Formal Approaches for Automatic Deobfuscation and Reverse-engineering of Protected Codes

PhD Defense - Robin David

January 6th, 2017

Start

Agenda of the presentation

1. **Introduction**
2. **Dynamic Symbolic Execution** extensions and variants
3. **Implementation**: Binsec
4. **Combination** of analyses
5. **Case-studies**
6. **Conclusion**

1.

Introduction

Context: Malware analysis

What is a malware and why does it matter to analyse them ?

Definition
Malware is a generic term grouping all softwares developed with the intention to harm and to threaten computer systems or their users.

Some numbers:

Average cost of a breach[1] *(almost always involving malware)*	**4M$**
Annual cost of cybercrime[2,3]	**> 400B$**
New malware sample detected daily[4,5]	**> 230K**

Context: Malware more & more critical

BBC | Sign in | News | Sport | Weather | Shop | Earth | Travel | More | Search

NEWS

Home | Video | World | UK | Business | Tech | Science | Magazine | Entertainment & Arts | Health | World News TV | More

Technology

Stuxnet worm 'targeted high-value Iranian assets'

By Jonathan Fildes
Technology reporter, BBC News

23 September 2010 | Technology

One of the most sophisticated pieces of malware ever detected was probably targeting "high value" infrastructure in Iran, experts have told the BBC.

Some have speculated the intended target was Iran's nuclear power plant

Stuxnet's complexity suggests it could only have been written by a "nation state", some researchers have claimed.

It is believed to be the first-known

BBC | Sign in | News | Sport | Weather | Shop | Earth | Travel | More | Search

NEWS

Home | Video | World | UK | Business | Tech | Science | Magazine | Entertainment & Arts | Health | World News TV | More

Technology

Flame: Massive cyber-attack discovered, researchers say

By Dave Lee
Technology reporter, BBC News

28 May 2012 | Technology | Share

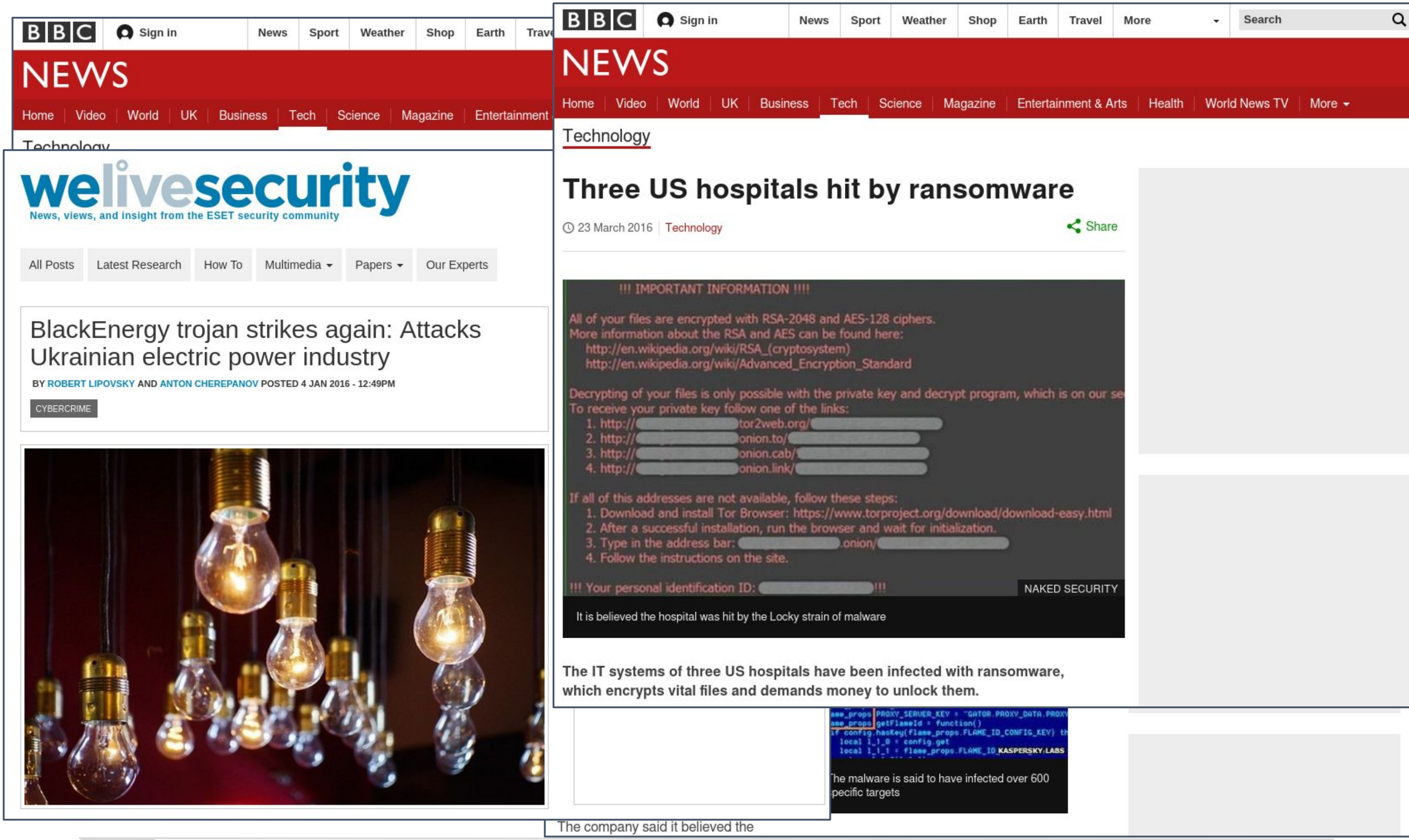
A complex targeted cyber-attack that collected private data from countries such as Israel and Iran has been uncovered, researchers have said.

The malware is said to have infected over 600 specific targets

Russian security firm Kaspersky Labs told the BBC they believed the malware, known as Flame, had been operating since August 2010.

The company said it believed the

Context: Malware more & more critical



BBC Sign in News Sport Weather Shop Earth Travel More Search

NEWS

Home Video World UK Business Tech Science Magazine Entertainment & Arts Health World News TV More

Technology

Three US hospitals hit by ransomware

23 March 2016 Technology Share

!!! IMPORTANT INFORMATION !!!!

All of your files are encrypted with RSA-2048 and AES-128 ciphers.
More information about the RSA and AES can be found here:
http://en.wikipedia.org/wiki/RSA_(cryptosystem)
http://en.wikipedia.org/wiki/Advanced_Encryption_Standard

Decrypting of your files is only possible with the private key and decrypt program, which is on our se
To receive your private key follow one of the links:
1. http://[redacted]tor2web.org/[redacted]
2. http://[redacted]onion.to/[redacted]
3. http://[redacted]onion.cab/[redacted]
4. http://[redacted]onion.link/[redacted]

If all of this addresses are not available, follow these steps:
1. Download and install Tor Browser: https://www.torproject.org/download/download-easy.html
2. After a successful installation, run the browser and wait for initialization.
3. Type in the address bar: [redacted].onion/[redacted]
4. Follow the instructions on the site.

!!! Your personal identification ID: [redacted]!!!

NAKED SECURITY

It is believed the hospital was hit by the Locky strain of malware

The IT systems of three US hospitals have been infected with ransomware, which encrypts vital files and demands money to unlock them.

welivesecurity

News, views, and insight from the ESET security community

All Posts Latest Research How To Multimedia Papers Our Experts

BlackEnergy trojan strikes again: Attacks Ukrainian electric power industry

BY ROBERT LIPOVSKY AND ANTON CHEREPANOV POSTED 4 JAN 2016 - 12:49PM

CYBERCRIME

KASPERSKY LABS

The malware is said to have infected over 600 specific targets

The company said it believed the

Binary analysis

Specificities inherent to binary analysis

Why on binary? Because source code generally not available on malware

Rule of the game *(w.r.t. source level)*

- compiler independent *(and potential issues)*
- language independent *(+ source free)*
- **no source code**

Handicap / Problematic

- **no distinction between code & data** *(jump eax)*
- only bitvector arithmetic
- memory not "typed" *(one flat array)*

Binary analysis: Example Switch

What is inside a blob of binary ? [Reps10] [Meng16]

Sections

.text

```
8D 4C 24 04 83 E4 F0 FF 71 FC 55 89 E5 53 51 83
EC 10 89 CB 83 EC 0C 6A 0A E8 A7 FE FF FF 83 C4
10 89 45 F0 8B 43 04 83 C0 04 8B 00 83 EC 0C 50
E8 C0 FE FF FF 83 C4 10 89 45 F4 83 7D F4 04 77
3B 8B 45 F4 C1 E0 02 05 98 85 04 08 8B 00 FF E0
C7 45 F4 00 00 00 00 EB 23 C7 45 F4 01 00 00 00
EB 1A C7 45 F4 02 00 00 00 EB 11 C7 45 F4 03 00
00 00 EB 08 C7 45 F4 04 00 00 00 90 83 EC 08 FF
75 F4 68 90 85 04 08 E8 29 FE FF FF 83 C4 10 8B
45 F4 8D 65 F8 59 5B 5D 8D 61 FC C3 66 90 66 90
66 90 66 90 90 55 57 31 FF 56 53 E8 85 FE FF FF
81 C3 89 12 00 00 83 EC 1C 8B 6C 24 30 8D B3 0C
FF FF FF E8 B1 FD FF FF 8D 83 08 FF FF FF 29 C6
C1 FE 02 85 F6 74 27 8D B6 00 00 00 00 8B 44 24
38 89 2C 24 89 44 24 08 8B 44 24 34 89 44 24 04
FF 94 BB 08 FF FF FF 83 C7 01 39 F7 75 DF 83 C4
1C 5B 5E 5F 5D C3 EB 0D 90 90 90 90 90 90 90 90
90 90 90 90 90 F3 C3 FF FF 53 83 EC 08 E8 13 FE
FF FF 81 C3 17 12 00 00 83 C4 08 5B C3 03 00 00
00 01 00 02 00 76 61 6C 3A 25 64 0A 00 AB 84 04
08 B4 84 04 08 BD 84 04 08 C6 84 04 08 CF 84 04
08 01 1B 03 3B 28 00 00 00 04 00 00 00 54 FD FF
```

.fini
.rodata
.eh_frame_hdr

Code (Functions)

main

unknown

__libc_csu_init

unknown

__libc_csu_fini

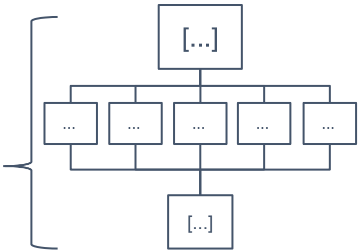
__term_proc

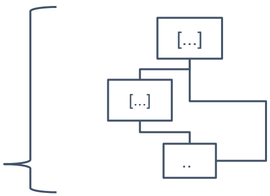
_fp_hw, _IO_stdin_used

"val%d\n"

switch jump table

Assembly






```
rep retn
```

```
push ebx
sub  esp, 8
call get_pc[..]
add ebx, 0x1217
add esp, 8
pop ebx
retn
```

■ code ■ dead bytes ■ global csts ■ strings ■ pointers ■ other

Disassembly process

The three different steps to get through in order to disassemble a program

Code discovery (aka. Decoding opcodes)

- Non-code bytes
- Missing symbols (function addr)
- Instruction overlapping

CFG reconstruction (aka. Building the graph, nodes & edges)

- Indirect control-flow
- Non-returning functions

CFG partitioning (aka. Finding functions, bounds etc)

- Function code sharing
- Non-contiguous function
- Tail calls

Malware now uses **obfuscation** and other tricks to **hide** their intents

How to find and to remove **obfuscation**?

How to differentiate the cat from the dogs ?

Obfuscation Techniques *(Some)*

What is obfuscation ? What are the different kinds of obfuscation ? [Collberg97] [Barak12]

Obfuscation: Any means aiming at slowing-down the analysis process for a human or an automated algorithm.

	Target		Against	
	Control	Data	Static	Dynamic
CFG flattening	●		●	
Jump encoding (direct → indirect/computed)	●		●	
Opaque predicates	●		●	
VM (Virtual-Machines)	●	●	●	●
Polymorphism (self-modification resource ciphering)	●	●	●	
Call stack tampering	●		●	
Anti-debug / Anti-tampering	●	●		●
Signal / Exception	●		●	

Opaque predicates

What is opaque predicate, and what is its purpose ?

- **Definition:** Predicate always evaluating to true *(resp false)* (but for which this property is difficult to deduce)
- **Can be based on:**
  - Arithmetic
  - Data-structure
  - Pointer
  - Concurrency
  - Environment
- **Corollary**, dead branch allows to:
  - Grow the code (artificially)
  - Drown the genuine code

eg: $7y^2 - 1 \neq x^2$

(for any value of x, y in modular arithmetic)

↧

```
mov  eax, ds:X
mov  ecx, ds:Y
imul ecx, ecx
imul ecx, 7
sub  ecx, 1
imul eax, eax
cmp  ecx, eax
jz   <dead_addr>
```

Call stack tampering

What is a call stack violation and its implication for analysis ?

- **Definition:** Alter the standard compilation scheme of a call and ret instructions

- **Corollary:**
  - Real ret target hidden and returnsite potentially not code
  - Impede the recovery of control flow edges
  - Impede the high-level function recovery

address	instr
80483d1	call +5
80483d6	pop edx
80483d7	add edx, 8
80483da	push edx
80483db	ret
80483dc	.byte{invalid}
80483de	[...]

General **Goal & Challenges**

What are the objectives of this thesis and the research challenges it implies ?

Objectives

- Analysis of obfuscated binaries and malware
- Recovering a high-level view of the program
- **Locating and removing obfuscation if any**
- **raising the difficulty of program obfuscation**
- **improving malware comprehension**
  (not necessarily detection)

Challenges

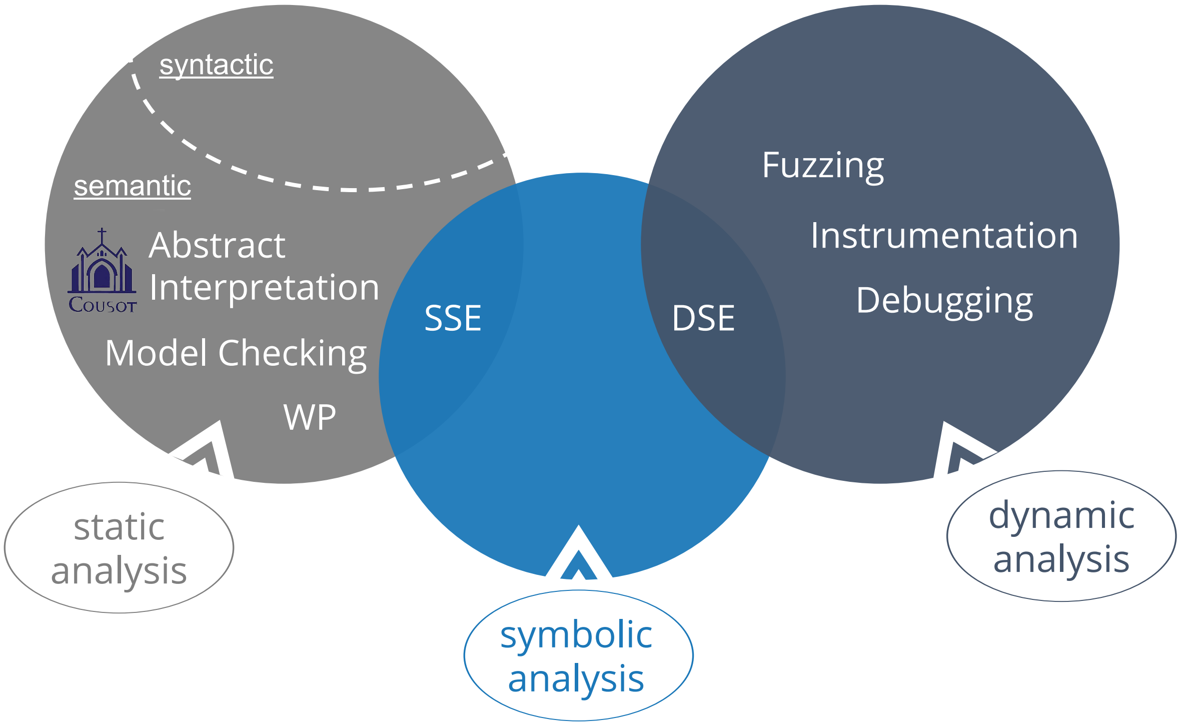
- Binary analysis
- **Scalability**
- **Robustness** w.r.t obfuscation

Deobfuscation

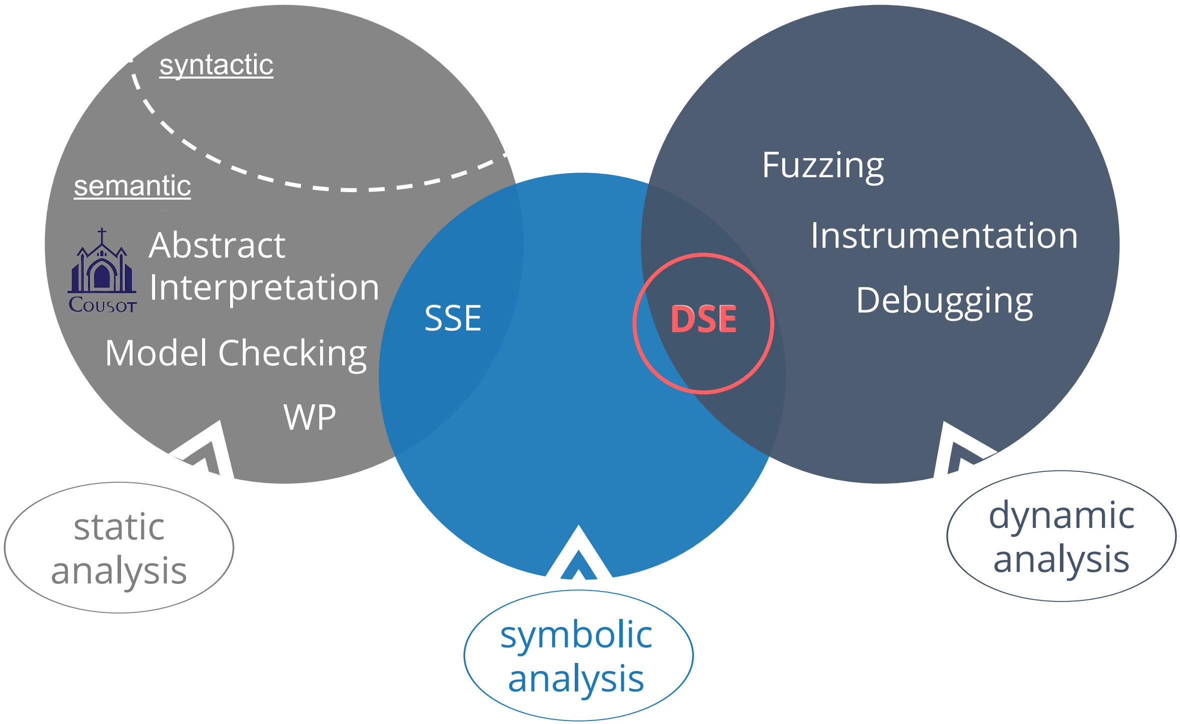
- Revert the transformation (often impossible)
- Simplify the code to facilitate later analysis

