SIGMA: A Semantic Integrated Graph Matching Approach For Identifying Reused Functions In Binary Code

By

Saed Alrabaee, Paria Shirani, Lingyu Wang and Mourad Debbabi

Presented At

The Digital Forensic Research Conference

DFRWS 2015 EU Dublin, Ireland (Mar 23rd- 26th)

DFRWS is dedicated to the sharing of knowledge and ideas about digital forensics research. Ever since it organized the first open workshop devoted to digital forensics in 2001, DFRWS continues to bring academics and practitioners together in an informal environment. As a non-profit, volunteer organization, DFRWS sponsors technical working groups, annual conferences and challenges to help drive the direction of research and development.

http:/dfrws.org

SIGMA:
A Semantic Integrated Graph Matching Approach for Identifying Reused Functions in Binary Code

Paria Shirani
Saed Alrabaee, Lingyu Wang, and Mourad Debbabi

UNIVERSITÉ Concordia UNIVERSITY

DFRWS

Outline

- Introduction
- Binary Representations
- SIGMA
- Case Study & Experimental Results
- Concluding Remarks

Introduction

- Why SIGMA?
  - SIGMA: Semantic Integrated Graph Matching Approach
  - A technique for identifying reused functions in binary code (recognizing reused functions are needed in different fields such as Malware Analysis, Copyright Infringement, Digital Forensics, etc.)
  - To improve the efficiency of reverse engineering
- How SIGMA works?
  - SIGMA enhances and merges several existing concepts from classic program analysis into a new graph called SIG
  - SIGMA applies exact and inexact graph matching to identify reused functions

Contributions

- Introduce a new representation (SIG) of binary code
  - To unify various semantic information
  - To facilitate more efficient graph matching
- Define different types of traces over SIG graphs such as
  - normal traces
  - AND-traces
  - OR-traces
- Demonstrate the effectiveness of our approach through a case study using two known malware

Outline

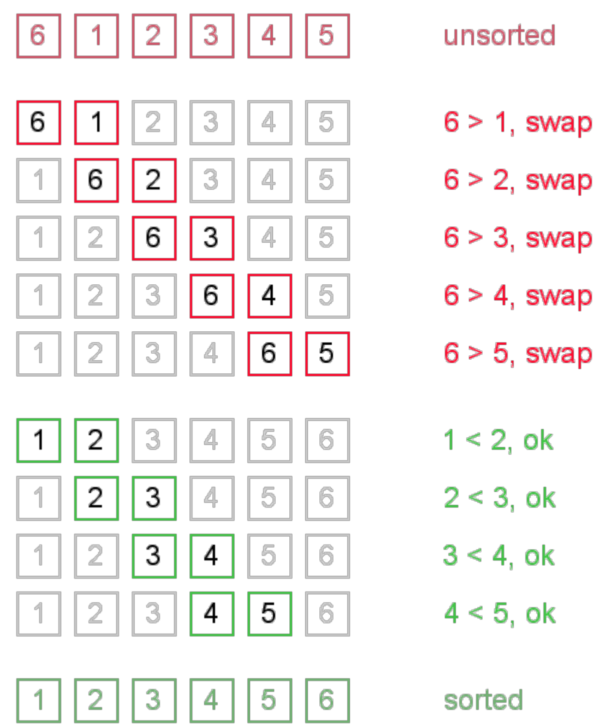
- Introduction
- Binary Representations
- SIGMA
- Case Study & Experimental Results
- Concluding Remarks

Binary Representation

- Take Bubble Sort as an example

```
procedure bubbleSort( A : list of sortable items )
  n = length(A)
  repeat
    swapped = false
    for i = 1 to n-1 inclusive do
      /* if this pair is out of order */
      if A[i-1] > A[i] then
        /* swap them and remember something changed */
        swap( A[i-1], A[i] )
        swapped = true
      end if
    end for
  until not swapped
end procedure
```

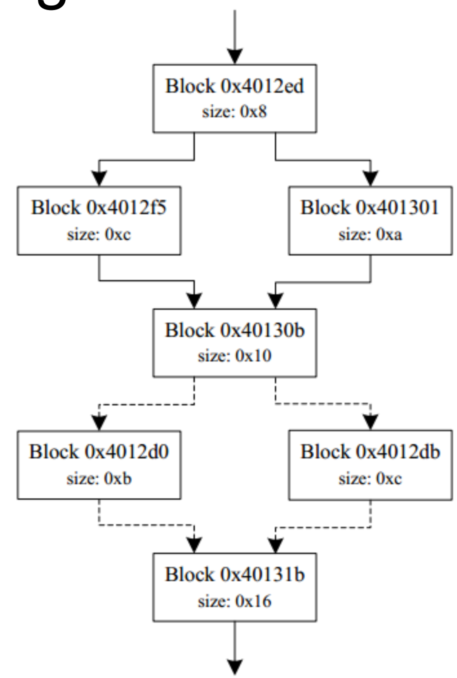
```
80483c4:   push   ebp
80483c5:   mov    ebp, esp
80483c7:   sub    esp, 0x10
80483ca:   movl   -0x4(ebp), 0x1
80483d1:   jmp    804844b
80483d3:   movl   -0x4(ebp), 0x0
80483da:   movl   -0x8(ebp), 0x1
80483e1:   jmp    8048443
80483e3:   mov    eax, -0x8(ebp)
...
```



Binary Representation (Cont'd)

- CFG (Control Flow Graph)
  - Represents the structure of the binaries
  - Consists of basic blocks and edges

