grap: define and match graph patterns within binaries

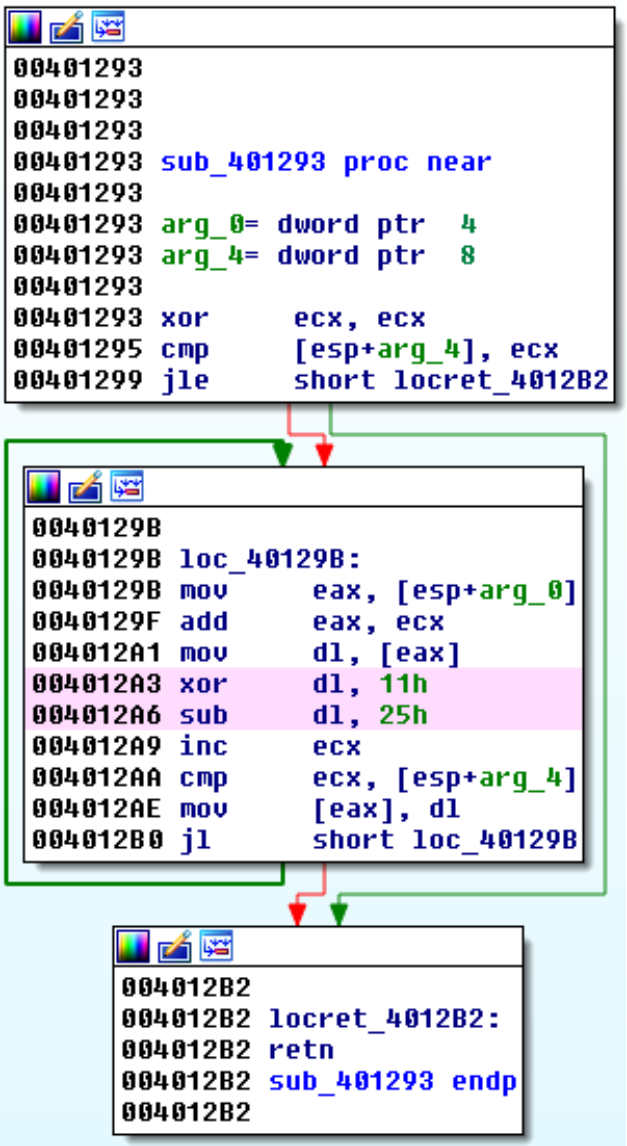
Aurélien Thierry (aurelien.thierry@airbus.com)
Jonathan Thieuleux (jonathan.thieuleux@stormshield.eu)

REcon 2017, January 28th

AIRBUS

STORMSHIELD

Malware analysis: Backspace

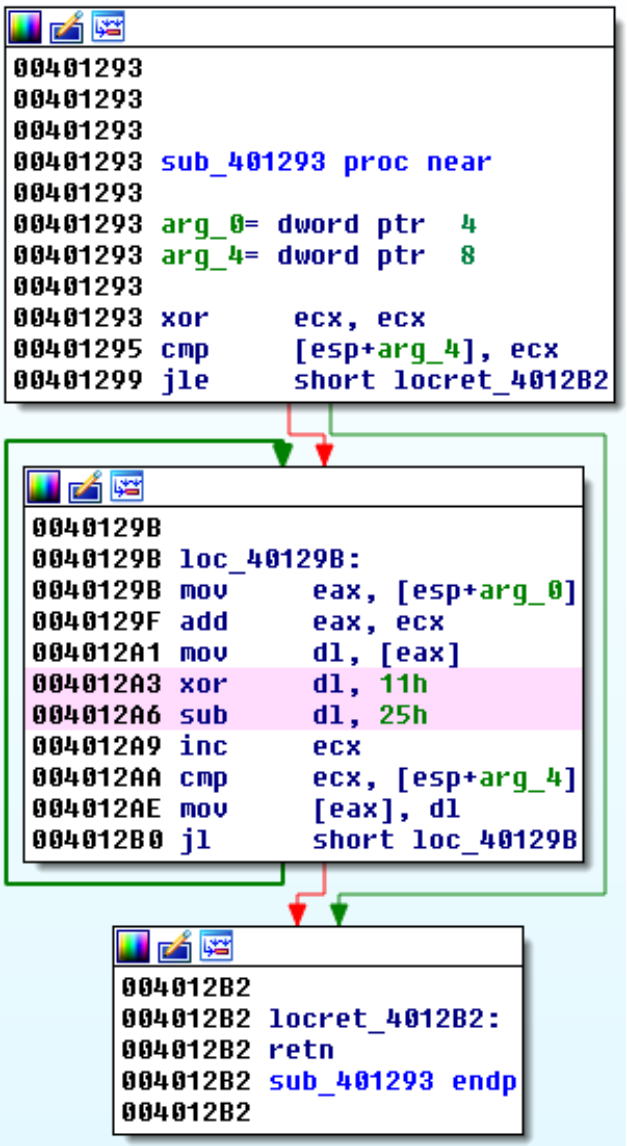


APT30 RAT (2015, see FireEye's whitepaper)

Encrypted configuration (C&Cs, ports...):

- Simple, custom "decryption" routine
- Many variants of the decryption algorithm

Malware analysis: Backspace



Goals:

- Detection
- Classify variants
- Decrypt configuration variables

YARA:

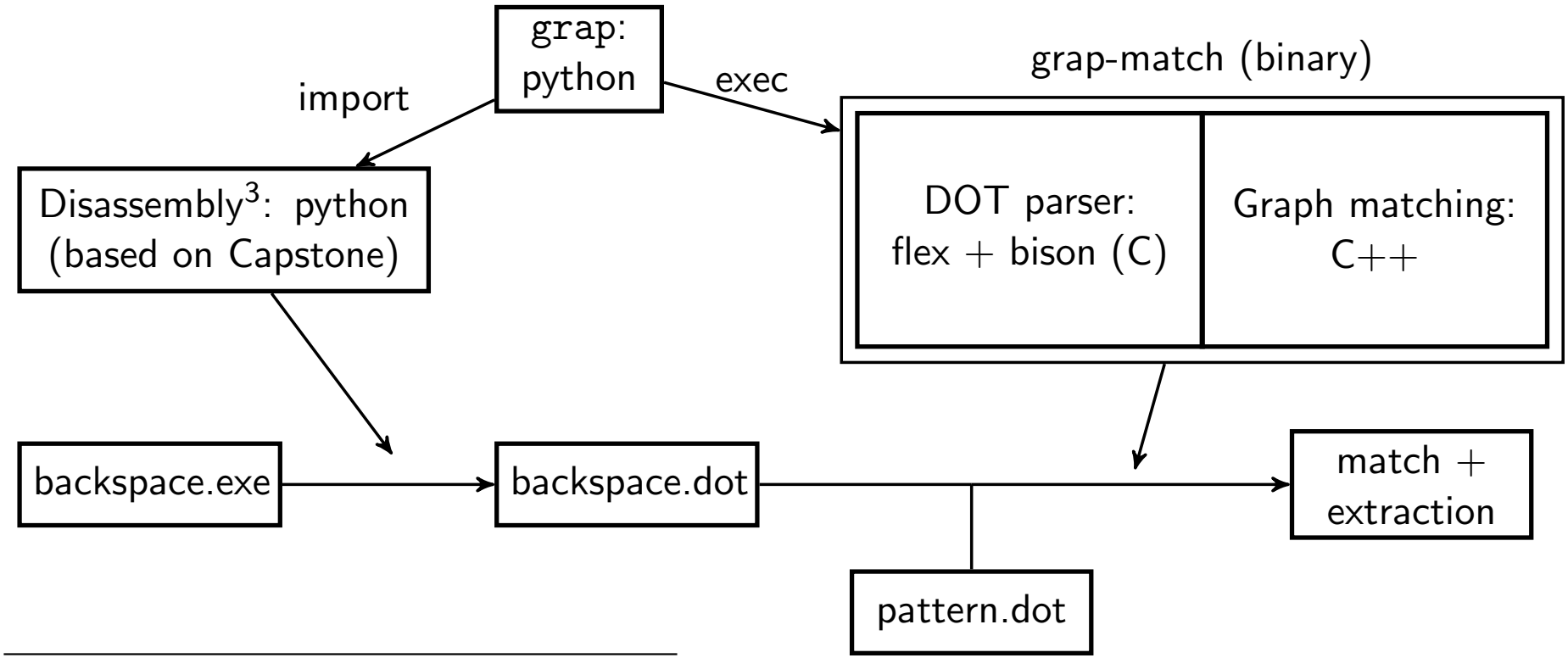
- Works on bytes (regular expression)
- What is "80??1180??25" ?

grap:

- Based on the instructions and their graph
- "xor ??, 0x11" **then** "sub ??, 0x25"

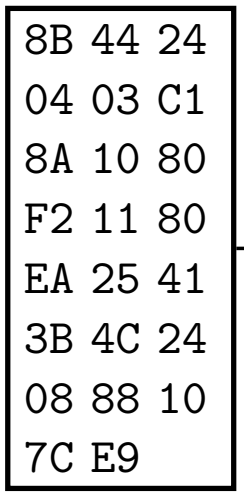
grap overview (standalone tool)

- graphs: DOT[1] files
- grap[2]: **standalone tool** + python bindings (pygrap) + IDA plugin

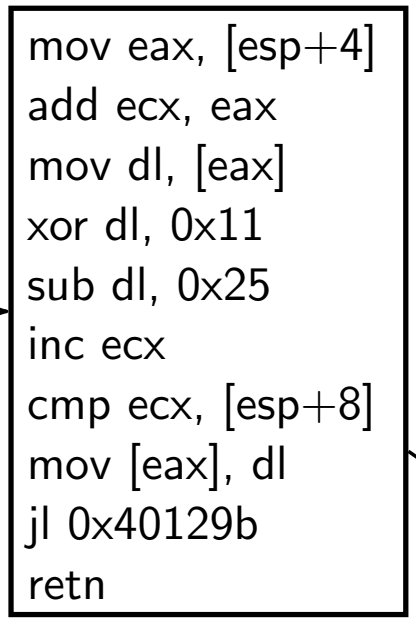


[1] The DOT Language: `http://www.graphviz.org/content/dot-language`
[2] Open source: `https://bitbucket.org/cybertools/grap`
[3] Thanks to @YoannFrancou for his work on the disassembler
[4]

Control flow graph (CFG)

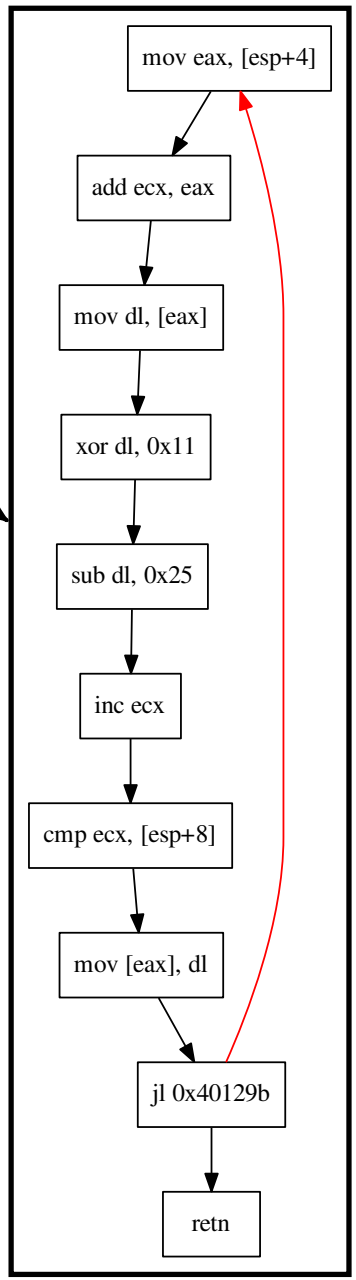


Bytes (hex)



Assembly listing

Control flow graph (CFG)



Standalone tool:

- Recursive (static) disassembler
- Based on Capstone

IDA plugin:

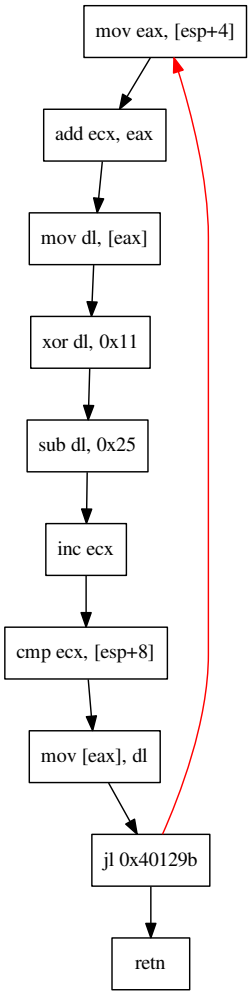
- Graph created by IDA

AIRBUS

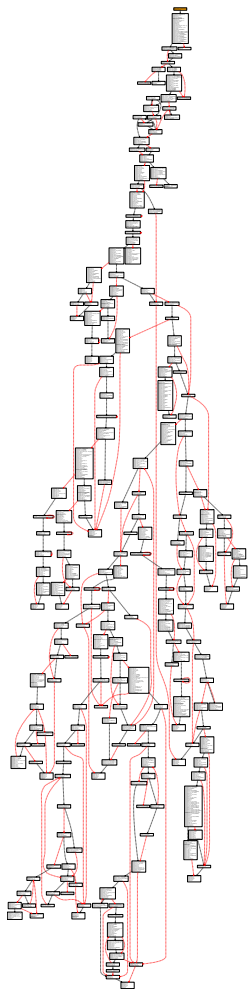
STORMSHIELD

Graph matching

Pattern (10 nodes)

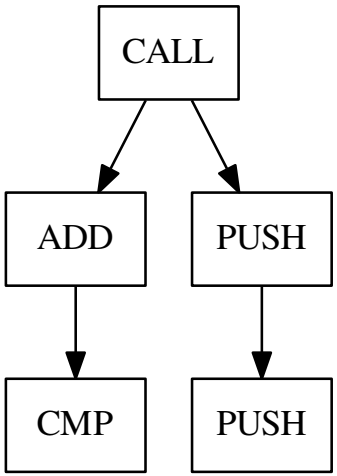


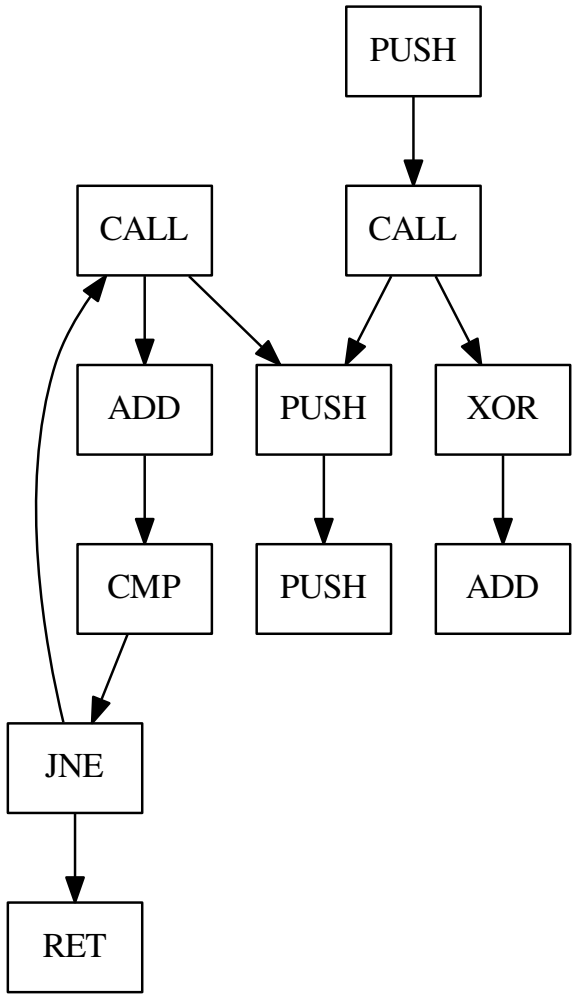
Test (8820 nodes)



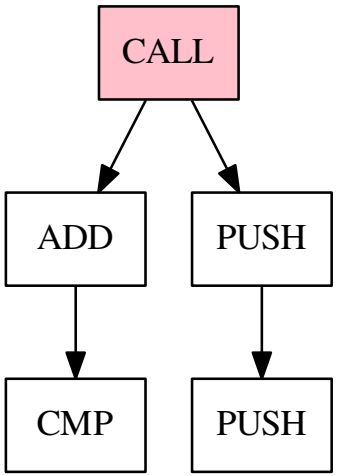
Can we find the pattern graph within the test graph ?

