Code Obfuscation against Static and Dynamic Reverse Engineering

Sebastian Schrittwieser
Stefan Katzenbeisser

Information Hiding 2011, Prague

Agenda

- Software Protection
- Static vs. Dynamic Code Analysis
- Obfuscation against Static Analysis
  - Branching Function
- Obfuscation against Dynamic Analysis
  - Control Flow Diversification
  - Path Signature
  - Code Block Diversification
- Evaluation
- Conclusion

Software Protection

- Today, software is usually distributed in binary form
- Reverse engineering aims at restoring a higher-level representation of software in order to analyze its structure and behavior
- In some applications there is a need to protect software against reverse engineering:
  - Intellectual property (e.g. proprietary algorithms) contained in software
  - confidentiality reasons
  - copy protection mechanisms

Reverse Engineering

	Static	**Dynamic**
Approach	analyzing code without actually executing it	analyzing code during execution
Pro	fast, automated, analyzes entire code	allows deeper understanding of the program’s behavior
Con	difficult to rebuild control flow (e.g. follow conditional jumps)	slow, mostly done by humans, only one trace at a time

Approach

- Prevent static code analysis
- Shift attacker’s effort to dynamic analysis
  - more time consuming
  - less tool support
  - difficult to automate
- Make dynamic analysis more time consuming

Branching Function

- First introduced by Linn and Debray[1]
- Idea: Replace CALL and JMP instructions with calls to a generic function, which decides at runtime where to jump
- For a static analyzer it is difficult to calculate jump target without executing the software
- Problem: Large code sections between calls allow local analysis

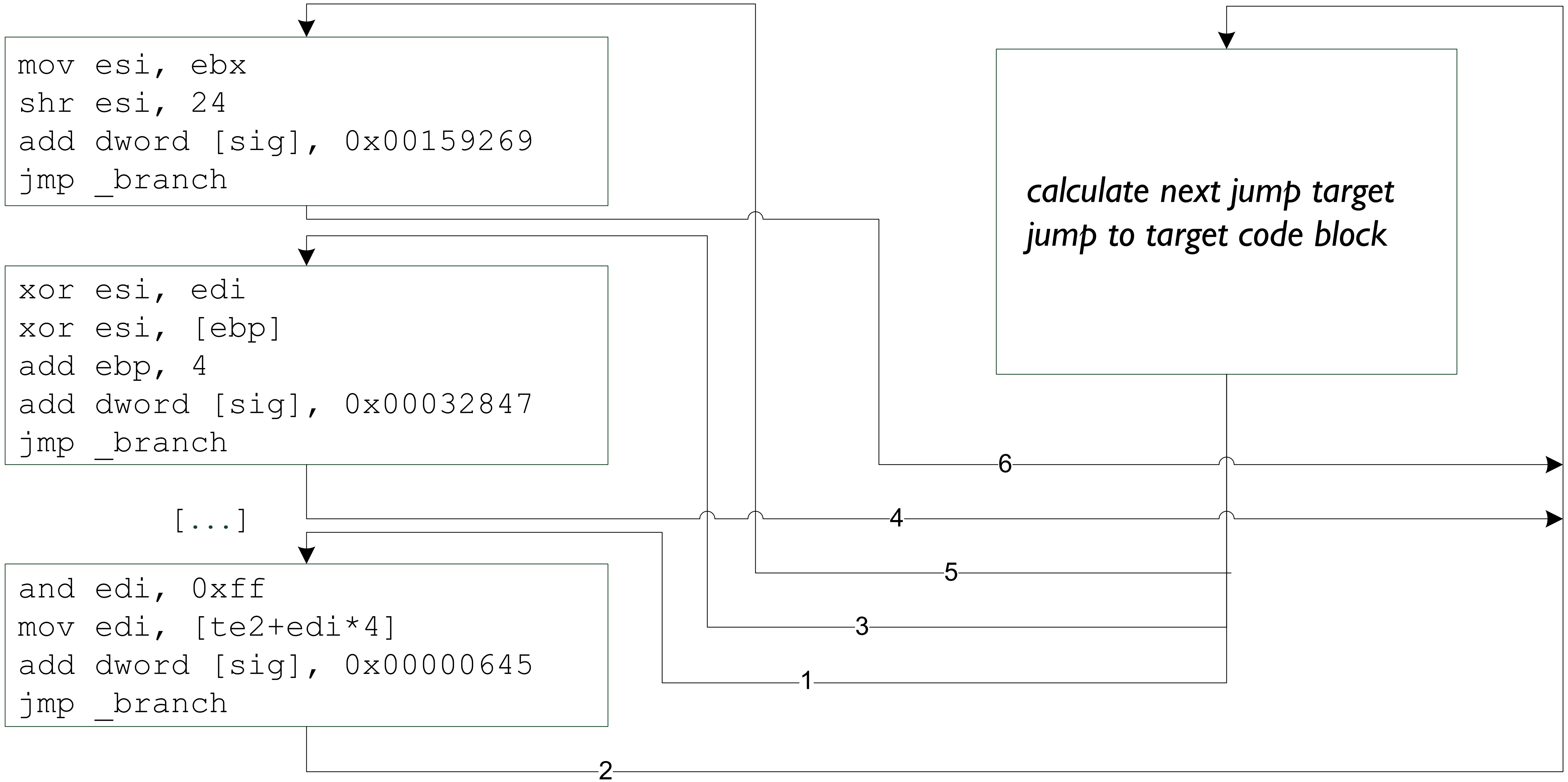
[1] C. Linn and S. Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. CCS 2003