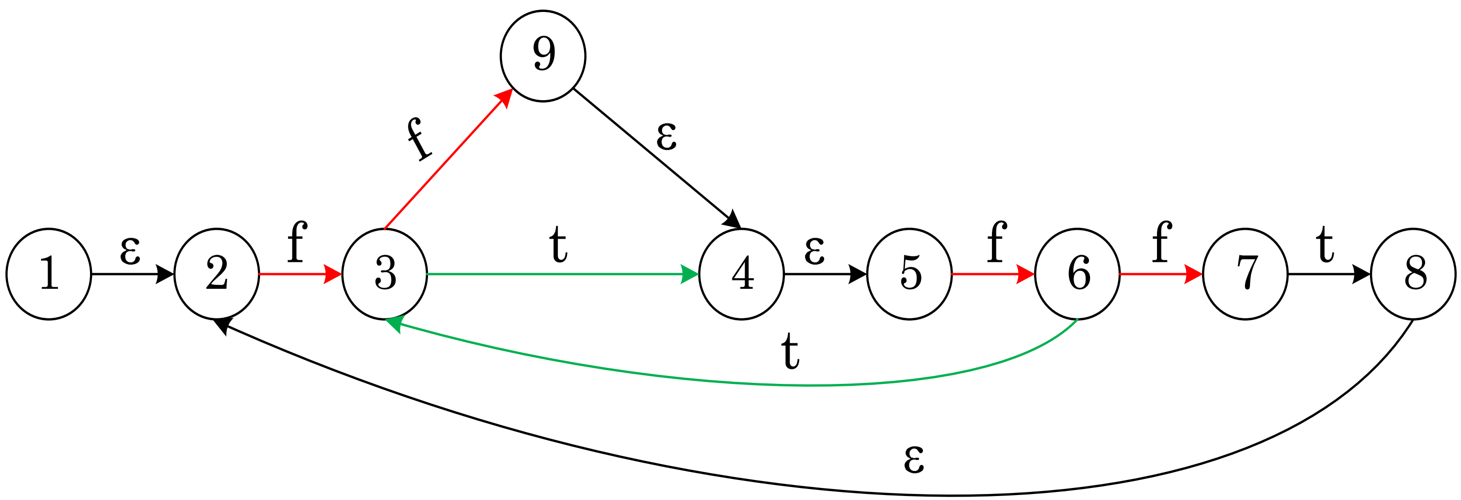
```
...
0x004012d0:  push   ebp
0x004012d1:  mov    ebp, esp
0x004012d3:  mov    eax, DWORD PTR [ebp+12]
0x004012d6:  add    eax, DWORD PTR [ebp+8]
0x004012d9:  pop    ebp
0x004012da:  ret
0x004012db:  push   ebp
0x004012dc:  mov    ebp, esp
0x004012de:  mov    eax, DWORD PTR [ebp+8]
0x004012e1:  imul   eax, DWORD PTR [ebp+12]
0x004012e5:  pop    ebp
0x004012e6:  ret
...
0x004012ed:  mov    eax, DWORD PTR [ebp+8]
0x004012f0:  cmp    eax, DWORD PTR [ebp+12]
0x004012f3:  jge    0x401301
0x004012f5:  mov    ds:0x403020, 0x4012d0
0x004012ff:  jmp    0x40130b
0x00401301:  mov    ds:0x403020, 0x4012db
0x0040130b:  sub    esp, 0x8
0x0040130e:  push   DWORD PTR [ebp+12]
0x00401311:  push   DWORD PTR [ebp+8]
0x00401314:  mov    eax, ds:0x403020
0x00401319:  call   eax
0x0040131b:  ...
```



Binary Representation (Cont'd)

- CFG of bubble sort
  - True (t)
  - False (f)
  - No condition (ϵ)

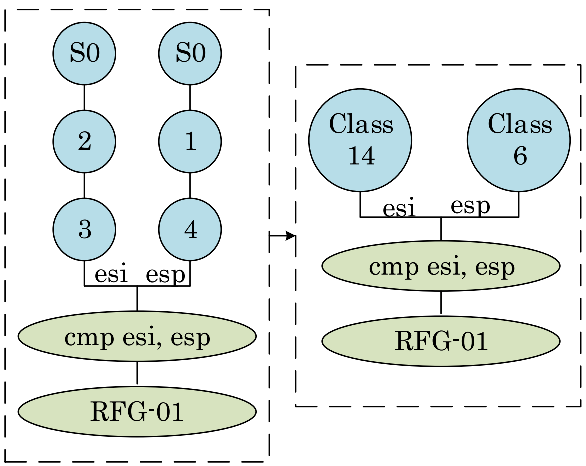
Two different functions may have the same CFG



Binary Representation (Cont'd)

- RFG (Register Flow Graph)

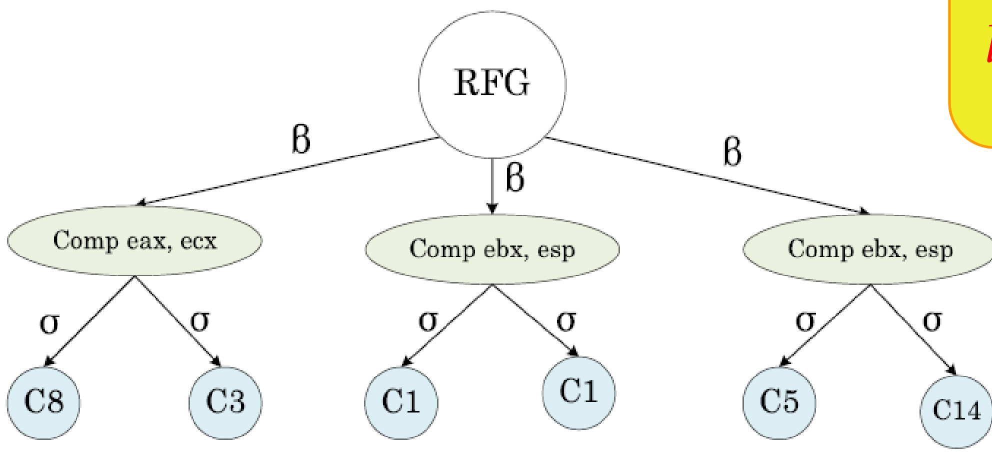
```
1.  sub     esp, 0D8h
2.  push    esi
3.  mov     esi, esp
4.  add     esp, 8
5.  cmp     esi, esp
```



Class	Arithmetic	Logical	Generic	Stack
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1
5	1	1	0	0
6	1	0	1	0
7	1	0	0	1
8	1	1	1	0
9	1	1	0	1
10	1	0	1	0
11	1	1	1	1
12	0	1	1	0
13	0	1	0	1
14	0	0	1	1
15	0	1	1	1

Binary Representation (Cont'd)

- RFG of bubble sort

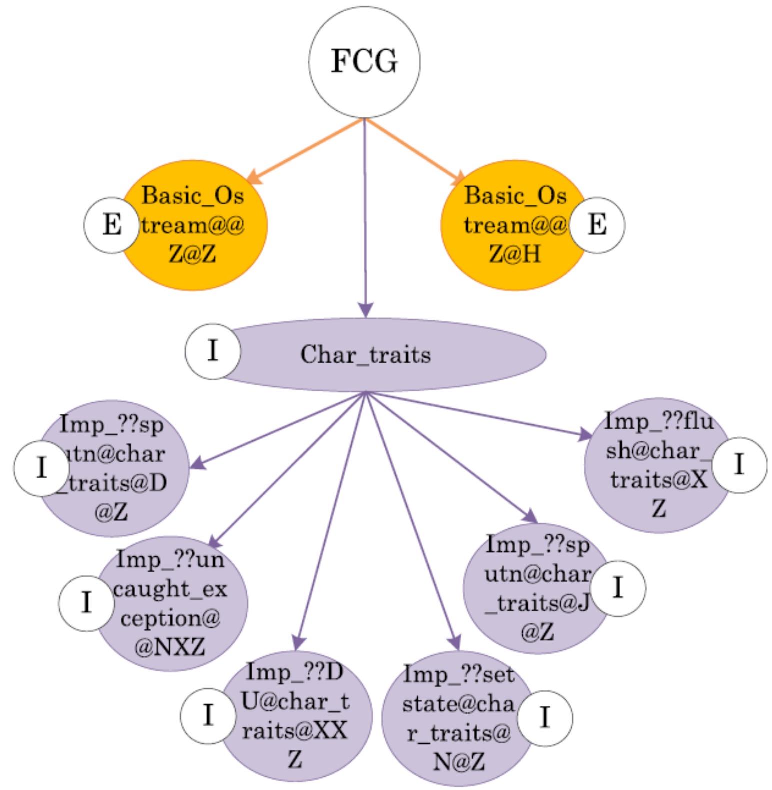


Some operands such as 'constants' and 'memory locations' are not considered as well as 'test' instruction

