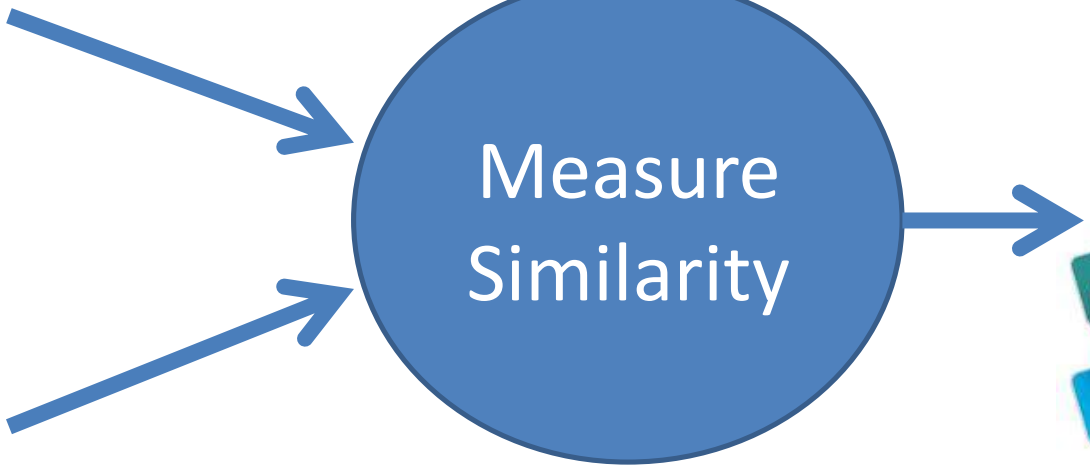
Function 2 – diff

Coreutils 7.15

# Search engine core

```
int foo() {
  …
 // buffer
 // overflow
  …
  printf(…)
  …
}
```

```
int patchedFoo()
{
  …
  // buffer
  // overflow
  …
  if (…) {}
  printf(…)
  …
}
```




Similarity score


- Fast & Scalable
- Accurate (low false positives)

# Challenge1: similarity at the binary level

printf(…)@foo():

```
int foo() {
  …
  // buffer
  // overflow
  …
  printf(…)
  …

}
```

printf(…)@patchedFoo():

```
int patchedFoo()
{
  …
  // buffer
  // overflow
  …
  if (…) {}
  printf(…)
  …
}
```

# Challenge1: similarity at the binary level

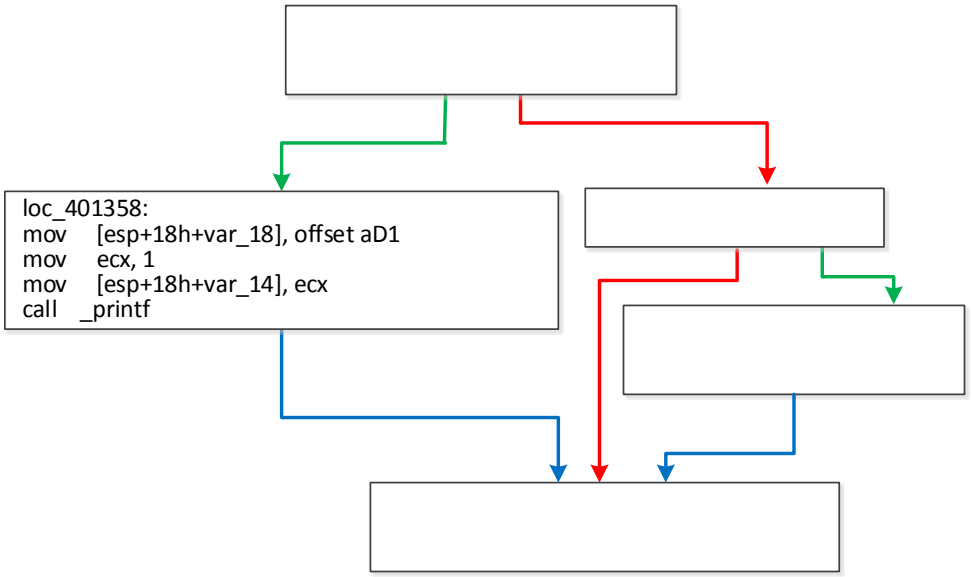
```
loc_401358:
mov [esp+18h+var_18],offset aD1
mov ecx,1
mov [esp+18h+var_14], ecx
call _printf
```

```
loc_401370:
mov [esp+28h+var_28],offset aD1
mov ebx,1
mov esi,4
mov [esp+28h+var_24], ebx
call _printf
```

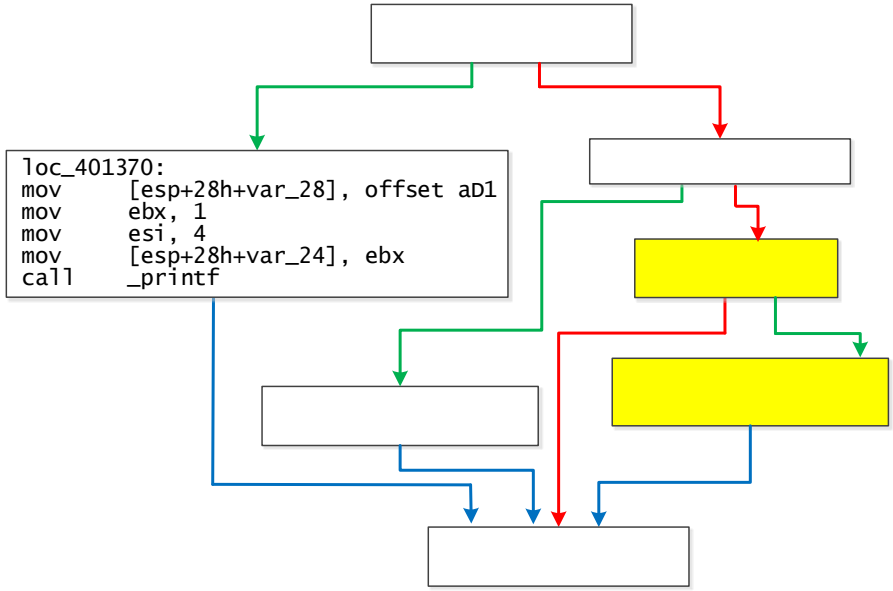
- Offsets in memory
- Register allocation
- New Instruction

# Challenge2: similarity between different structures

foo's CFG:




patchedFoo's CFG:

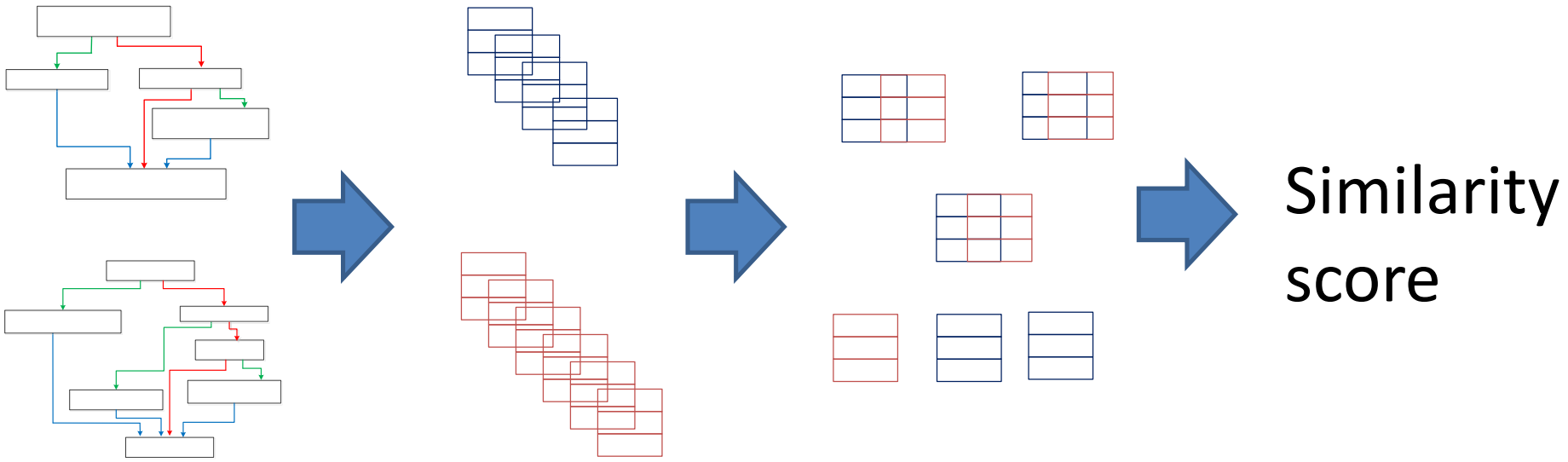


# In this talk

- A system for searching code in executables
  - Based on tracelet decomposition of each function
  - Works by solving a set of alignment and dataflow constraints with minimal violations on tracelets

- An evaluation methodology based on tools from Information Retrieval
  - How do we know that our search engine is good?