⇒ **best effort approach *(undecidable problems)***

Existing **Analysis Techniques**



Existing **Analysis Techniques**

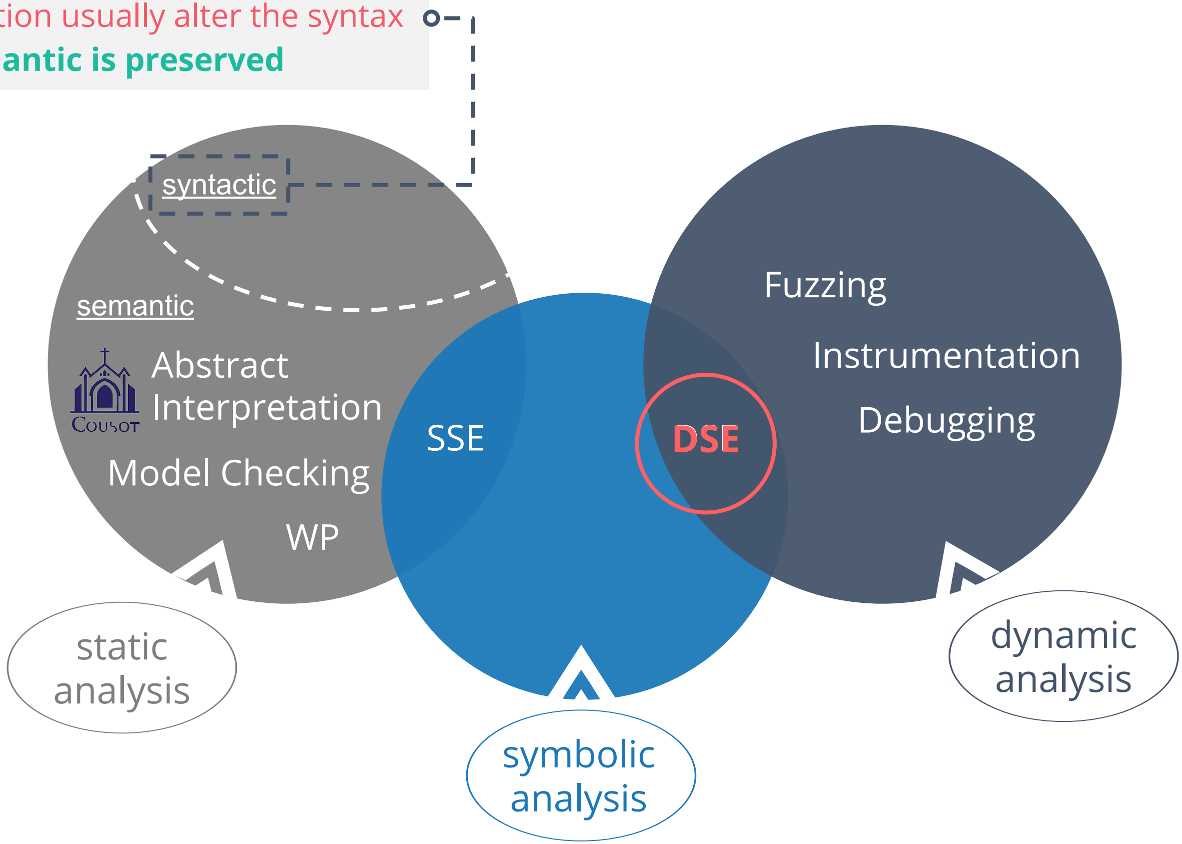


Existing **Analysis Techniques**

Why not syntactic analysis ?

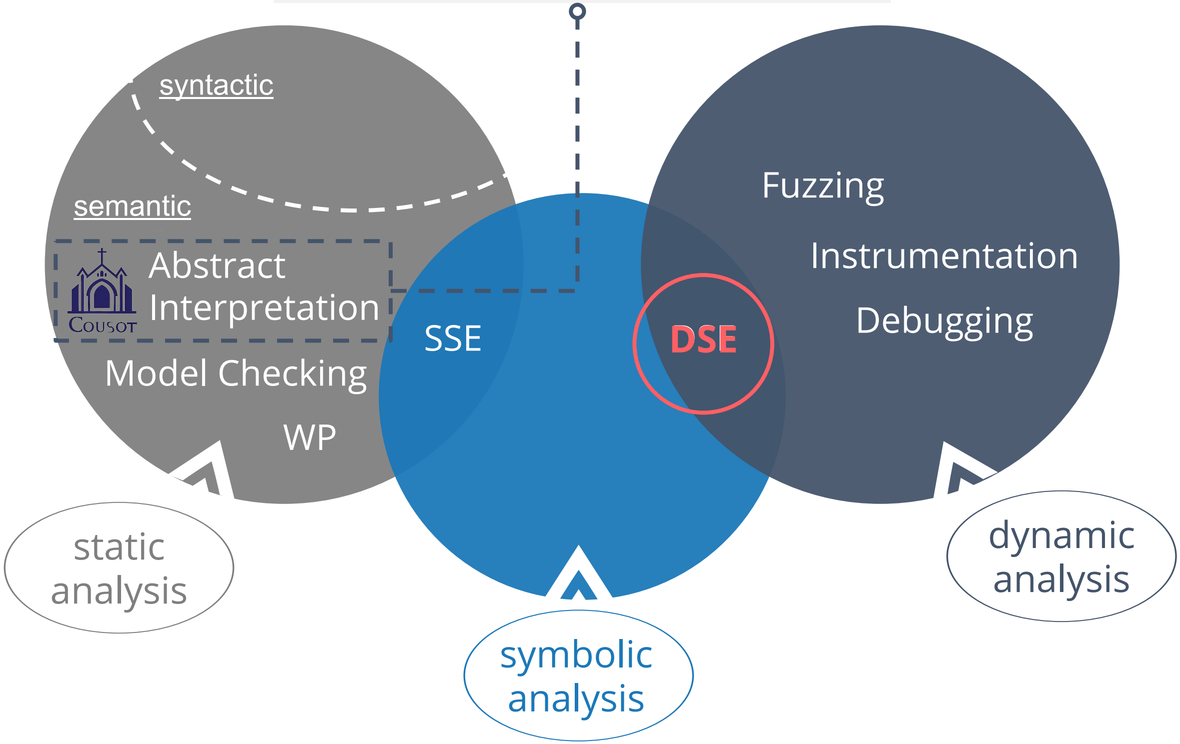
✖ Obfuscation usually alter the syntax

⇒ **But semantic is preserved**



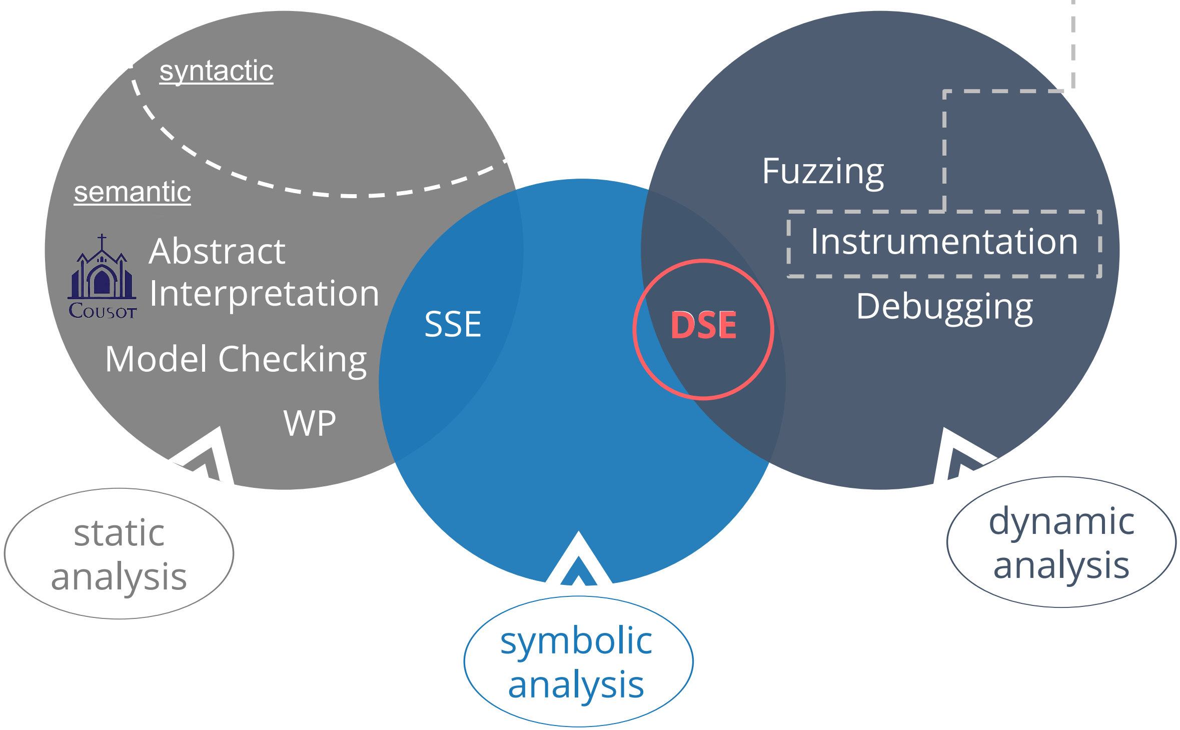
Existing **Analysis Techniques**

Why not abstract interpretation ?
✖ hindered by SMC and tricks against static analysis
⇒ **DSE takes advantage of dynamic**



Existing **Analysis Techniques**

Why not dynamic analysis ?
✖ Only cover on path
⇒ **DSE can find new paths**

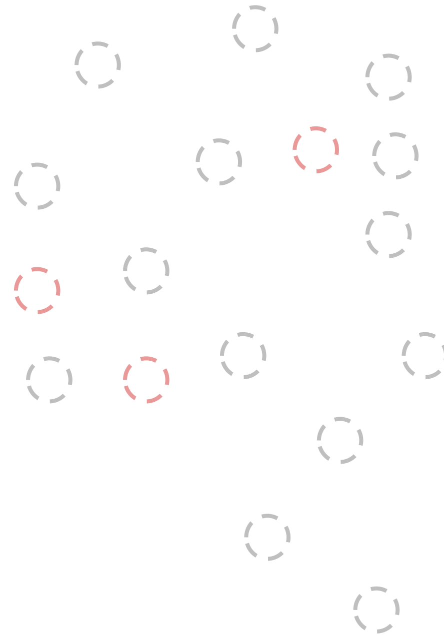


State of the Technique in disassembly

The different disassembly approaches and their shortcomings and strength

Notation

- **Correct:** only genuine (executable) instructions are disassembled
- **Complete:** all genuine instructions are disassembled

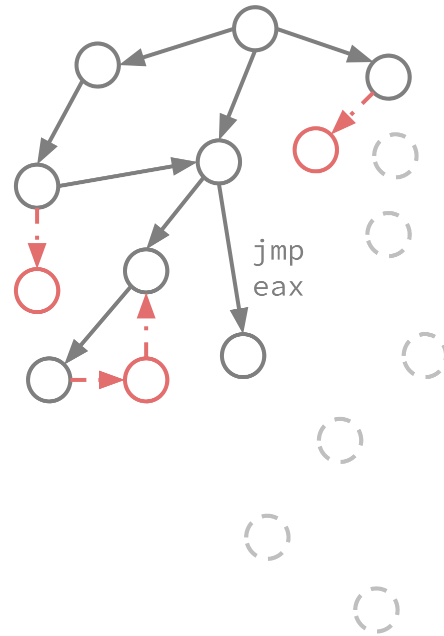


Standard approaches:

State of the Technique in disassembly

The different disassembly approaches and their shortcomings and strength

Notation
• **Correct:** only genuine (executable) instructions are disassembled • **Complete:** all genuine instructions are disassembled



Standard approaches:

○ static disassembly

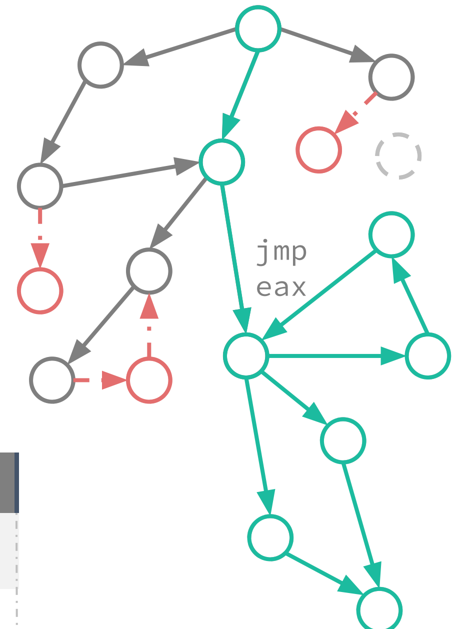
	static
scale	●
robust *(obfuscation)*	●
correct	●
complete *(coverage)*	●

dynamic jump

State of the Technique in disassembly

The different disassembly approaches and their shortcomings and strength

Notation
• **Correct:** only genuine (executable) instructions are disassembled • **Complete:** all genuine instructions are disassembled



Standard approaches:

- static disassembly
- dynamic disassembly

	static	dynamic
scale	●	●
robust *(obfuscation)*	●	●
correct	●	●
complete *(coverage)*	●	●

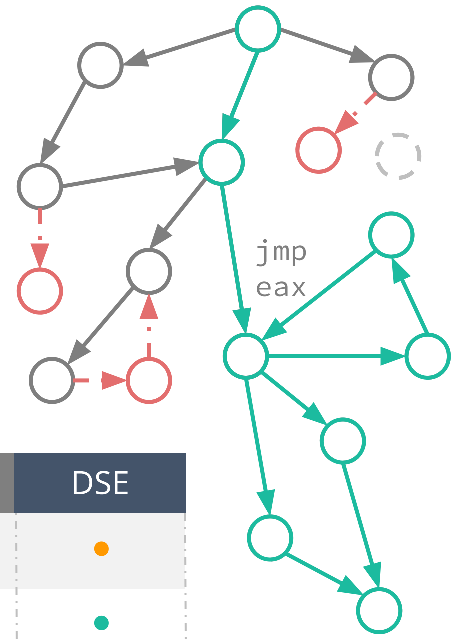
dynamic jump

input dependent

State of the Technique in disassembly

The different disassembly approaches and their shortcomings and strength

Notation
• **Correct:** only genuine (executable) instructions are disassembled • **Complete:** all genuine instructions are disassembled



Standard approaches:

○ static disassembly
○ dynamic disassembly

	static	dynamic	DSE
scale	●	●	●
robust *(obfuscation)*	●	●	●
correct	●	●	●
complete *(coverage)*	●	●	●

coverage + obfuscation infos

dynamic jump

input dependent

Symbolic Execution

Definition and how it works in practice ? [King76]

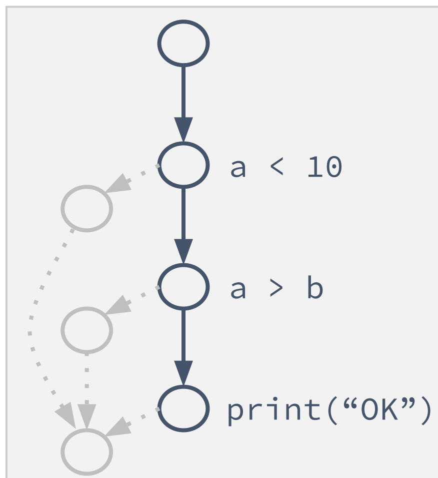
Definition

Mean of executing a program using symbolic values (logical symbols) rather than real values (bitvectors) in order to obtain an in-out relationship of a path.

How to reach “OK” ?