Binary Representation (Cont'd)

- FCG (Function Call Graph) of bubble sort
  - I : Internal call
  - E: External call

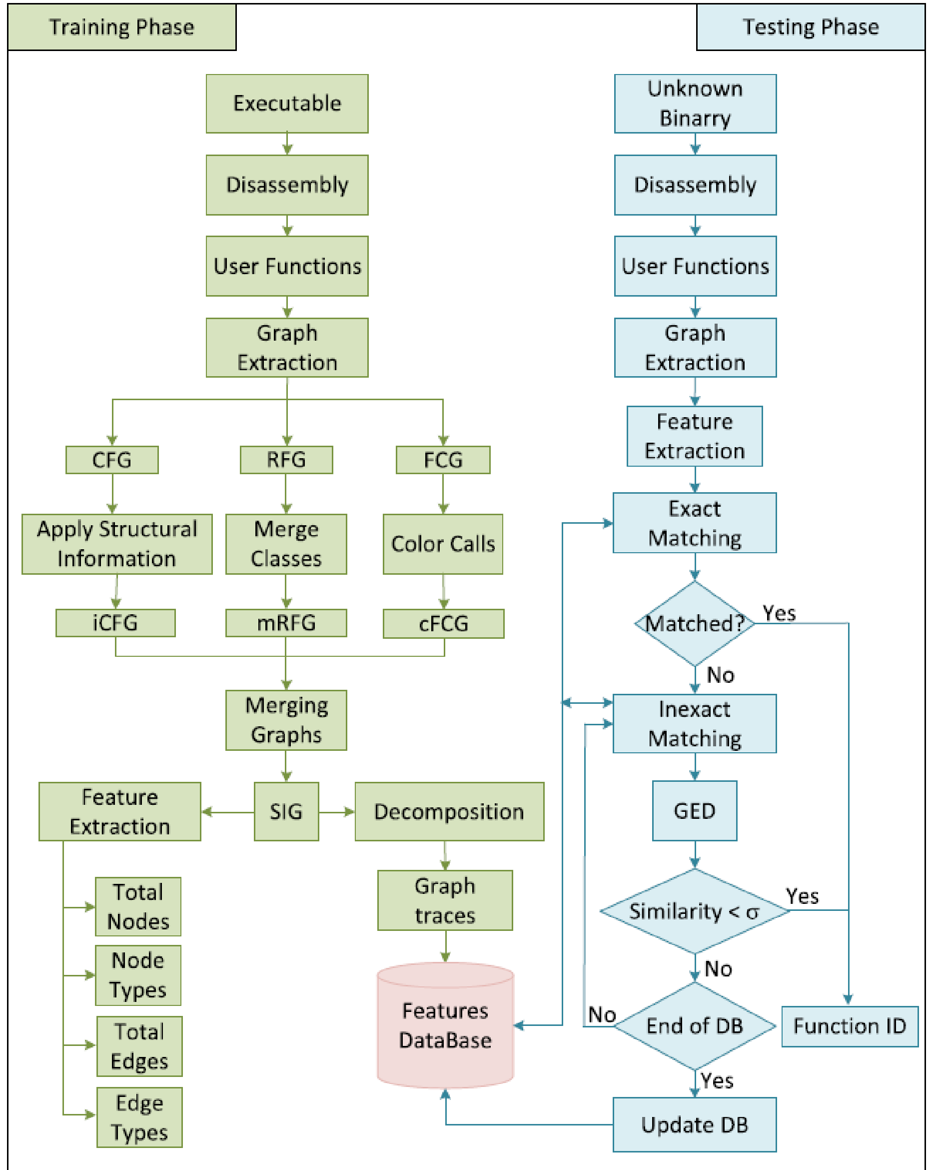
Internal calls are not comparable across different programs



Outline

- Introduction
- Binary Representations
- SIGMA
- Experimental Results & Case studies
- Concluding Remarks

SIGMA Architecture



Concordia University

SIG Components

- *i*CFG
  - Applying structural information to color the nodes

Category	Description
Data Transfer (DT)	Data transfer instructions such as mov, movzx, movsx
Test(T)	Test instructions such as cmp, test
ArLo	Arithmetic and logical instructions such as add, sub, mul, div, imul, idiv, and, or, xor, sar, shr
CaLe	System call, API call, and Load effective instructions such as lea
Stack	Stack instructions such as push, pop

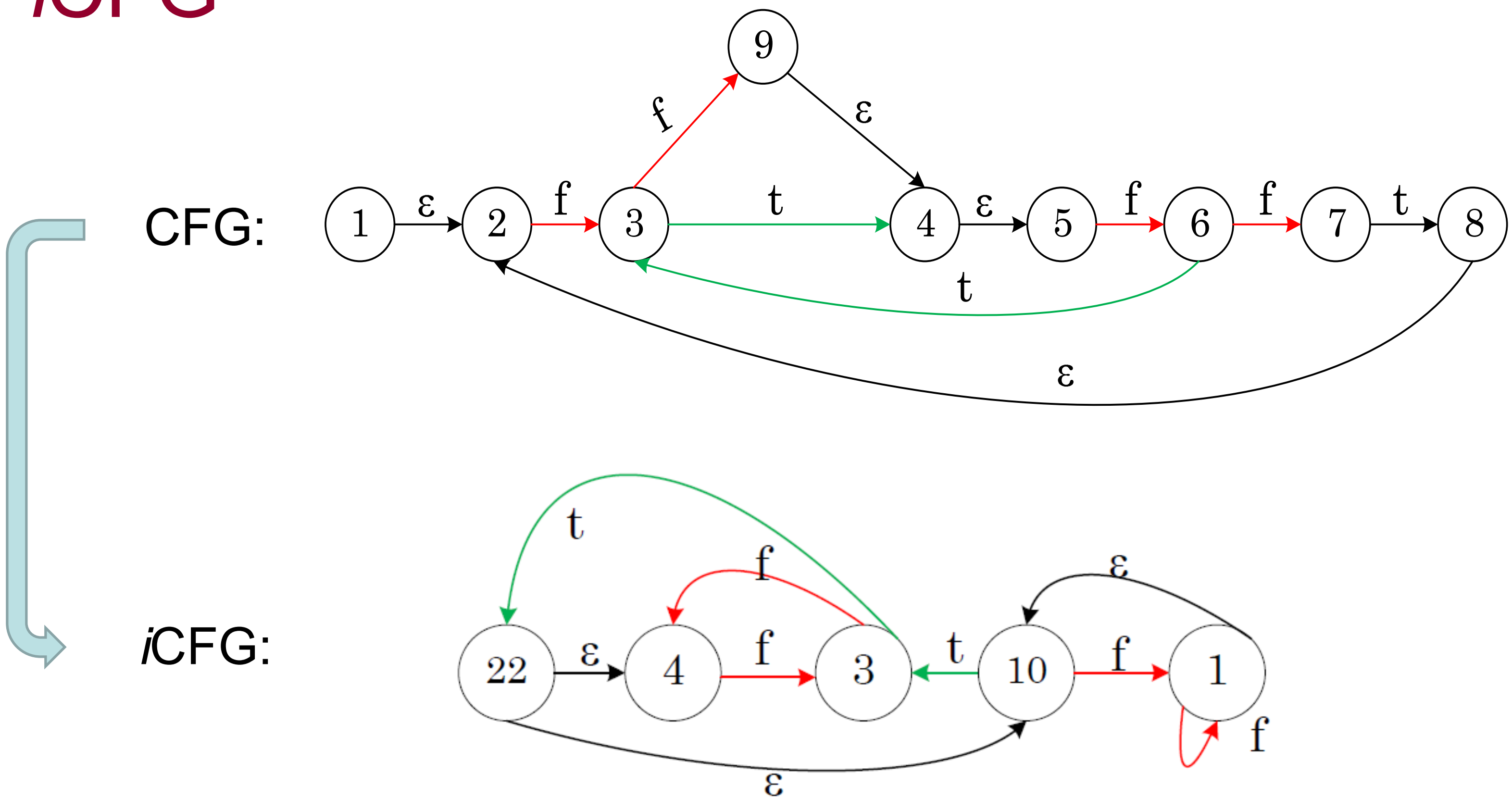
SIG Components (Cont'd)