# Our Approach




Extract tracelets

Deal with structural changes

Pair tracelets using **alignment** and **rewrite**

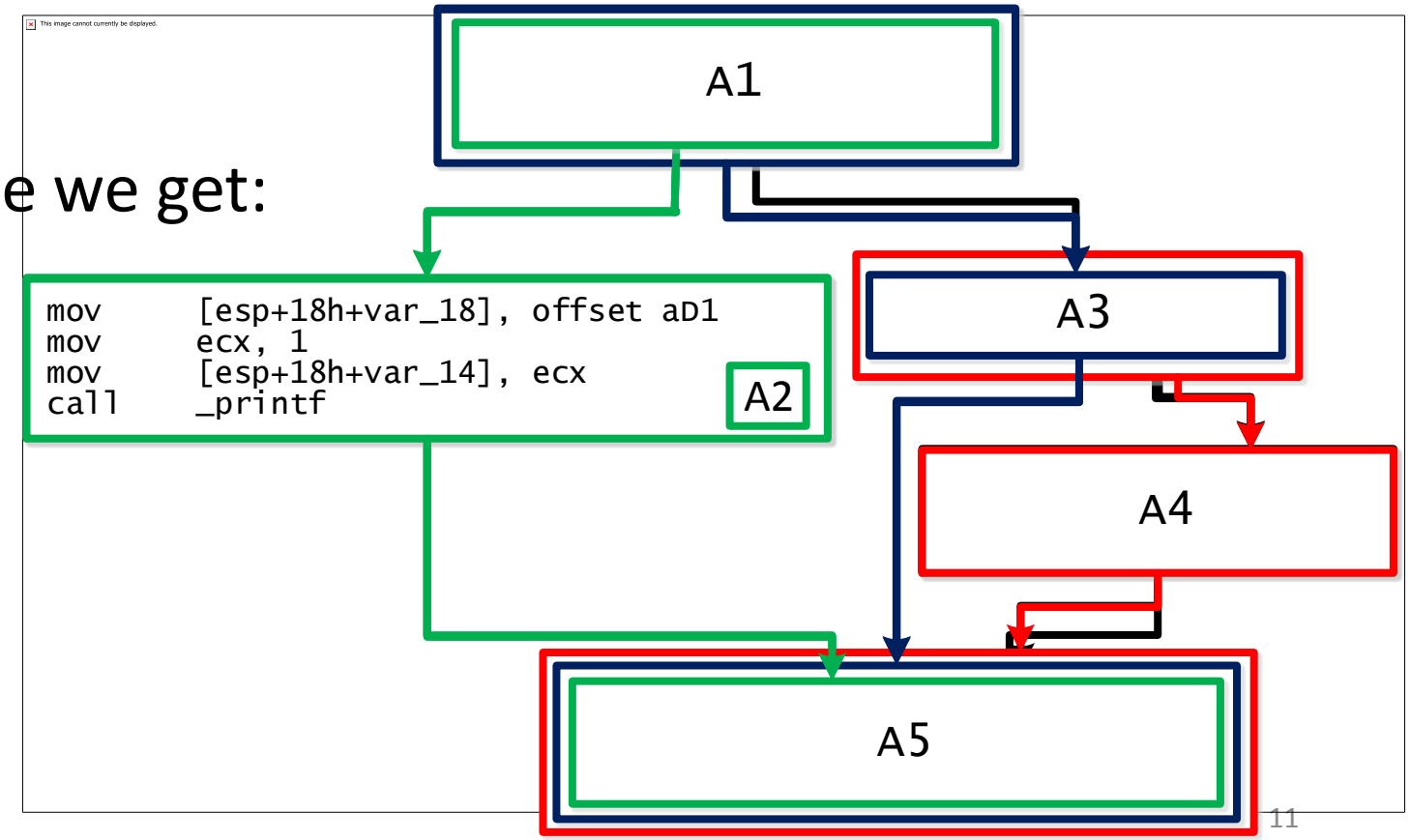
Deal with the code changes

# Using tracelets to deal with CFG structural changes

A tracelet is a fixed length sub-trace

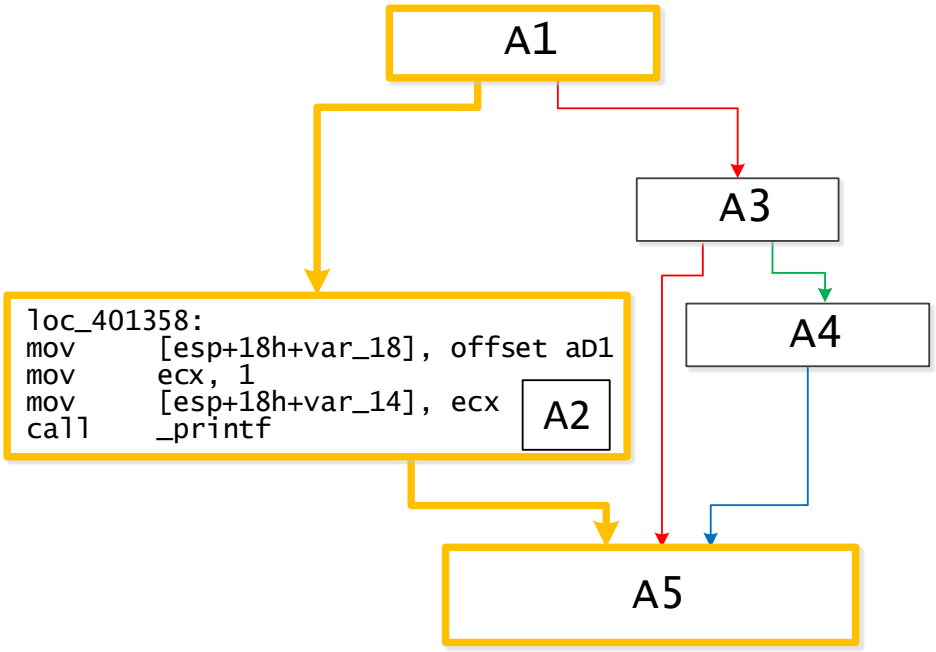
For length=3,
In this example we get:

(A1,A2,A5)
(A1,A3,A5)
(A3,A4,A5)

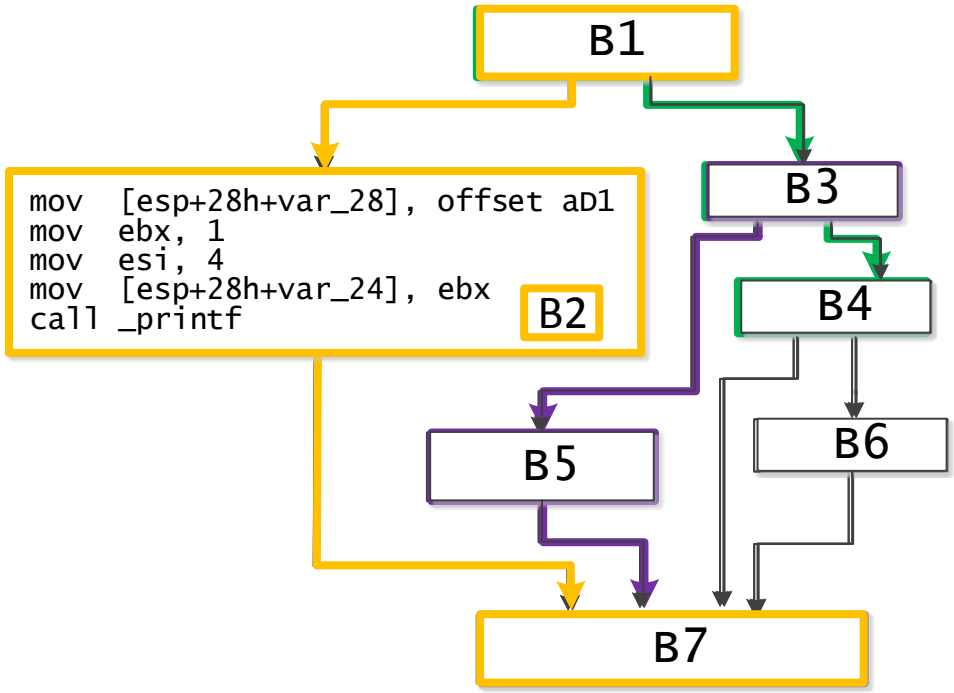


# Using tracelets calculate similarity between different structures

foo's CFG:



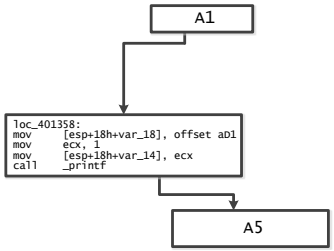

patchedFoo's CFG:



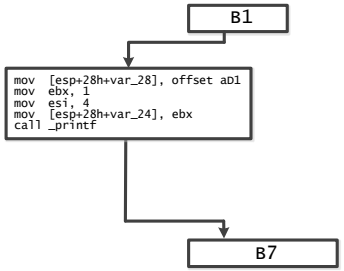

We need to find the corresponding tracelet

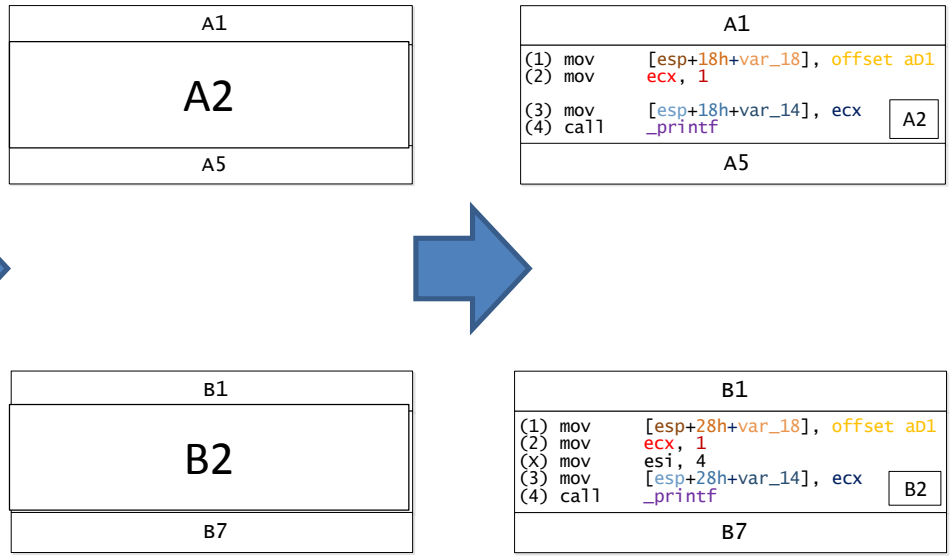
# Comparing tracelets

foo's tracelet




patchedFoo's tracelet:




Edit distance

Graph -> linear code

Align & RW

# Dealing with code changes: Align

| A1 | |
|---|---|
| mov [esp+18h+var_18], offset aD1<br>mov ecx, 1<br>mov [esp+18h+var_14], ecx<br>call _printf | A2 |
| A5 | |

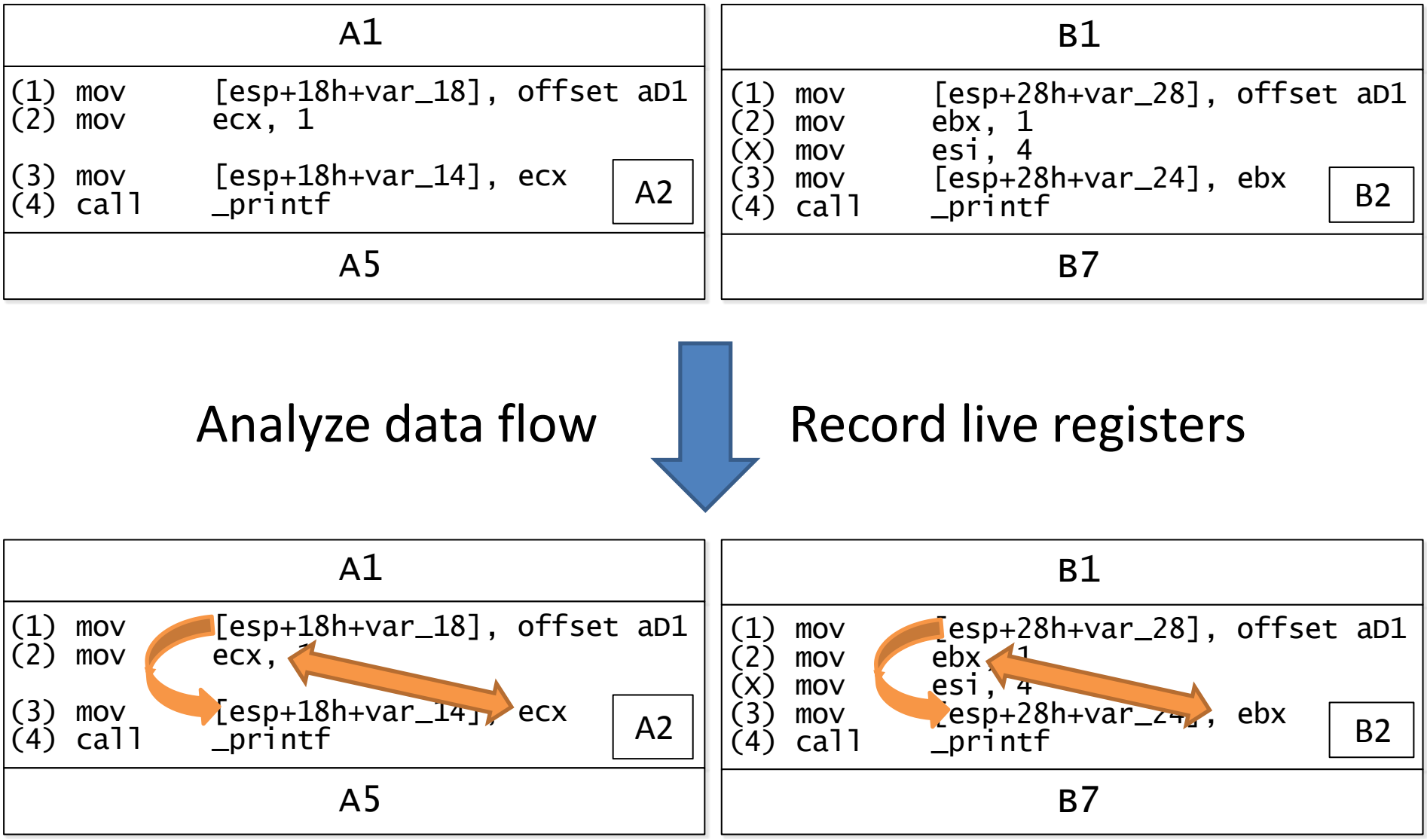
| B1 | |
|---|---|
| mov [esp+28h+var_28], offset aD1<br>mov ebx, 1<br>mov esi, 4<br>mov [esp+28h+var_24], ebx<br>call _printf | B2 |
| B7 | |

Align tracelets using specialized edit-distance

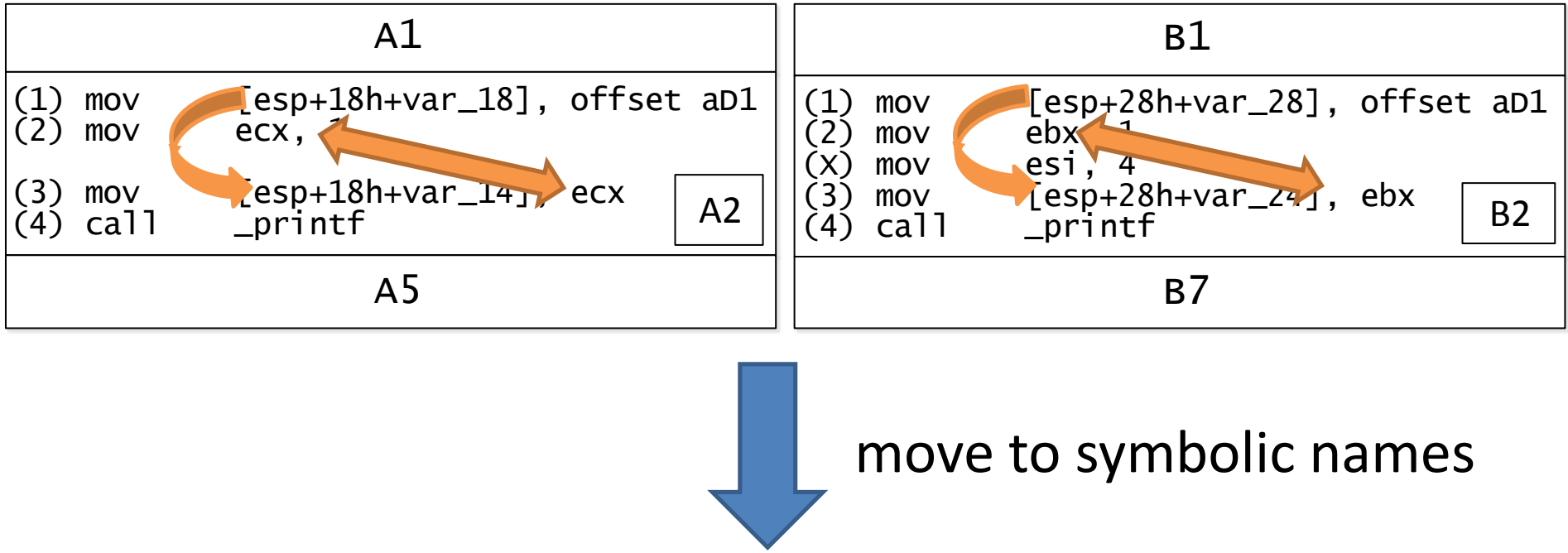
| A1 | |
|---|---|
| (1) mov [esp+18h+var_18], offset aD1<br>(2) mov ecx, 1<br><br>(3) mov [esp+18h+var_14], ecx<br>(4) call _printf | A2 |
| A5 | |

| B1 | |
|---|---|
| (1) mov [esp+28h+var_28], offset aD1<br>(2) mov ebx, 1<br>(X) mov esi, 4<br>(3) mov [esp+28h+var_24], ebx<br>(4) call _printf | B2 |
| B7 | |