Source Code (C)

```
int f(int a, int b) {
  if (a < 10) {
    if (a > b) {
      printf(“OK”);
    }
  }
}
```



Formula:

$a < 10 \wedge a > b$

Solution:
a=5, b=1

(using SMT solvers)

Dynamic Symbolic Execution (aka concolic)

What is dynamic symbolic execution and advantages? [Godefroid05]

- **Main properties:**
  - works on a dynamically generated path
  - can take advantage of runtime values [concretization]

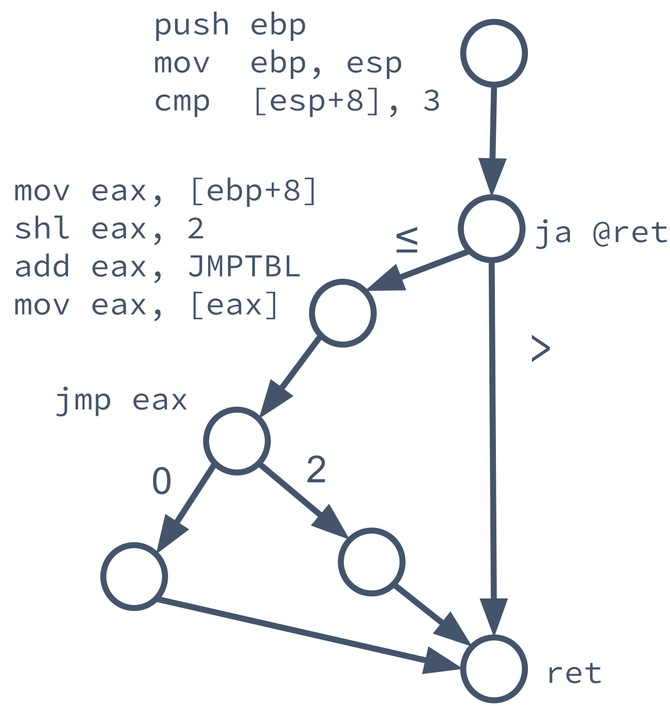
Advantages

- **path sure to be feasible** [unlike static]
- **can generate new inputs** [unlike dynamic]
- **thwart basic tricks** [code-overlapping, SMC, etc]
- easier than static semantic analysis
  - next instruction always known
  - loops unrolled

DSE Path Coverage: Switch example

Extending the disassembly by covering new paths

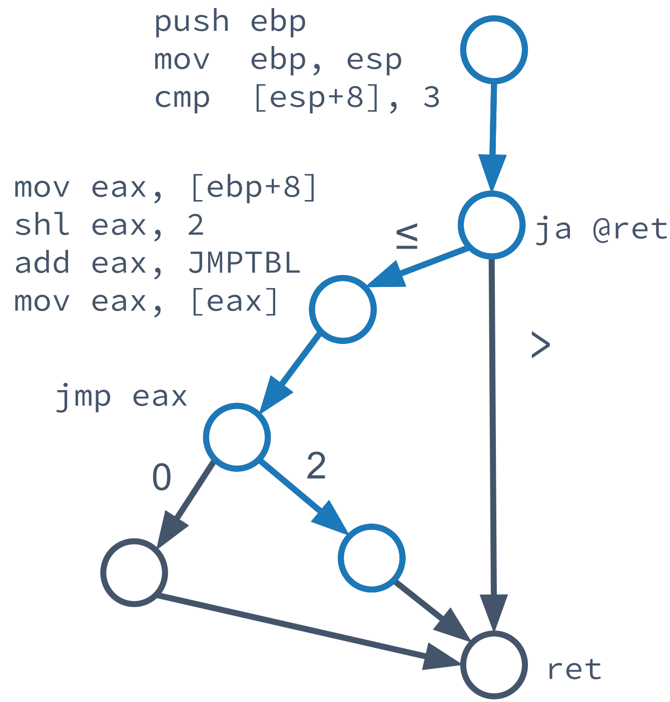
x86 assembly	**Symbolic Execution** (input:esp, ebp, memory)
push ebp	@[esp] := ebp
mov ebp, esp	ebp1 := esp
cmp [ebp+8], 3	@[ebp1+8] < 3
ja @ret	
mov eax, [ebp+8]	eax1 := @[esp+8]
shl eax, 2	eax2 := eax1 << 2
add eax, JMPTBL	eax3 := eax2 + JMPTBL
mov eax, [eax]	eax4 := @[eax3]
jmp eax	eax4 == 2



DSE Path Coverage: Switch example

Extending the disassembly by covering new paths

x86 assembly	Symbolic Execution (input:esp, ebp, memory)
push ebp	@[esp] := ebp
mov ebp, esp	ebp1 := esp
cmp [ebp+8], 3	@[ebp1+8] < 3
ja @ret	
mov eax, [ebp+8]	eax1 := @[esp+8]
shl eax, 2	eax2 := eax1 << 2
add eax, JMPTBL	eax3 := eax2 + JMPTBL
mov eax, [eax]	eax4 := @[eax3]
jmp eax	eax4 == 2



Path predicate φ :

@[ebp1+8] < 3 ∧ eax4 == 2

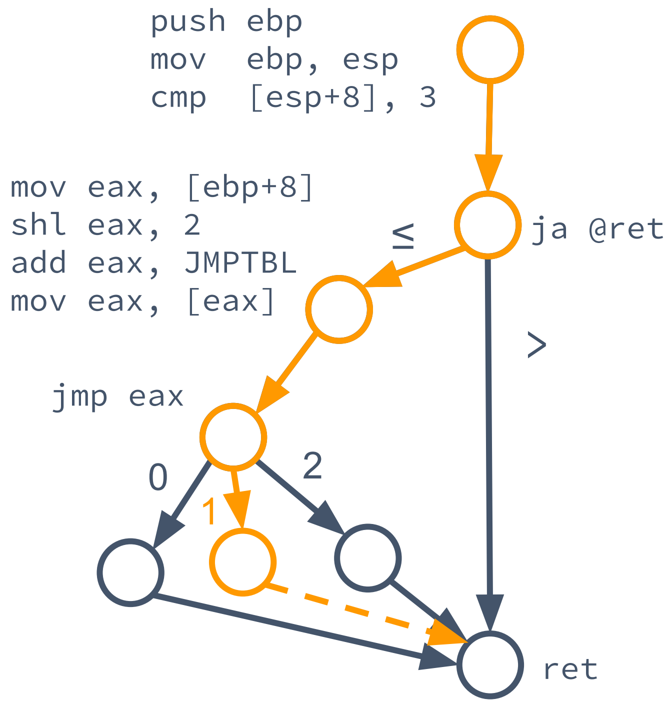
@[esp+8] < 3 ∧ @[(@[esp+8]≪ 2) + JMPTBL] == 2

DSE Path Coverage: Switch example

Extending the disassembly by covering new paths

x86 assembly
push ebp
mov ebp, esp
cmp [ebp+8], 3
ja @ret
mov eax, [ebp+8]
shl eax, 2
add eax, JMPTBL
mov eax, [eax]
jmp eax

Symbolic Execution (input:esp, ebp, memory)
@[esp] := ebp
ebp1 := esp
@[ebp1+8] < 3
eax1 := @[esp+8]
eax2 := eax1 << 2
eax3 := eax2 + JMPTBL
eax4 := @[eax3]
eax4 == 2



Path predicate φ :

@[ebp1+8] < 3 ∧ eax4 ≠ [0,2]

@[esp+8] < 3 ∧ @[(@[esp+8] ≪ 2) + JMPTBL] ≠ [0,2]

DSE limitations

Why is DSE limited in some ways to address obfuscation?

1

Scalability
path predicate solving, and path coverage

2

Flexibility
Difficulty to tune execution in existing engines

3

No infeasibility
DSE Solve reachability issues on a given path *(while some issues are infeasibility issues)*

Thesis **Contributions**

The four main contributions in terms of binary analysis for obfuscated binaries

DSE for obfuscation

#1

#1 flexible C/S policies via **CSML**

#2 infeasibility with **backward bounded DSE**.

[ISSTA16]
[S&P17]

Implementation in Binsec

#2

#1 Binsec/SE with **solver optimizations**

#2 instrumentation with **Pinsec**

#3 IDA plugin **Idasec**.

[SANER16]
[BHEU16]

Analysis combinations

#3

#1 sparse disassembly for obfuscated code disassembly

#2 vulnerability discovery

#3 software testing

[ICST15]
[SSPREW16]

Case-studies

#4

#1 packers large scale study

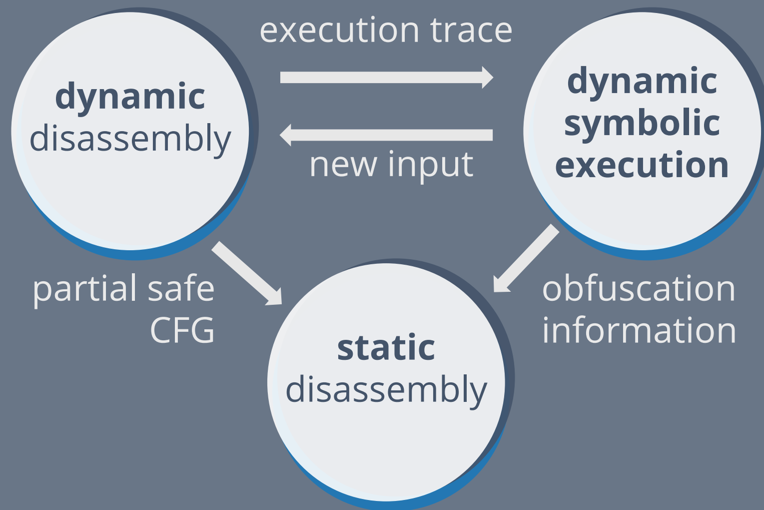
#2 X-Tunnel deobfuscation

[BHEU16]
[S&P17]

Toward semantic-aware disassembly

Long term objective aimed by this thesis

Focus: Combination of symbolic, static and dynamic for deobfuscation



2.

Dynamic Symbolic Execution **extensions and variants**

Concretization & Symbolization modulation

What are concretization and symbolization?

program

```
input: a, b
x := a × b
x := x + 1
//assert x > 10
```

Concretization & Symbolization modulation

What are concretization and symbolization?

- **Propagation:** logical propagation *(without approximation)*

program	Propagation (path predicate)
input: a, b x := a × b x := x + 1 **//assert x > 10**	x1 = a × b ∧ x2 = x1 + 1 ∧ x2 > 10

Concretization & Symbolization modulation

What are concretization and symbolization?

- **Propagation:** logical propagation *(without approximation)*

- **Concretization:** replace a logical variable by its runtime value [Godefroid05]
  - simplify the formula (but under-approximate it)
  - simplify the computation of irrelevant parts of the program

program	Propagation (path predicate)	Concretization
input: a, b x := a × b x := x + 1 **//assert x > 10**	x1 = a × b ∧ x2 = x1 + 1 ∧ x2 > 10	**a = 5** ∧ x1 = **5** × b ∧ x2 = x1 + 1 ∧ x2 > 10

Concretization & Symbolization modulation

What are concretization and symbolization?

- **Propagation:** logical propagation *(without approximation)*
- **Concretization:** replace a logical variable by its runtime value [Godefroid05]
  - simplify the formula (but under-approximate it)
  - simplify the computation of irrelevant parts of the program
- **Symbolization:** replace a logical variable by a new symbol
  - simulate non-deterministic effect (but over-approximate)
  - injecting inputs in the execution

program	Propagation (path predicate)	Concretization	Symbolization

`input: a, b`
`x := a × b`
`x := x + 1`
`//assert x > 10`
	`x1 = a × b`
	`∧ x2 = x1 + 1`
	`∧ x2 > 10`
		`a = 5`
		`∧ x1 = 5 × b`
		`∧ x2 = x1 + 1`
		`∧ x2 > 10`
			`x1 = `**`fresh`**
			`∧ x2 = x1 + 1`
			`∧ x2 > 10`

➡ The goal is to find the right trade-off which is **extremely important in practice**

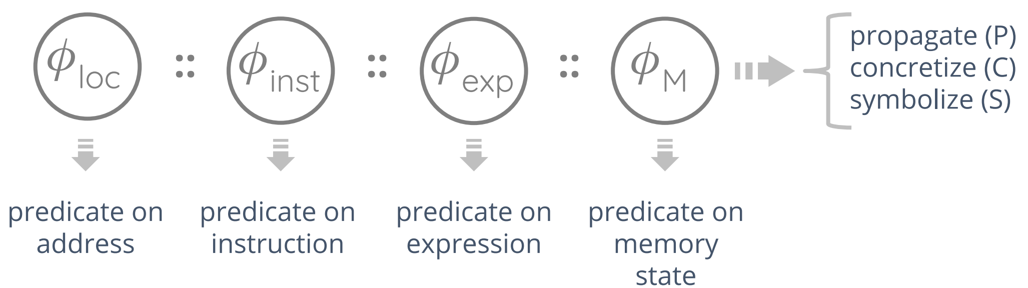
What is the issue of C/S ?

- Hardcoded in most engines
- Not well-documented (with its implication on soundness)
- **Important to modulate in order to scale !**

CSML: C/S Meta-Language [ISSTA16]

Modulating concretization and symbolization via a simple language.

- **Why:** need to find the balance between C & S to scale
- **Need:** an easy and generic specification system for C/S
- **Properties:**
  - language running dynamically over the DSE algorithm
  - defines the action to perform on each expression of the computation (i.e C,S,P)
  - defined as a rule-based language to match any expression



Allowed to tune finely the performance of the path predicate computation

CSML: Example

Example of how a CSML rule works and matches the expression of a DBA instruction

X86 instr : 804876: inc [ebp]

DBA instr : @[ebp] := @[ebp] + 1

CSML rule : $\star$ :: @[e?] := $\star$ :: e! :: $\star$ $\Rightarrow$ C

CSML: Example

Example of how a CSML rule works and matches the expression of a DBA instruction

X86 instr : 804876: inc [ebp]

DBA instr : @[ebp] := @[ebp] + 1

CSML rule : $\star$ :: @[e?] := $\star$ :: e! :: $\star$ $\Rightarrow$ C

match any location

CSML: Example

Example of how a CSML rule works and matches the expression of a DBA instruction

X86 instr : 804876: inc [ebp]

⇓

DBA instr : @[ebp] := @[ebp] + 1

CSML rule : ⋆ :: @[e?] := ⋆ :: e! :: ⋆ ⇒ C

- ⋆ (first): match any location
- @[e?] := ⋆ : match a memory write instruction

CSML: Example

Example of how a CSML rule works and matches the expression of a DBA instruction

X86 instr : 804876: inc [ebp]

DBA instr : @[ebp] := @[ebp] + 1

CSML rule : $\star$ :: @[e?] := $\star$ :: e! :: $\star \Rightarrow$ C

- $\star$: match any location
- @[e?] := $\star$: match a memory write instruction
- e! : match the expression **of write address** in the instr

CSML: Example

Example of how a CSML rule works and matches the expression of a DBA instruction

X86 instr : 804876: inc [ebp]

DBA instr : @[ebp] := @[ebp] + 1

CSML rule : $\star$ :: @[e?] := $\star$ :: e! :: $\star$ $\Rightarrow$ C

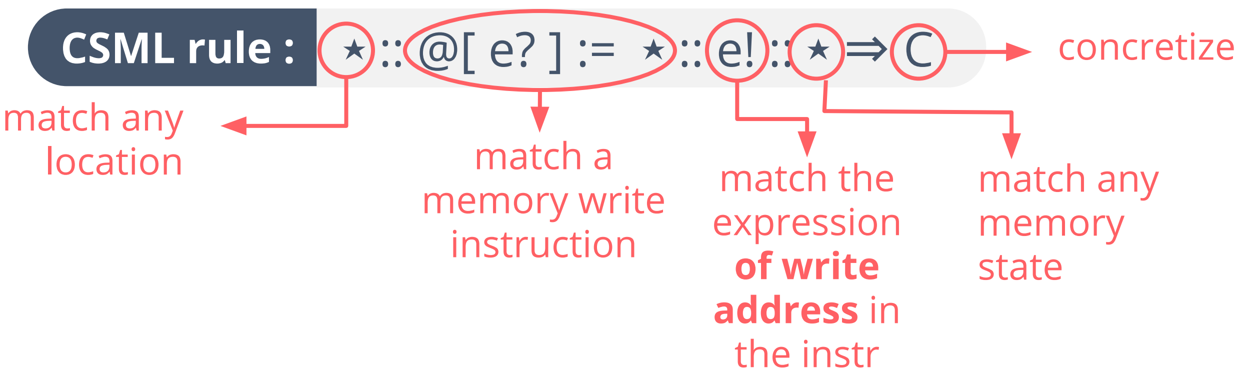
- $\star$: match any location
- @[e?] := $\star$: match a memory write instruction
- e! : match the expression **of write address** in the instr
- $\star$: match any memory state

CSML: Example

Example of how a CSML rule works and matches the expression of a DBA instruction

X86 instr : 804876: inc [ebp]

DBA instr : @[ebp] := @[ebp] + 1



CSML: Example

Example of how a CSML rule works and matches the expression of a DBA instruction

X86 instr : 804876: inc [ebp]

DBA instr : @[ebp] := @[ebp] + 1

@[ebp]+1

804876 @[ebp] := @[ebp]+1

CSML rule : ⋆ :: @[e?] := ⋆ :: e! :: ⋆ ⇒ C

True **True** **False** **True**

Logical term : **(bvadd +)**

CSML: Example

Example of how a CSML rule works and matches the expression of a DBA instruction

X86 instr : 804876: inc [ebp]

DBA instr : @[ebp] := @[ebp] + 1

@[ebp]

804876 @[ebp] := @[ebp]+1

CSML rule : ⋆ :: @[e?] := ⋆ :: e! :: ⋆ ⇒ C

True **True** **False** **True**

Logical term : (bvadd **(select mem**) +)

CSML: Example

Example of how a CSML rule works and matches the expression of a DBA instruction

X86 instr : 804876: inc [ebp]

DBA instr : @[ebp] := @[ebp] + 1

ebp

804876 @[ebp] := @[ebp]+1

CSML rule : ⋆ :: @[e?] := ⋆ :: e! :: ⋆ ⇒ C

True **True** **False** **True**

Logical term : (bvadd (select mem **ebp**) +)

CSML: Example

Example of how a CSML rule works and matches the expression of a DBA instruction

X86 instr : 804876: inc [ebp]