- *i*CFG
  - Considering the two highest and lowest amounts

Color Classes	Majority	Minority
1/2/3	DT, T	ArLo/Stack/CaLe
4/5/6	DT, ArLo	T/CaLe/Stack
7/8/9	DT, CaLe	ArLo/Stack/T
10/11/12	DT, Stack	T/CaLe/ArLo
13/14/15	T, ArLo	DT/CaLe/Stack
16/17/18	T, CaLe	DT/ArLo/Stack
19/20/21	T, Stack	DT/ArLo/CaLe
22/23/24	ArLo, Stack	T/DT/CaLe
25/26/27	ArLo, CaLe	Stack/DT/T
28/29/30	Stack, Cale	T/DT/ArLo

SIG Components (Cont'd)

- *i*CFG



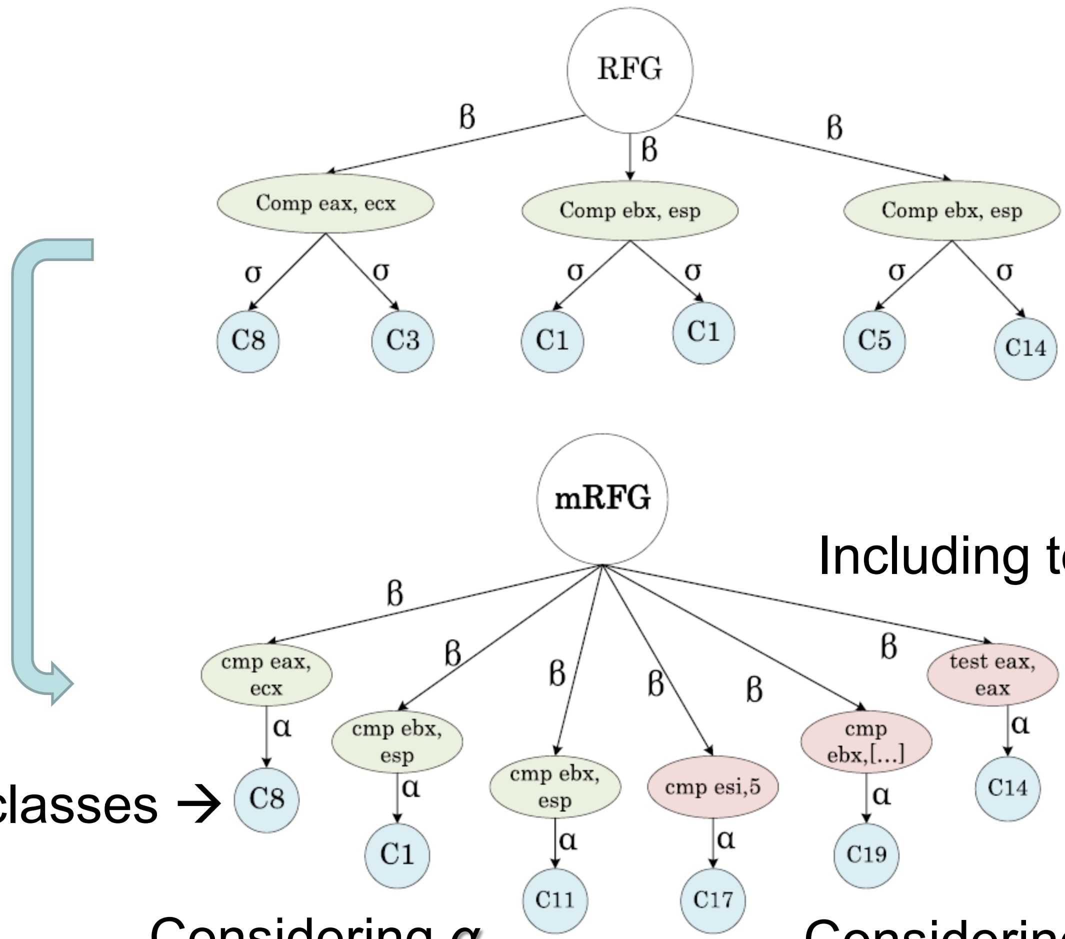
SIG Components (Cont'd)

- *m*RFG
  - Include *constants (C)*, and *memory locations (ML)*
  - Include *test* instructions
  - Merge classes
  - Consider *α* (cost for instruction counts)

Class	Arithmetic	Logical	Generic	Stack	C C	C Reg	ML ML	ML Reg	ML C
1	1	0	0	0	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0
3	0	0	1	0	0	0	0	0	0
4	0	0	0	1	0	0	0	0	0
5	1	1	0	0	0	0	0	0	0
6	1	0	1	0	0	0	0	0	0
7	1	0	0	1	0	0	0	0	0
8	1	1	1	0	0	0	0	0	0
9	1	1	0	1	0	0	0	0	0
10	1	0	1	0	0	0	0	0	0
11	1	1	1	1	0	0	0	0	0
12	0	1	1	0	0	0	0	0	0
13	0	1	0	1	0	0	0	0	0
14	0	0	1	1	0	0	0	0	0
15	0	1	1	1	0	0	0	0	0
16	0	0	0	0	1	0	0	0	0
17	0	0	0	0	0	1	0	0	0
18	0	0	0	0	0	0	1	0	0
19	0	0	0	0	0	0	0	1	0
20	0	0	0	0	0	0	0	0	1

SIG Components (Cont'd)