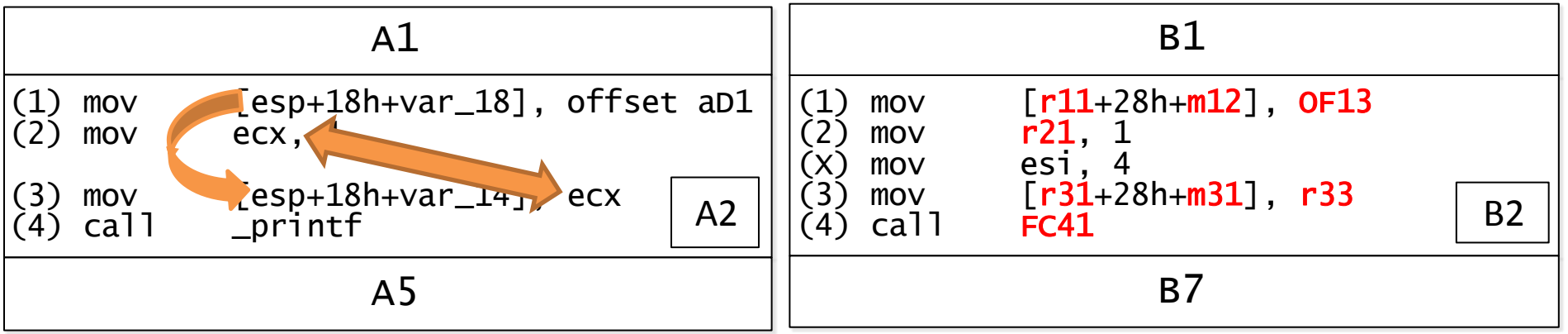
# Dealing with code changes: DFA



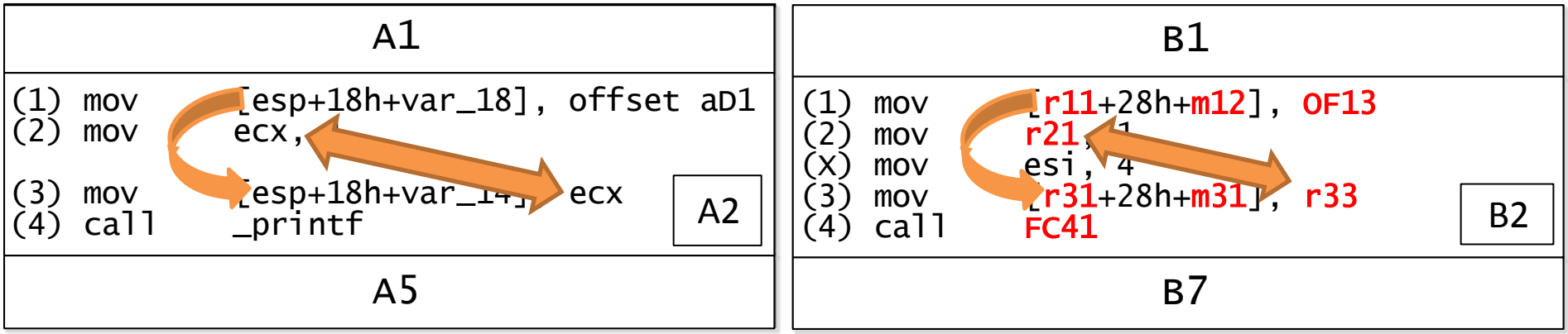
# Dealing with code changes: Symbolize




move to symbolic names



# Dealing with code changes: Solve & Rewrite




Use alignment & DFA to create constraints

Solve them using constraint solver with minimal conflicts

Data Flow constraints:

r21=r33;

r11=r31;

Alignment constraints:

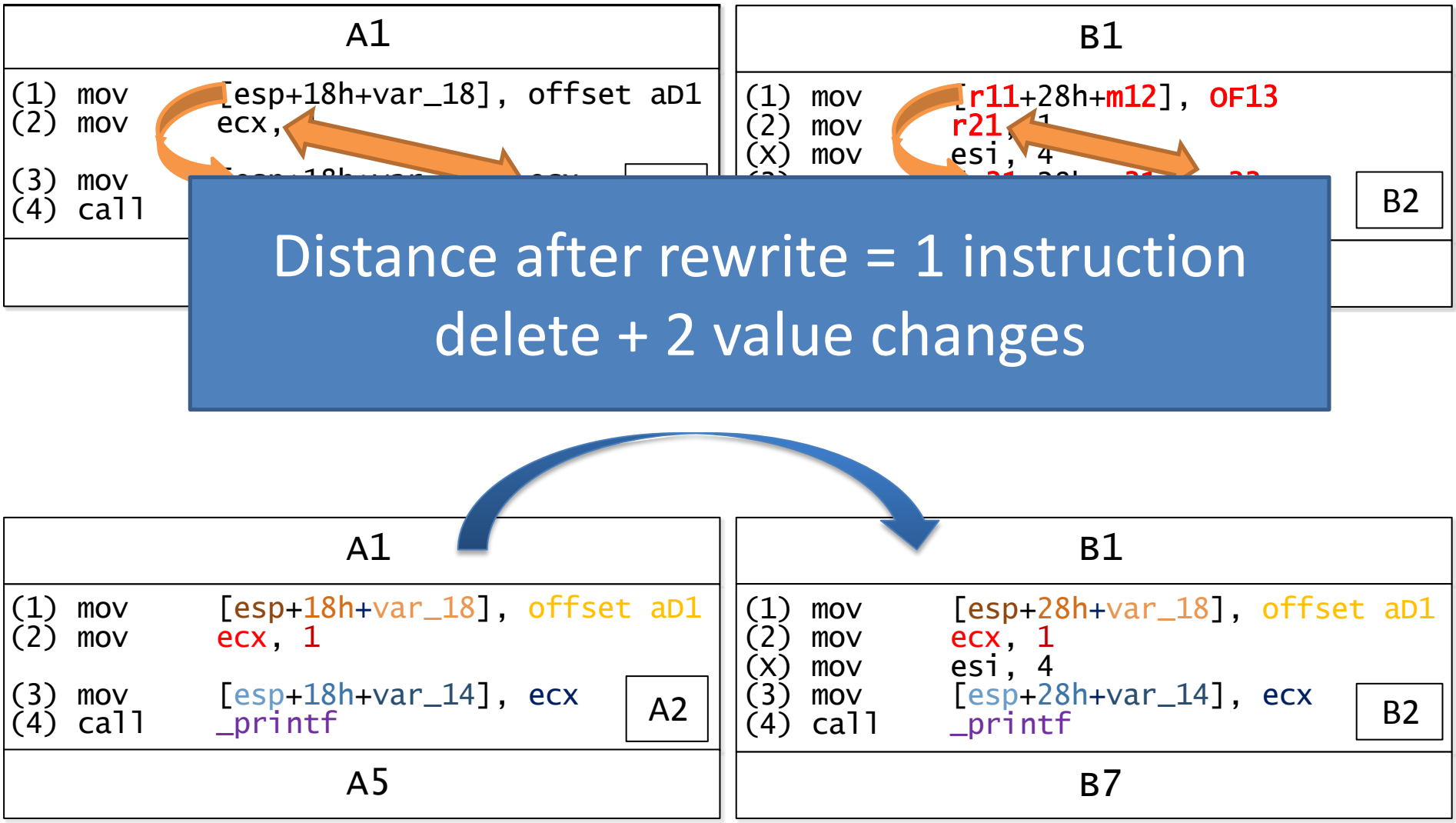
r11=esp;F13=…; m12=var_18;

r21=ecx;e31=esp;