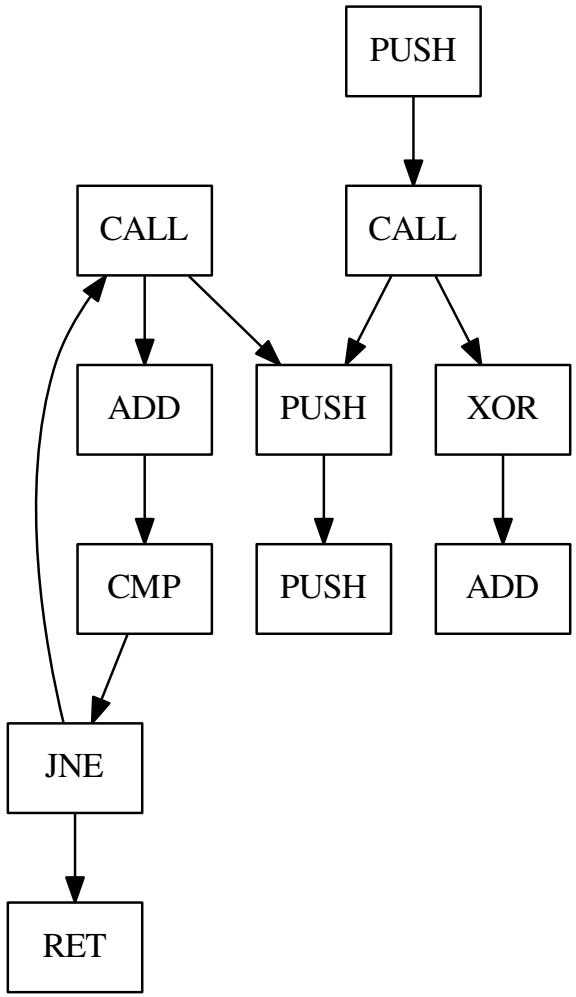
Subgraph isomorphism



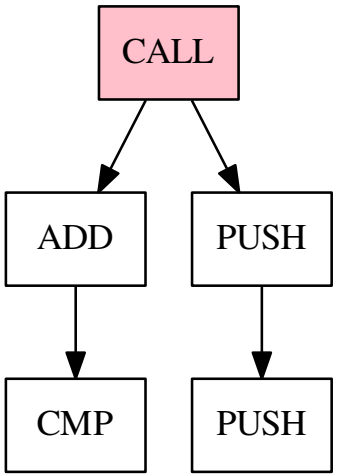


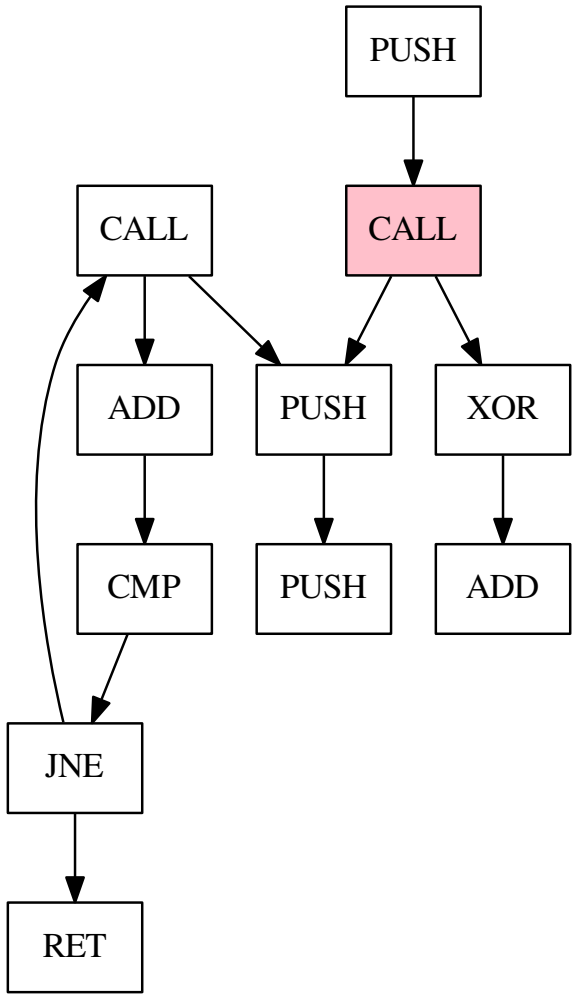
Subgraph isomorphism



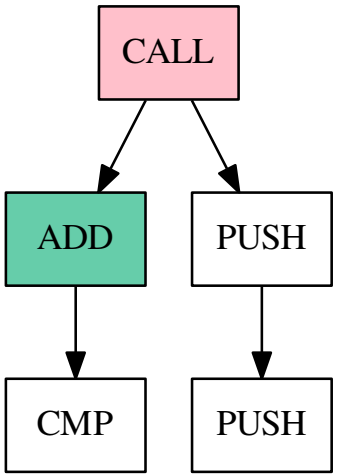


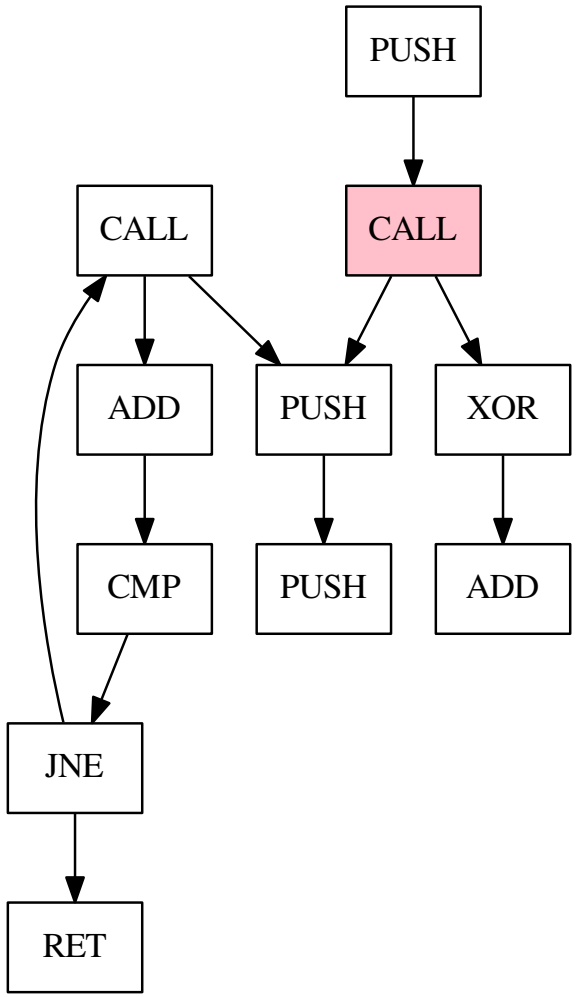
Subgraph isomorphism



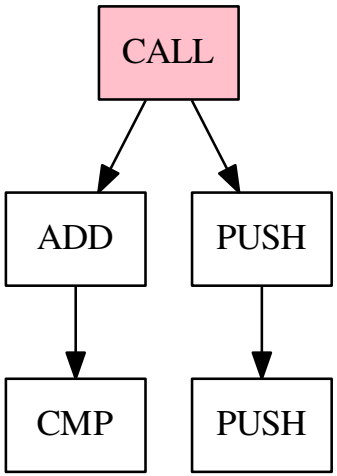


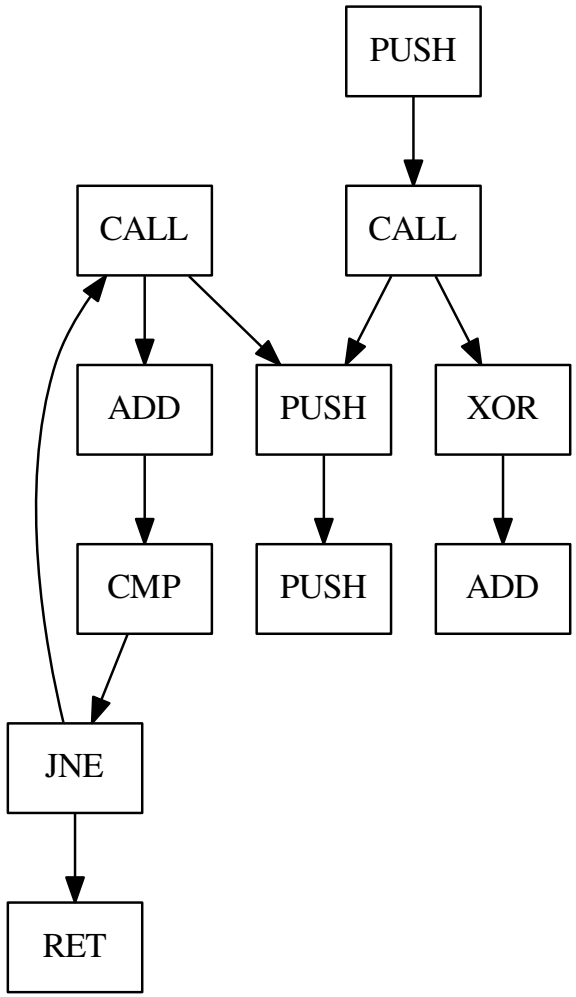
Subgraph isomorphism



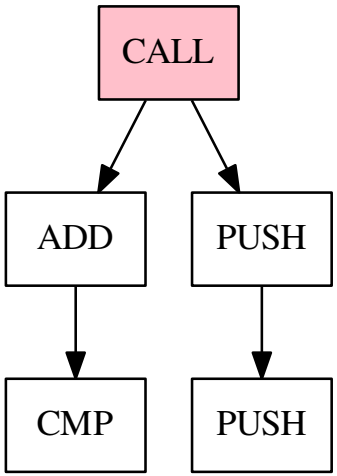


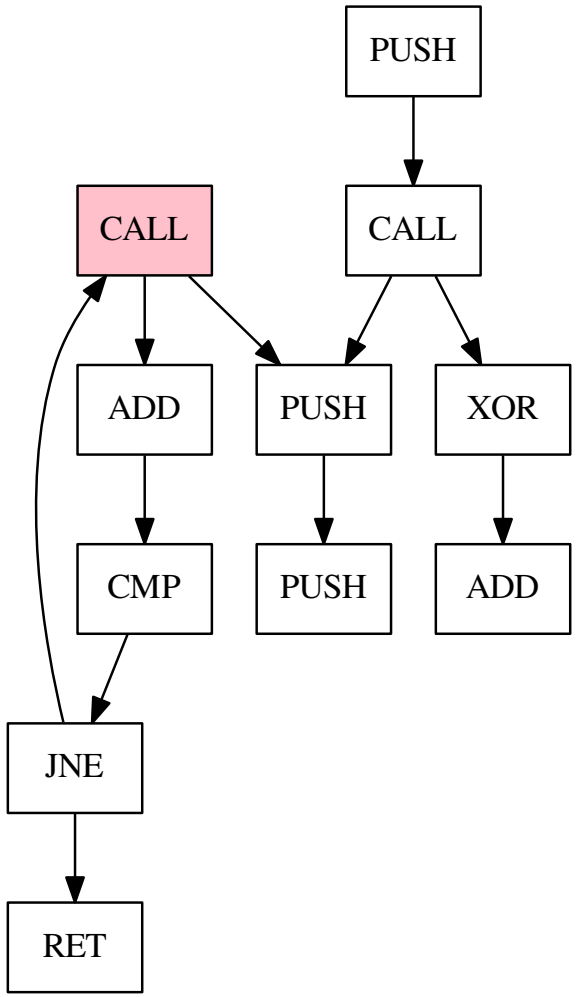
Subgraph isomorphism



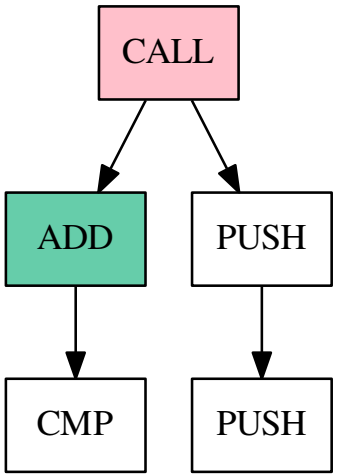


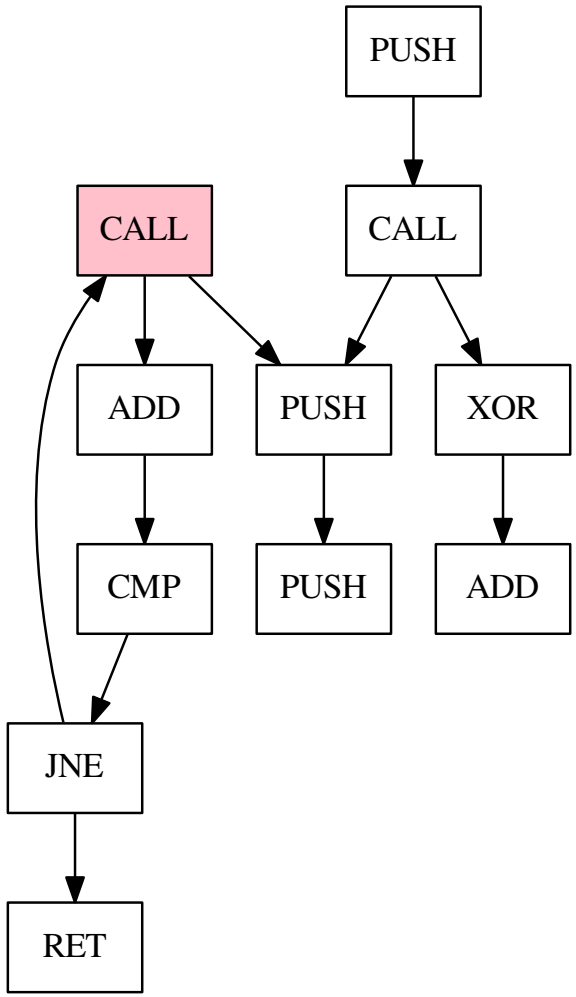
Subgraph isomorphism



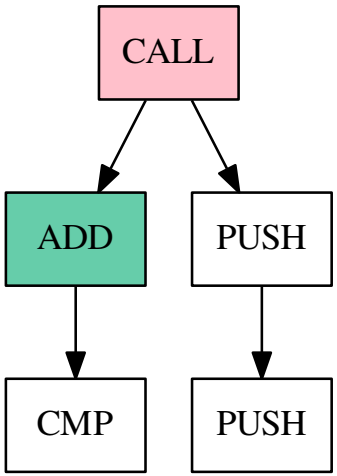


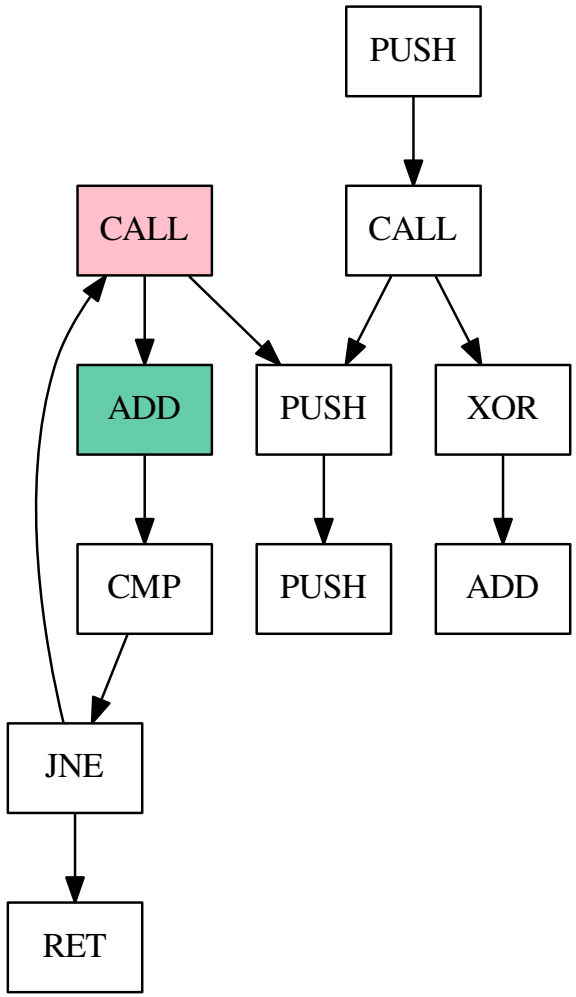
Subgraph isomorphism



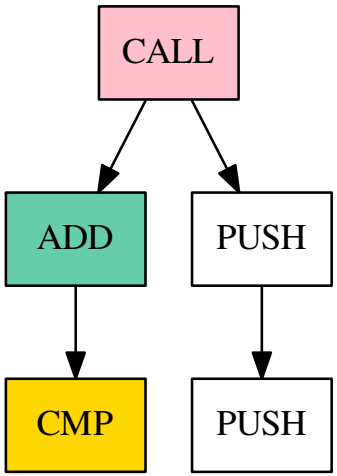


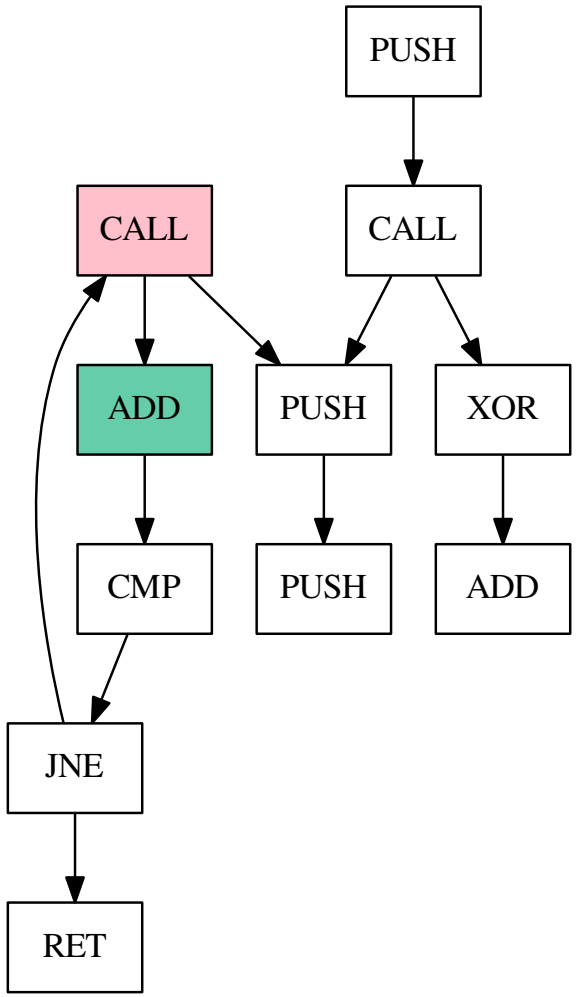
Subgraph isomorphism



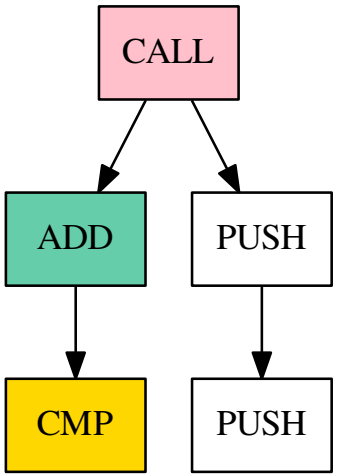


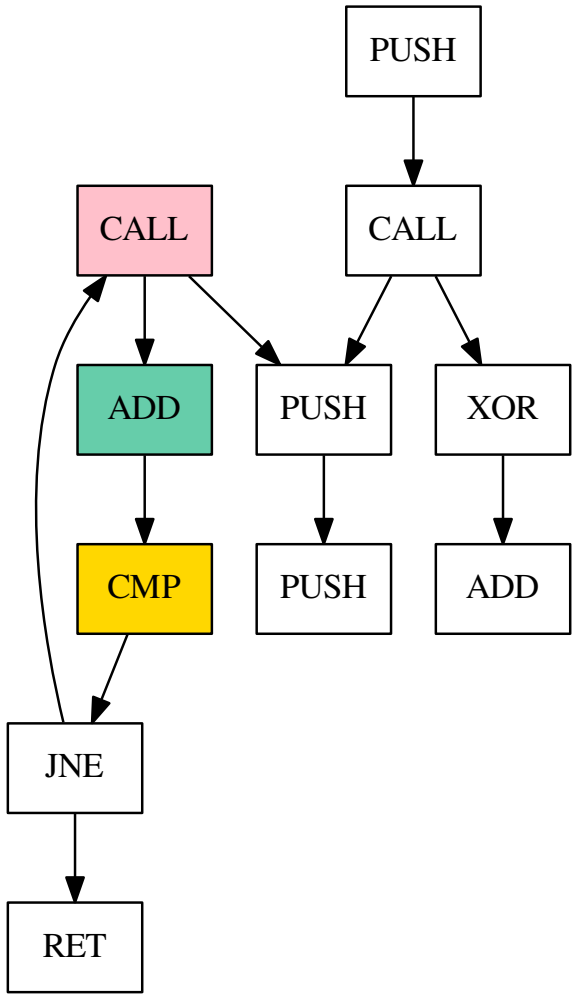
Subgraph isomorphism



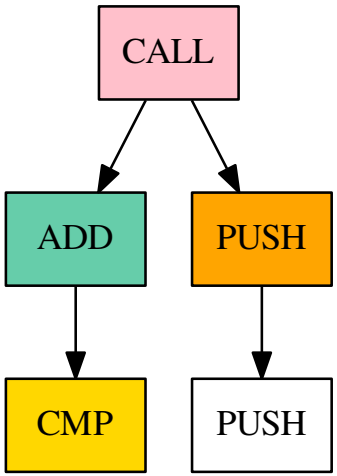