- *m*RFG



Including test instructions

Merging classes →

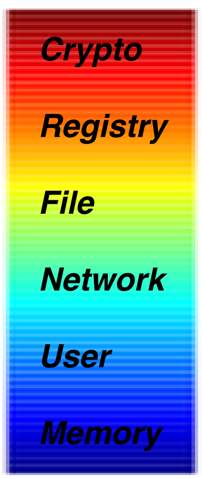
Considering *α*

Considering Cs and MLs

SIG Components (Cont'd)

- *c*FCG
  - C system calls

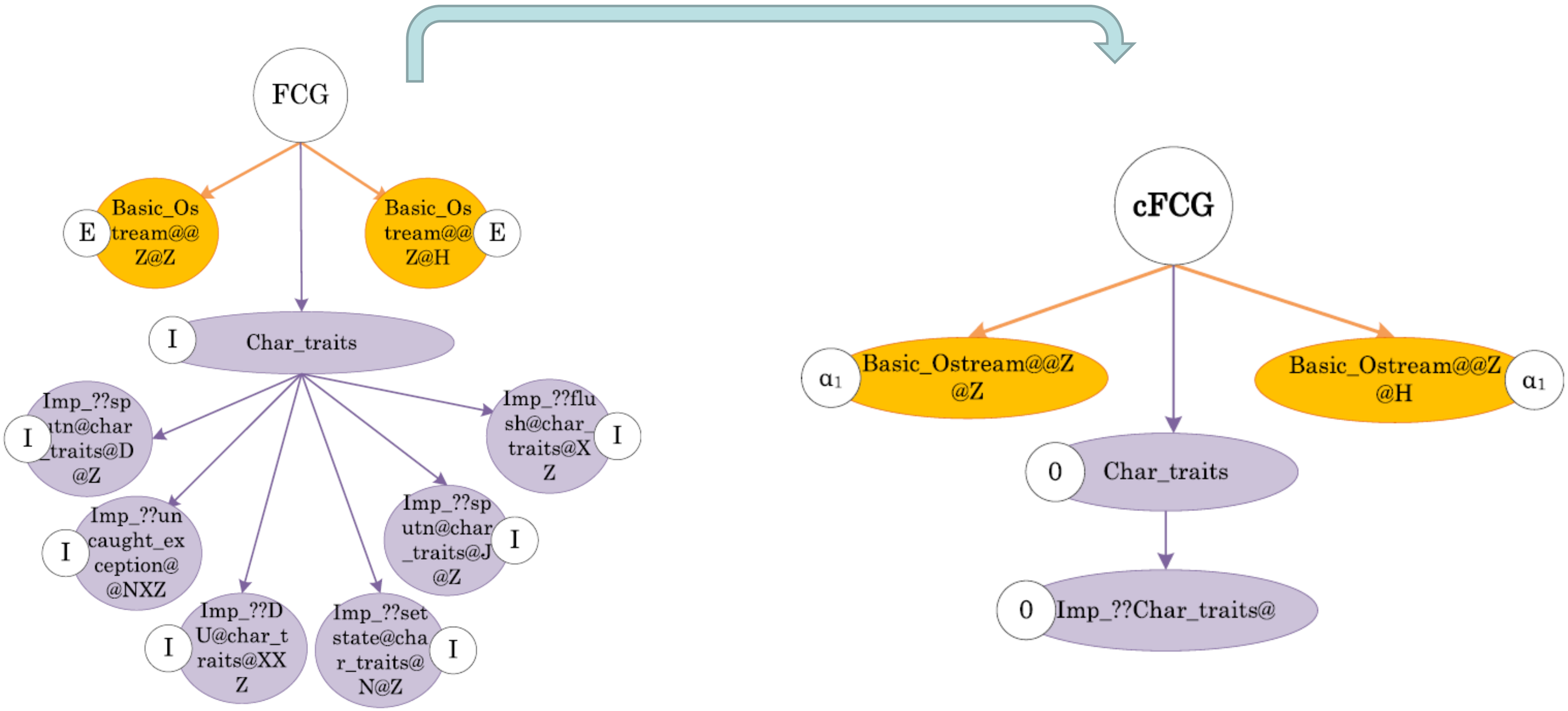
$$f(c) = \begin{cases} \alpha = 0 & \text{if } c \text{ is internal system call} \\ 0 < \alpha < 1 & \text{if } c \text{ is external system call} \end{cases}$$



Category	API Functions
File	CreateFileMapping, GetFileAttributes, ReplaceFile
Network	gethostbyname, getaddrinfo, recv, WSAAccept
Registry	RegCreateKey, RegQueryValue, SHRegSetPath
Crypto	CryptGenKey, CryptSetKey, CryptDecodeObject
Service	QueryServiceLockStatus, SetServiceObjectSecurity
Memory	VirtualAlloc, VirtualUnlock, ReadProcessMemory

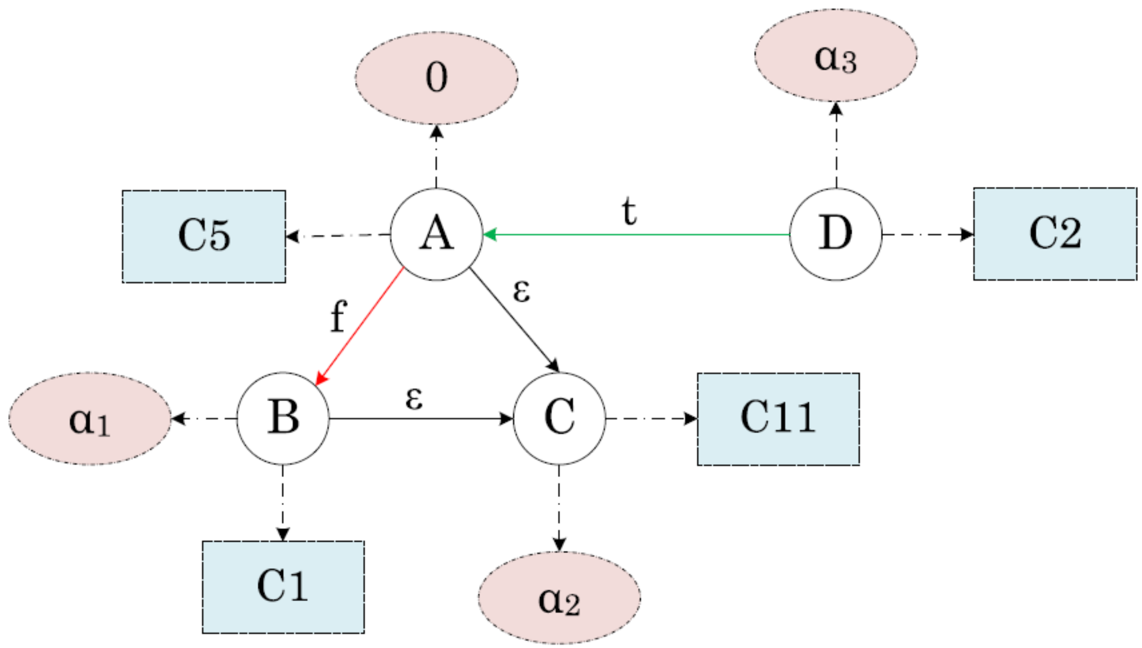
SIG Components (Cont'd)