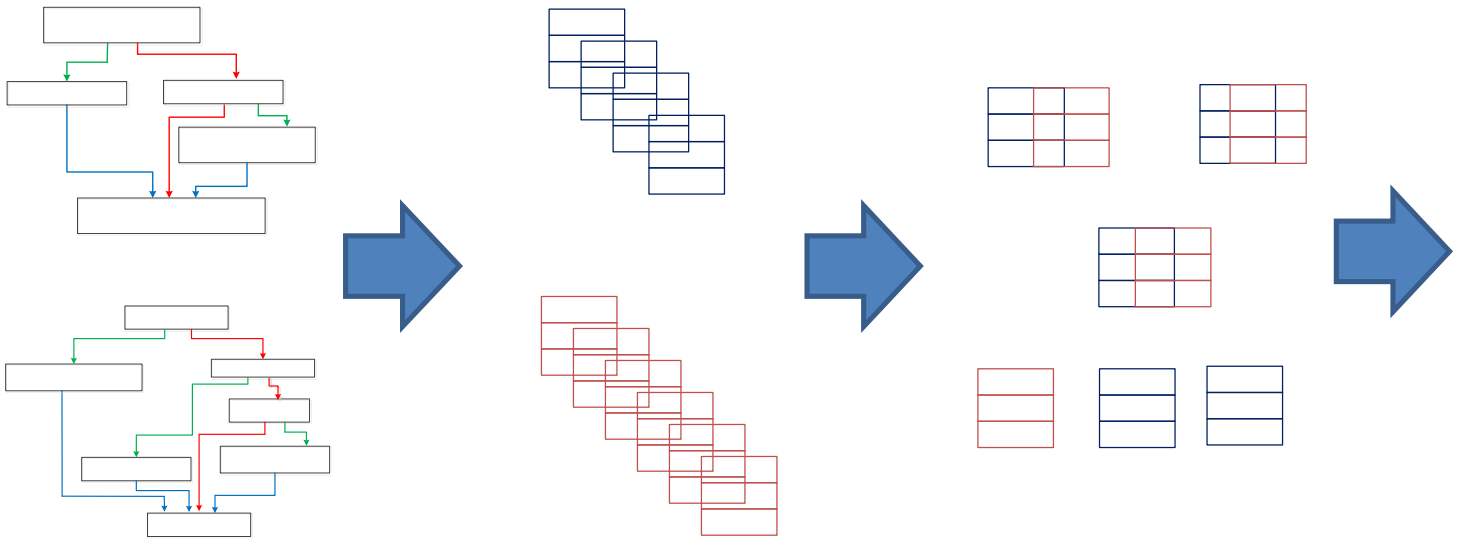
m32=var_14; r33=ecx;

FC41=_printf;

# Dealing with code changes: Solve & Rewrite



# Our Approach



Extract tracelets

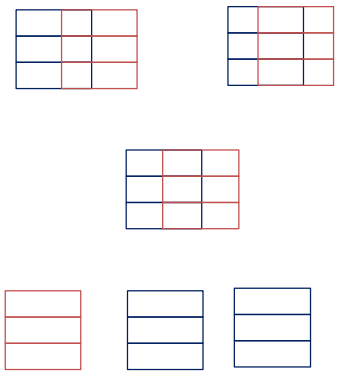
Deal with structural changes

Pair tracelets using **alignment** and **rewrite**

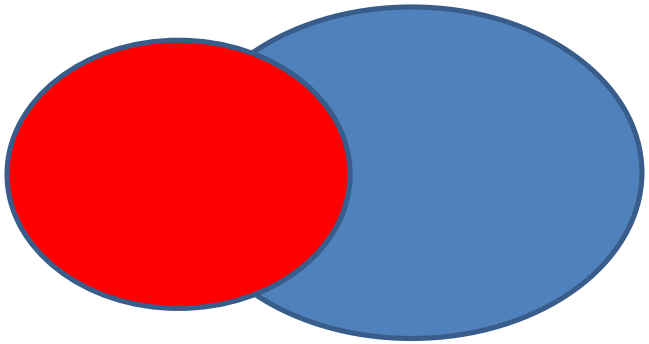
Deal with the code changes

Similarity score

# From paired tracelets to function similarity score

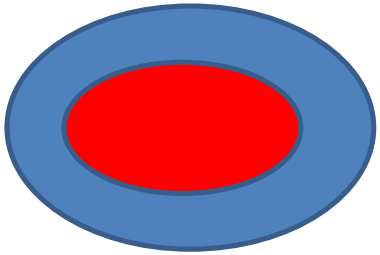


**Ratio**



$$\frac{2 * \#PairedTracelets(f1, f2)}{\#Tracelets(f1) + \#Tracelets(f2)}$$

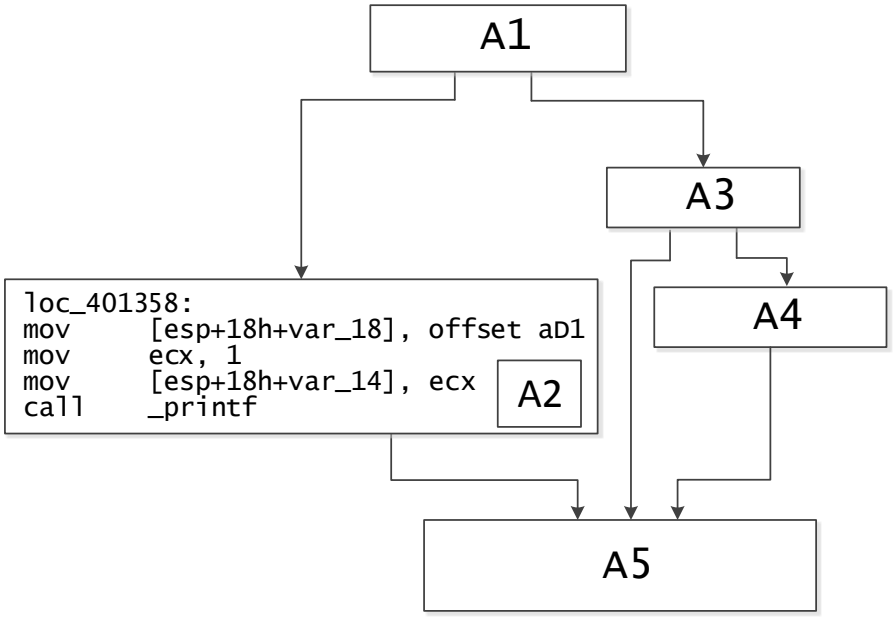
**Containment**



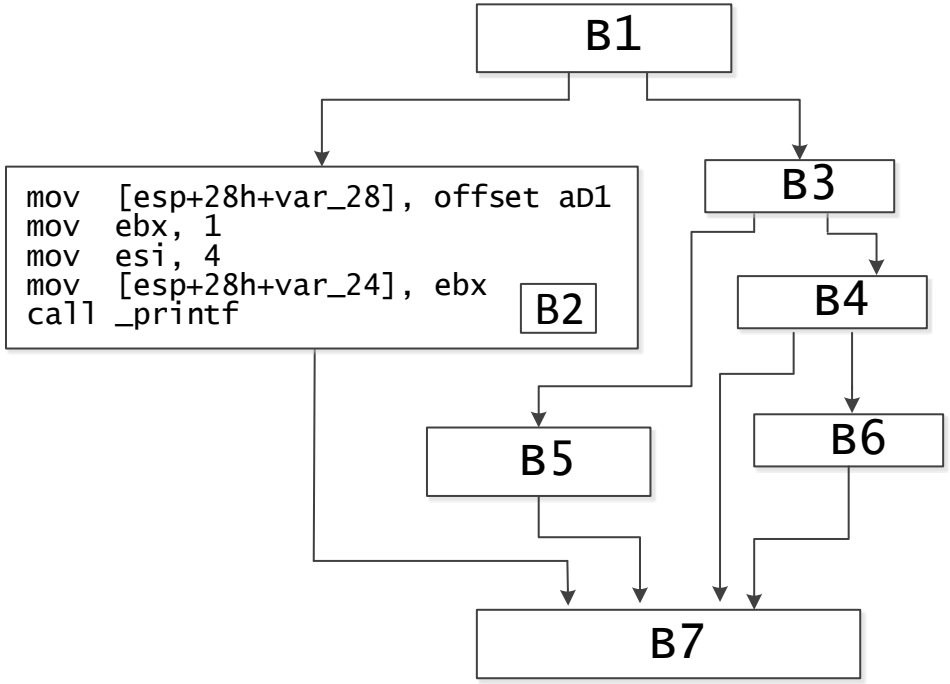
$$\frac{\#PairedTracelets(f1, f2)}{Min(\#Tracelets(f1), \#Tracelets(f2))}$$