DBA instr : @[ebp] := @[ebp] + 1

804876 | @[ebp] := @[ebp]+1 | **1**

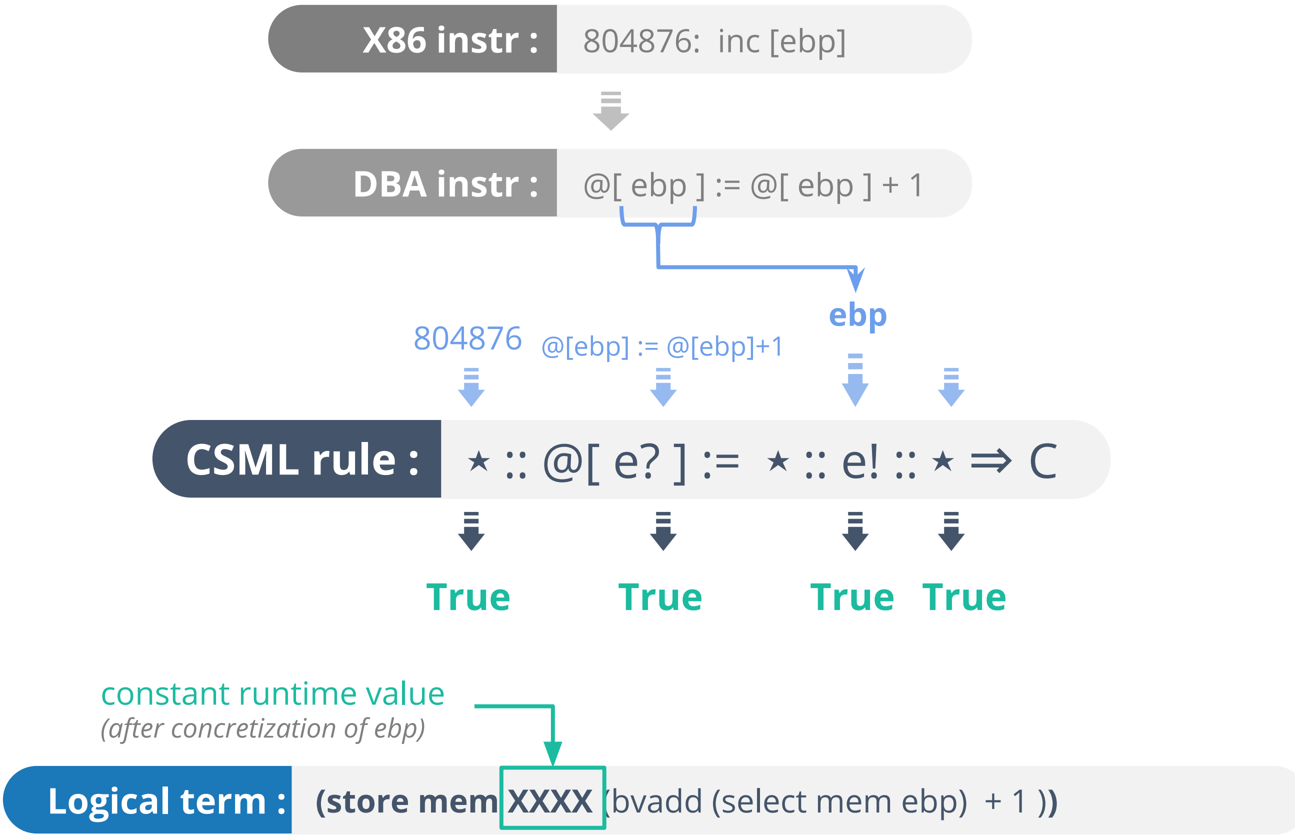
CSML rule : ⋆ :: @[e?] := ⋆ :: e! :: ⋆ ⇒ C

True **True** **False** **True**

Logical term : (bvadd (select mem ebp) + **1**)

CSML: Example

Example of how a CSML rule works and matches the expression of a DBA instruction



CSML: DSE algorithm revisited

How is CSML integrated in the path predicate computation of the DSE algorithm

$$\mathbb{E}\mathbf{xpr}: \quad cst\ \frac{}{\Sigma^*, bv \vdash_{cs\circ} bv, true} \quad var\ \frac{}{\Sigma^*, v \vdash_{cs\circ} \Sigma^*(v), true} \quad binop\ \frac{\Sigma^*, e_1 \vdash_{cs\bullet} \varphi_1, \phi_1 \quad \Sigma^*, e_2 \vdash_{cs\bullet} \varphi_2, \phi_2}{\Sigma^*, e_1 \diamond_b e_2 \vdash_{cs\circ} \varphi_1 \diamond_b^* \varphi_2, \phi_1 \wedge \phi_2}$$

$$unaryop\ \frac{\Sigma^*, e \vdash_{cs\bullet} \varphi_e, \phi_e \quad \varphi' \triangleq \diamond_u^* \varphi_e}{\Sigma^*, \diamond_u e \vdash_{cs\circ} \varphi', \phi_e} \quad @\ \frac{\Sigma^*, e \vdash_{cs\bullet} \varphi_e, \phi_e \quad \varphi \triangleq select(\Sigma^*(Mem), \varphi_e)}{\Sigma^*, @\ e \vdash_{cs\circ} \varphi, \phi_e}$$

$$\mathbb{I}\mathbf{nstr}: \quad goto\ l_1\ \frac{}{l, \Sigma^*, \phi, goto\ l_1 \rightsquigarrow l_1, \Sigma^*, \phi, \Delta(l_1)} \quad l_e - goto\ e\ \frac{\Sigma^*, e \vdash_{cs\bullet} \varphi_e, \phi_e \quad \phi' \triangleq (\phi \wedge \phi_e \wedge to\_val(l_e) = \varphi_e)}{l, \Sigma^*, \phi, goto\ e \rightsquigarrow l_e, \Sigma^*, \phi', \Delta(l_e)}$$

$$T - ite\ \frac{\Sigma^*, e \vdash_{cs\bullet} \varphi_e, \phi_e \quad \phi' \triangleq \phi \wedge \phi_e \wedge \varphi_e}{l, \Sigma^*, \phi, ite(e):\ l_1;\ l_2 \rightsquigarrow l_1, \Sigma^*, \phi', \Delta(l_1)} \quad F - ite\ \frac{\Sigma^*, e \vdash_{cs\bullet} \varphi_e, \phi_e \quad \phi' \triangleq \phi \wedge \phi_e \wedge \neg\varphi_e}{l, \Sigma^*, \phi, ite(e):\ l_1;\ l_2 \rightsquigarrow l_2, \Sigma^*, \phi', \Delta(l_2)}$$

$$var\ assign\ \frac{\Sigma^*, e \vdash_{cs\bullet} \varphi_e, \phi_e \quad \Sigma^*_{new} \triangleq \Sigma^*[v \leftarrow fresh] \quad \phi' \triangleq (\phi \wedge \phi_e \wedge fresh = \varphi_e)}{l, \Sigma^*, \phi, v := e \rightsquigarrow l+1, \Sigma^*_{new}, \phi', \Delta(l+1)}$$

$$@\ assign\ \frac{\Sigma^*, e \vdash_{cs\bullet} \varphi, \phi_e \quad \Sigma^*, e' \vdash_{cs\bullet} \varphi', \phi_{e'} \quad m' \triangleq store(\Sigma^*(Mem), \varphi', \varphi) \quad \phi_m \triangleq (\phi \wedge \phi_e \wedge \phi_{e'} \wedge fresh_m = m')}{l, \Sigma^*, \phi, @\ e' := e \rightsquigarrow l+1, \Sigma^*[Mem \leftarrow fresh_m], \phi_m, \Delta(l+1)}$$

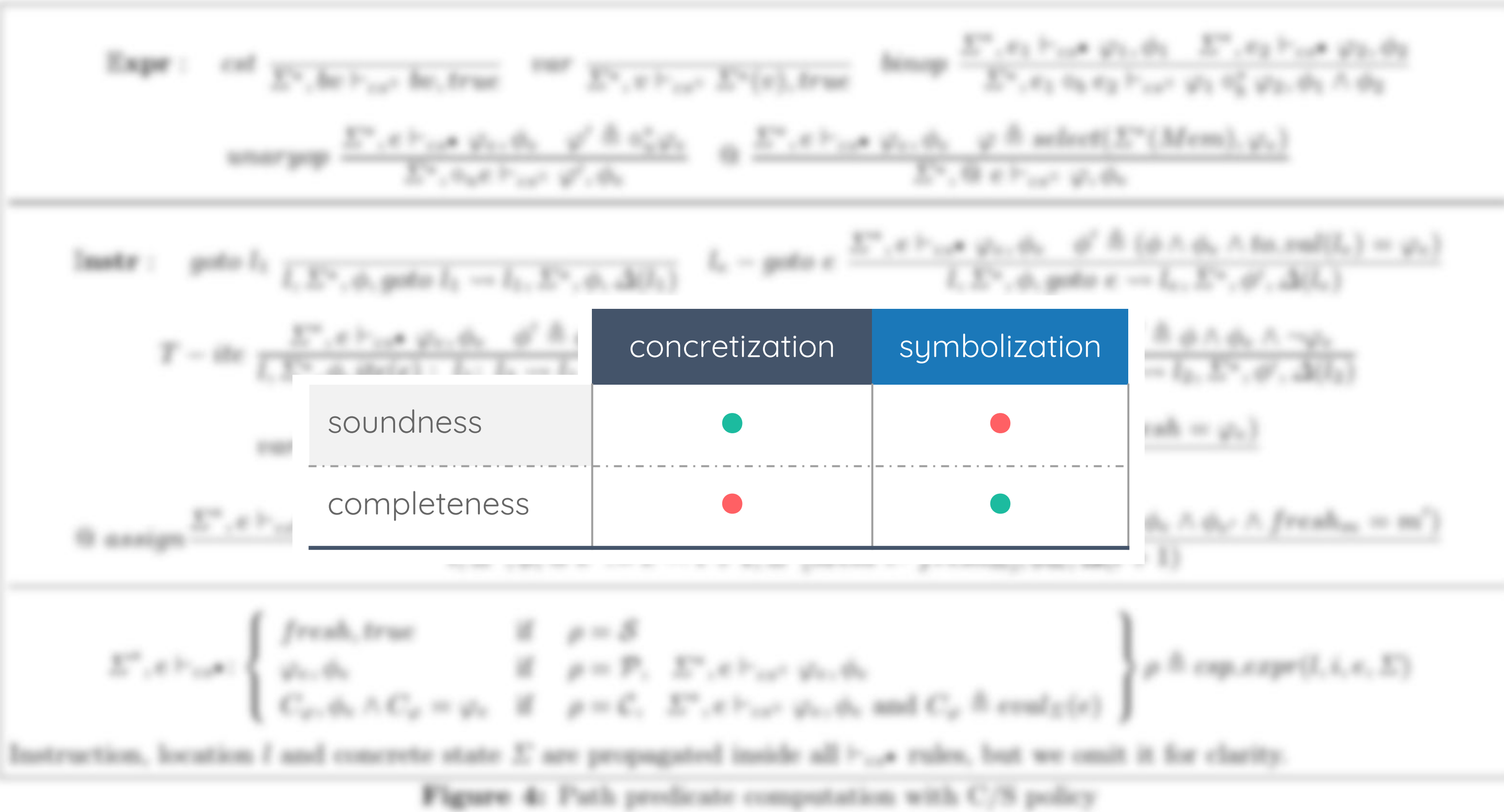
$$\Sigma^*, e \vdash_{cs\bullet}: \left\{ \begin{array}{lll} fresh, true & \text{if} & \rho = \mathcal{S} \\ \varphi_e, \phi_e & \text{if} & \rho = \mathcal{P}, \quad \Sigma^*, e \vdash_{cs\circ} \varphi_e, \phi_e \\ C_\varphi, \phi_e \wedge C_\varphi = \varphi_e & \text{if} & \rho = \mathcal{C}, \quad \Sigma^*, e \vdash_{cs\circ} \varphi_e, \phi_e \text{ and } C_\varphi \triangleq eval_\Sigma(e) \end{array} \right\} \rho \triangleq csp\_expr(l, i, e, \Sigma)$$

Instruction, location l and concrete state Σ are propagated inside all $\vdash_{cs\bullet}$ rules, but we omit it for clarity.

Figure 4: Path predicate computation with C/S policy

CSML: DSE algorithm revisited

How is CSML integrated in the path predicate computation of the DSE algorithm



	concretization	symbolization
soundness	● (green)	● (red)
completeness	● (red)	● (green)

CSML: Results

Example of how a CSML rule works and match the expression of a DBA instruction

Flexible C/S specification mechanism:

- clear formal semantic & integration into DSE
- **encode all literature policies**
- can be improved with various extensions

[first] Quantitative Evaluation:

- 5 differents policies on memory
- on some SAMATE benchmarks and all coreutils *(169 programs)*
- **rule matching computation cost negligible**, avg: 1.45% *(amortized by solving)*
- **significant time difference between policies, but no clear winner**

⇒ Validates the need for a flexible mechanism

Forward DSE allows to check **feasibility** properties

- find new targets for dynamic jumps
- cover a new branch

If we want to check **infeasibility** properties, better to go **backward**

- dynamic jump closure
- opaque predicates, stack tampering
- conditional self-modification etc...

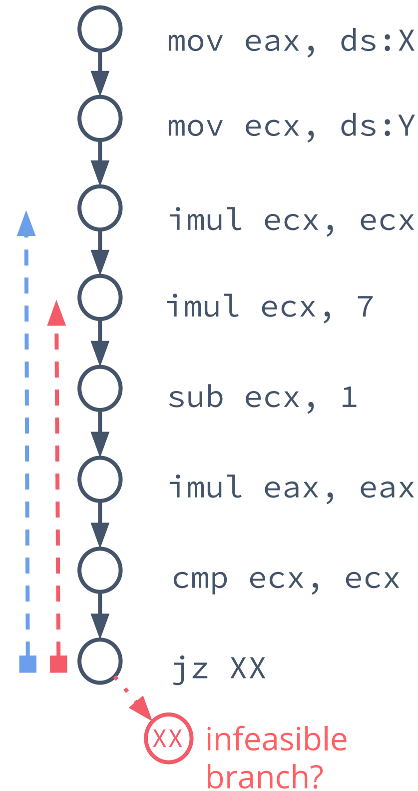
Backward-Bounded DSE: General idea

How it can be helpful for solving obfuscation problems.

- **Goal:** check that the branch to XX is infeasible
  - **false negative**

    (still feasible w.r.t. ecx, eax)
  - **true positive**

    (backtrack enough constraints to prove the infeasibility)



Insight: Turning a potential infinite set of paths to a finite path suffixes

BB-DSE: Call stack tampering

BB-DSE applied on call stack tampering when with multiple paths

- ## Goal

  check that the return address cannot be tampered by the function

  - **false negative**

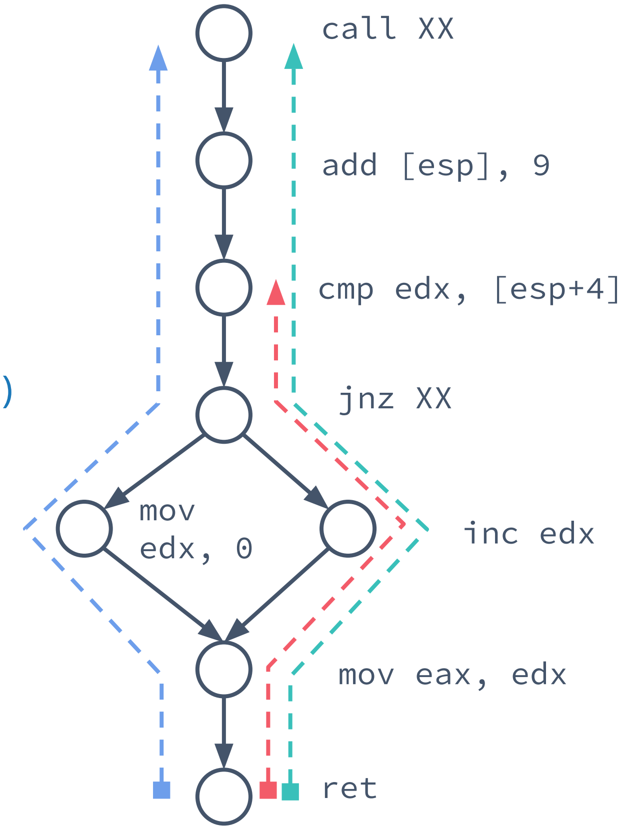
    miss the tampering (too small bound)

  - **correct**

    find the tampering

  - **+ complete**

    validate the tampering for all paths

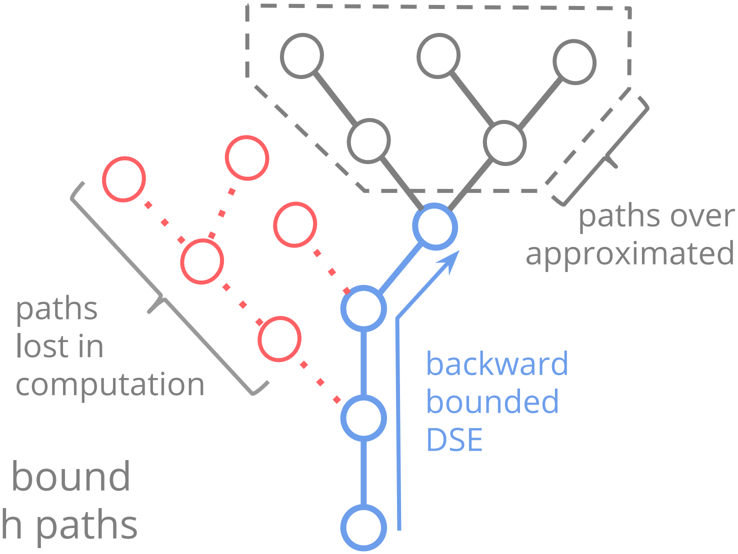


Backward-Bounded DSE [S&P17(submitted)]

Overall behavior, properties and strength

Summary:

- backward for **infeasibility**
- bounded reasoning for **scale**
- adaptable bound *(for the need)*
- **dynamic** for **robustness**
  (hence false positive)



Shortcomings:

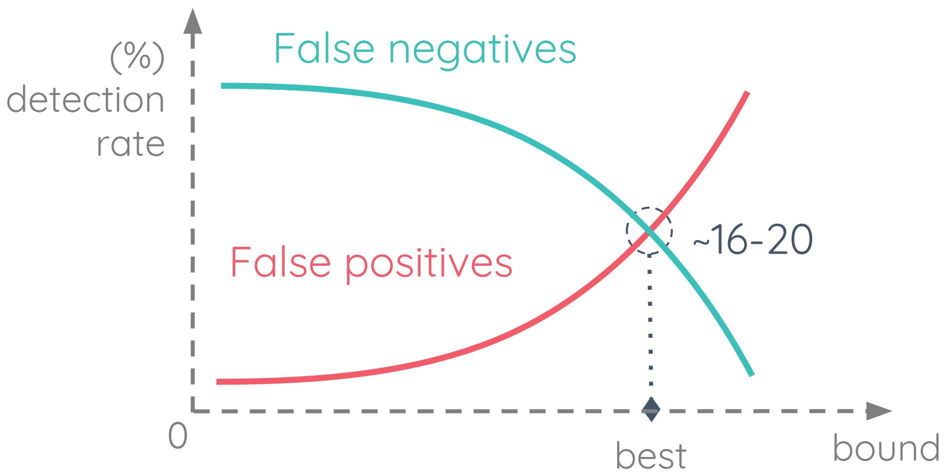
- False negative (FN): too small bound
- False positive (FP): not enough paths

	(forward) DSE	bb-DSE
feasibility queries	● (green)	● (red)
infeasibility queries	● (red)	● (green)
scale	● (orange)	● (green)

BB-DSE: Bound selection

Overall behavior, properties and strength

- **Need to be adapted to the problem to solve**
- **Application to obfuscation:**
  - Call stack tampering: ret →call
  - Opaque predicates: Trade-off FP/FN



FN: OP missed (backtracking not enough)

empiric results obtained through benchmarking

FP: not OP but infeasible w.r.t. path taken

BB-DSE: Results

Overall behavior, properties and strength

- **Scalability:**
  - get rid of path length issue
  - k bound allows to adjust to “hardness” of formulas
- **Evaluation** (ground truth value):
  - Opaque predicates on test files obfuscated with O-LLVM
  - Call stack tampering on coreutils obfuscated with Tigress
  - Yield very few FP /FN *(3.17% with k=16)*
- **Performances** (against forward DSE on a 115K instrs trace)

	bound k	#UNSAT	#Timeout	Total time
forward DSE	/	**7749**	2460	**17h43m**
backward DSE	∞	7748	2461	17h48m
BB-DSE	100	7406	0	18m78s
BB-DSE	20	**54**	0	**4m14s**

too many false positives

BB-DSE: Results

Overall behavior, properties and strength

- **Scalability:**
  - get rid of path length issue
  - k bound allows to adjust to “hardness” of formulas
- **Evaluation** (ground truth value):
  - Opaque predicates on test files obfuscated with O-LLVM
  - Call stack tampering on coreutils obfuscated with Tigress
  - Yield very few FP /FN *(3.17% with k=16)*
- **Performances** (against forward DSE on a 115K instrs trace)

large scale benchmarks given in section (case-studies)

	bound k	#UNSAT	#Timeout	Total time
forward DSE	/	**7749**	2460	**17h43m**
backward DSE	∞	7748	2461	17h48m
BB-DSE	100	7406	0	18m78s
BB-DSE	20	**54**	0	**4m14s**

too many false positives

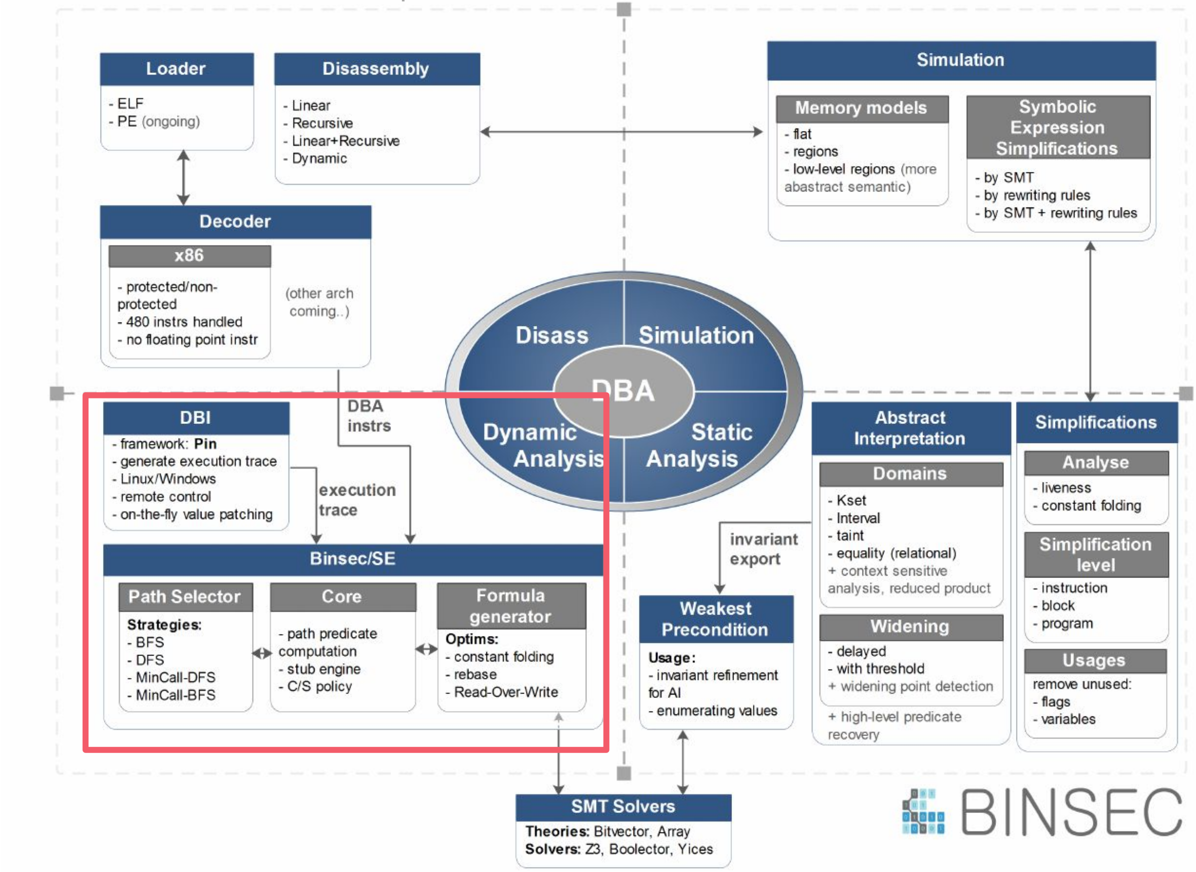
3.

Implementation

[IDA|Pin|Bin] sec

Binsec platform overview

Overview of Binsec, all its component and interaction between them



Intermediate Representation (IR)

Encode the semantic (and all side-effect) of a machine instruction

Avantages

- bitvector size statically known
- side-effect free
- bit-precise

Shortcomings

- no floats
- no thread modeling
- no self-modification
- no exception
- x86(32) only

Language DBA

`bv`	`bitvector` *(constant value)*
`l :=`	`loc` *(addr + offset)*

`e :=`	`v | bv | ⊥ | ⊤`
	`@ [ e ]` *(read memory)*
	`e ◇ e | ◇ e`

`lhs :=`	`v` *(variable)* `v{i,j}` *(extraction)* `@[ e ]` *(write memory)*

`inst :=`	`lhs := e`
	`goto e | goto l`
	`ite (c)? goto l1; goto l2`
	`assert e | assume e ..`

Many other similar IR: REIL: BIL, VEX, LLVM IR, MIASM IR, Binary Ninja IR

DBA: Example

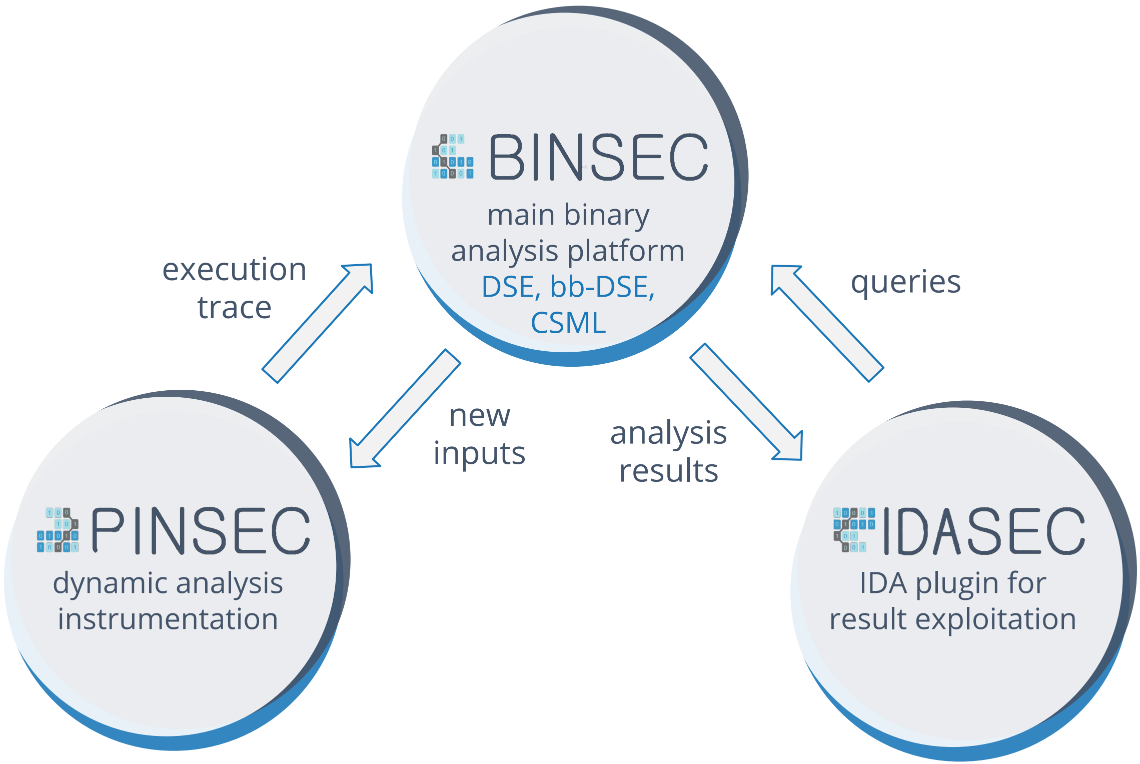
Example of how an instruction is modeled in the DBA language

Decoding: `imul eax, dword ptr[esi+0x14], 7`

res32	:=	$@[esi_{(32)} + 0x14_{(32)}] * 7_{(32)}$
temp64	:=	$(exts\ @[esi_{(32)} + 0x14_{(32)}]\ 64) * (exts\ 7_{(32)}\ 64)$
OF	:=	$(temp64_{(64)} \neq (exts\ res32_{(32)}\ 64))$
SF	:=	$\bot$
ZF	:=	$\bot$
CF	:=	$OF_{(1)}$
eax	:=	$res32_{(32)}$

Binsec/SE: Platform architecture [SANER16]

Three components of the Dynamic Symbolic Execution engine



PINSEC

Pinsec dynamic instrumentation based on Pin 2.14-71313 to generate execution trace

Execution Trace

As a protobuf file containing all the runtime values

Limit Instrumentation

either in time (with timeout) or in space (number of instructions)

Windows & Linux

Tested on Windows 7 and Debian (kernel officially compatible < 4.0)

Configuration JSON

All parameters can be specified in a JSON file for reproducibility

On-The-Fly Patching

Allow to patch, registers or memory addresses at any moment of execution

Function Stubs

Allow to retrieve function parameters of known library calls

Remote Control

Provide more interaction with breakpoints and value patching *(beta)*

Polymorphism tracking

Track self-modification occurring during execution

Streaming Trace

Streaming instructions in real-time to Binsec for online analysis

still lacks many anti-debug/anti-VM countermeasures

IDASEC

IDA Pro (from 6.4) plugin to assist reverse-engineering tasks

Goal: Leveraging Binsec features into IDA (triggering analyses and post-processing)

DBA **decoding**

Decode any instruction and shows graphically the DBA semantic of the instruction

Dynamic disassembly

Allows to disassemble in IDA by following the execution trace. (For now, stop on the first self-modification layer)

Reading execution traces

Load execution trace, generated by Pinsec, shows runtime values, allows to vizualize the path taken on the CFG etc.

Binsec **remote** connection

Allows to trigger analyses on Binsec and to retrieve results for post-analysis data exploitation.

BINSEC/SE

Dynamic Symbolic Execution engine performing the core execution

Path **selection**

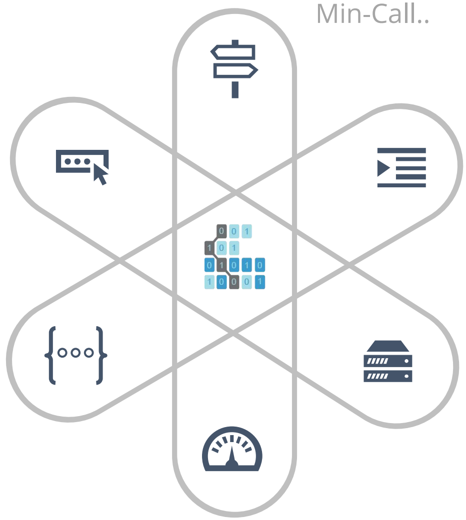
for coverage with DSE thanks to different strategies DFS, BFS, Min-Call..

CSML policy engine

generic C/S policy engine implementing the CSML language for dynamic modulation of C/S

Stub engine

allows to over-approximate side-effect of library call without executing them



Configurable JSON

Analyses configurable by a JSON file (common with Pinsec)

Multiple **Solvers**

Supports officially, Z3, boolector, Yices, CVC4 by using the common SMTLIB2 format

predicate **optimizations**

Implement various path predicate optimizations providing a great performances

Many other DSE engines: Mayhem (ForAllSecure)**, Triton** (QuarksLab)**, S2E …**

Optimizations: for path predicate

Practical examples of optimizations

call XXX	...
pop ebp	ebp0 := (select mem0 esp0)
	esp1 := esp0 + 0x20
inc ebp	ebp1 := ebp0 + 1
	OF0 := ebp0 +1
	SF0 := 0
	...
push ebp	mem1 := (store mem0 (esp1 - 0x20) ebp1)
	esp2 := esp1 - 0x20
mov eax, [804856]	eax0 := (select mem1 0x084858)
ret	(assert (= (select mem1 esp2) ebp0)

Query

Check that the ret value read in memory is equal to ebp0 meant to hold the ret address

Optimizations: for path predicate

Practical examples of optimizations

Optimizations:

- rebase

rebase
Rebase a new symbol definition by reusing older definition of it.

call XXX	...
pop ebp	ebp0 := (select mem0 esp0)
	esp1 := esp0 + 0x20
inc ebp	ebp1 := ebp0 + 1
	OF0 := ebp0 +1
	SF0 := 0
	...
push ebp	mem1 := (store mem0 (esp1 - 0x20) ebp1)
	esp2 := esp1 - 0x20
mov eax, [804856]	eax0 := (select mem1 0x084858)
ret	(assert (= (select mem1 esp2) ebp0)

Optimizations: for path predicate

Practical examples of optimizations

Optimizations:

- rebase

call XXX	...
pop ebp	ebp0 := (select mem0 esp0)
	esp1 := esp0 + 0x20
inc ebp	ebp1 := ebp0 + 1
	OF0 := ebp0 +1
	SF0 := 0
	...
push ebp	mem1 := (store mem0 (esp1 - 0x20) ebp1)
	esp2 := (**esp0**
mov eax, [804856]	eax0 := (select mem1 0x084858)
ret	(assert (= (select mem1 **esp0**) ebp0)

rebase

Rebase a new symbol definition by reusing older definition of it.

Optimizations: for path predicate

Practical examples of optimizations

Optimizations:

- rebase
- Read-Over-Write #1

`call XXX`	`...`
`pop ebp`	`ebp0 := (select mem0 esp0)` `esp1 := esp0 + 0x20`

`inc ebp`	`ebp1 := ebp0 + 1`
	`OF0  := ebp0 +1`
	`SF0  := 0`
	`...`

`push ebp`	`mem1 := (store mem0 (esp1 - 0x20) ebp1)` `esp2 := esp0`
`mov eax, [804856]`	`eax0 := (select mem1 0x084858)`
`ret`	`(assert (= (select mem1 esp0) ebp0)`

`0x084858 =? (esp1-0x20)`
`(cannot compare)`

Read-Over-Write #1

A select in an array can be replace by the value written **iff** performed on the same logical indexes

Optimizations: for path predicate

Practical examples of optimizations

Optimizations:

- rebase
- Read-Over-Write #1

Read-Over-Write #1

A select in an array can be replace by the value written **iff** performed on the same logical indexes

`call XXX`	`...`
`pop ebp`	`ebp0 := (select mem0 esp0)` `esp1 := esp0 + 0x20`

`inc ebp`	`ebp1 := ebp0 + 1`
	`OF0  := ebp0 +1`
	`SF0  := 0`
	`...`

`push ebp`	`mem1 := (store mem0 (esp1 − 0x20) ebp1)` `esp2 := esp0`
`mov eax, [804856]`	`eax0 := (select mem1 0x084858)`
`ret`	`(assert (= (select mem1 esp0) ebp0)`

`esp0 == (esp1 − 0x20)`
`because:`
`esp1 = esp0 + 0x20`
`(same base)`

Optimizations: for path predicate

Practical examples of optimizations

Optimizations:

- rebase
- Read-Over-Write #1

Read-Over-Write #1

A select in an array can be replace by the value written **iff** performed on the same logical indexes

call XXX	...
pop ebp	ebp0 := (select mem0 esp0)
	esp1 := esp0 + 0x20
inc ebp	ebp1 := ebp0 + 1
	OF0 := ebp0 +1
	SF0 := 0
	...
push ebp	mem1 := (store mem0 (esp1 - 0x20) ebp1)
	esp2 := esp0
mov eax, [804856]	eax0 := (select mem1 0x084858)
ret	(assert (= **ebp1** ebp0)

Optimizations: for path predicate

Practical examples of optimizations

Optimizations:

- rebase
- Read-Over-Write #1
- constant propagation

constant propagation

Standard optimization evaluating all operations involving only constant values.

call XXX	**esp0 := 0x6ff68**
pop ebp	ebp0 := (select mem0 esp0)
	esp1 := esp0 + 0x20
inc ebp	ebp1 := ebp0 + 1
	OF0 := ebp0 +1
	SF0 := 0
	...
push ebp	mem1 := (store mem0 esp1 ebp1)
	esp2 := esp0
mov eax, [804856]	eax0 := (select mem1 0x084858)
ret	(assert (= ebp1 ebp0))

Optimizations: for path predicate

Practical examples of optimizations

Optimizations:

- rebase
- Read-Over-Write #1
- constant propagation

constant propagation

Standard optimization evaluating all operations involving only constant values.

call XXX	**esp0 := 0x6ff68**
pop ebp	ebp0 := (select mem0 **0x6ff68**)
	esp1 := **0x6ff88**
inc ebp	ebp1 := ebp0 + 1
	OF0 := ebp0 +1
	SF0 := 0
	...
push ebp	mem1 := (store mem0 **0x6ff88** ebp1)
	esp2 := **0x6ff68**
mov eax, [804856]	eax0 := (select mem1 0x084858)
ret	(assert (= ebp1 ebp0)

Optimizations: for path predicate

Practical examples of optimizations

Optimizations:

- rebase
- Read-Over-Write #1
- constant propagation
- Read-Over-Write #2

Read-Over-Write #2

For a select, if the index of the previous store is disjoint, the select can be performed on the previous array.

call XXX	esp0 := 0x6ff68
pop ebp	ebp0 := (select mem0 0x6ff68)
	esp1 := 0x6ff88
inc ebp	ebp1 := ebp0 + 1
	OF0 := ebp0 +1
	SF0 := 0
	...
push ebp	mem1 := (store mem0 0x6ff88 ebp1)
	esp2 := 0x6ff68
mov eax, [804856]	eax0 := (select mem1 0x084858)
ret	(assert (= ebp1 ebp0))

disjoint

Optimizations: for path predicate

Practical examples of optimizations

Optimizations:

- rebase
- Read-Over-Write #1
- constant propagation
- Read-Over-Write #2

`call XXX`	`esp0 := 0x6ff68`
`pop ebp`	`ebp0 := (select mem0 0x6ff68)`
	`esp1 := 0x6ff88`
`inc ebp`	`ebp1 := ebp0 + 1`
	`OF0  := ebp0 +1`
	`SF0  := 0`
	`...`
`push ebp`	`mem1 := (store mem0 0x6ff88 ebp1)`
	`esp2 := 0x6ff68`
`mov eax, [804856]`	`eax0 := (select` **`mem0`** `0x084858)`
`ret`	`(assert (= ebp1 ebp0))`

Read-Over-Write #2

For a select, if the index of the previous store is disjoint, the select can be performed on the previous array.