- *c*FCG

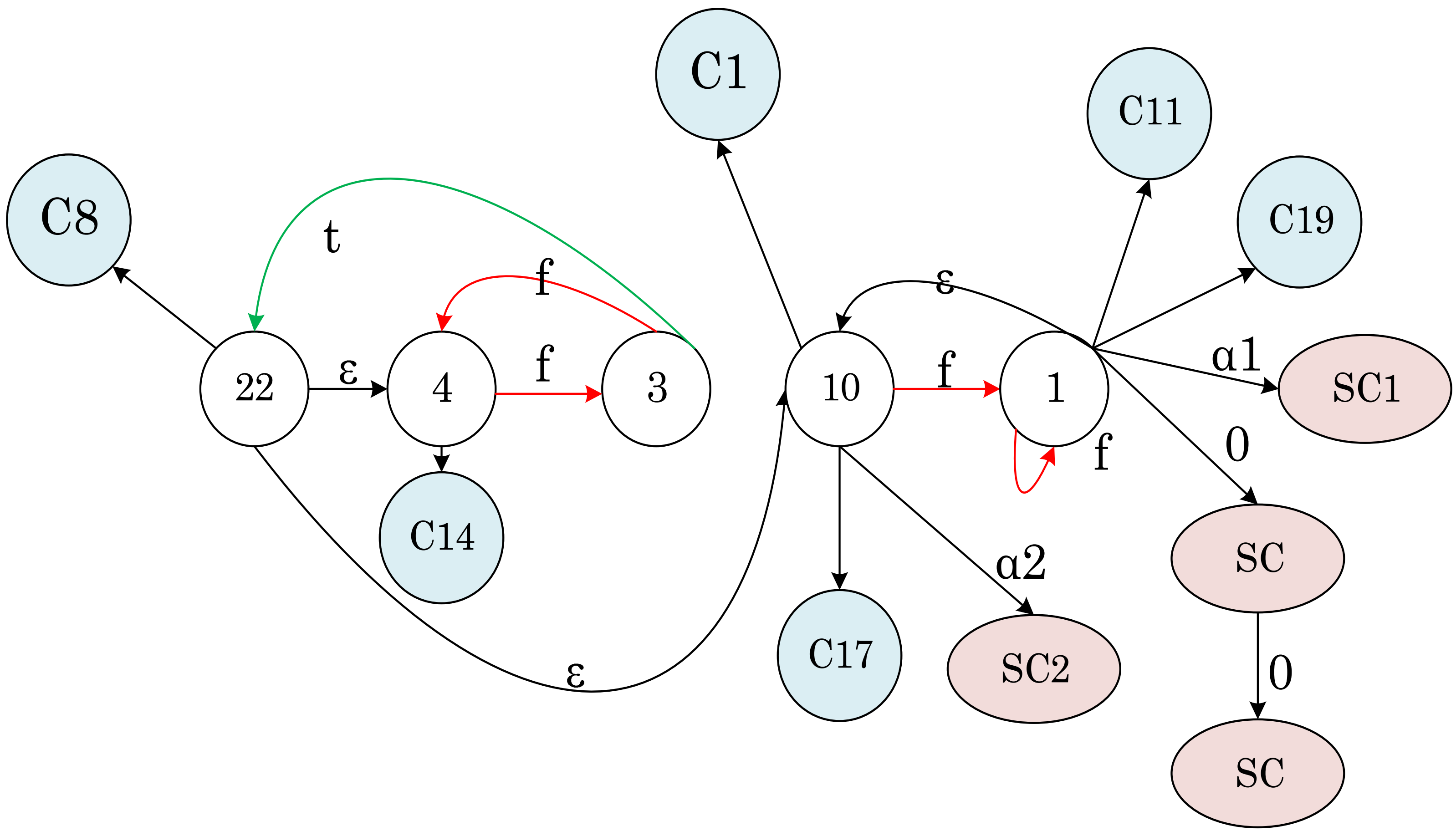


SIG Graph

- SIG
  - *i*CFG
  - *m*RFG
  - *c*FCG

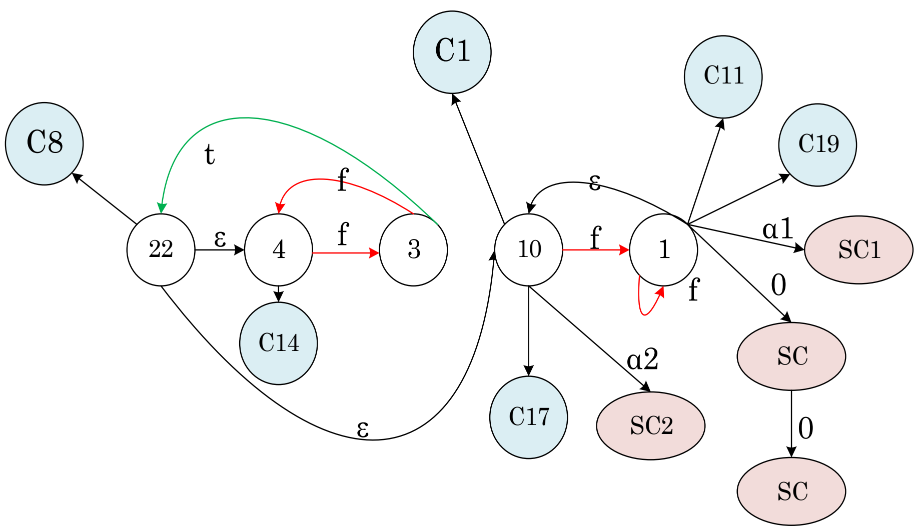


SIG Graph : Bubble Sort



SIG Features and Traces

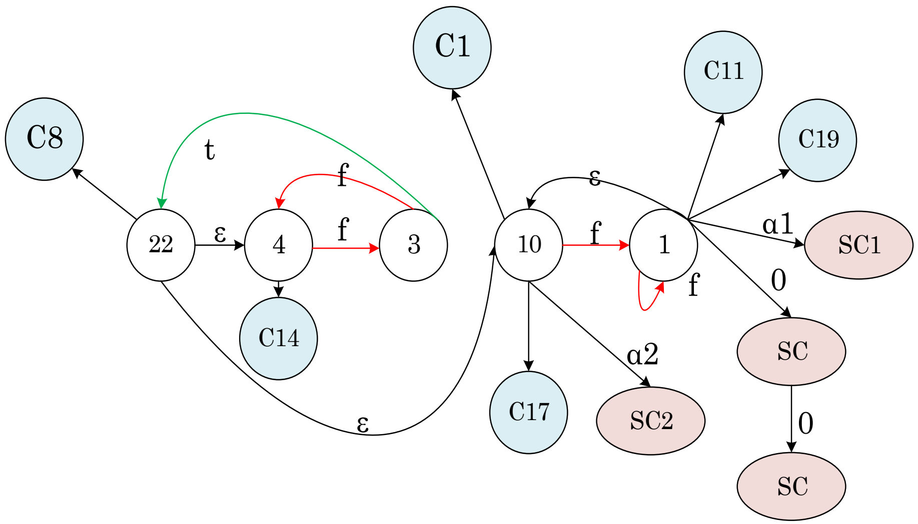
- Features
  - Exact matching



Features	Frequency
Total # of Nodes	15
Total # of Edges	18
# of Control Nodes	5
# of Control Edges	8
# of Call Nodes	4
# of Register Nodes	6
Connected Graphs	3
k-Cone	1, 2