# Using tracelets calculate similarity between different structures

foo's CFG:



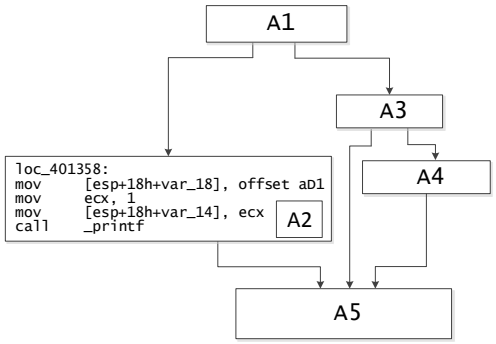

patchedFoo's CFG:



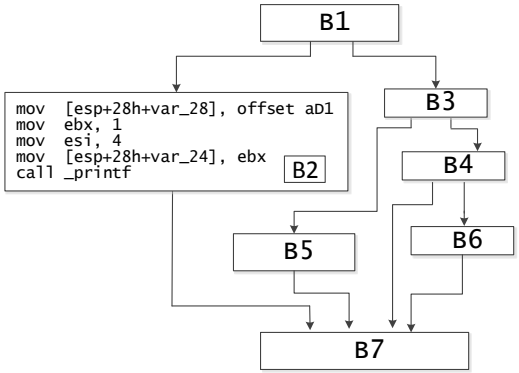

(A1,A2,A5)~(B1,B2,B7),(A1,A3,A4)~(B1,B3,B4),
(A3,A4,A5)~(B3,B4,B7),(A1,A3,A5) -> "lost"

# Using tracelets calculate similarity between different structures

foo's CFG:




patchedFoo's CFG:




$$\frac{2 * \#PairedTracelets(f1, f2)}{\#Tracelets(f1) + \#Tracelets(f2)}$$

(A1,A2,A5)~(B1,B2,B7),(A1,A3,A4)~(B1,B3,B4),
(A3,A4,A5)~(B3,B4,B7),(A1,A3,A5) -> "lost"

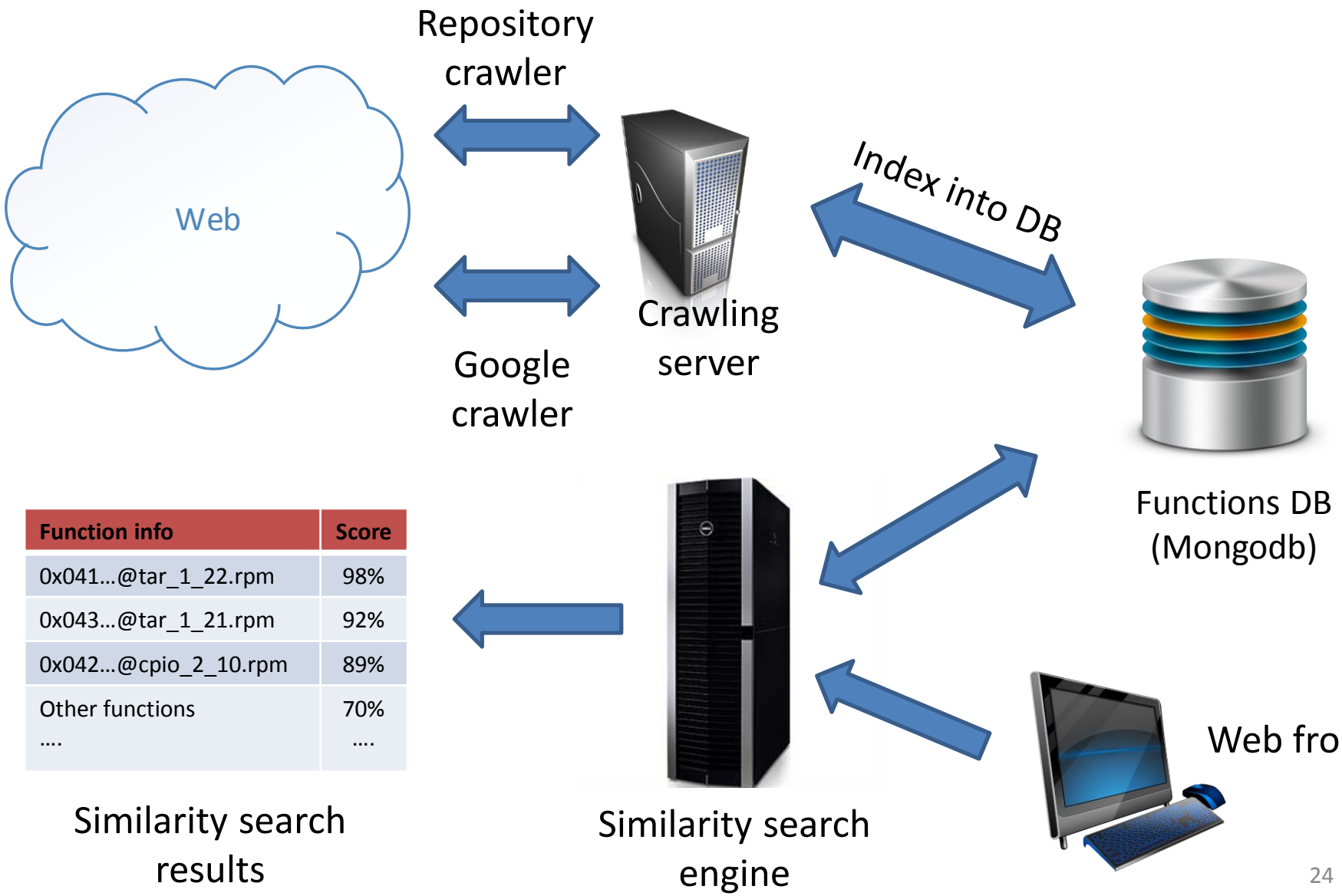
$$\frac{2 * 3}{4 + 7} = \frac{6}{11} = 54\% \; Similarity \; (ratio)$$

# Our system


| Function info | Score |
|---|---|
| 0x041…@tar_1_22.rpm | 98% |
| 0x043…@tar_1_21.rpm | 92% |
| 0x042…@cpio_2_10.rpm | 89% |
| Other functions …. | 70% …. |

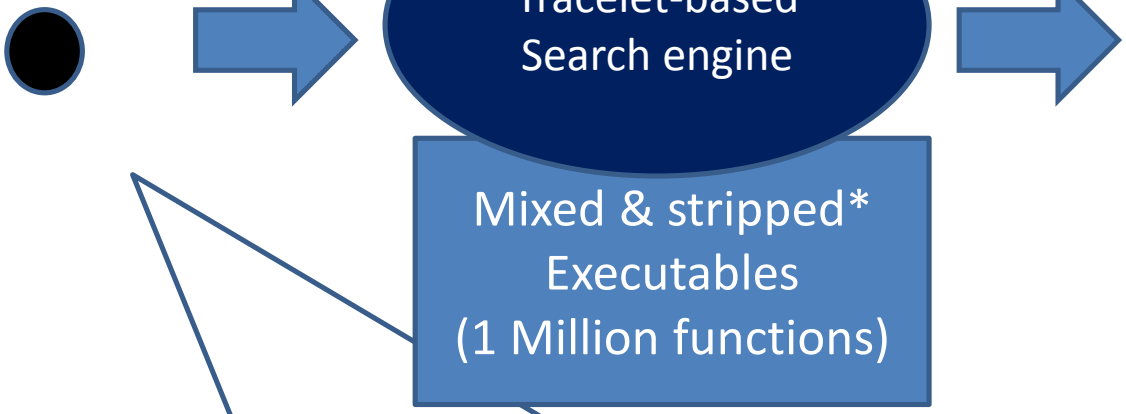
Similarity search results

# Our system


Repository crawler
Web
Google crawler
Crawling server
Index into DB
Functions DB (Mongodb)
Function info
Score
0x041…@tar_1_22.rpm
98%
0x043…@tar_1_21.rpm
92%
0x042…@cpio_2_10.rpm
89%
Other functions ….
70% ….
Similarity search results
Similarity search engine
Web front

# One experiment – find my Heartbleed (CVE-2014-0160)

tls1_heartbeat
@ openssl 1.0.1f




| Function info | Score |
|---|---|
| tls1_heartbeat @openssl_1_0_1f.rpm | 98% |
| dtls1_process_heartbeat @openssl_1_0_1f.rpm | 96% |
| …@openssl_1_0_1e.rpm | 89% |
| more vulnerable functions …. | …. |

TLS implementation does not properly handle Heartbeat Extension packets causes information disclosure

# Using a single threshold

90% similarity score is…good?