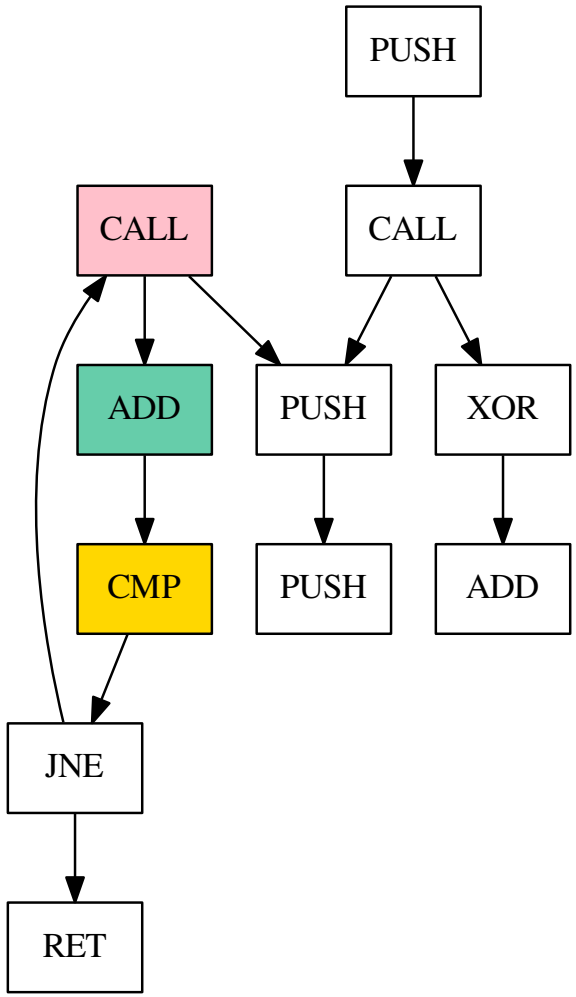
Subgraph isomorphism



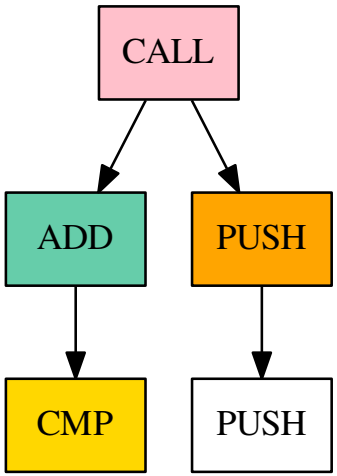


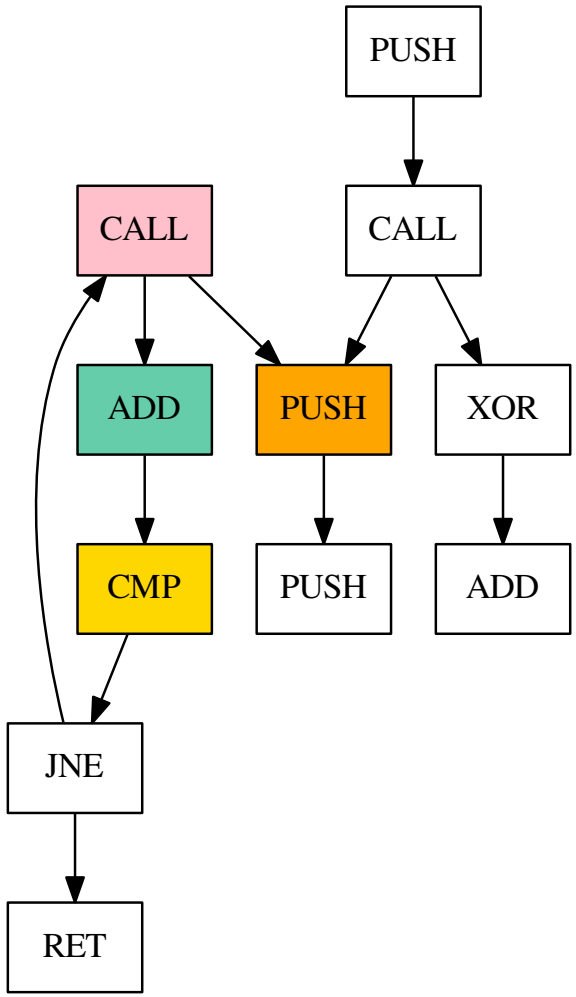
Subgraph isomorphism



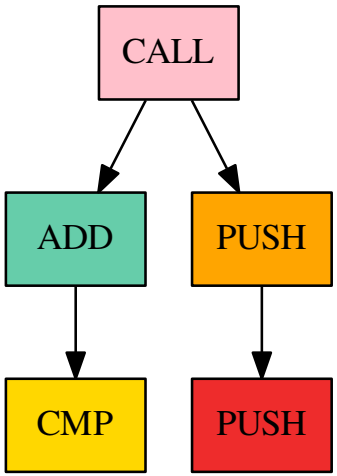


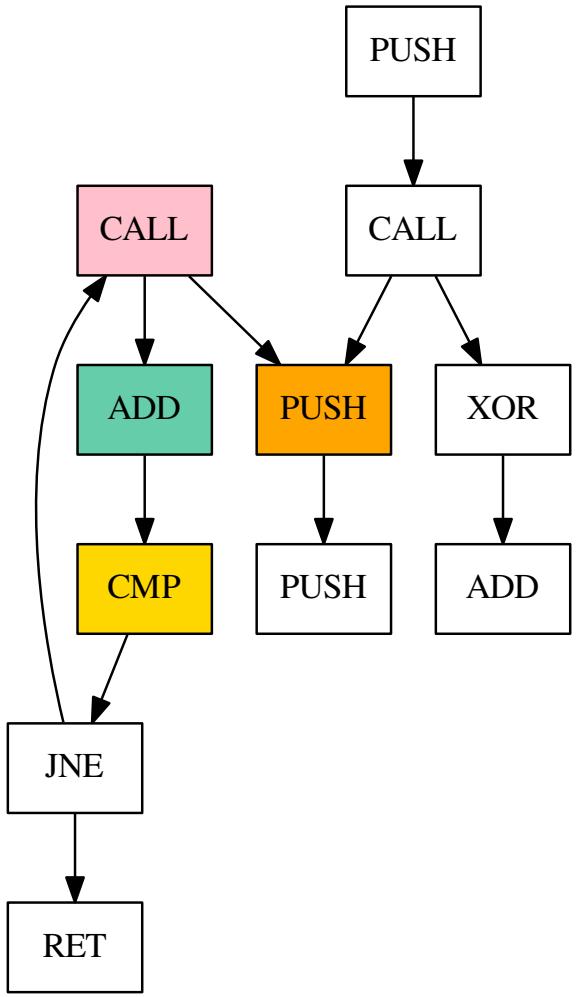
Subgraph isomorphism



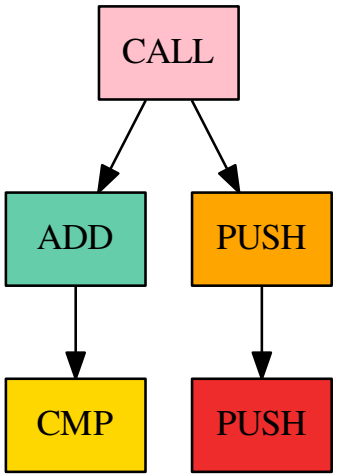


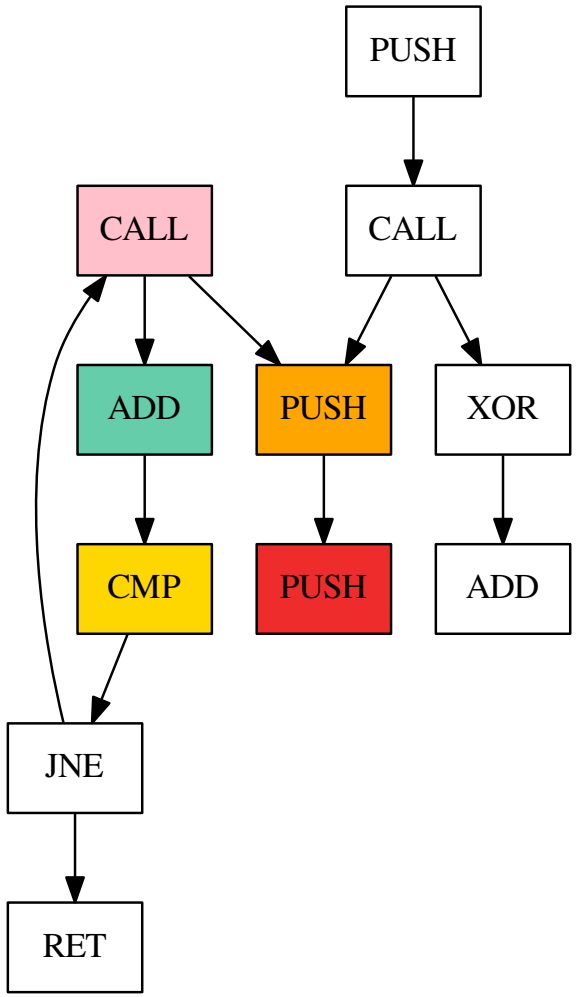
Subgraph isomorphism



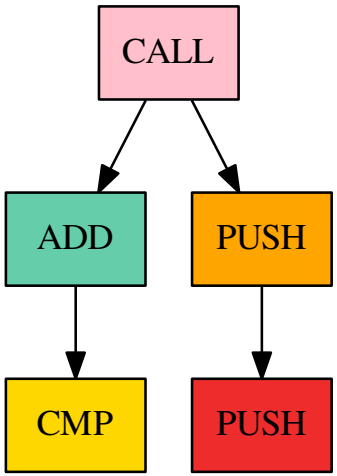


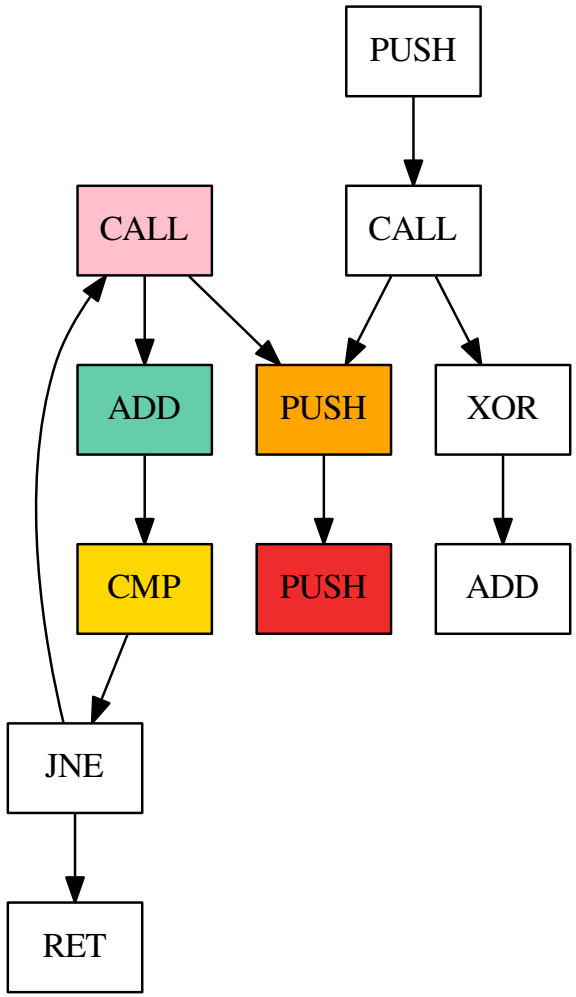
Subgraph isomorphism





Subgraph isomorphism





- Need to check every child node
- Exponential time (NP-Complete)
- Slow on big graphs

Simplify the problem $\rightarrow$ Fast resolution (polynomial time)

Simplification on pattern and test graphs

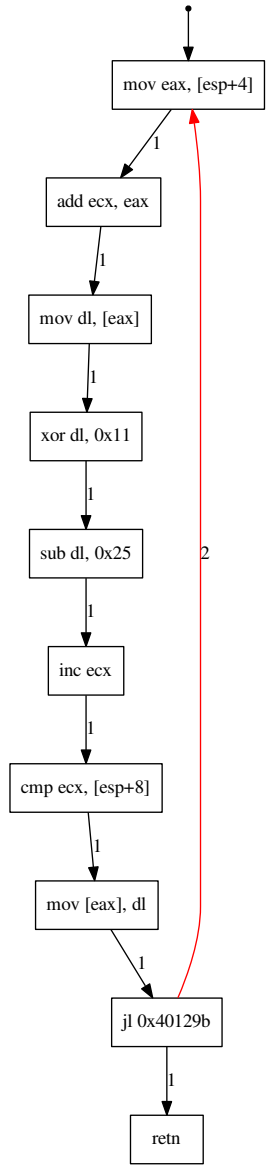
Control flow graphs:

- At most two children
- Children order: (1) the following instruction
  (2) a remote instruction
- Children ordered $\rightarrow$ not really graphs