Code Splitting

- Code is split into small blocks (gadgets)
- Branching function
- The calculation of the next jump target depends on all predecessors of the current block
- Static analysis of a code block reveals only limited local information
- Difficult to obtain a complete view of the software

Code Blocks

Branching Function



```
mov esi, ebx
shr esi, 24
add dword [sig], 0x00159269
jmp _branch
```

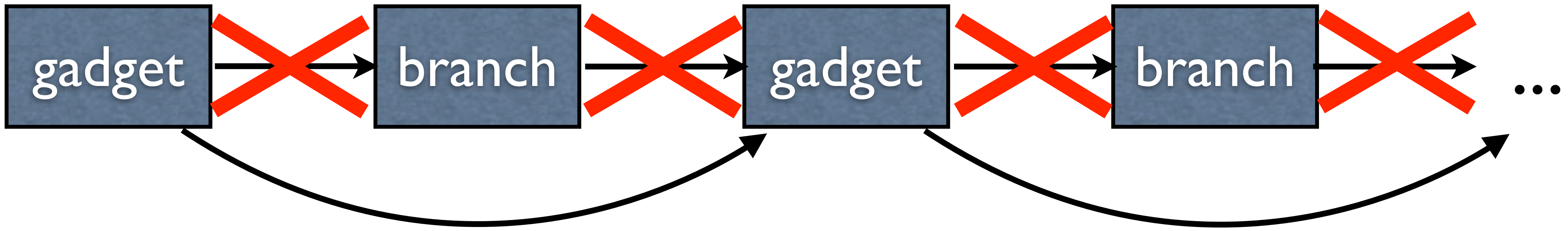
```
xor esi, edi
xor esi, [ebp]
add ebp, 4
add dword [sig], 0x00032847
jmp _branch
```

[...]

```
and edi, 0xff
mov edi, [te2+edi*4]
add dword [sig], 0x00000645
jmp _branch
```