Optimizations: for path predicate

Practical examples of optimizations

Optimizations:

- rebase
- Read-Over-Write #1
- constant propagation
- Read-Over-Write #2
- memory flattening

memory flattening

Optimization removing the array theory if all select operation performed on initial memory (mem0).

call XXX	esp0 := 0x6ff68
pop ebp	ebp0 := (select mem0 0x6ff68)
	esp1 := 0x6ff88
inc ebp	ebp1 := ebp0 + 1
	OF0 := ebp0 +1
	SF0 := 0
	...
push ebp	mem1 := (store mem0 0x6ff88 ebp1)
	esp2 := 0x6ff68
mov eax, [804856]	eax0 := (select mem0 0x084858)
ret	(assert (= ebp1 ebp0)

all select in memory performed on mem0

Optimizations: for path predicate

Practical examples of optimizations

Optimizations:

- rebase
- Read-Over-Write #1
- constant propagation
- Read-Over-Write #2
- memory flattening

memory flattening

Optimization removing the array theory if all select operation performed on initial memory (mem0).

call XXX	esp0 := 0x6ff68
pop ebp	ebp0 := **mem_dw_6ff68**
	esp1 := 0x6ff88
inc ebp	ebp1 := ebp0 + 1
	OF0 := ebp0 +1
	SF0 := 0
	...
push ebp	mem1 := (store mem0 0x6ff88 ebp1)
	esp2 := 0x6ff68
mov eax, [804856]	eax0 := **mem_dw_084858**
ret	(assert (= ebp1 ebp0)

bitvector symbols, representing memory cells of initial memory

Optimizations: for path predicate

Practical examples of optimizations

Optimizations:

- rebase
- Read-Over-Write #1
- constant propagation
- Read-Over-Write #2
- memory flattening
- backward pruning

backward pruning

Remove all unused terms for the formula to solve

call XXX	~~esp0 := 0x6ff68~~	made removable by: cst prop
pop ebp	ebp0 := mem_dw_6ff68	
	~~esp1 := 0x6ff88~~	made removable by: cst prop + RoW
inc ebp	ebp1 := ebp0 + 1	
	~~OF0 := ebp0 +1~~	unused
	~~SF0 := 0~~	unused
	...	
push ebp	~~mem1 := (store mem0 0x6ff88~~	removable by: RoW + mem flat
	~~esp2 := 0x6ff68~~	made removable by: rebase + cst prop
mov eax, [804856]	~~eax0 := mem_dw_084858~~	unused
ret	(assert (= ebp1 ebp0)	

Optimizations: for path predicate

Practical examples of optimizations

Optimizations:

- rebase
- Read-Over-Write #1
- constant propagation
- Read-Over-Write #2
- memory flattening
- backward pruning

backward pruning
Remove all unused terms for the formula to solve

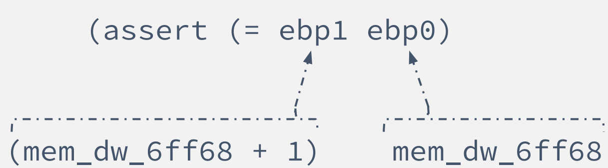
`call XXX`	
`pop ebp`	`ebp0 := mem_dw_6ff68`
`inc ebp`	`ebp1 := ebp0 + 1`
`push ebp`	
`mov eax, [804856]`	
`ret`	`(assert (= ebp1 ebp0))`

Optimizations: for path predicate

Example of how an instruction is modeled in the DBA language

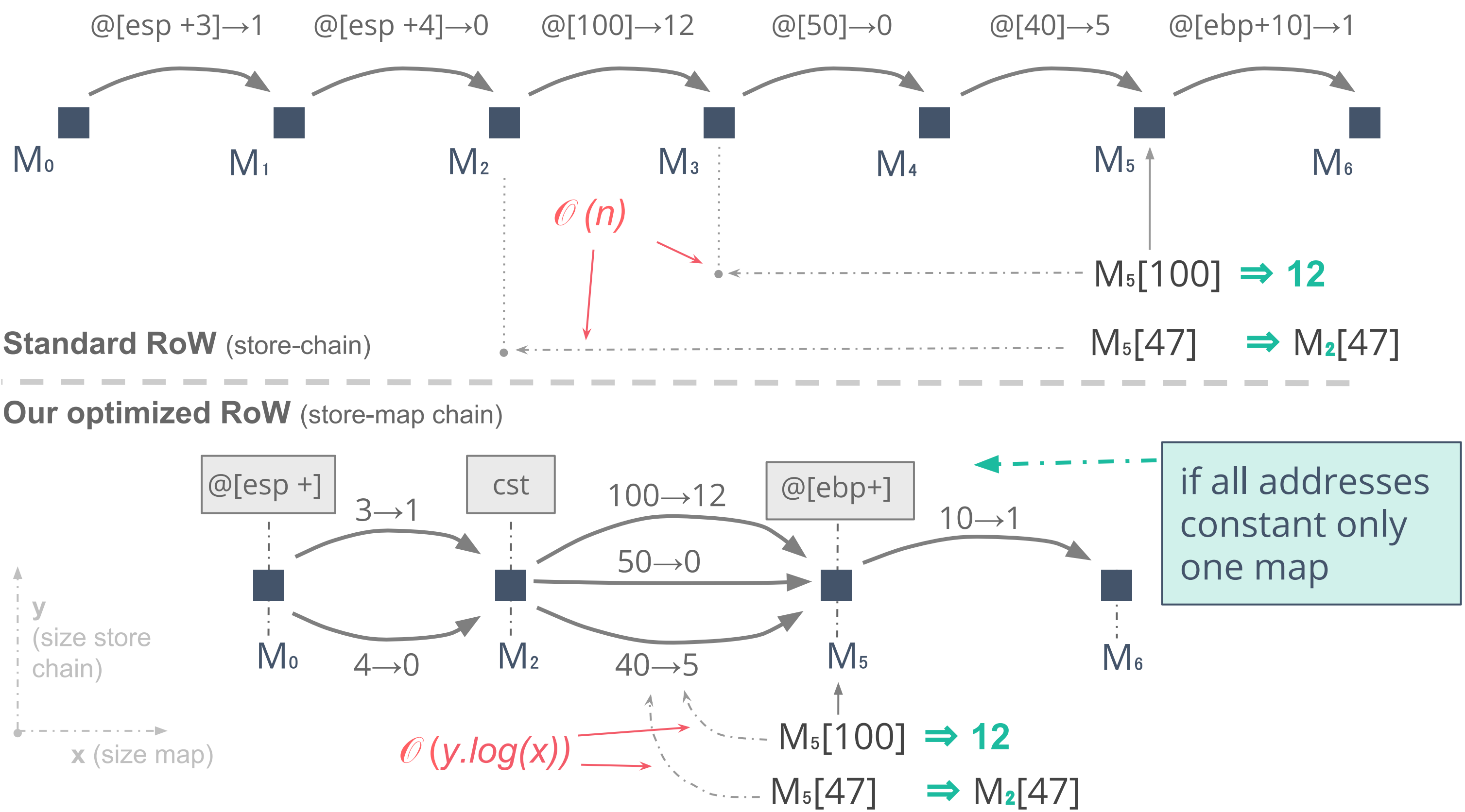
Without optimization

After optimization



Read-Over-Write: Design

How we turned a standard RoW quadratic complexity into n (log x)



Read-Over-Write: Discussions

What are the difference in complexity and time depending on the policy

Complexity:

	standard RoW	optimized RoW
constant addresses	n x m	n x log(m)
symbolic addresses	n x m	n x y x log(z)

m: nb store
n: nb load
y: nb maps
z: max card map

Benchmark on a path predicate (337k instrs):

	standard RoW	optimized RoW
constant addresses	79.32s	26.61s
symbolic addresses	**40.84s**	26.97s

The structure can be enhanced to improve the base comparison (in progress)

4.

Analysis Combinations

Analysis Combinations

The three combinations designed and implemented during the course of my PhD

Software Testing
Infeasibility test requirements

Abstract Interpretation and Weakest-Precondition Calculus greybox combination.

Use-After-Free
Vulnerability Discovery

Abstract Interpretation and Dynamic Symbolic Execution to detect and validate UaF

Sparse
Disassembly

Dynamic disassembly improved by static disassembly guided by DSE and obfuscation data

98% Infeasible test objectives detected [ICST15]

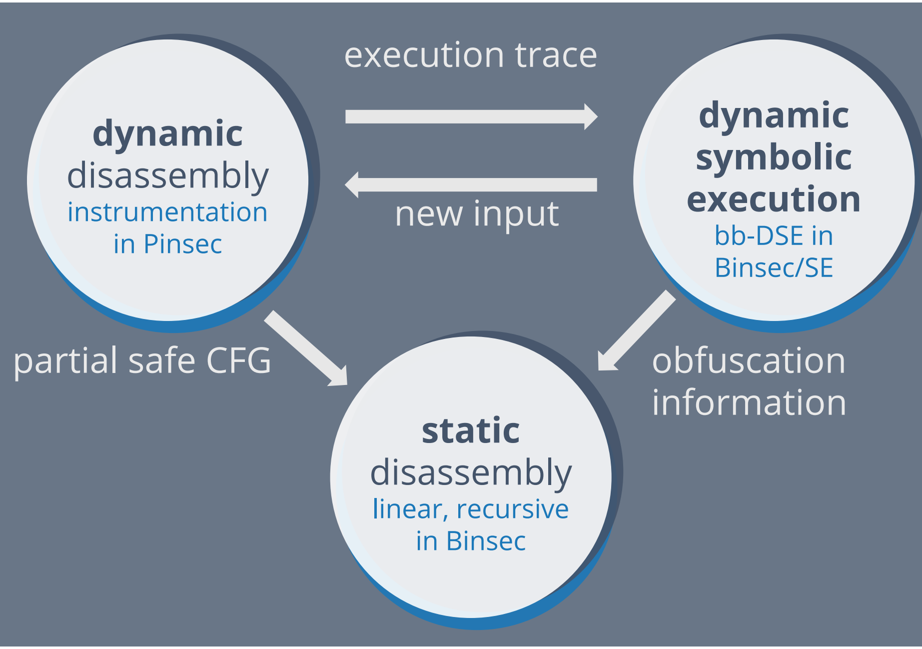
CVE-2015-5221 validated in JasPer.
joint work CEA, Verimag with Josselin Feist [SSPREW16]

~50% gain on a real world malware [S&P17](submitted)

Sparse disassembly: Components

Main components of the sparse disassembly combination

Goal: enlarging disassembly in a **safe and more precise** manner

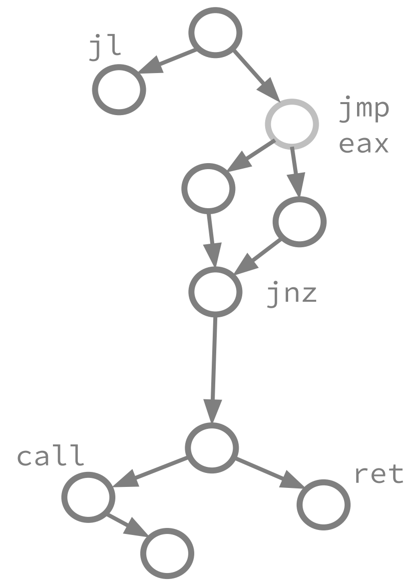


➡ The ultimate goal is to provide a semantic-aware disassembly based on information computed by symbolic execution

Sparse disassembly: Application

Result of applying the combination using obfuscation related data

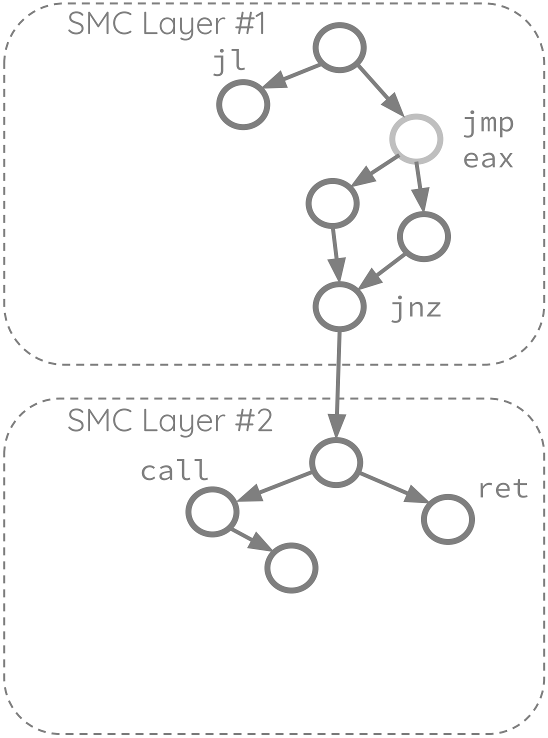
- ◦ ■ + ■ safe dynamic disassembly with dynamic jumps



Sparse disassembly: Application

Result of applying the combination using obfuscation related data

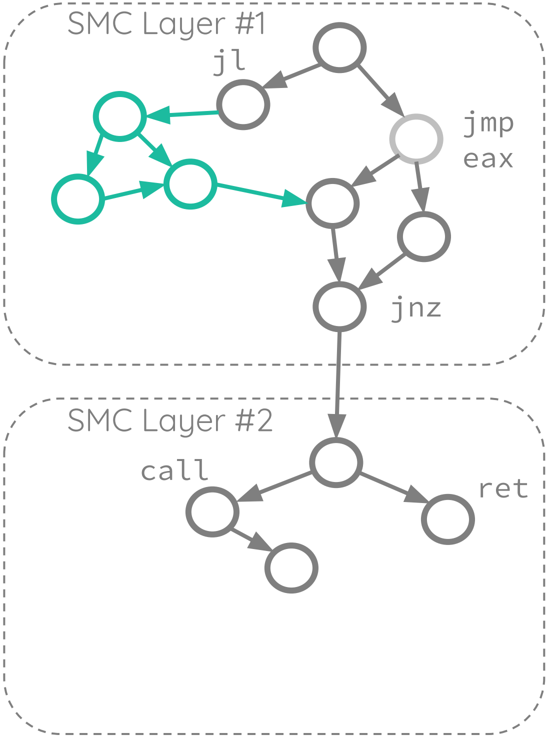
- ■ + ■ safe dynamic disassembly with dynamic jumps
- ⬚ multiple self-modification segmentation



Sparse disassembly: Application

Result of applying the combination using obfuscation related data

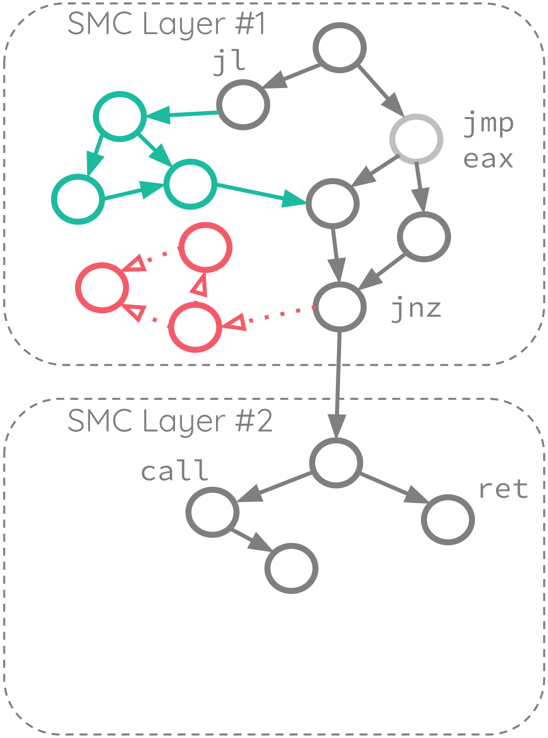
- ■ + ■ safe dynamic disassembly with dynamic jumps
- ⬚ multiple self-modification segmentation
- ■ enlarge partial CFG on genuine conditional jump