Pattern graphs can be matched from their first node:

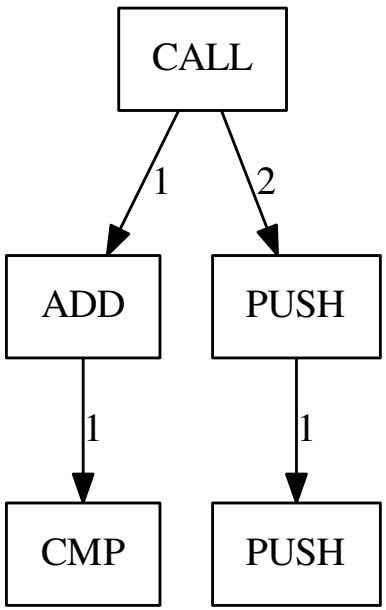
- Pattern: CFG with a root node

$\rightarrow$ Fast matching (polynomial time)

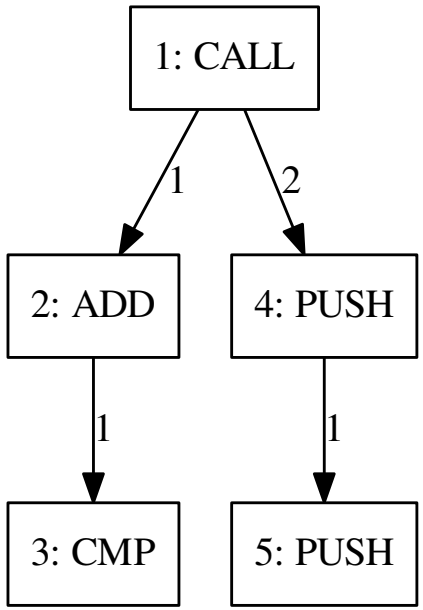


Node numbering

Pattern



Numbering: Depth First Search

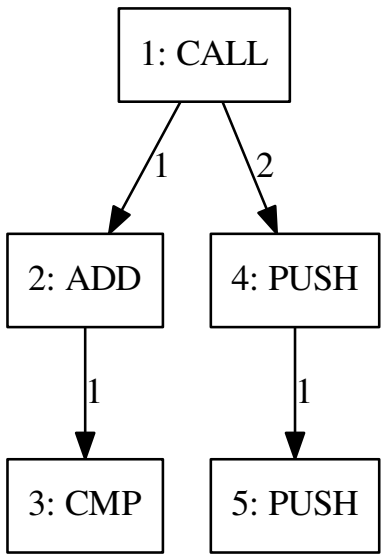


Traversal description:

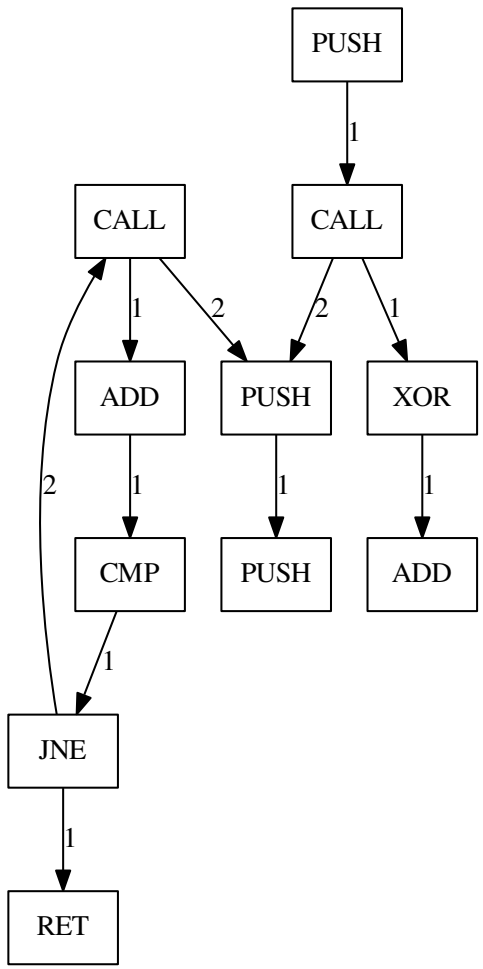
$$1 : \mathit{CALL} \xrightarrow{1} 2 : \mathit{ADD} \xrightarrow{1} 3 : \mathit{CMP} \xrightarrow{R} 1 \xrightarrow{2} 4 : \mathit{PUSH} \xrightarrow{1} 5 : \mathit{PUSH}$$

Traversal within a test graph

Can we perform the traversal (of the pattern) within the test graph ?

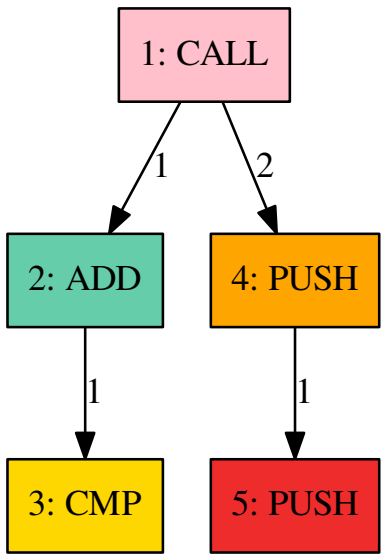


$1 : CALL \xrightarrow{1} 2 : ADD \xrightarrow{1} 3 : CMP \xrightarrow{R} 1$
$\xrightarrow{2} 4 : PUSH \xrightarrow{1} 5 : PUSH$

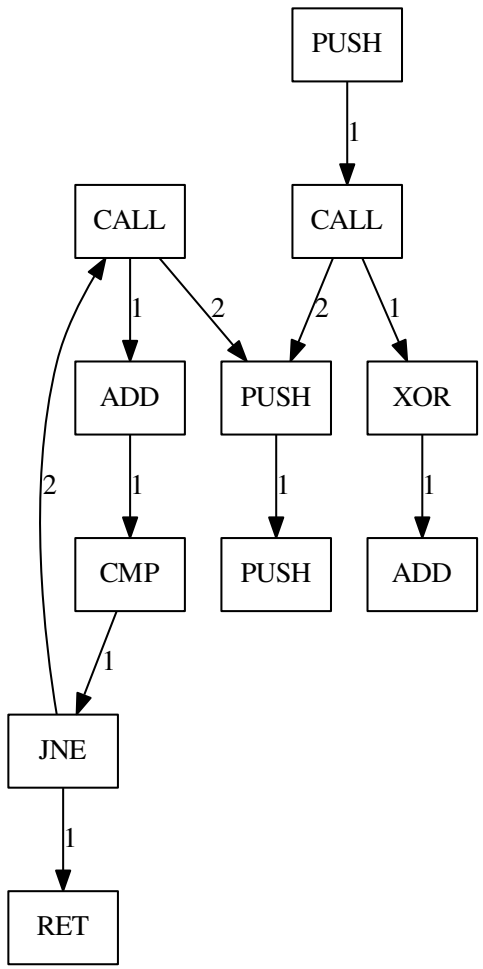


Traversal within a test graph

Can we perform the traversal (of the pattern) within the test graph ?