Dynamic Analysis

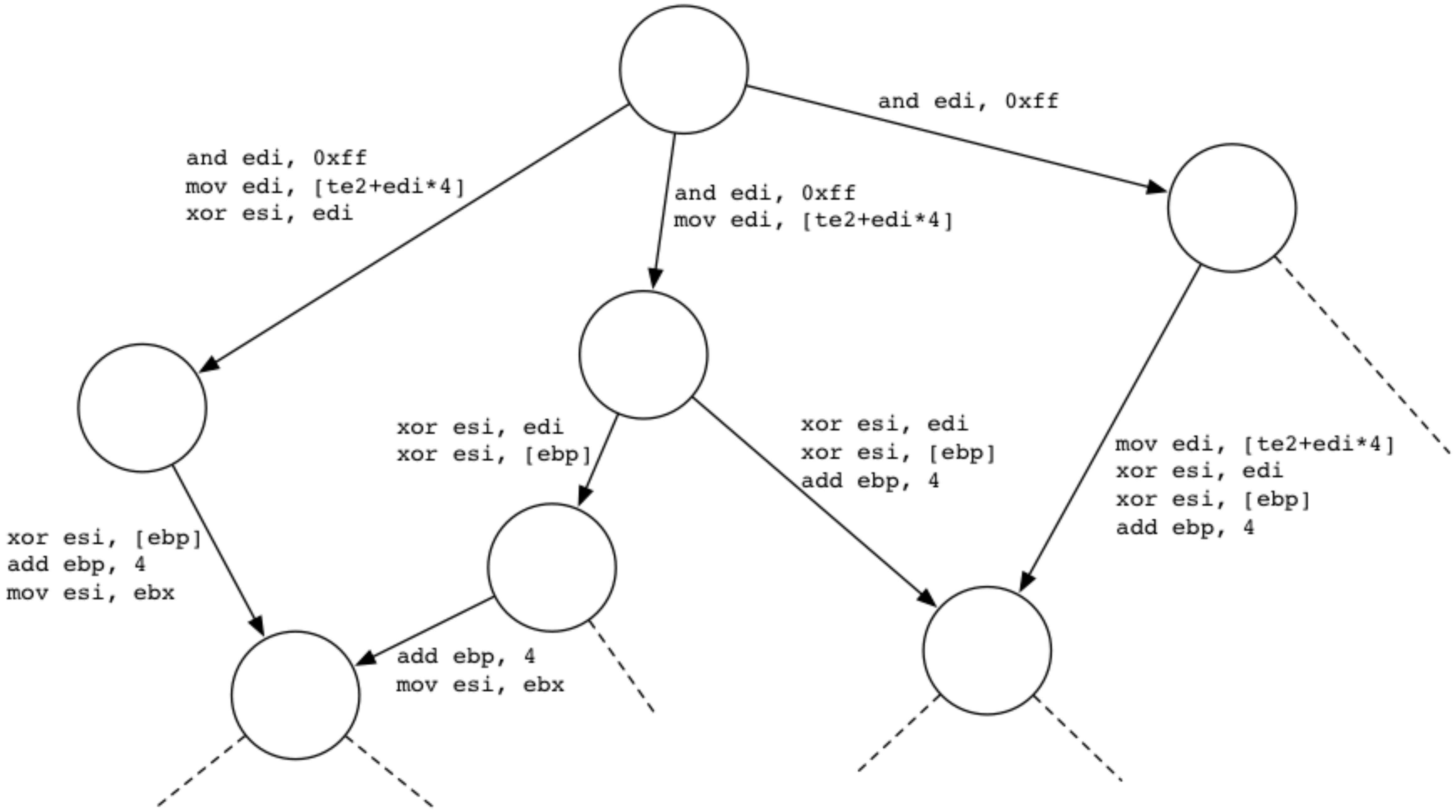
- Problem: Dynamic analysis reveals all code blocks used in a single invocation of the software as well as their order.
- Easy to remove the jumps to the branching function by just concatenating called gadgets in their correct order.