Sparse disassembly: Application

Result of applying the combination using obfuscation related data

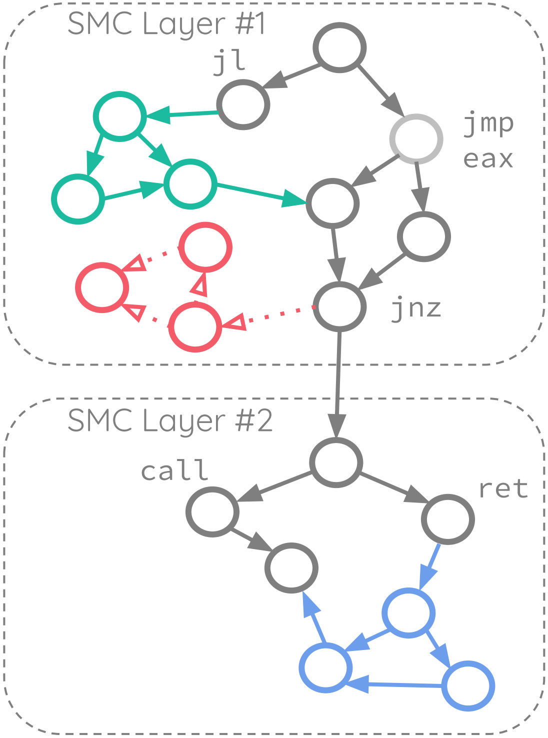
- ◼ + ◼ safe dynamic disassembly with dynamic jumps
- ⬚ multiple self-modification segmentation
- ◼ enlarge partial CFG on genuine conditional jump
- ◼ do not disassemble dead branch of opaque predicate



Sparse disassembly: Application

Result of applying the combination using obfuscation related data

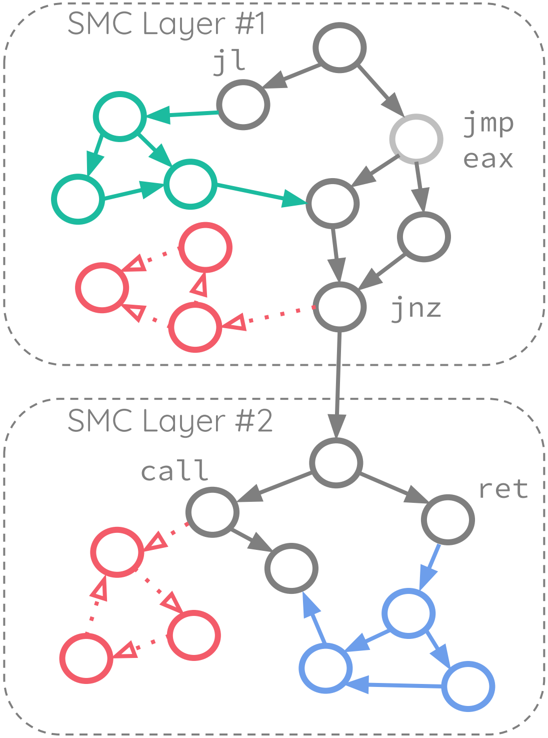
- ■ + ■ safe dynamic disassembly with dynamic jumps
- ⸬ multiple self-modification segmentation
- ■ enlarge partial CFG on genuine conditional jump
- ■ do not disassemble dead branch of opaque predicate
- ■ disassemble the target of tampered ret



Sparse disassembly: Application

Result of applying the combination using obfuscation related data

- + safe dynamic disassembly with dynamic jumps
- multiple self-modification segmentation
- enlarge partial CFG on genuine conditional jump
- do not disassemble dead branch of opaque predicate
- disassemble the target of tampered ret
- do not disassemble the return site of tampered ret



Sparse disassembly: Results

Disassembly results obtained with sparse disassembly

Benchmark:

- compared the disassembly coverage with Objdump, IDA, Binsec
- a controlled environment *(5 toy examples, 5 coreutils from State-of-the-Art)*
- opaque predicates, call stack tampering *(separately)*

Results: Opaque predicates

sample	no obf	perfect	IDA	Objdump	Binsec **(sparse)**	gain (vs IDA)
simple-if	37	**185**	240	244	**185**	23.23%
huffman	558	**3226**	3594	3602	**3226**	10.26%
mat_mult	249	**854**	1075	1080	**854**	20.67%
bin_search	105	**833**	1110	1115	**833**	24.95%
bubble_sort	121	**1026**	1531	1537	**1026**	32.98%

On-going work, functionalities not yet implemented (disassembly across waves)

Sparse disassembly: Results

Disassembly results obtained with sparse disassembly

Benchmark:

- compared the disassembly coverage with Objdump, IDA, Binsec
- a controlled environment *(5 toy examples, 5 coreutils from State-of-the-Art)*
- opaque predicates, call stack tampering *(separately)*

Results: Call stack tampering

sample	no obf	perfect	IDA	Objdump	Binsec **(sparse)**	gain (vs IDA)
simple-if	37	**83**	95	98	**83**	14.45%
huffman	558	**659**	678	683	**659**	2.80%
mat_mult	249	**461**	524	533	**461**	12.0%
bin_search	105	**207**	231	238	**207**	10.39%
bubble_sort	121	**170**	182	185	**170**	6.6%

On-going work, functionalities not yet implemented (disassembly across waves)

5.

Case-Studies

Packers & X-Tunnel

Packers: Case-study #1

Evaluation aiming at finding opaque predicates and call stack tampering

- **Evaluation of 33 packers**
  (packed with a stub binary)

- **Why packers ?**
  - realistic protections
  - do contain obfuscation
  - usually first protection layer
    (if not the single)

- Looking for (with bb-DSE):
  - **opaque predicates**
  - **call stack tampering**
  - record of self-modification layers

- **Goal:**
  - perform a systematic and fully automated evaluation of BB-DSE on packers *(for robustness, scale etc)*

Packers: Analysis results

packers	trace len.	#proc	#th	#SMC	opaque predicates OK	opaque predicates OP	call stack tampering OK	call stack tampering tamper
ACProtect v2.0	1.8M	1	1	4	83	159	0	48
ASPack v2.12	377K	1	1	2	168	24	11	6
Crypter v1.12	1.1M	1	1	1	399	24	125	78
Expressor	635K	1	1	1	81	8	14	0
FSG v2.0	68k	1	1	1	24	1	6	0
Mew	59K	1	1	1	28	1	6	1
PE Lock	2.3M	1	1	6	95	90	4	3
RLPack	941K	1	1	1	46	2	14	0
TELock v0.51	406K	1	1	5	5	2	3	1
Upack v0.39	711K	1	1	2	41	1	7	1

- Several have no such obfuscation, NeoLite, nPack, Packman, PE Compact ....
- Several packers still evade the DBI, Armadillo, BoxedApp, EP Protector, VMProtect....
- 3 reached the 10M instructions limit, Enigma, svk, Themida

Packers: Analysis results

packers	trace len.	#proc	#th	#SMC	opaque predicates OK	OP	call stack tampering OK	tamper
ACProtect v2.0	1.8M	1	[illegible]	[illegible]	[illegible]	159	0	48
ASPack v2.12	377K	1	[illegible]	[illegible]	[illegible]	24	11	6
Crypter v1.12	1.1M	1	[illegible]	[illegible]	[illegible]	24	125	78
Expressor	635K	1	1	1	81	8	14	0
FSG v2.0	68k	1	1	1	24	1	6	0
Mew	59K	1	1	1	28	1	6	1
PE Lock	2.3M	1	1	6	95	90	4	3
RLPack	941K	1	1	1	46	2	14	0
TELock v0.51	406K	1	1	5	5	2	3	1
Upack v0.39	711K	1	1	2	41	1	7	1

The technique scale on significant traces

- Several have no such obfuscation, NeoLite, nPack, Packman, PE Compact ....
- Several packers still evade the DBI, Armadillo, BoxedApp, EP Protector, VMProtect....
- 3 reached the 10M instructions limit, Enigma, svk, Themida

Packers: Analysis results

packers	trace len.	#proc	#th	#SMC	opaque predicates OK	opaque predicates OP	call stack tampering OK	call stack tampering tamper
ACProtect v2.0	1.8M	1	[illegible]	[illegible]	[illegible]	159	0	48
ASPack v2.12	377K	1	[illegible]	[illegible]	[illegible]	24	11	6
Crypter v1.12	1.1M	1	[illegible]	[illegible]	[illegible]	24	125	78
Expressor	635K	1	1	[illegible]	[illegible]	[illegible]	14	0
FSG v2.0	68k	1	1	[illegible]	[illegible]	[illegible]	6	0
Mew	59K	1	1	1	28	1	6	1
PE Lock	2.3M	1	1	6	95	90	4	3
RLPack	941K	1	1	1	46	2	14	0
TELock v0.51	406K	1	1	5	5	2	3	1
Upack v0.39	711K	1	1	2	41	1	7	1

The technique scale on significant traces

Many true positives. Some packers are using it intensively

- Several have no such obfuscation, NeoLite, nPack, Packman, PE Compact ....
- Several packers still evade the DBI, Armadillo, BoxedApp, EP Protector, VMProtect....
- 3 reached the 10M instructions limit, Enigma, svk, Themida

Packers: Analysis results

packers	trace len.	#proc	#th	#SMC	opaque predicates		call stack tampering	
					OK	OP	OK	tamper
ACProtect v2.0	1.8M	1	[illegible]	[illegible]	[illegible]	159	0	48
ASPack v2.12	377K	1	[illegible]	[illegible]	[illegible]	24	11	6
Crypter v1.12	1.1M	1	[illegible]	[illegible]	[illegible]	24	125	78
Expressor	635K	1	1	[illegible]	[illegible]	[illegible]	14	0
FSG v2.0	68k	1	1	[illegible]	[illegible]	[illegible]	6	0
Mew	59K	1	1	1	28	1	6	1
PE Lock	2.3M	1	1	6	95	90	4	3
RLPack	941K	1	1	[illegible]	[illegible]	[illegible]	14	0
TELock v0.51	406K	1	1	[illegible]	[illegible]	[illegible]	3	1
Upack v0.39	711K	1	1	[illegible]	[illegible]	[illegible]	7	1

The technique scale on significant traces

Many true positives. Some packers are using it intensively

Packers using ret to perform the final tail transition to the entrypoint

- Several have no such obfuscation, NeoLite, nPack, Packman, PE Compact ....
- Several packers still evade the DBI, Armadillo, BoxedApp, EP Protector, VMProtect....
- 3 reached the 10M instructions limit, Enigma, svk, Themida

Packers: Tricks and patterns found

Several of the tricks detected by the analysis

OP in ACProtect		
1018f7a	js	0x1018f92
1018f7c	jns	0x1018f92

(and all possible variants ja/jbe, jp/jnp, jo/jno..)

OP in Armadillo		
10330ae	xor	ecx, ecx
10330b0	jnz	0x10330ca

CST in ASPack		
10043a9	mov	[ebp+0x3a8], eax
10043af	popa	
10043b0	jnz	0x10043ba
Enter SMC Layer 1		
10043ba	push 0	
10043bf	ret	

CST in ACProtect	
1001000	push 16793600
1001005	push 16781323
100100a	ret
100100b	ret

CST in ACProtect	
1004328	call 0x1004318
1004318	add [esp], 9
100431c	ret

Packers: Tricks and patterns found

Several of the tricks detected by the analysis

OP in ACProtect		
1018f7a	js	0x1018f92
1018f7c	jns	0x1018f92

(and all possible variants ja/jbe, jp/jnp, jo/jno..)

OP in Armadillo		
10330ae	xor	ecx, ecx
10330b0	jnz	0x10330ca

CST in ACProtect	
1001000	push 16793600
1001005	push 16781323
100100a	ret
100100b	ret

CST in ACProtect	
1004328	call 0x1004318
1004318	add [esp], 9
100431c	ret

CST in ASPack		
10043a9	mov	[ebp+0x3a8], eax
10043af	popa	0x10043bb at runtime
10043b0	jnz	0x10043ba
	Enter SMC Layer 1	
10043ba	push	0x10011d7
10043bf	ret	

Packers: Tricks and patterns found

Several of the tricks detected by the analysis

OP in ACProtect

1018f7a	js	0x1018f92
1018f7c	jns	0x1018f92

(and all possible variants ja/jbe, jp/jnp, jo/jno..)

OP in Armadillo

10330ae	xor	ecx, ecx
10330b0	jnz	0x10330ca

CST in ASPack

10043a9	mov	[ebp+0x3a8], eax
10043af	popa	0x10043bb at runtime
10043b0	jnz	0x10043ba
	Enter SMC Layer 1	
10043ba	push	0x10011d7
10043bf	ret	

CST in ACProtect

1001000	push 16793600
1001005	push 16781323
100100a	ret
100100b	ret

OP (decoy) in ASPack

```
10040fe: mov bl, 0x0
10041c0: cmp bl, 0x0
1004103: jnz 0x1004163
```

ZF = 0

```
1004163: jmp 0x100416d
[...]
```

ZF = 1

```
1004105: inc [ebp+0xec]
[...]
```

CST in ACProtect

1004328	call 0x1004318
1004318	add [esp], 9
100431c	ret

Packers: Tricks and patterns found

Several of the tricks detected by the analysis

OP in ACProtect		
1018f7a	js	0x1018f92
1018f7c	jns	0x1018f92

(and all possible variants ja/jbe, jp/jnp, jo/jno..)

OP in Armadillo		
10330ae	xor	ecx, ecx
10330b0	jnz	0x10330ca

CST in ACProtect	
1001000	push 16793600
1001005	push 16781323
100100a	ret
100100b	ret

CST in ACProtect		
1004328	call	0x1004318
1004318	add	[esp], 9
100431c	ret	

CST in ASPack		
10043a9	mov	[ebp+0x3a8], eax
10043af	popa	0x10043bb at runtime
10043b0	jnz	0x10043ba
Enter SMC Layer 1		
10043ba	push	0x10011d7
10043bf	ret	

OP (decoy) in ASPack

```
10040fe: mov bl, 0x0
10041c0: cmp bl, 0x1
1004103: jnz 0x1004163
```

0x10040ff at runtime

ZF = 0

```
1004163: jmp 0x100416d
[...]
```

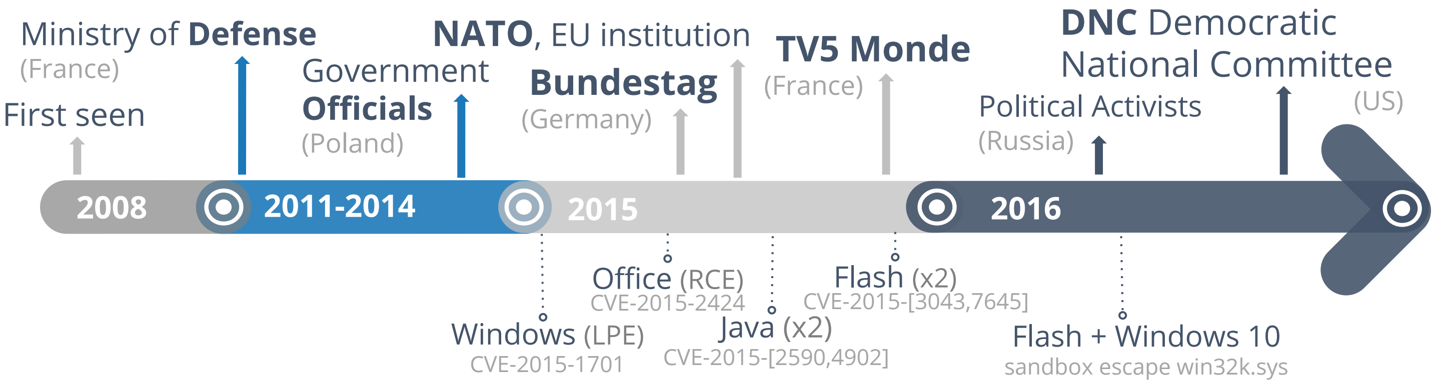
ZF = 1

```
1004105: inc [ebp+0xec]
[...]
```

X-Tunnel: Case-study #2

Introduction of the Sednit group, alleged attacks, methods and techniques used

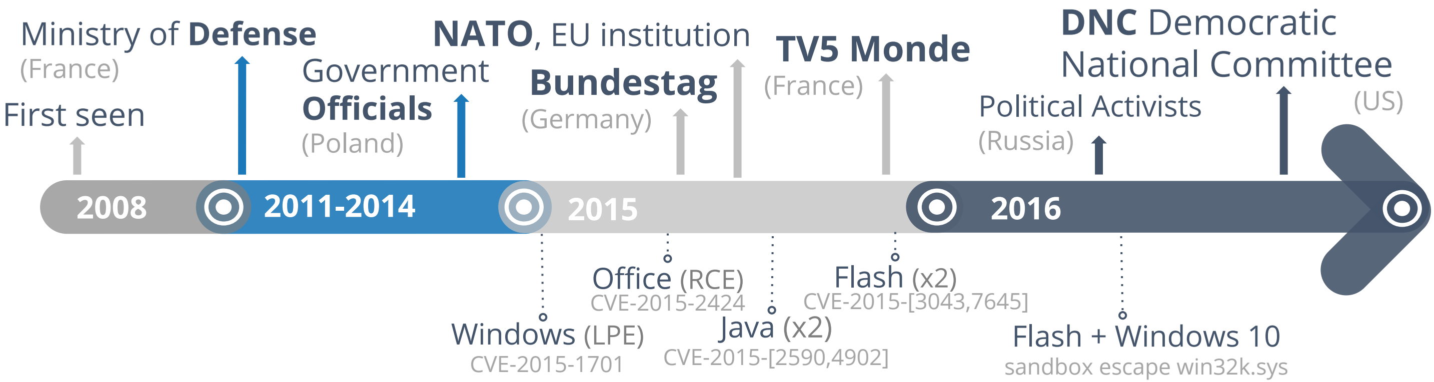
Nicknames: APT28, Fancy Bear, Sofacy, Sednit, Pawn Storm