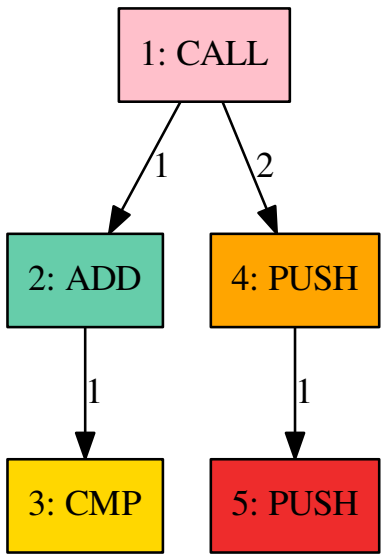


$1 : CALL \xrightarrow{1} 2 : ADD \xrightarrow{1} 3 : CMP \xrightarrow{R} 1$
$\xrightarrow{2} 4 : PUSH \xrightarrow{1} 5 : PUSH$

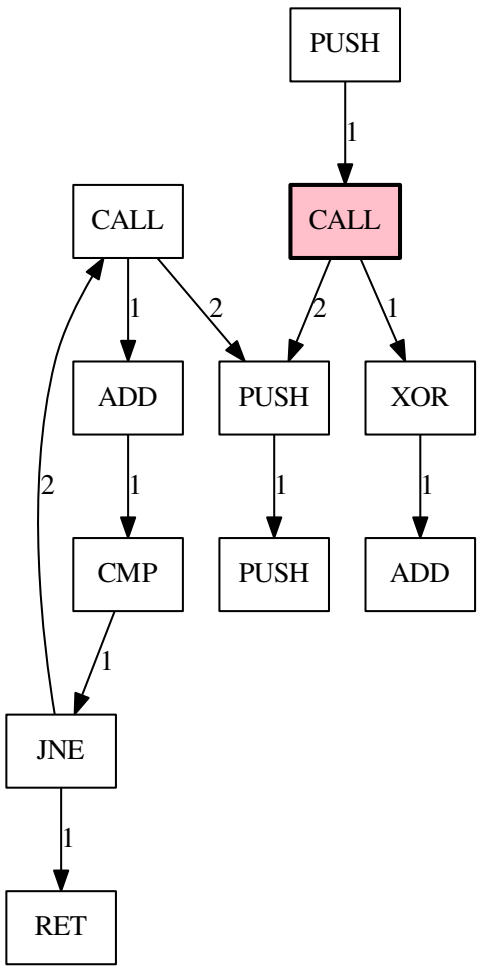


Traversal within a test graph

Can we perform the traversal (of the pattern) within the test graph ?

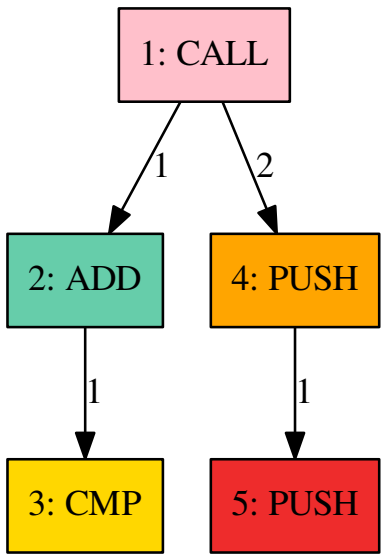


$$1 : \mathit{CALL} \xrightarrow{1} 2 : \mathit{ADD} \xrightarrow{1} 3 : \mathit{CMP} \xrightarrow{R} 1$$
$$\xrightarrow{2} 4 : \mathit{PUSH} \xrightarrow{1} 5 : \mathit{PUSH}$$

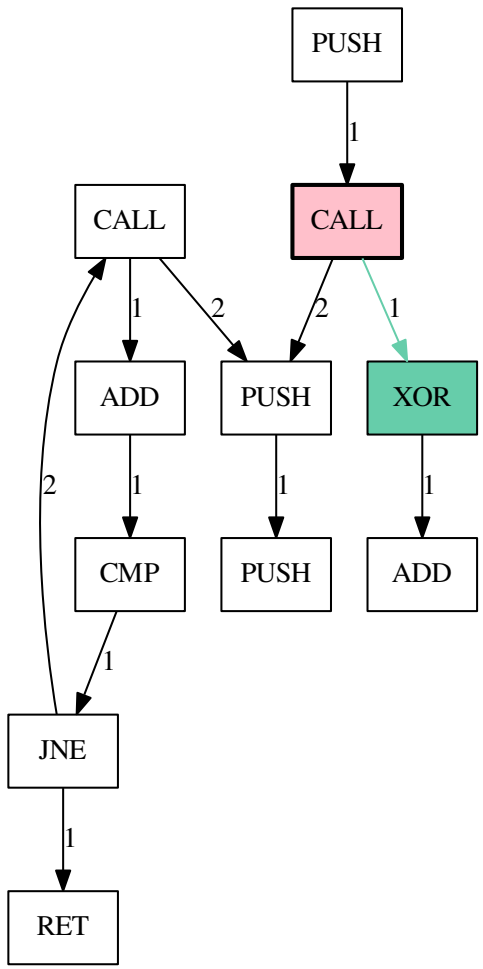


Traversal within a test graph

Can we perform the traversal (of the pattern) within the test graph ?

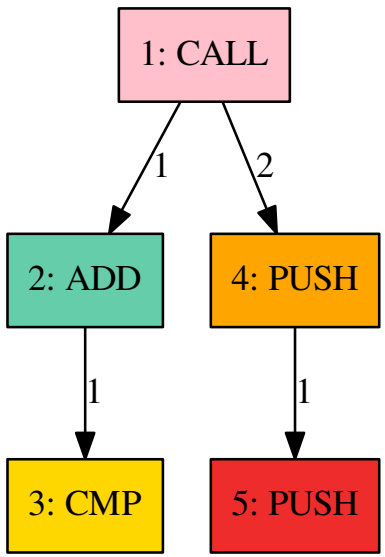


$1 : \mathit{CALL} \xrightarrow{1} 2 : \mathbf{\mathit{ADD}} \xrightarrow{1} 3 : \mathit{CMP} \xrightarrow{R} 1$
$\xrightarrow{2} 4 : \mathit{PUSH} \xrightarrow{1} 5 : \mathit{PUSH}$

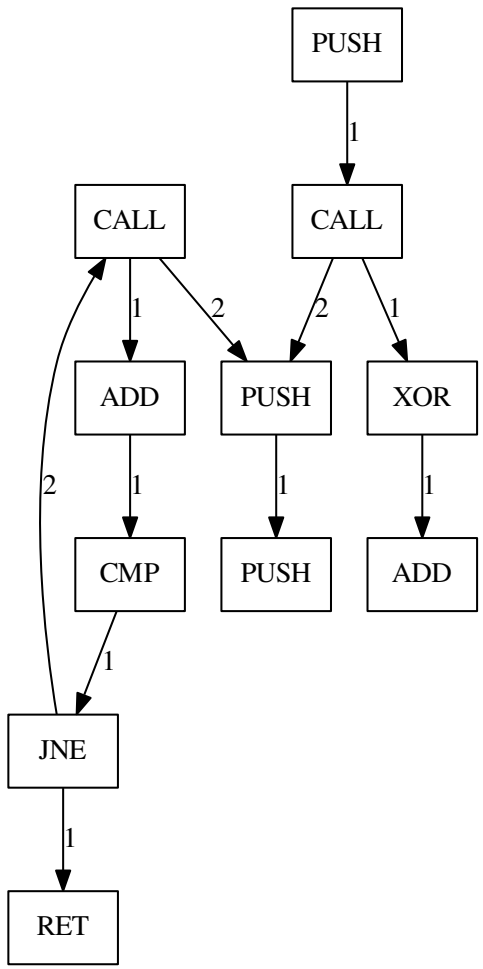


Traversal within a test graph

Can we perform the traversal (of the pattern) within the test graph ?

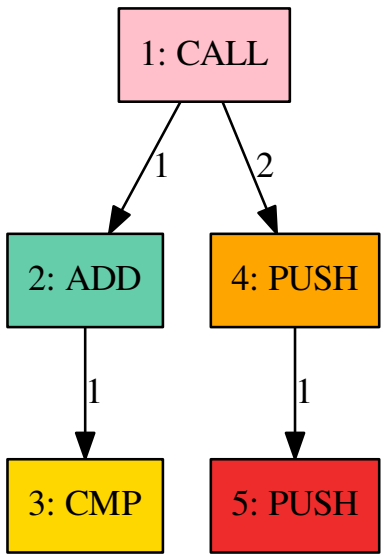


$1 : \mathit{CALL} \xrightarrow{1} 2 : \mathit{ADD} \xrightarrow{1} 3 : \mathit{CMP} \xrightarrow{R} 1$
$\xrightarrow{2} 4 : \mathit{PUSH} \xrightarrow{1} 5 : \mathit{PUSH}$

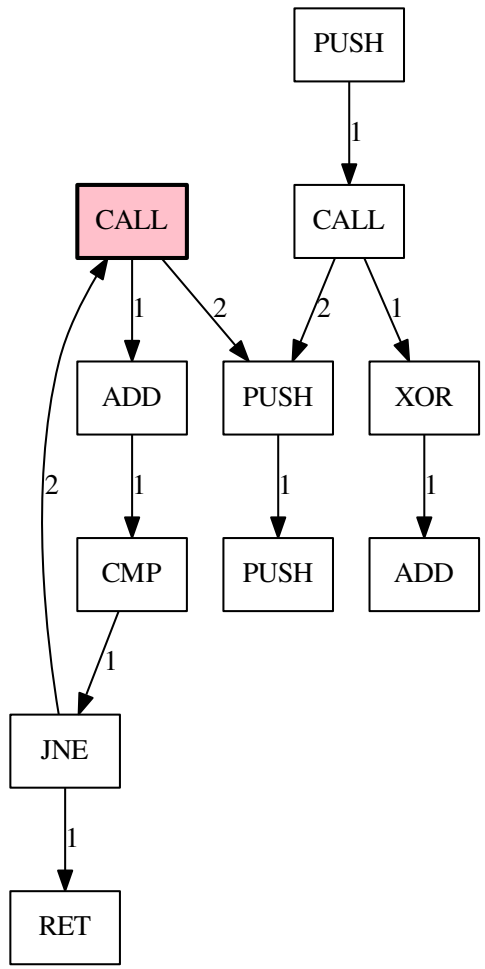


Traversal within a test graph

Can we perform the traversal (of the pattern) within the test graph ?

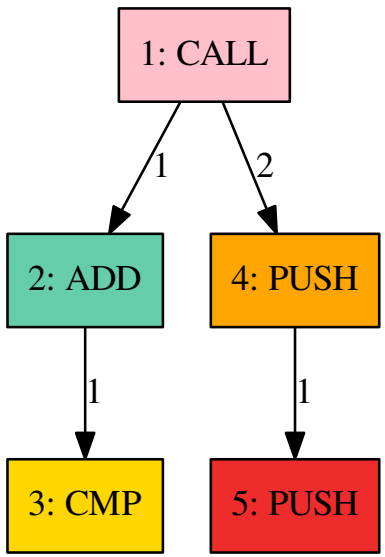


$1 : \mathit{CALL} \xrightarrow{1} 2 : \mathit{ADD} \xrightarrow{1} 3 : \mathit{CMP} \xrightarrow{R} 1$
$\xrightarrow{2} 4 : \mathit{PUSH} \xrightarrow{1} 5 : \mathit{PUSH}$

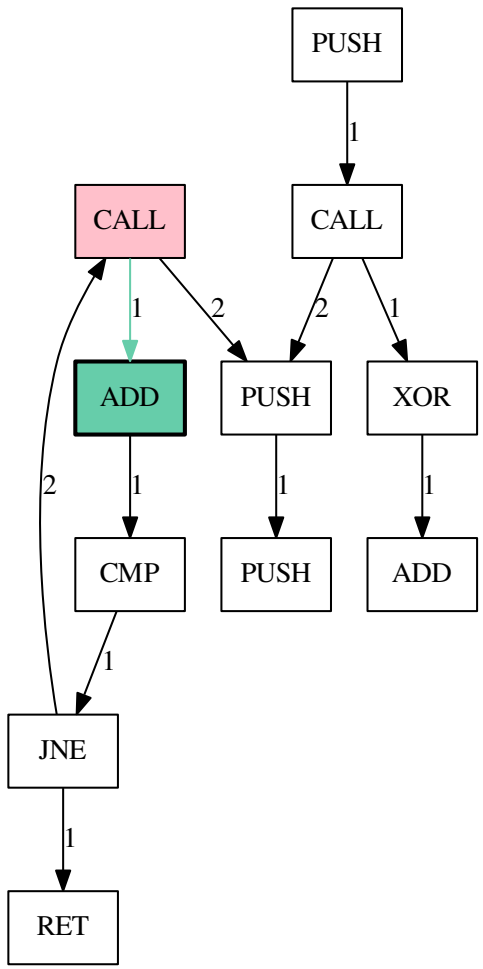


Traversal within a test graph

Can we perform the traversal (of the pattern) within the test graph ?

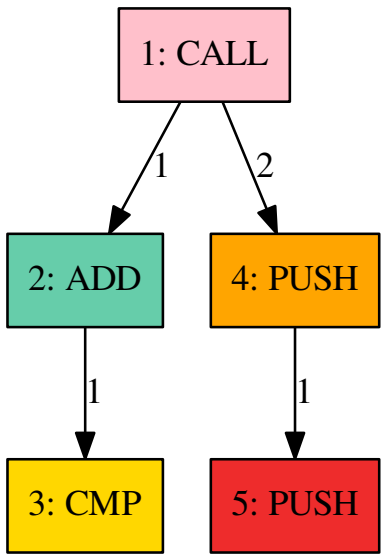


$1 : \mathit{CALL} \xrightarrow{1} 2 : \mathit{ADD} \xrightarrow{1} 3 : \mathit{CMP} \xrightarrow{R} 1$
$\xrightarrow{2} 4 : \mathit{PUSH} \xrightarrow{1} 5 : \mathit{PUSH}$

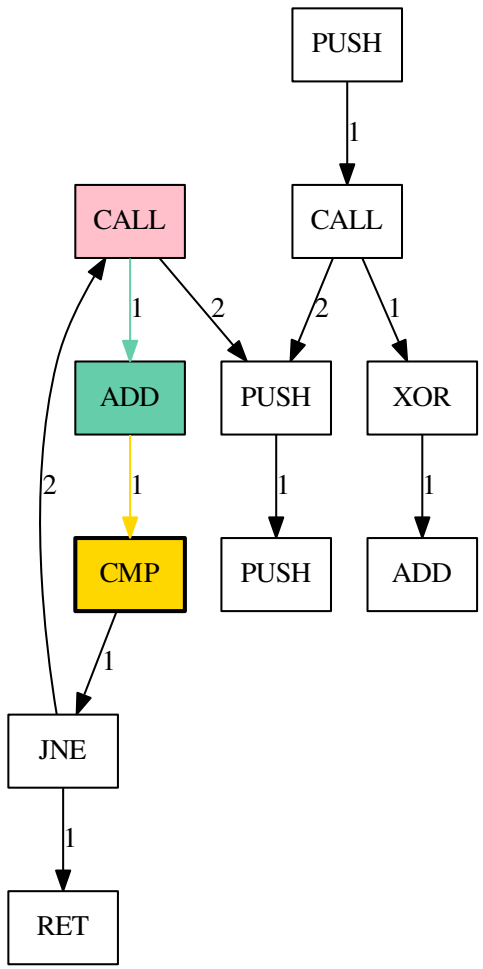


Traversal within a test graph

Can we perform the traversal (of the pattern) within the test graph ?