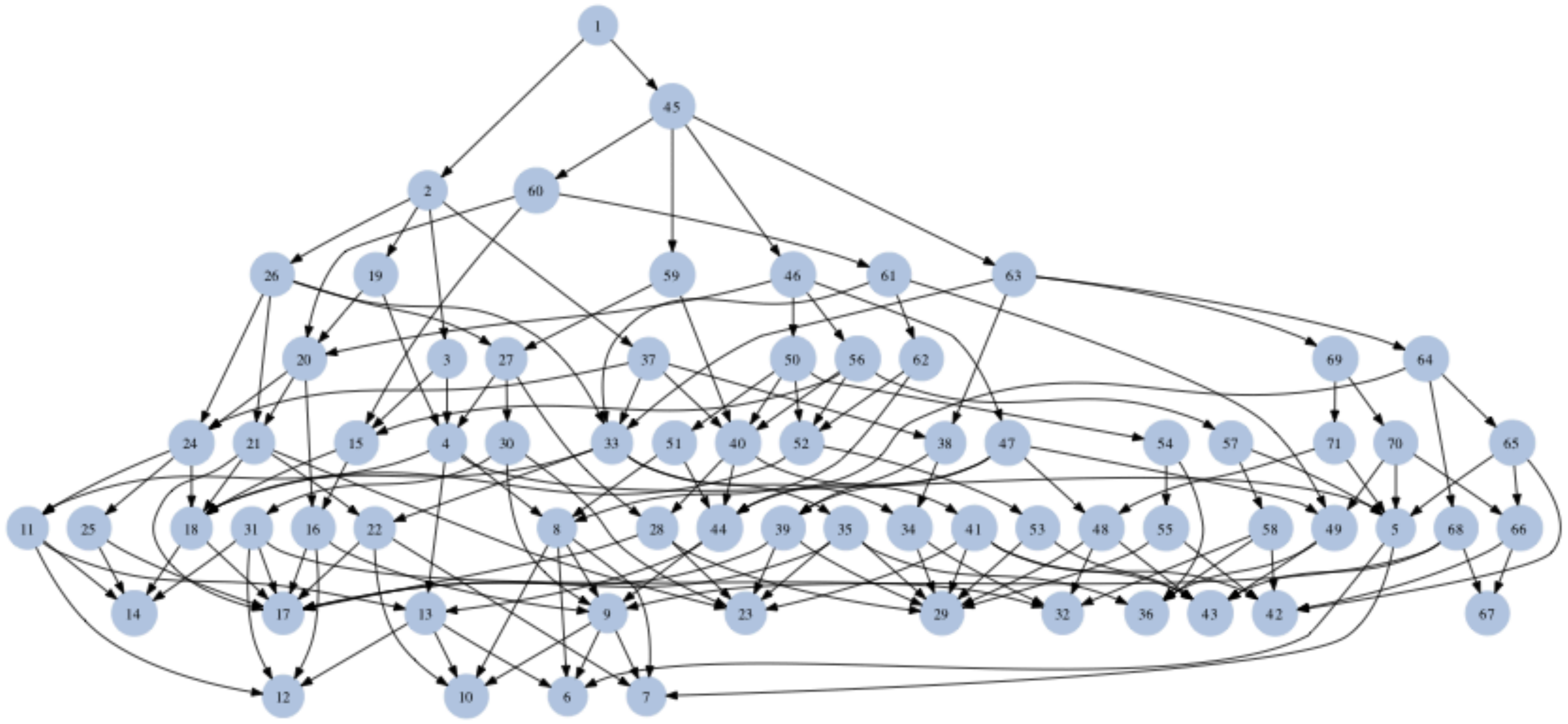


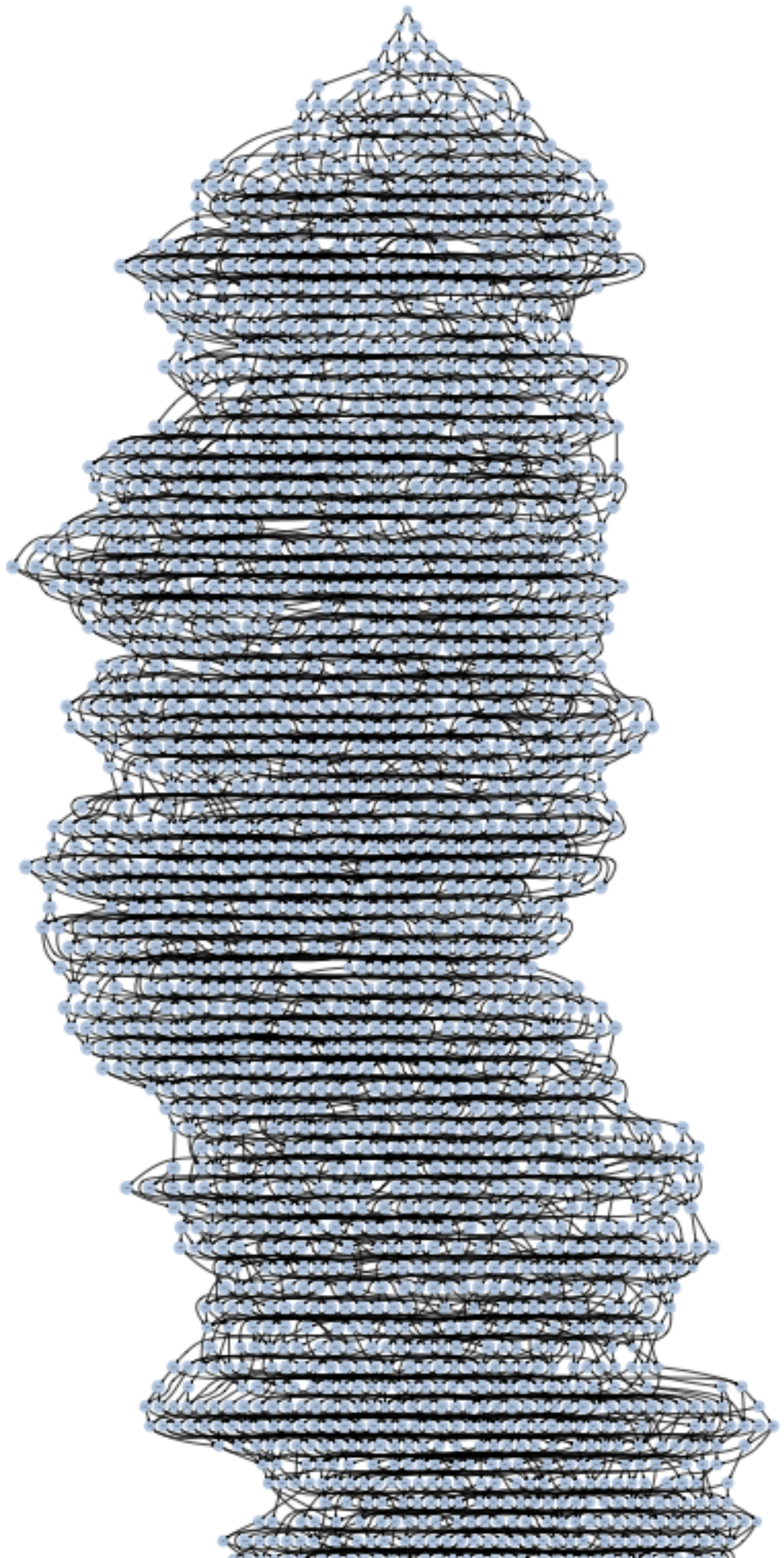
- Idea: control flow diversification

Control Flow Diversification

- Applying the concept of software diversification to the control flow graph of a program
- Each copy contains exactly the same code
- Control flow depends on the program's input data