SIG Features and Traces

- Traces
  - Inexact Matching



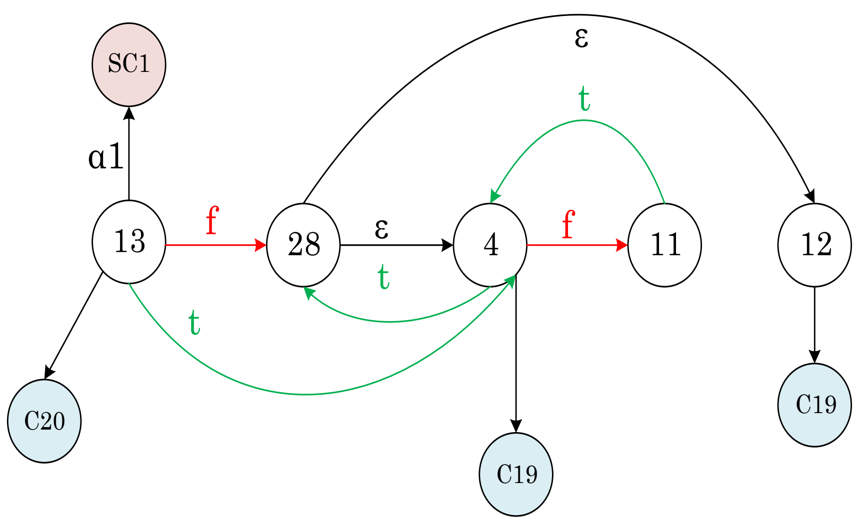
Node	Traces	Traces type
22	ε, ε, C8	out
	t	in
4	f, C14	out
	ε, f	in
22 OR 4	ε, C8, C14, f	out
	f, t, ε	in
4 AND 3	f	out
	f	in

Outline

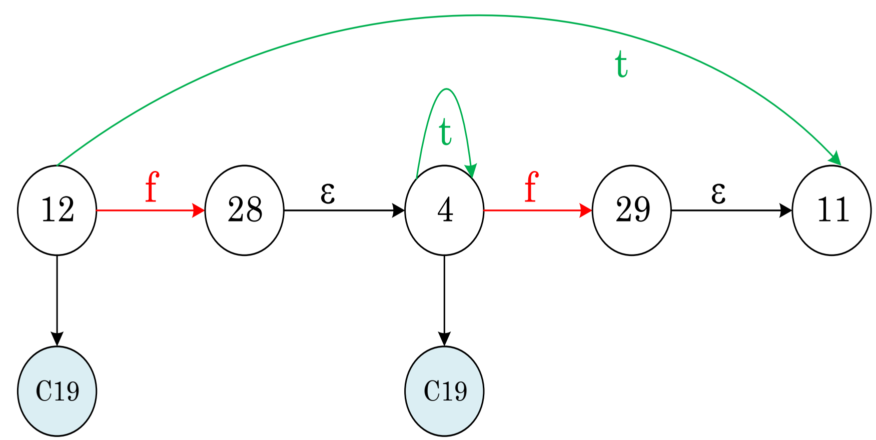
- Introduction
- Binary Representations
- SIGMA
- Case Study & Experimental Results
- Concluding Remarks

Case Study : Citadel to Zeus

- SIG for RC4 in Citadel



- SIG for RC4 in Zeus



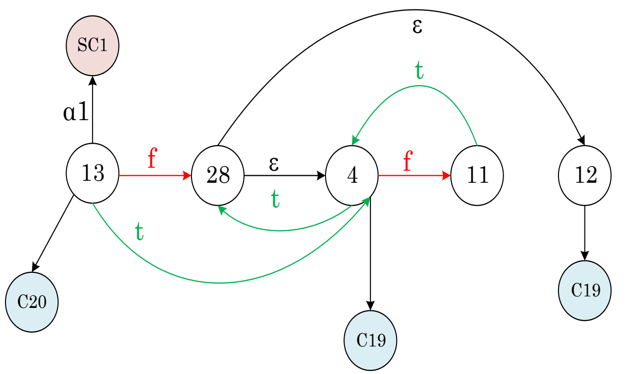
Case Study : Citadel to Zeus

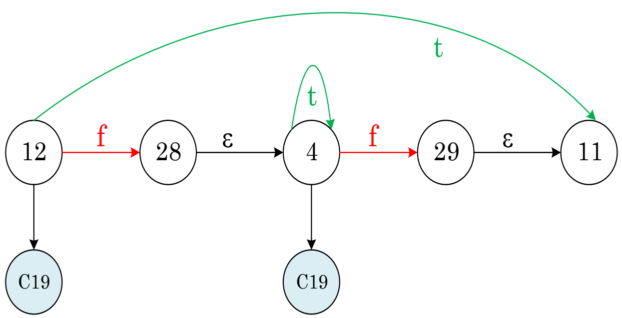
- Exact matching

Features	*SIG*-RC4 (Zeus)	*SIG*-RC4 (Citadel)	Similarity
Total # of Nodes	7	9	78%
Total # of Edges	8	11	72%
# of Control Nodes	5	5	100%
# of Control Edges	6	7	86%
# of Call Nodes	1	0	50%
# of Register Nodes	2	3	67%
Connected Graphs	3	3	100%
K-Cone	1,2,3,4	1,2,3	75%
Average Similarity			**78.5%**