Can we really choose one threshold?

| Function info | Score |
|---|---|
| tls1_heartbeat @openssl_1_0_1f.rpm | 98% |
| dtls1_process_heartbeat @openssl_1_0_1f.rpm | 96% |
| …@openssl_1_0_1e.rpm | 89% |
| other functions …. | …. |

| Function info | Score |
|---|---|
| 0x041…@tar_1_22.rpm | 88% |
| 0x043…@tar_1_21.rpm | 83% |
| 0x042…@cpio_2_10.rpm | 89% |
| Other functions …. | 70% …. |

| Function info | Score |
|---|---|
| 0x042…@wget_1_12.rpm | 94% |
| 0x045…@wget_1_14.rpm | 91% |
| Other functions …. | 60% …. |

# Using a single threshold

90% similarity score is…good?

Can we really choose one threshold?

| Function info | Score |
|---|---|
| tls1_heartbeat @openssl_1_0_1f.rpm | 98% |
| dtls1_process_heartbeat @openssl_1_0_1f.rpm | 96% |
| other functions …. | …. |

| Function info | Score |
|---|---|
| 0x041…@tar_1_22.rpm | 88% |
| 0x043…@tar_1_21.rpm | 83% |
| Other functions …. | 70% …. |

| Function info | Score |
|---|---|
| 0x042…@wget_1_12.rpm | 94% |
| 0x045…@wget_1_14.rpm | 91% |
| …. | …. |

Threshold

There should be a more accurate way

# ROC – trying all thresholds

- Receiver operating characteristic
- Try every threshold (=>binary classifier)
- Get a number representing the method's accuracy

# Experiment example

The function we are searching for

Threshold: XX%

Tracelet-based Search engine

| Function info | Score |
|---|---|
| tls1_heartbeat @openssl_1_0_1f.rpm | 98% |
| dtls1_process_heartbeat @openssl_1_0_1f.rpm | 96% |
| …@openssl_1_0_1e.rpm | 89% |
| other functions …. | …. |

Remove any functions below Threshold

| Function info | Score |
|---|---|
| tls1_heartbeat @openssl_1_0_1f.rpm | 98% |
| dtls1_process_heartbeat @openssl_1_0_1f.rpm | 96% |
| …@openssl_1_0_1e.rpm | 89% |

Check results (manually)

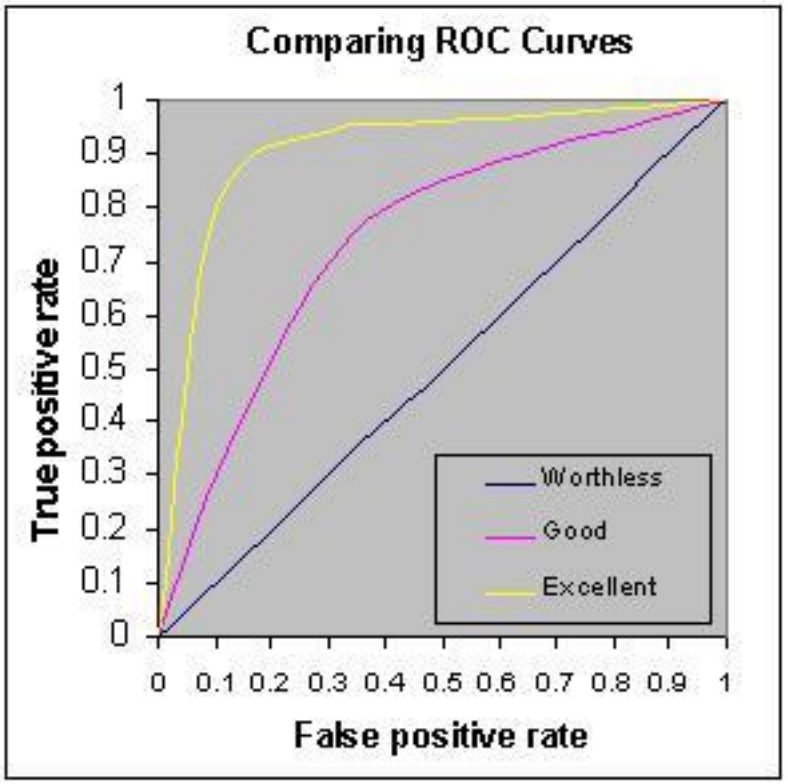
| Function info | Score |
|---|---|
| tls1_heartbeat @openssl_1_0_1f.rpm | ✓ |
| dtls1_process_heartbeat @openssl_1_0_1f.rpm | ✓ |
| …@openssl_1_0_1e.rpm | ✗ |

Calculate Accuracy

$$Accuracy = (TP + TN)/(P + N)$$

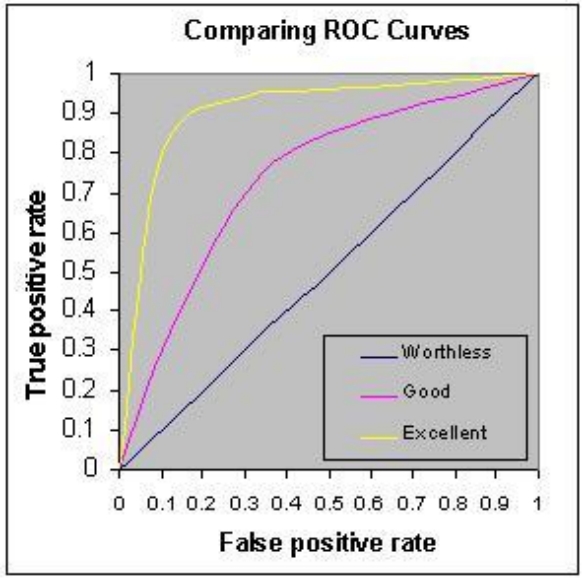
# ROC – trying all thresholds

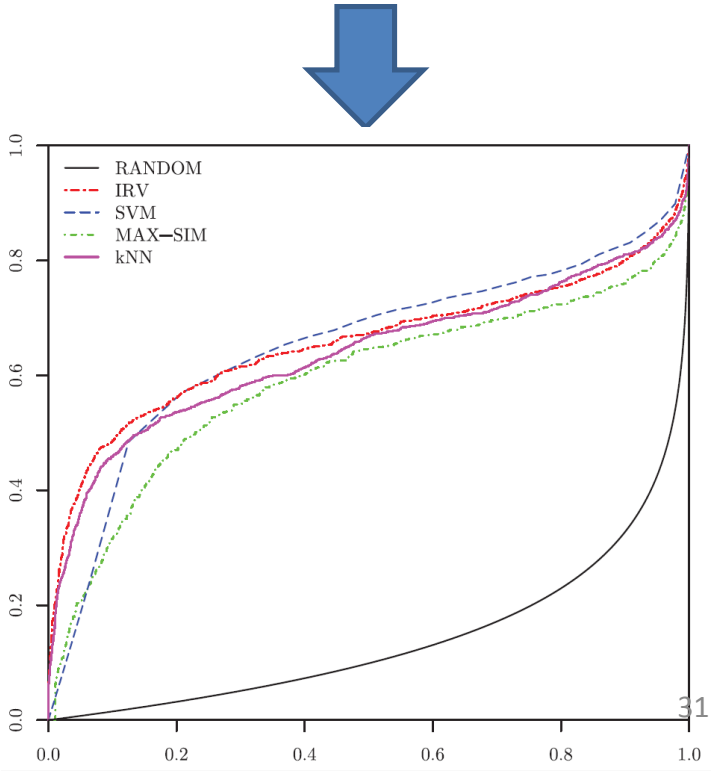
- Method's accuracy is Area Under Curve (AUC) determines precision