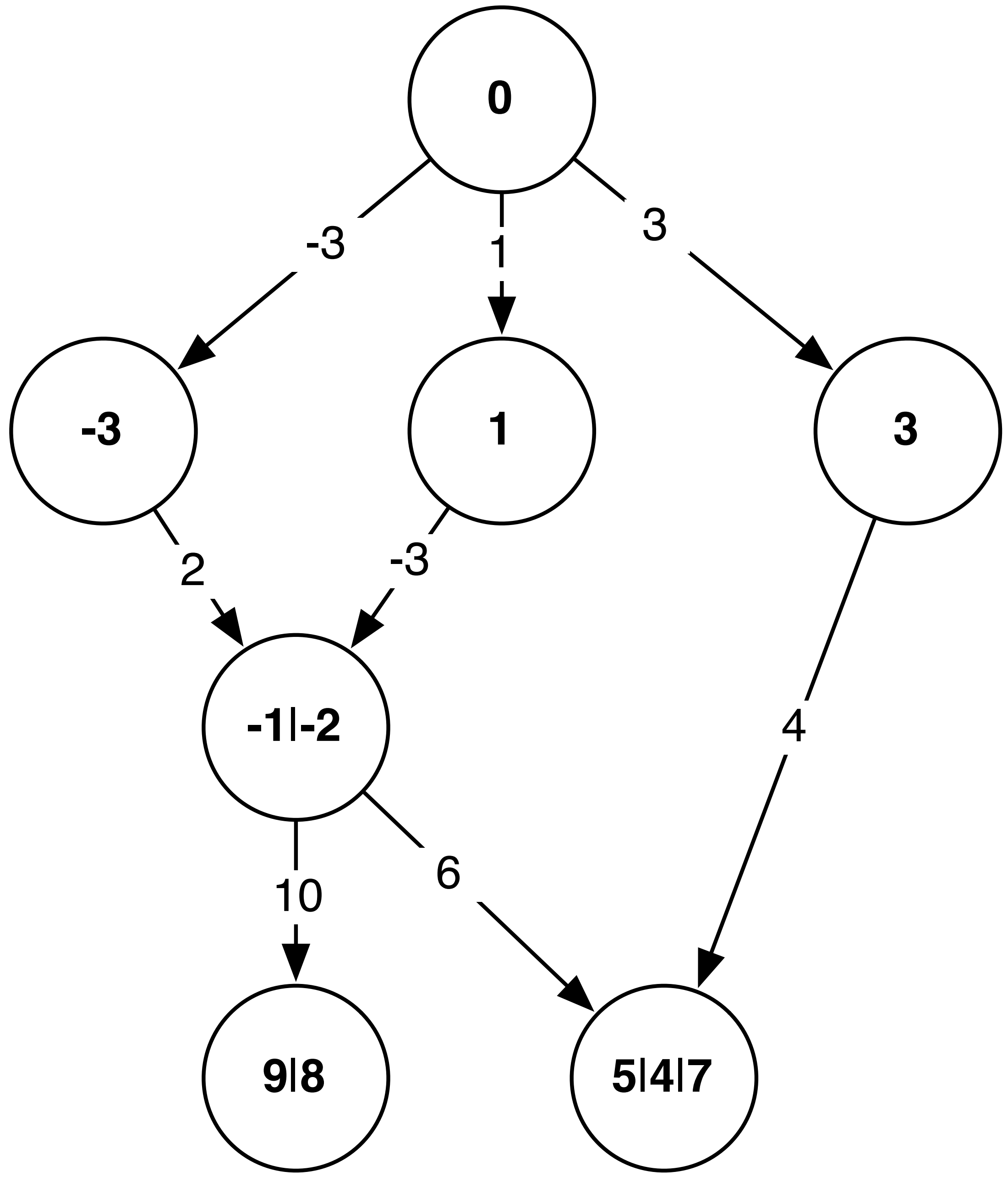

and edi, 0xff
and edi, 0xff
mov edi, [te2+edi*4]
xor esi, edi
and edi, 0xff
mov edi, [te2+edi*4]
xor esi, edi
xor esi, [ebp]
xor esi, edi
xor esi, [ebp]
add ebp, 4
mov edi, [te2+edi*4]
xor esi, edi
xor esi, [ebp]
add ebp, 4
xor esi, [ebp]
add ebp, 4
mov esi, ebx
add ebp, 4
mov esi, ebx


1
45
2
60
26
19
59
46
61
63
20
3
27
37
50
56
62
69
64
24
21
15
4
30
33
51
40
52
38
47
54
57
71
70
65
11
25
18
31
16
22
8
28
44
39
35
34
41
53
48
55
58
49
5
68
66
14
17
13
9
23
29
32
36
43
42
67
12
10
6
7



Path Signature

- The path signature uniquely identifies a gadget and all its predecessors
- The branching function decides, based on the path signature and the program's input, where to jump next
- Graph representation: Lookup table