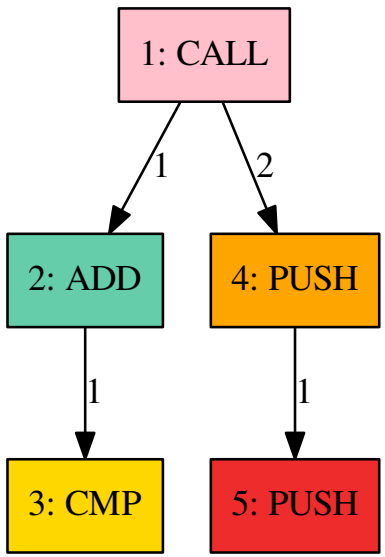


$1 : \mathit{CALL} \xrightarrow{1} 2 : \mathit{ADD} \xrightarrow{1} 3 : \mathit{CMP} \xrightarrow{R} 1$
$\xrightarrow{2} 4 : \mathit{PUSH} \xrightarrow{1} 5 : \mathit{PUSH}$

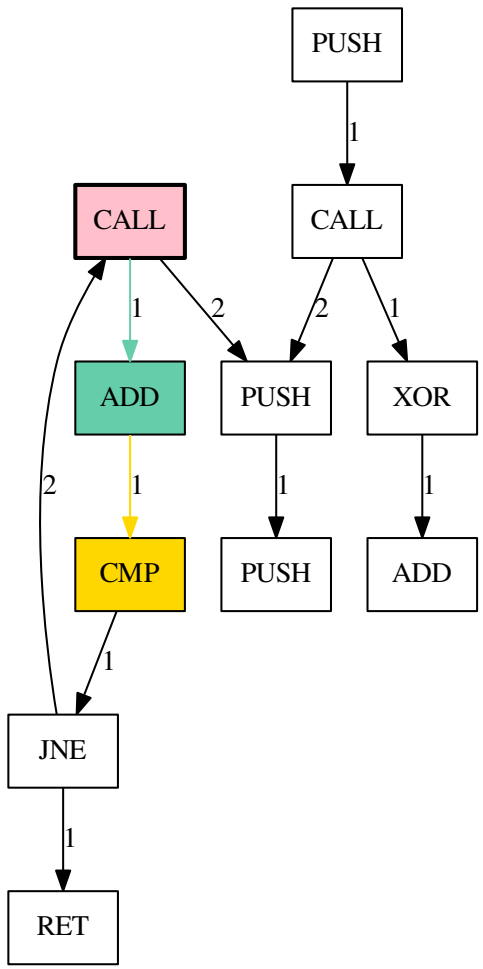


Traversal within a test graph

Can we perform the traversal (of the pattern) within the test graph ?

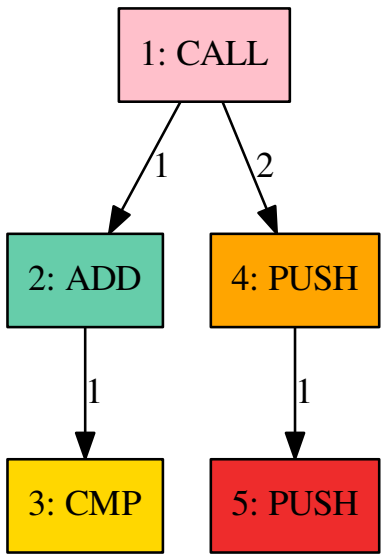


$1 : CALL \xrightarrow{1} 2 : ADD \xrightarrow{1} 3 : CMP \xrightarrow{R} 1$
$\xrightarrow{2} 4 : PUSH \xrightarrow{1} 5 : PUSH$

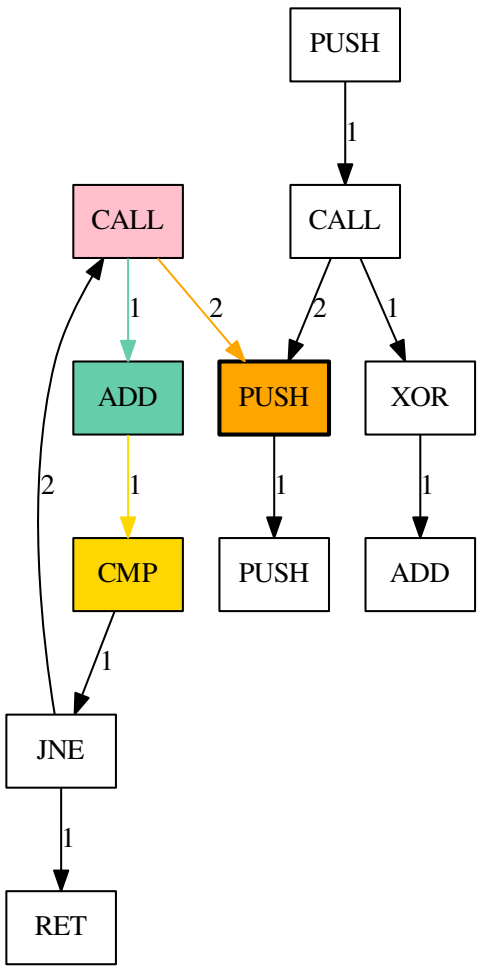


Traversal within a test graph

Can we perform the traversal (of the pattern) within the test graph ?

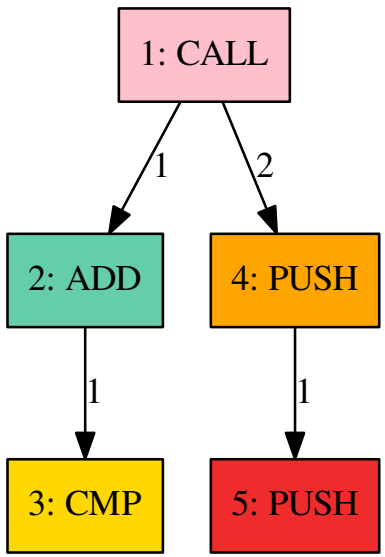


$1 : CALL \xrightarrow{1} 2 : ADD \xrightarrow{1} 3 : CMP \xrightarrow{R} 1$
$\xrightarrow{2} 4 : PUSH \xrightarrow{1} 5 : PUSH$

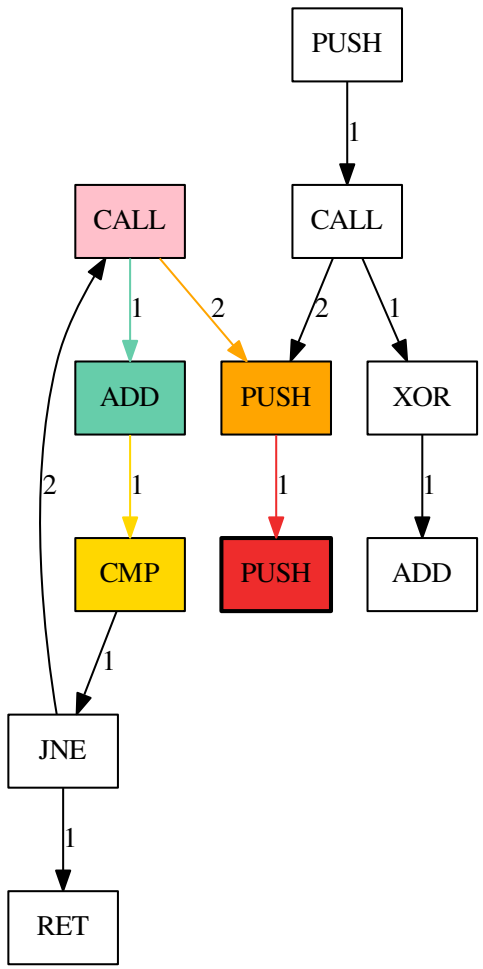


Traversal within a test graph

Can we perform the traversal (of the pattern) within the test graph ?

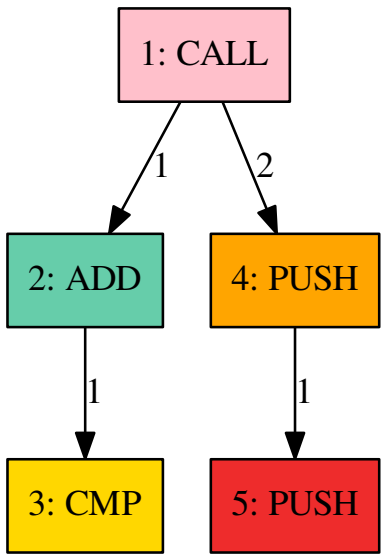


$1 : CALL \xrightarrow{1} 2 : ADD \xrightarrow{1} 3 : CMP \xrightarrow{R} 1$
$\xrightarrow{2} 4 : PUSH \xrightarrow{1} \mathbf{5 : PUSH}$



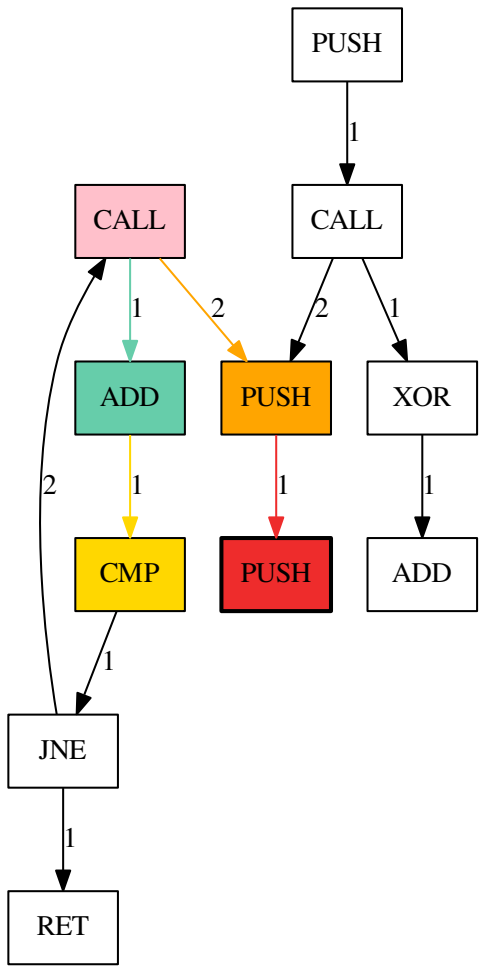
Traversal within a test graph

Can we perform the traversal (of the pattern) within the test graph ?

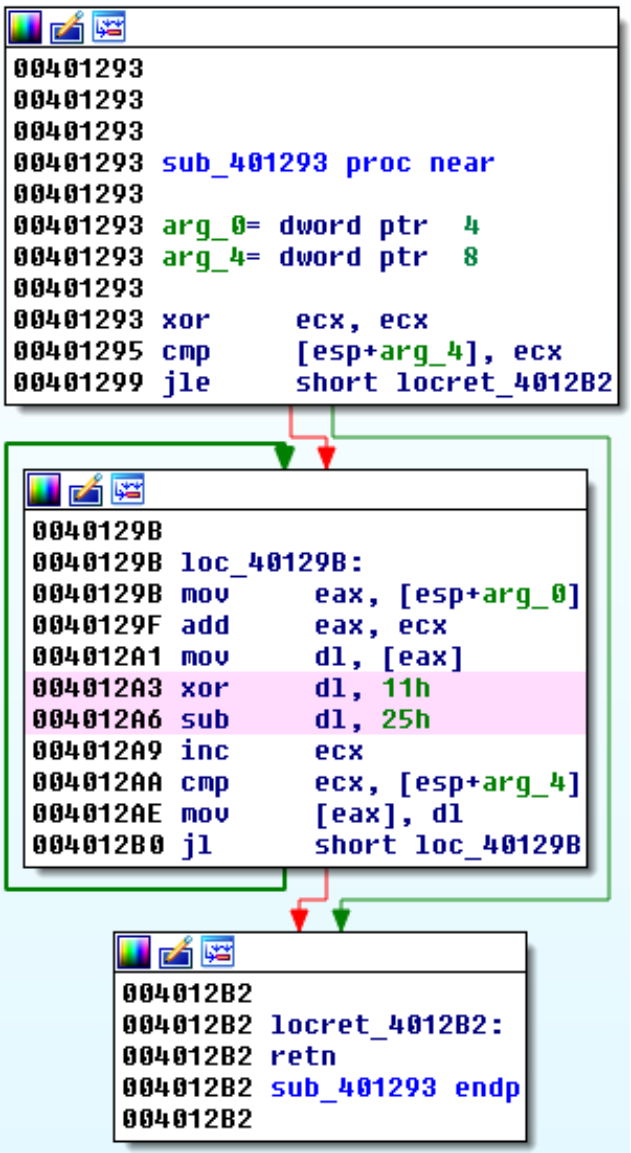


$1 : \mathit{CALL} \xrightarrow{1} 2 : \mathit{ADD} \xrightarrow{1} 3 : \mathit{CMP} \xrightarrow{R} 1$
$\xrightarrow{2} 4 : \mathit{PUSH} \xrightarrow{1} \mathbf{5 : PUSH}$

- Check one child at a time (1 or 2)
- Fast (polynomial time)



Pattern example



Patterns:

- DOT files with specific fields
- **condition** on opcode, arguments, address, number of incoming and outgoing edges

digraph decrypt_xor_sub {
A [**cond**="opcode is xor and arg2 is 0x11"]
B [**cond**="opcode is sub and arg2 is 0x25"]

A -> B
}

Pattern syntax: node and edge options

Node options:

- **root=true**: specify pattern's root
- **cond**: condition to match against
- **getid**: keep matched node with specified id

Edge option:

- **childnumber**: 1 (following instruction) or 2 (remote instruction)

```
digraph decrypt_xor_sub {
A [cond="opcode is xor and arg2 is 0x11", root=true, getid=A]
B [cond="opcode is sub and arg2 is 0x25", getid=B]

A -> B [childnumber=1]
}
```

Pattern syntax: condition fields

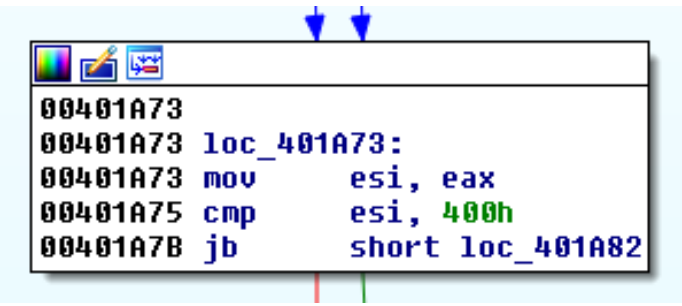
- **instruction** (string): full disassembled instruction
- **address** (number): address (VA) of the instruction
- **opcode** (string): mnemonics
- **nargs** (number): number of arguments
- **arg1** and **arg2** (string)
- **nfathers** and **nchildren**: number of incoming and outgoing edges

```
digraph call_frequent_function {
A [cond="opcode is call"]
B [cond="nfathers >= 5"]

A -> B [childnumber=2]
}
```

Node repetition

How to allow node repetition and define basic blocks ?