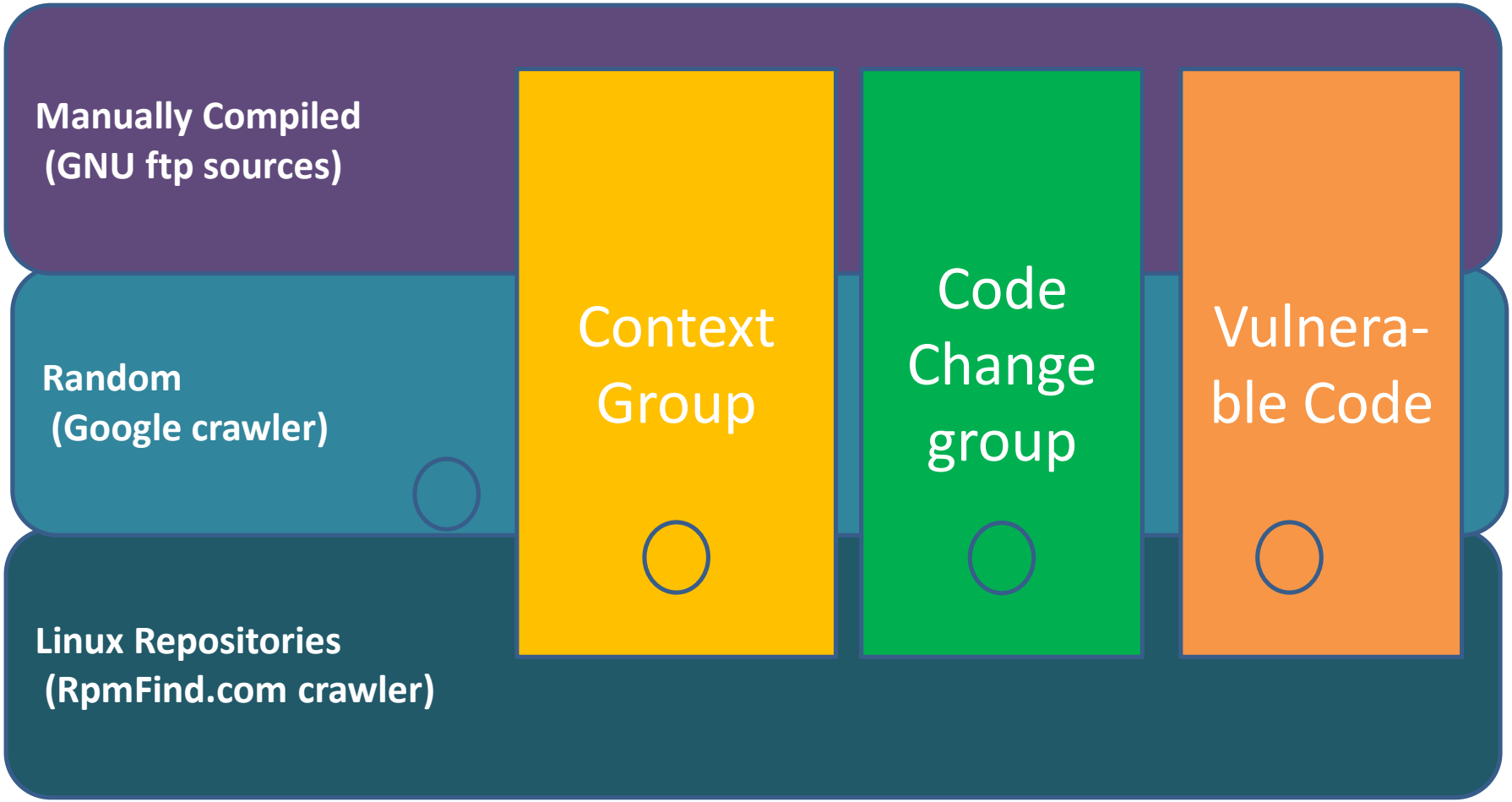
# CROC is better then ROC

- The matches we expect are **very sparse**
- We need to “punish” false positives – they have a high cost
- CROC does exactly that

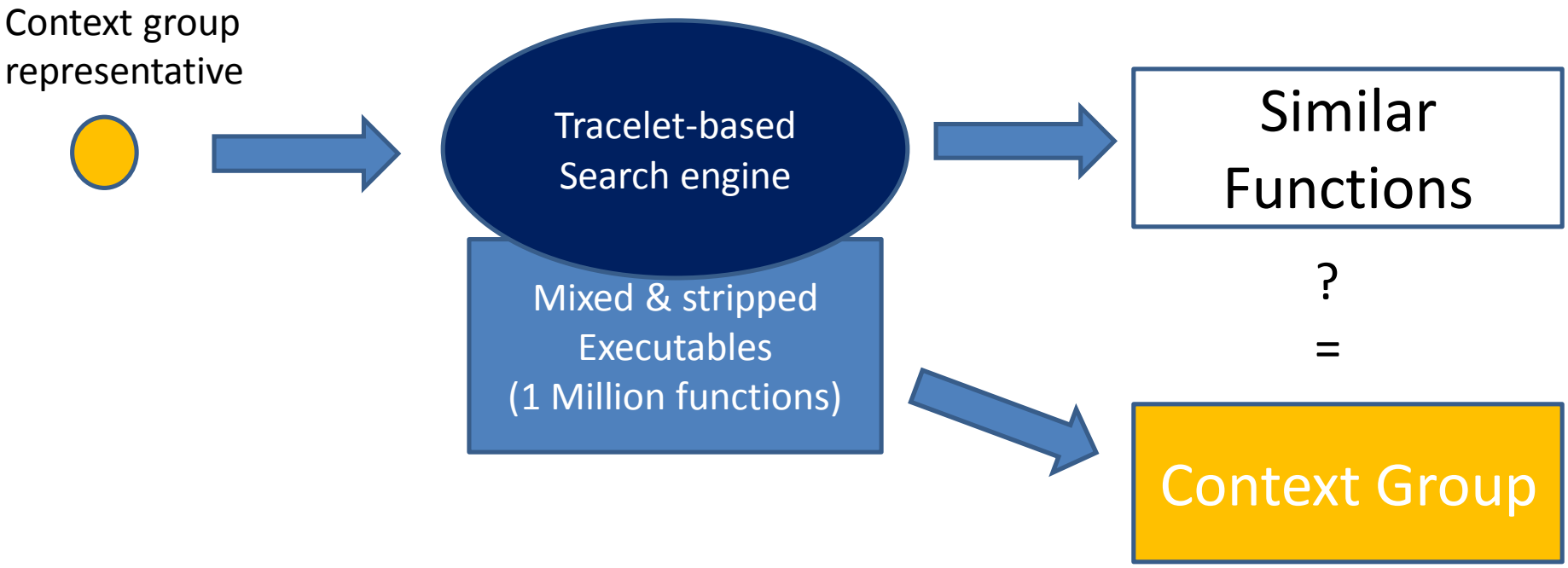


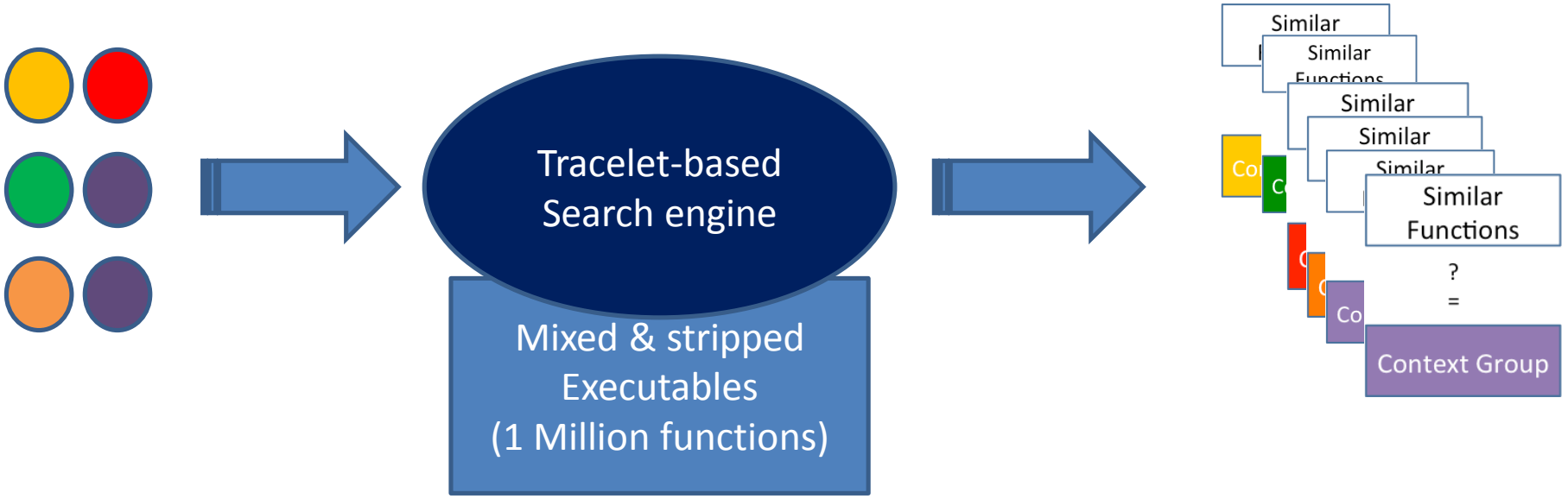
# Experiment Structure



# Experiment goal



# Experiment Setup & Results




| | N-grams Size 5,Delta 1 | Graphlets K=5 | Tracelets K=3 |
|---|---|---|---|
| AUC[ROC] | 72% | 60% | 99% |
| AUC[CROC] | 25% | 12% | 99% |

# Conclusions

- Tracelets based code search system
  - Effective in finding exact and near matches
  - Provides a quantitative similarity score

- Evaluated using Information Retrieval tools
  - Achieves good precision and recall
  - Tested against other leading methods