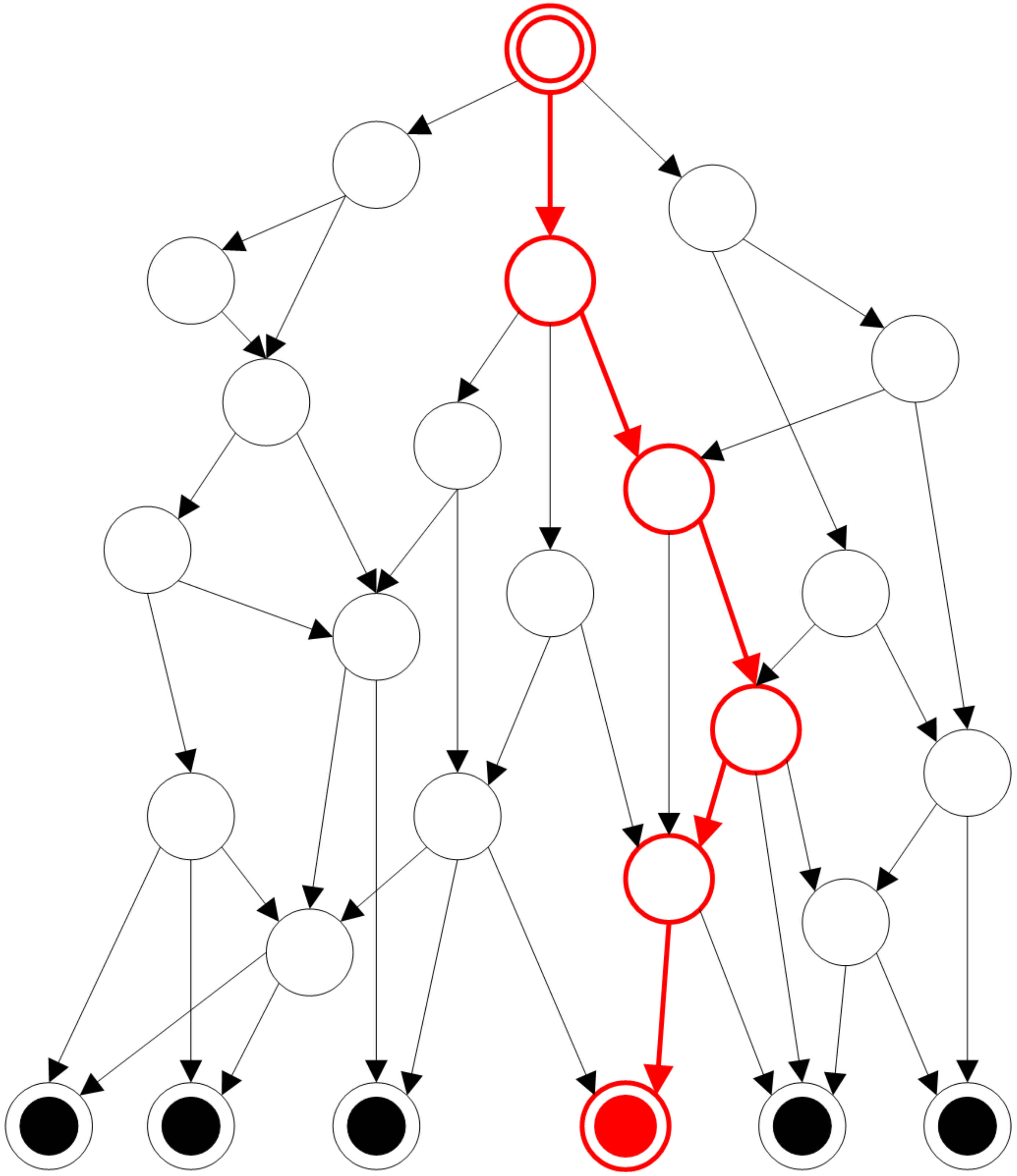
Gadget Diversification

- All paths through the graph are valid and semantically equal traces of the program
- Gadget Diversification: one specific path yields correct computation only for a specific input of the program

```
xor esi, [ebp]
add ebp, 4
add ebx, 4
mov eax, [esp+4]
jmp _branch
```

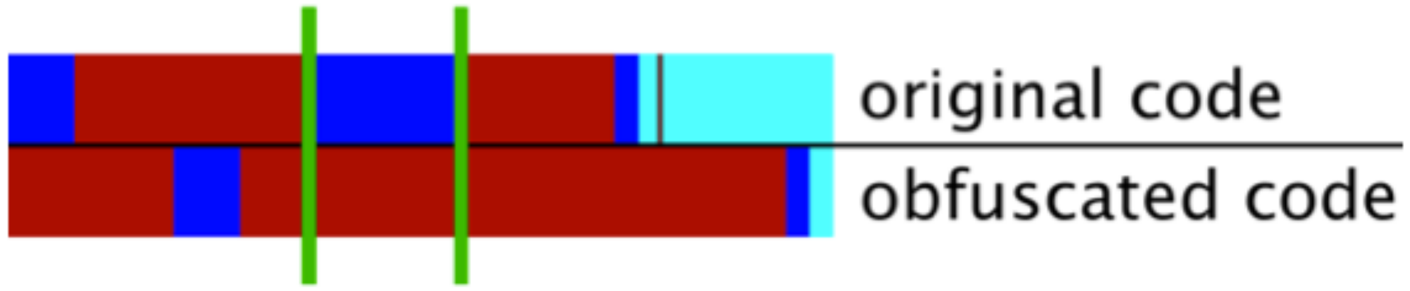
$\overset{\mathcal{T}}{\Rightarrow}$

```
xor esi, [ebp]
sub ebp, eax
add ebp, 12
add eax, 5
add ebx, 2
mov dword [0x0040EA00], ebx
add ebx, 2
mov eax, [esp+4]
jmp _branch
```