- Repetition (*, +, {n, m}) on sequential instructions (1 father, 1 child)



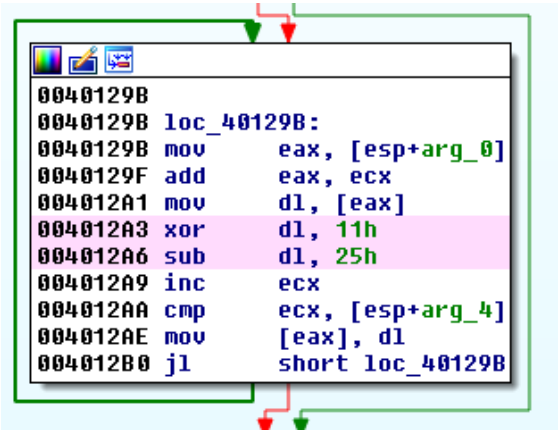

```
digraph basic_block {
A [cond=true, repeat=+]
}
```

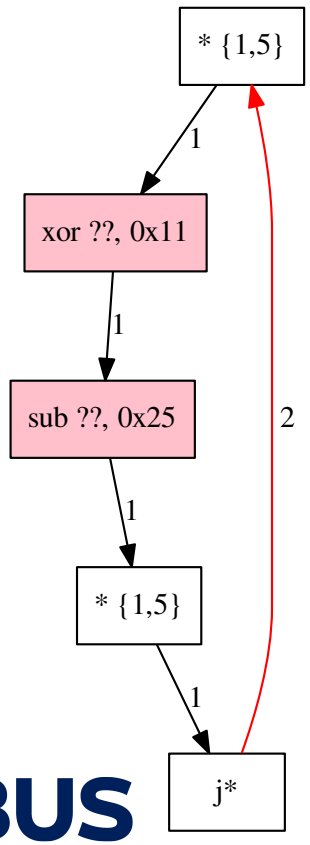
3 push instructions ?

```
digraph pushes {
A [cond="opcode is push", repeat=3]
}
```

- By default: take the **most** matching instructions
- **lazyrepeat = true**: stop when the next condition is fulfilled

Repetition: loop detection





- 1 to 5 any instructions,
- `xor` then `sub`,
- 1 to 5 any instructions,
- conditional jump (loop).

digraph decrypt_sample_4ee {
A [**cond**=true, **maxrepeat**=5, **lazyrepeat**=true]
B [**cond**="opcode is xor and arg2 is 0x11"]
C [**cond**="opcode is sub and arg2 is 0x25"]
D [**cond**=true, **maxrepeat**=5, **lazyrepeat**=true]
E [**cond**="opcode beginswith j and nchildren == 2"]

A −> B [**childnumber**=1]
B −> C [**childnumber**=1]
C −> D [**childnumber**=1]
D −> E [**childnumber**=1]
E −> A [**childnumber**=2]
}

Back to Backspace

100 backspace samples:

- Disassemble them
- Detect the known decryption algorithm
- Find variants of the decryption algorithm
- Detect those decryption algorithms

Demo time (or not)

Back to Backspace: decryption variants

7 variants:

```
xor dl, 0x11
sub dl, 0x25
```

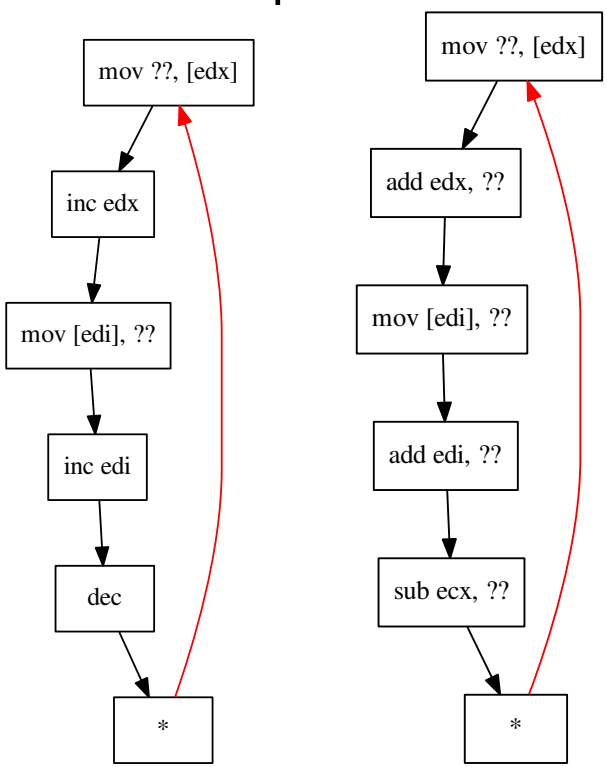
```
sub dl, cl
xor dl, 0xb
sub dl, 0x12
```

```
sub dl, al
add bl, dl
```

```
sub dl, al
xor dl, 0x19
add dl, 0x13
```

...

2 UPX loops:



100 Backspace samples:

- Disassembly: 21s
- Detect 9 patterns: 2.4s
- 17 packed with UPX
- 51 `xor` then `sub` pattern
- 14 `sub`, `xor`, `sub` pattern
- 7 have one of the other patterns
- 11 unidentified patterns

AIRBUS

STORMSHIELD

Python bindings with SWIG

With python:

- Disassembly
- Load pattern and test graphs
- Match patterns
- Parse results

```
import pygrap
from grap_disassembler import disassembler

pattern_graph = pygrap.getGraphFromPath("pattern.dot")
disassembler.disassemble_file(bin_path="test.exe", dot_path="test.dot")
test_graph = pygrap.getGraphFromPath("test.dot")

matches = pygrap.match_graph(pattern_graph, test_graph)
```

Parsing matches


```
digraph pushes {
A [cond="opcode is push", repeat=3, getid="P"]
}
```

```
matches = pygrap.match_graph(pattern_graph, test_graph)
```

- `matches`: **dict** with the names of matching patterns
- `matches["pushes"]`: **list** of matches for pattern "pushes"
- `matches["pushes"][0]`: **dict** of matched instructions
- `matches["pushes"][0]["P"]`: **list** of instructions with **getid** "P"
- `matches["pushes"][0]["P"][0]`: first matched `push` instruction

Parsing matched instructions

```
digraph pushes {
A [cond="opcode is push", repeat=3, getid="P"]
}
```

```
matches = pygrap.match_graph(pattern_graph, test_graph)
inst = matches["pushes"][0]["P"][0]
```

`inst` is a push instruction:

- `inst.info.address`: address (number)
- `inst.info.inst_str`: disassembled instruction (string)
- `inst.info.opcode`: mnemonics (string)
- `inst.info.arg1`, `inst.info.arg2`: arguments (string)

Backspace: decrypt configuration strings

Calls to the decryption algorithm:

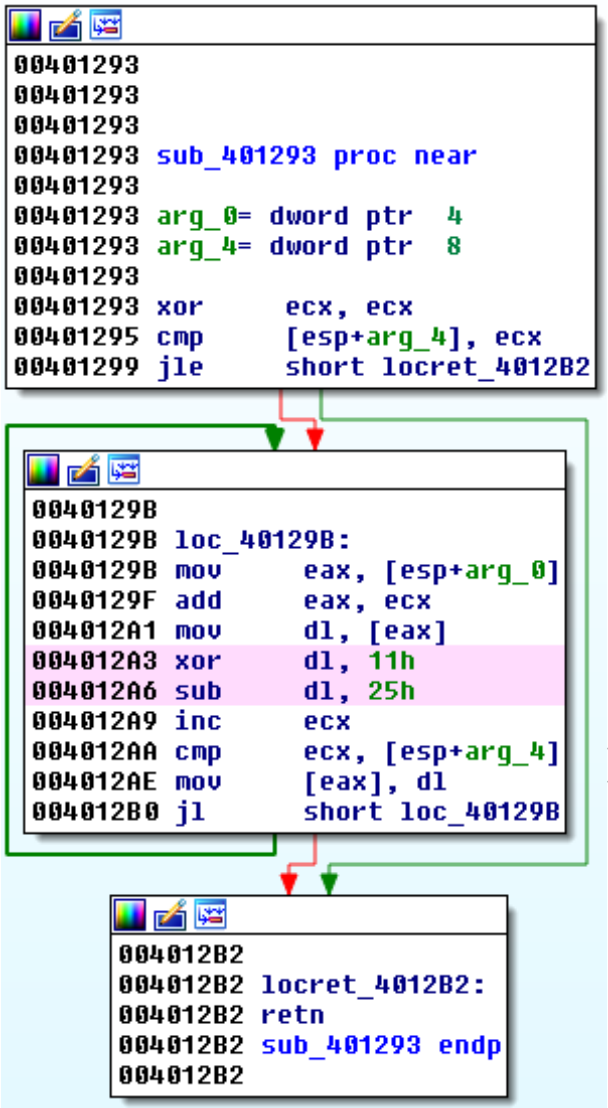
```
push    0Ah
push    offset aEjsialiii ; "Ùà§Öø0ïøëê"
call    sub_401293
```

```
push len
push str_addr
call DECRYPT_ENTRYPOINT
```

- `DECRYPT_ENTRYPOINT`: address of the decryption routine
- Let's write a pattern to get **len** and **str_addr** !
- We need `DECRYPT_ENTRYPOINT` !

Backspace: decrypt configuration strings

Finding the decryption routine's address (DECRYPT_ENTRYPOINT) ?



← DECRYPT_ENTRYPOINT - **30**

← DECRYPT_ENTRYPOINT

match from backspace_decrypt_algos.dot:

← DECRYPT_MATCH

DECRYPT_ENTRYPOINT :

- Before DECRYPT_MATCH
- After DECRYPT_MATCH - **30**
- It has 5 (or more) incoming nodes
- Let's write a pattern !

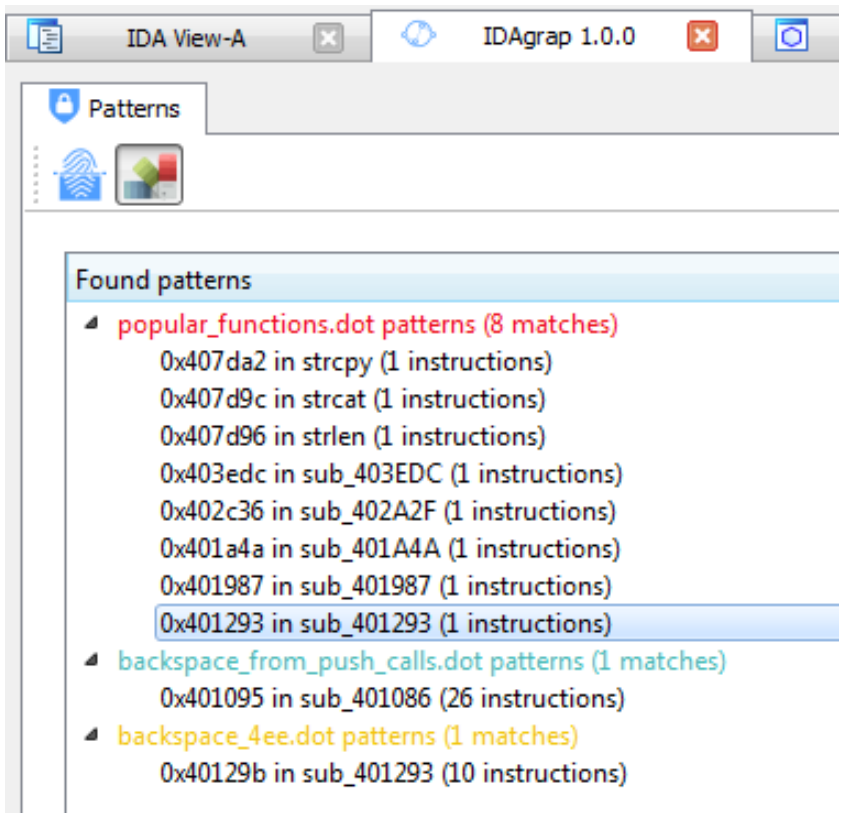
AIRBUS STORMSHIELD

Backspace: decrypt configuration strings

Demo !

IDA plugin

- Convert IDA's graph for `grap`
- Match patterns with pygrap
- Allows to browse and color matches
- → Filtering techniques



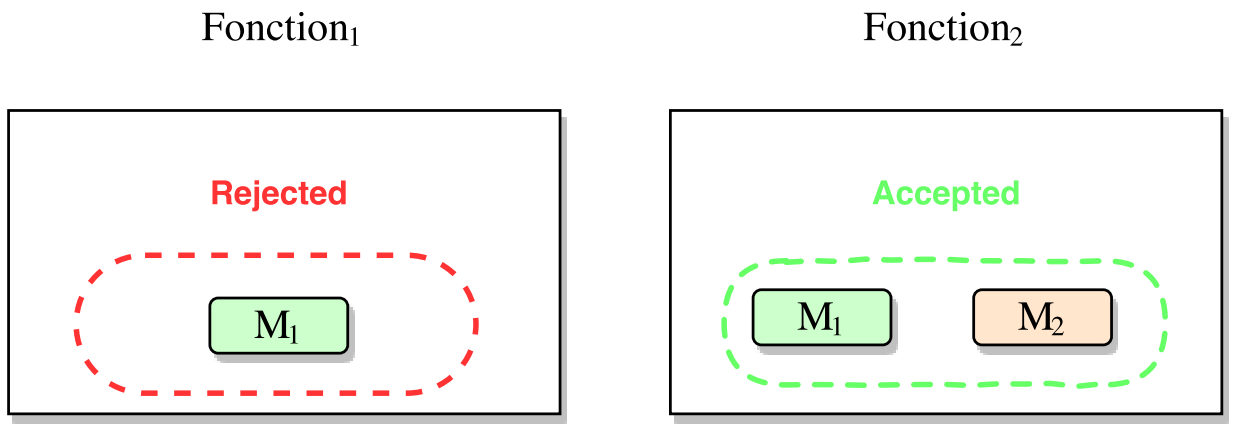
IDAgrap and correlation

grap: match a single pattern

- Use on RC4: two small loops
- With only one pattern: many false positives

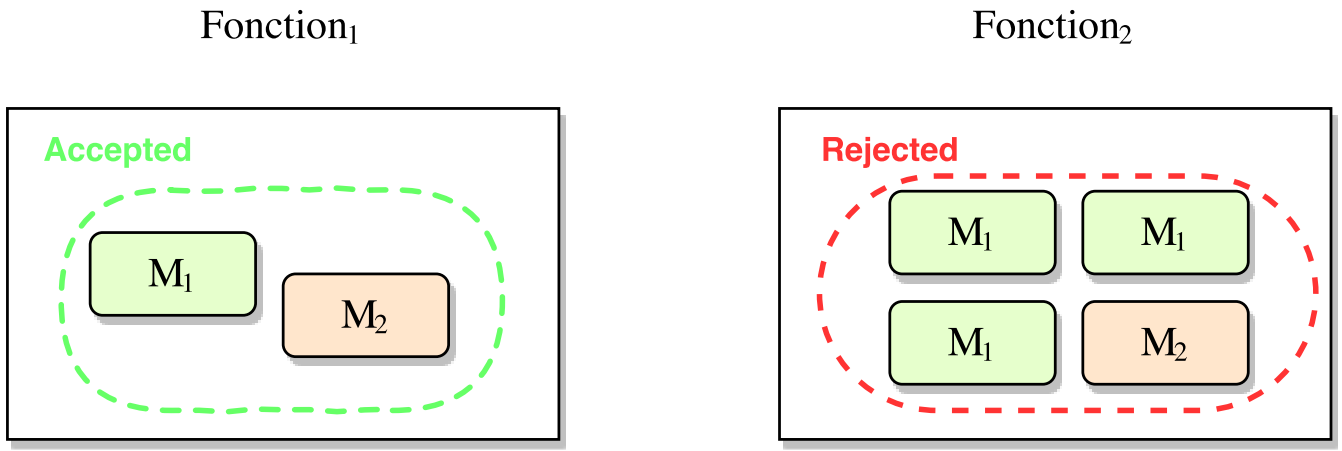
Zone restriction

M_1 and M_2 in the same function ?