Case Study : Citadel to Zeus

- Inexact matching



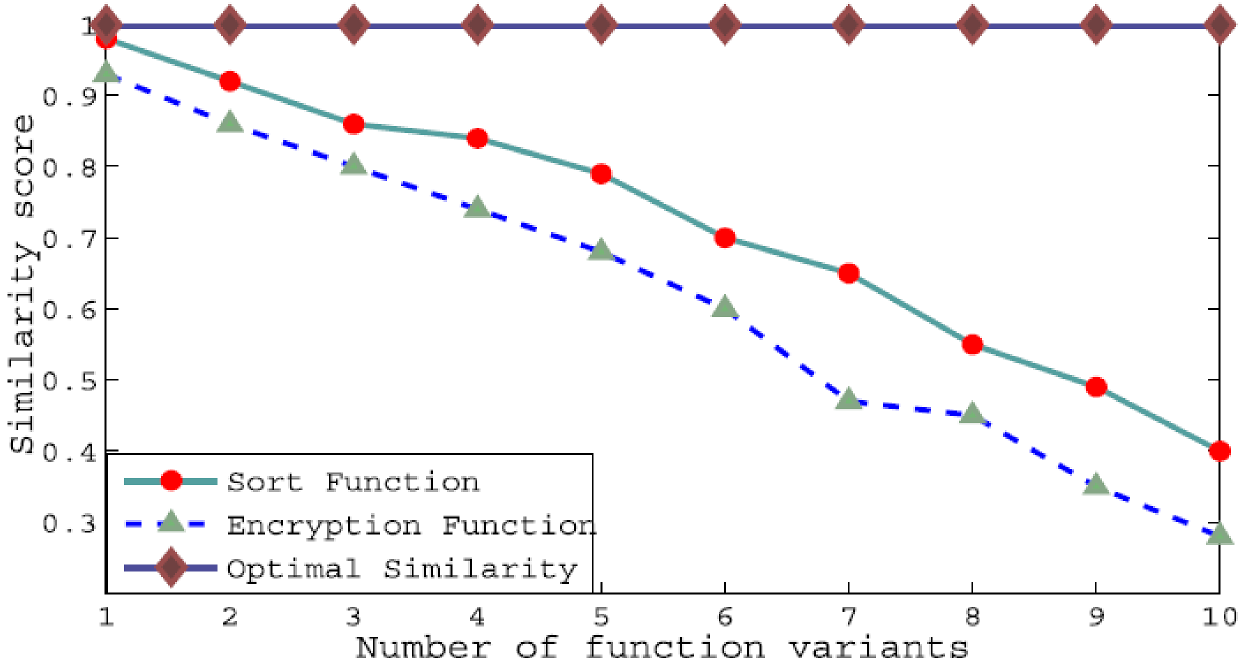


Threshold = 0.5 >

Citadel node	Zeus node	Costs	Node(s) with minimum cost	Cost %
13	12	1 out, 0 in	12	(1/10)
	28	1 out, 1 in	–	
	4	1 out, 2 in	–	
	29	1 out, 1 in	–	
	11	0 out, 2 in	–	
28	12	3 out, 0 in	–	
	28	0 out, 0 in	28 (Select this)	0
	4	3 out, 1 in	–	
	29	0 out, 0 in	29	
	11	0 out, 1 in	–	
4	12	0 out, 0 in	12 (Already chosen)	
	28	1 out, 1 in	–	
	4	0 out, 0 in	4 (Select this)	0
	29	1 out, 1 in	–	
	11	0 out, 0 in	11	
11	12	2 out, 0 in	–	
	28	1 out, 0 in	28 (Already chosen)	
	4	2 out, 2 in	–	
	29	1 out, 0 in	29 (Select this)	(1/10)
	11	0 out, 2 in	–	
12	12	2 out, 0 in	–	
	28	1 out, 1 in	–	
	4	2 out, 0 in	–	
	29	1 out, 1 in	–	
	11	0 out, 1 in	11	(1/9)
Total Cost				**0.311**

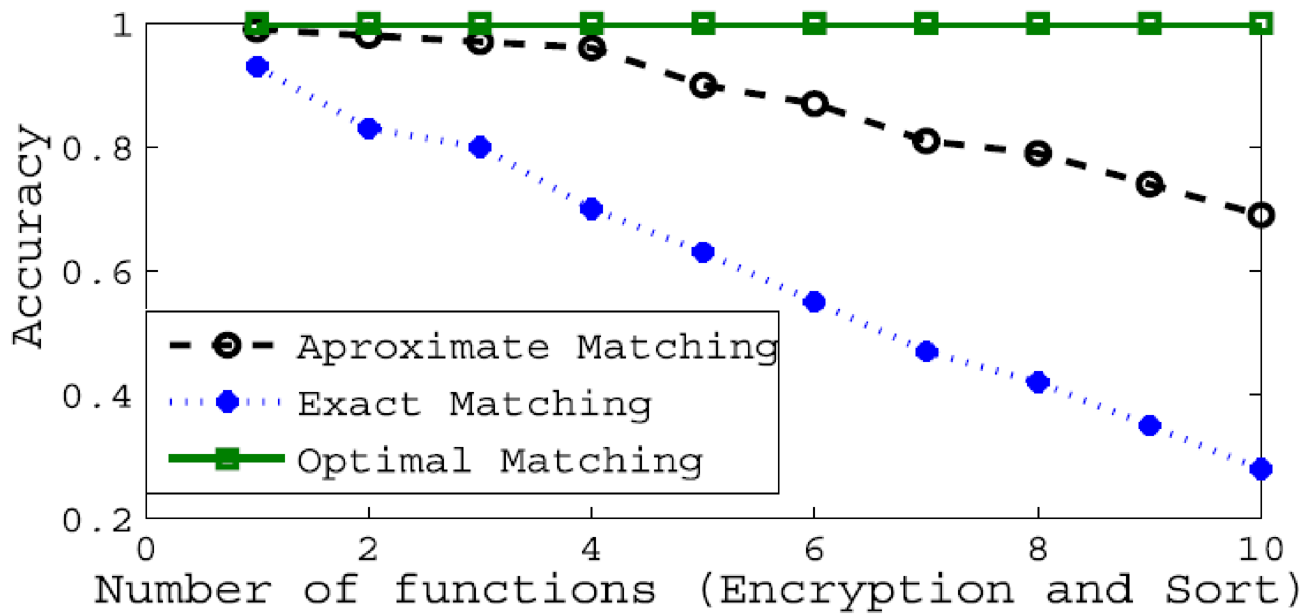
Experimental Results

- Similarity Score



Experimental Results (Cont’d)

- Accuracy



Outline

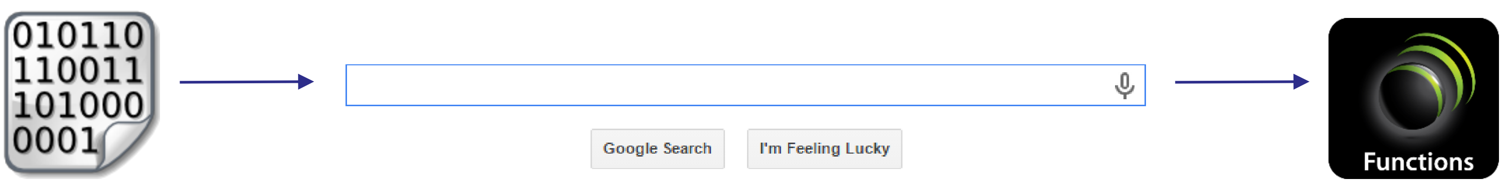
- Introduction
- Binary Representations
- SIGMA
- Case Study & Experimental Results
- Concluding Remarks

Summary

- Introduce a novel approach for effectively identifying reused functions in binary code
- Enhance and combine multiple representations into one joint data structure, called SIG
- Apply exact and inexact graph matching
- Evaluate our approach using sort and encryption functions
- Both experimental results have demonstrated the effectiveness of our method

Future Work

- Test larger datasets
- Improve graph matching algorithm
- Develop a search engine for identifying assembly fragments in binary code



Thank YOU