Evaluation

- No provable security
- Two state-of-the-art reverse engineering tools (IDA Pro & Jakstab) for evaluation of the static part



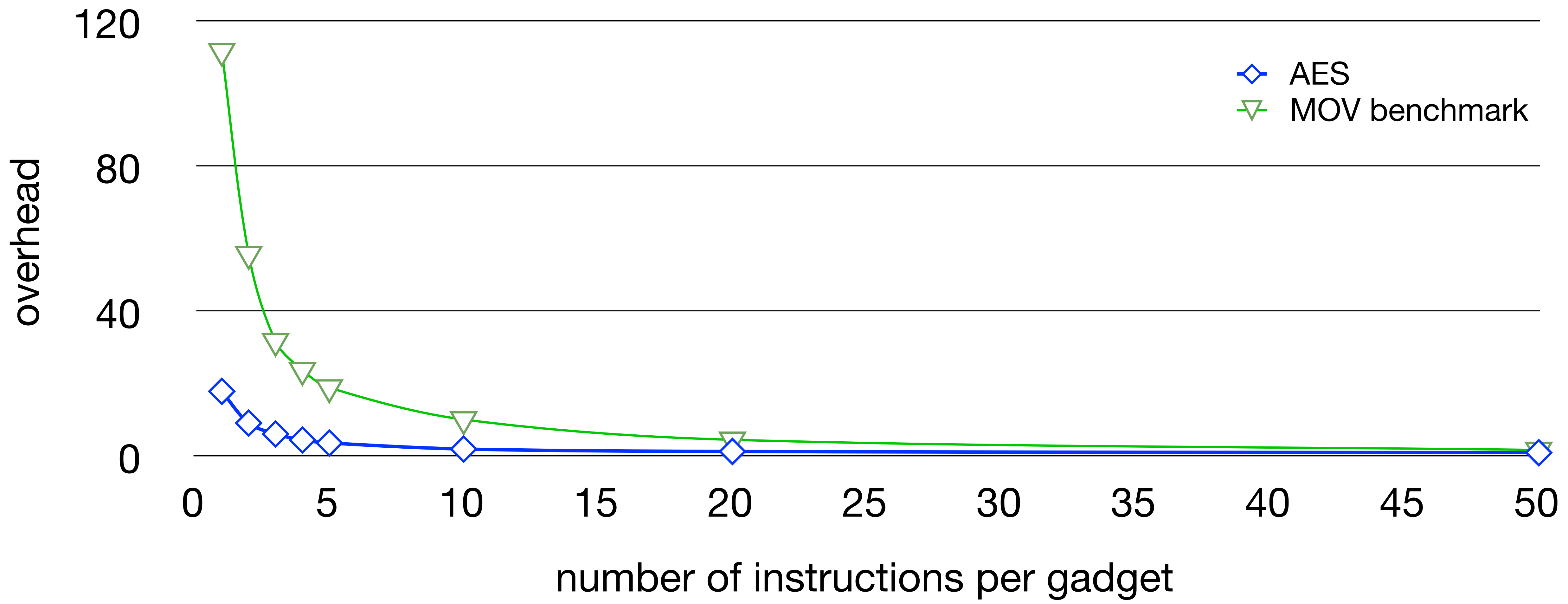
- Collberg's classification for the dynamic part
  - Resilience: *strong*
  - Potency: *high*

Information Gap

- Aim: increasing the information gap between developer and attacker
- Obfuscated software does not contain an explicit representation of the graph's structure
- Attacker's perspective:
  - Reconstruct the entire graph
  - Remove diversity of a single trace

Performance

- Heavily depends on code block size



Conclusion

- Novel code obfuscation method, based on control flow diversification
- By splitting code in to small portions, local analysis can only reveal very limited local information of the program
- Future work: inter-gadget diversification

Tack Vielen Dank

Obrigado

Merci

ありがとうございます

Bedankt Takk

感謝您

Terima Kasih

谢谢

Grazie

ขอบคุณ

Спасибо

Thank You

Kiitos Tak

Teşekkür Ederiz 감사합니다

Gracias

Dziękujemy Σας ευχαριστούμε