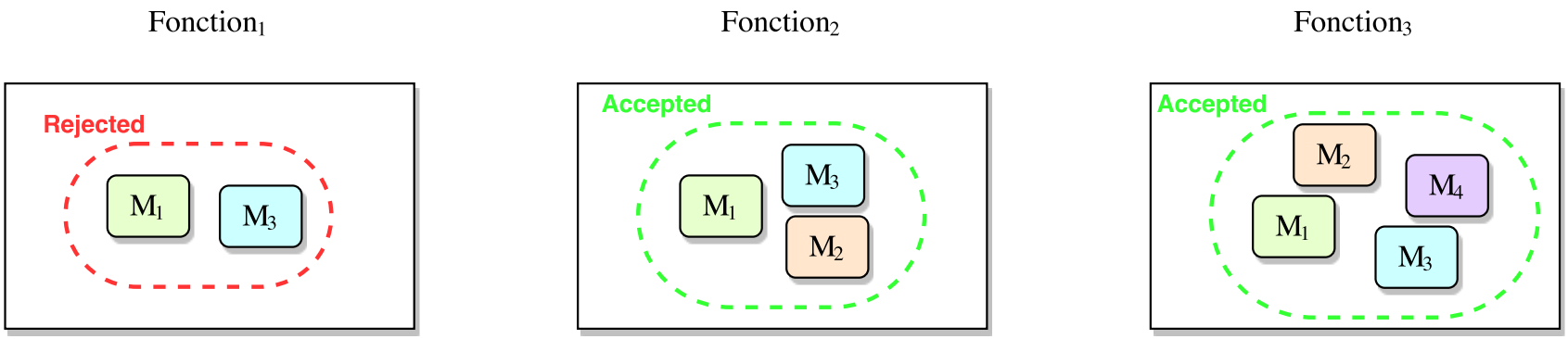
Zone rate

Between 1 and 3 matches of M_1 in the same function ?



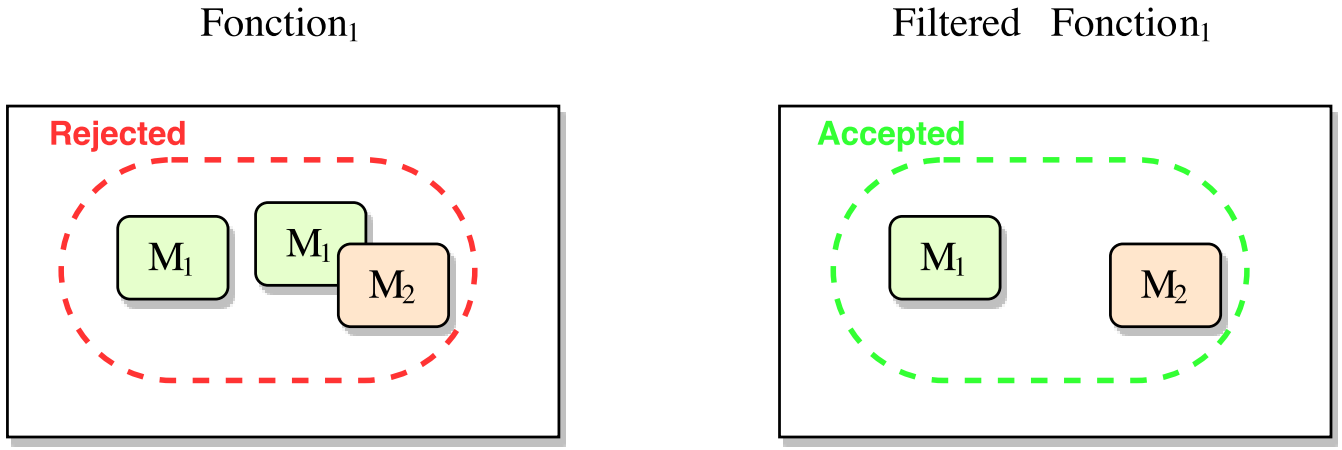
Matching threshold

At least 3 out of 4 (0.75) unique patterns need to be matched ?

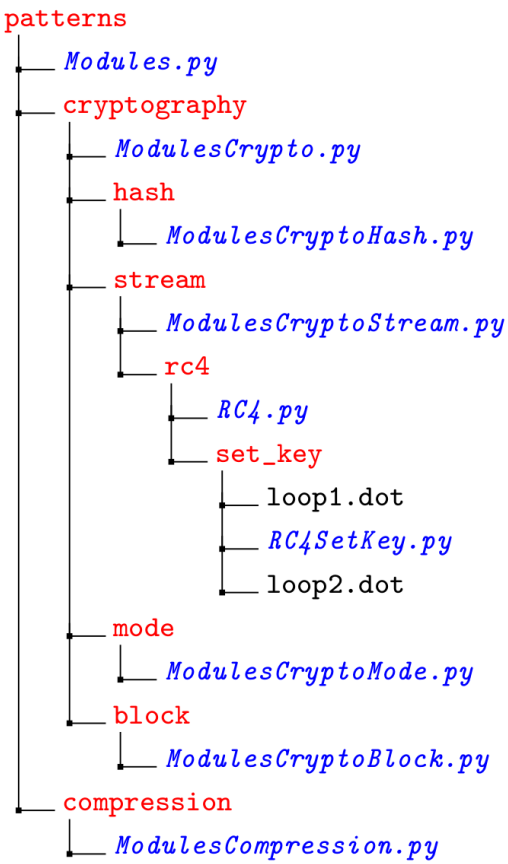


Overlapping

Reject overlapping patterns ?

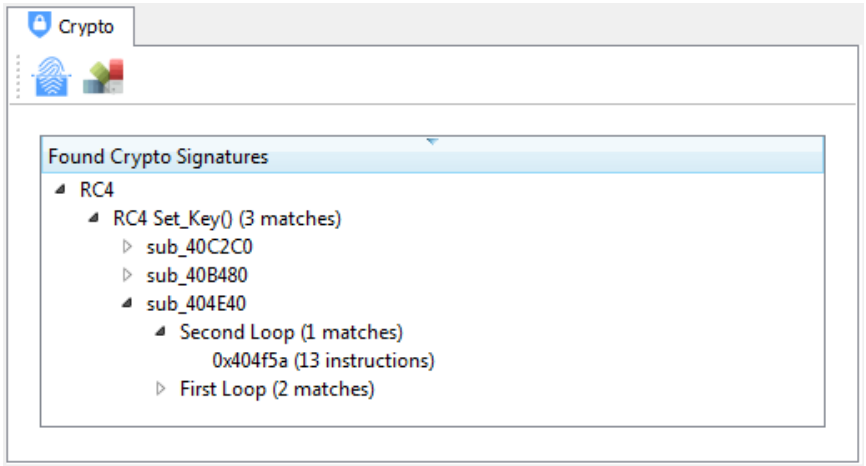


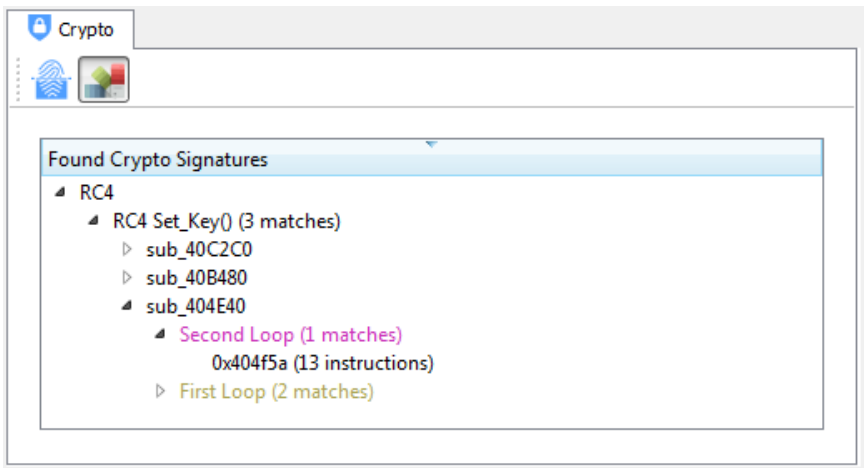
Define new rules: cryptography and more

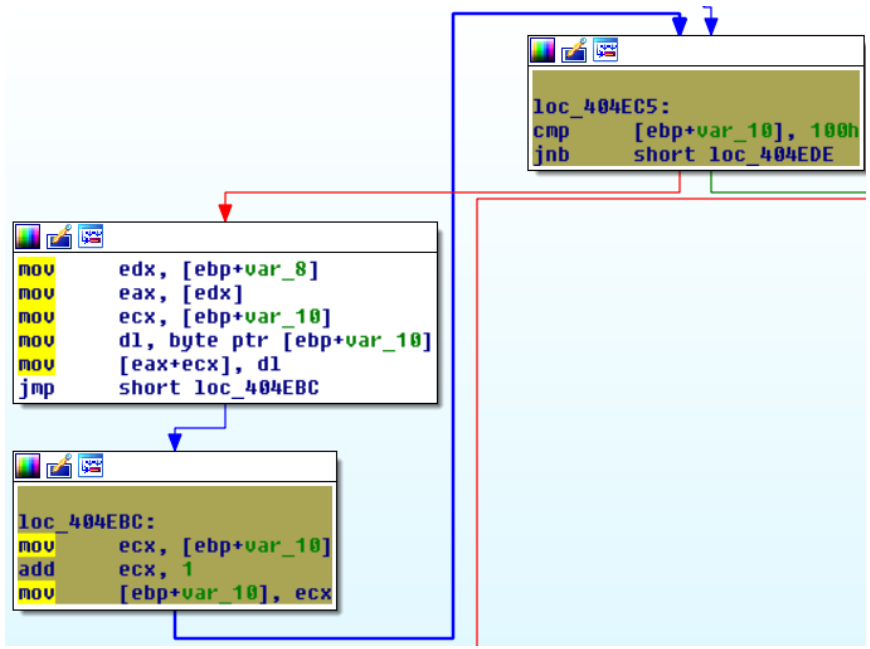



```python
# RC4 set key first loop
loop1 = Pattern(f=ROOT + "/loop1.dot",
                name="First Loop",
                description="First Initialization loop of RC4 set_key.",
                min_pattern=1,
                max_pattern=1)

# RC4 set key second loop
loop2 = Pattern( ... )


RC4_SET_KEY = Patterns(
    patterns=[
        loop1,
        loop2
    ],
    threshold=1.0,
    name="RC4 Set_Key()",
    description="Initialization function of the RC4 algorithm."
)
```

User experience







Demo !

Pattern correlation on RC4

Not efficient on RC4:

- Two small loops, very generic
- Still many false positives
- A few false negatives

Correlation:

- The tooling is appropriate
- Try on other signatures (crypto, packers...)
- Allow correlation to be used in pygrap (not only IDAgrap) ?

Text-based fields: limitation

- **arg1**, **arg2** and **arg3** (string)
- **opcode** (string): mnemonics

opcode is a conditional jump (`jne 0x40129b`) ?

cond="**opcode** **beginswith** j **and** **nchildren** == 2"

IDA vs Capstone

```
sub     dl, 25h
inc     ecx
cmp     ecx, [esp+arg_4]
mov     [eax], dl
jl      short loc_40129B
```

```
sub dl, 0x25
inc ecx
cmp ecx, dword ptr [esp + 8]
mov byte ptr [eax], dl
jl 0x40129b
```

cond="**arg2** **is** 25h **or** **arg2** **is** 0x25"

Text-based fields: perspectives

- **arg1**, **arg2** and **arg3** (string)
- **opcode** (string): mnemonics

Solution: semantics

- Parse instructions with Capstone (or IDA)
- Condition: "**arg1** class REG"
- Condition: "**opcode** class JCC"
- Condition: "int(**arg1**) == 0x25"

→ `grap v2 !`

Repeat: min OR max of instructions

```
digraph any_xor_call {
any [cond=true, minrepeat=1, maxrepeat=4, lazyrepeat=??]
xor [cond="opcode is xor"]
call [cond="opcode is call"]
any -> xor
xor -> call
}
```

Match pattern on `push, push, xor, xor, call` ?

lazyrepeat=true: stop "any" on `xor` (or non basic-block instruction)

- any: `push, push`
- xor: `xor`
- call: `xor`, **no match!**

lazyrepeat=false: stop "any" only on non basic-block instruction

- any: `push, push, xor, xor`
- xor: `call`, **no match!**

Repeat: min OR max of instructions

```
digraph any_xor_call {
any [cond=true, minrepeat=1, maxrepeat=4, lazyrepeat=??]
xor [cond="opcode is xor"]
call [cond="opcode is call"]
any -> xor
xor -> call
}
```

Match pattern on `push, push, xor, xor, call` ?

Solution (v2!): try with repeat=1, 2, 3, 4

- Slower
- Impact on performance ?

Other improvements

Postconditions on basic blocks:

- "1 with 'opcode is xor'"
- "2 with 'opcode is xor' and 1 with "opcode is cmp"'

Children are numbered:

- Allow "childnumber=?"

Meta patterns:

- Pattern rate (3 of 5)
- $P_1 \rightarrow P_2$: one instruction matching for P_1 has a child that matched for P_2

Create patterns with IDA plugin:

- Select nodes within IDA
- Export DOT pattern file

Conclusion

Conclusion

Standalone tool, python bindings and IDA plugin:

- Patterns are easy to write and to understand
- Useful for detection and automatic analysis (Backspace)
- Open source (MIT License):
  `https://bitbucket.org/cybertools/grap`

Perspectives:

- Add pattern features (semantics, basic block...)
- Pattern creation within IDA plugin
- Write patterns for crypto algorithms and packers

Thank you !

`https://bitbucket.org/cybertools/grap`

Aurélien Thierry (@yaps8)
Jonathan Thieuleux (@coldshell)

AIRBUS

STORMSHIELD