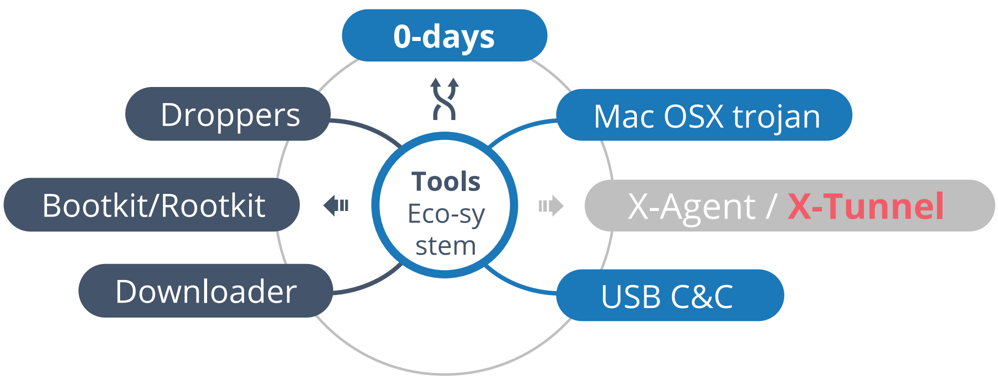


X-Tunnel: Case-study #2

Introduction of the Sednit group, alleged attacks, methods and techniques used

Nicknames: APT28, Fancy Bear, Sofacy, Sednit, Pawn Storm

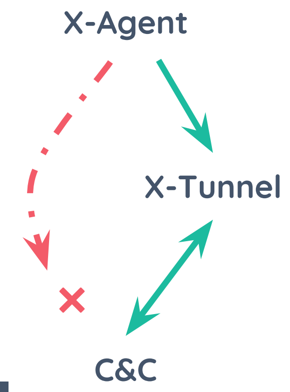




X-Tunnel: Proxy component

What it is, features and samples description

- **What is it:** Ciphering proxy allowing X-Agent(s) not able to reach the C&C directly to connect to it through X-Tunnel
- **Features:** Encapsulate any TCP-based traffic into a RC4 cipher stream embedded into a TLS connection
- **Where:** Used in at least **Bundestag**[6] & **DNC**[7,8] attacks



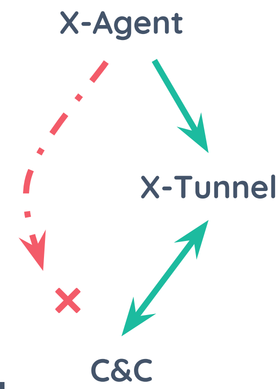
	Sample #0	Sample #1	Sample #2
Hash	42DEE3[..]	C637E0[...]	99B454[...]
Size	1.1 Mo	2.1 Mo	1.8 Mo
Creation date	25/06/2015	02/07/2015	02/11/2015
#functions	3039	3775	3488
#instructions (IDA)	231907	505008	434143

A huge thanks to Joan Calvet

X-Tunnel: Proxy component

What it is, features and samples description

- **What is it:** Ciphering proxy allowing X-Agent(s) not able to reach the C&C directly to connect to it through X-Tunnel
- **Features:** Encapsulate any TCP-based traffic into a RC4 cipher stream embedded into a TLS connection
- **Where:** Used in at least **Bundestag**[6] & **DNC**[7,8] attacks



	Sample #0	Sample #1	Sample #2
Hash	42DEE3[..]	C637E0[...]	99B454[...]
Size	1.1 Mo	2.1 Mo	1.8 Mo
Creation date	25/06/2015	02/07/2015	02/11/2015
#functions	3039	3775	3488
#instructions (IDA)	**231907**	**505008**	**434143**

Widely obfuscated with opaque predicates

A huge thanks to Joan Calvet

X-Tunnel: Questions

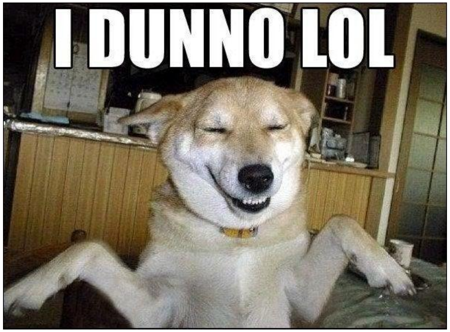
Experimental issues intended to be solved in this use-case

Q1

Can we **remove** the obfuscation?

Q2

Are there new **functionalities**?



X-Tunnel: Analysis

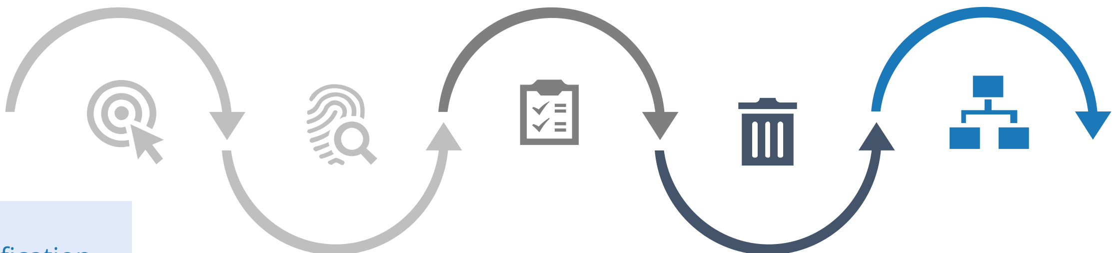
Analysis process and different steps followed

Goal: Detect and remove all opaque predicates to extract a clean CFG

Analysis context
fully static analysis

2. High-level predicate recovery
to identify predicates used

4. Reduced CFG extraction
using data computed by previous steps



because:
- no self-modification
- need to contact C&C
- need to wait clients

1. Opaque predicates detection
with bb-DSE and IDAsec

3.Dead and spurious instruction removal
with liveness propagation

High-level predicate recovery

Synthesis and extraction of the different opaque predicates used

- **Behavior:** Computes the dependency, generates the predicate

```
mov     esi, dword_5D7A84
mov     edi, dword_5D7A80
jz      loc_44D9FA
```

```
imul    esi, esi
imul    eax, esi, 7
dec     eax
imul    edi, edi
cmp     eax, edi
jnz     loc_44D922
```

```
(define-fun esi2 (load32_at memory #x005d7a84))
(define-fun edi0 (load32_at memory #x005d7a80))
(assert (not (= ZF2 #b1)))

(define-fun esi3 (bvmul esi2 esi2))
(define-fun eax2 (bvmul esi3 #x00000007))
(define-fun eax3 (bvsub eax2 #x00000001))
(define-fun edi1 (bvmul edi0 edi0))
(define-fun res328 (bvsub eax3 edi1))
(define-fun ZF4    (bvcomp res328 #x00000000))
(assert (= ZF4 #b1))
```

High-level predicate recovery

Synthesis and extraction of the different opaque predicates used

- **Behavior:** Computes the dependency, generates the predicate

```
mov     esi, dword_5D7A84
mov     edi, dword_5D7A80
jz      loc_44D9FA

imul    esi, esi
imul    eax, esi, 7
dec     eax
imul    edi, edi
```

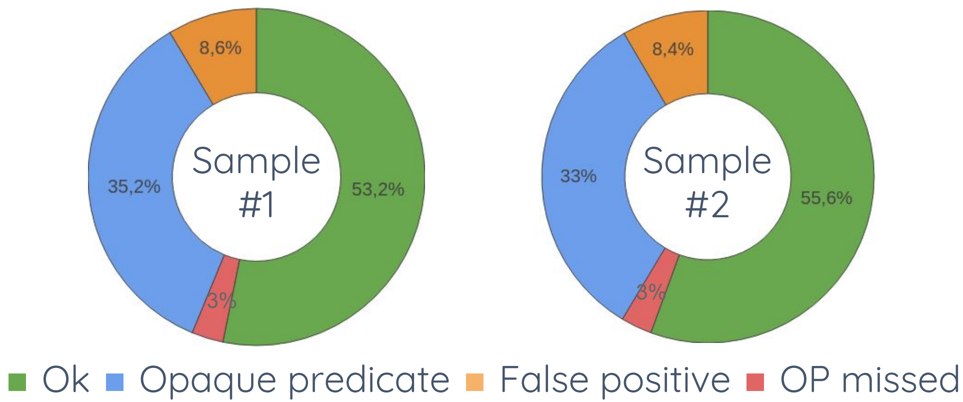
```
(define-fun esi2 (load32_at memory #x005d7a84))
(define-fun edi0 (load32_at memory #x005d7a80))
(assert (not (= ZF2 #b1)))

(define-fun esi3 (bvmul esi2 esi2))
(define-fun eax2 (bvmul esi3 #x00000007))
(define-fun eax3 (bvsub eax2 #x00000001))
(define-fun edi1 (bvmul edi0 edi0))
(define-fun res328 (bvsub eax3 edi1))
```

((bvsub (bvmul (bvmul esi2 esi2) 7) 1) ≠ (bvmul edi0 edi0)) ↦ $7x^2 - 1 \neq y^2$

X-Tunnel: Results

Results in terms of opaque predicates detections and false positive/negative



- **2 predicates synthesized:** $7y^2 - 1 \neq x^2$ $\quad \frac{2}{x^2+1} \neq y^2 + 3$

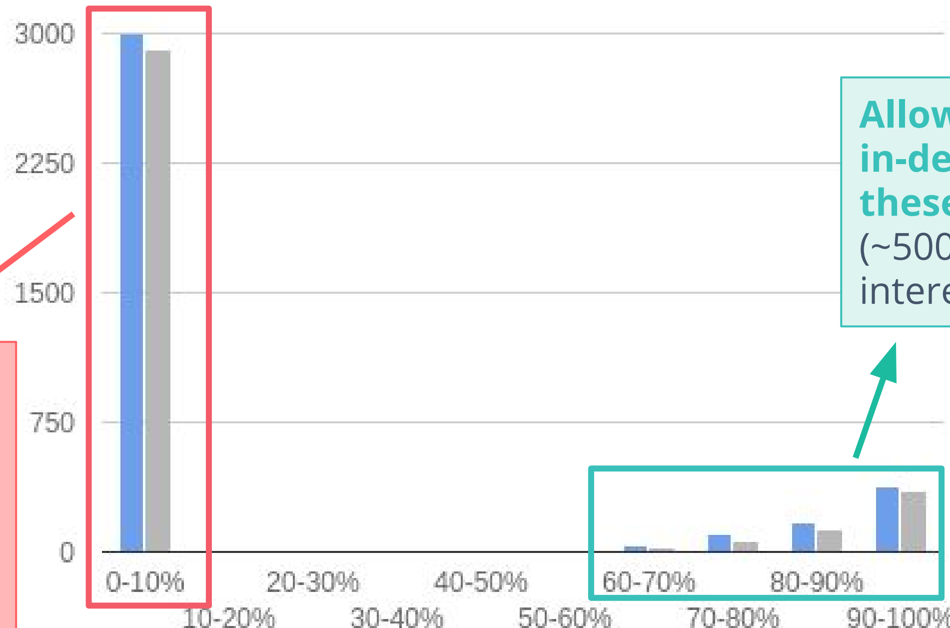
possible signature

	#cond jmp	bb-DSE	Synthesis	Total
Sample #1	34505	57m36	48m33	1h46m
Sample #2	30147	50m59	40m54	1h31m

Analysis: Obfuscation distribution

Obfuscation accross functions in both binaries

- **Goal:** Compute the percentage of conditional jump obfuscated within a function



Allow to narrow the in-depth analysis on these functions (~500 more likely of interest)

Many not obfuscated functions (statically linked library OpenSSL...)

X-Tunnel: Code coverage

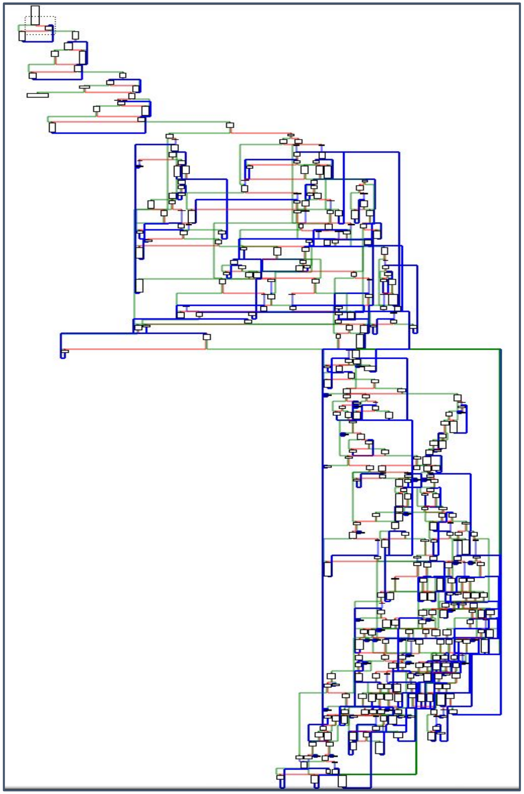
Results of the liveness propagation and identification of spurious instructions

	C637 Sample #1	99B4 Sample #2
#total instruction	**505,008**	**434,143**
#alive	+279,483	+241,177
#dead	-121,794	-113,764
#spurious	-103,731	-79,202
#delta with sample #0	**47,576**	**9,270**

In both samples the difference with the un-obfuscated binary is very low (probably due to some noise)

X-Tunnel: Reduced CFG extraction

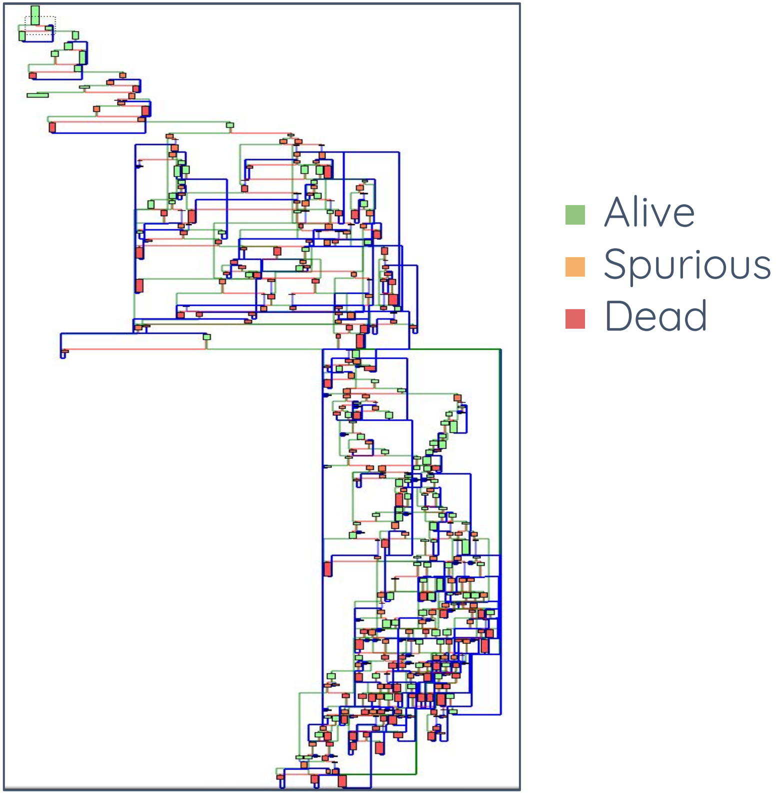
Results of extracting a CFG without the obfuscation



Original CFG

X-Tunnel: Reduced CFG extraction

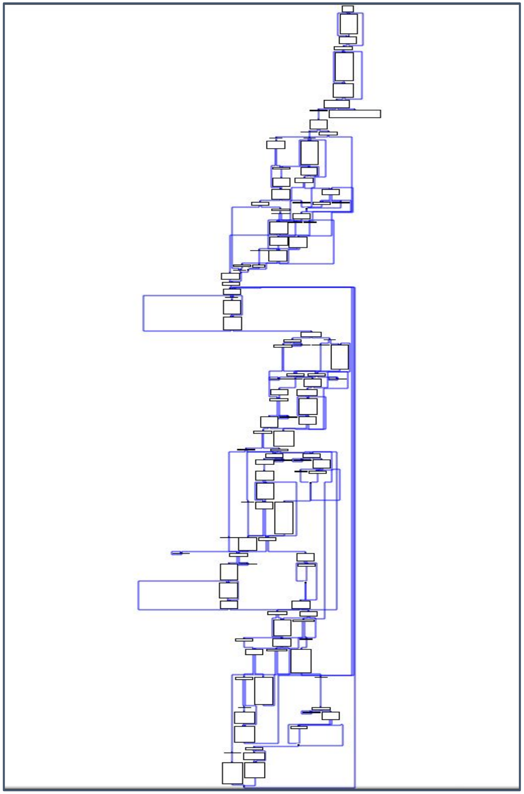
Results of extracting a CFG without the obfuscation



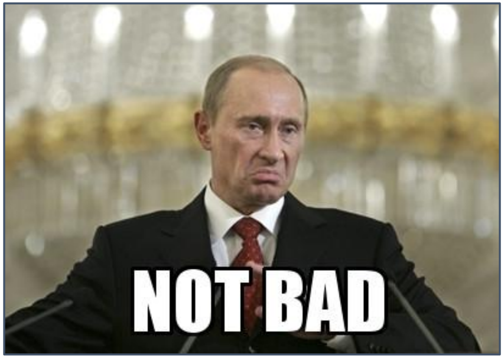
Tagged CFG

X-Tunnel: Reduced CFG extraction

Results of extracting a CFG without the obfuscation



Extracted CFG



X-Tunnel: Conclusion

Reversing conclusion and future work opening

New functionalities ?

Manual checking of difference did not appeared to yield significant differences or any new functionalities…

- **Obfuscation:** Difference with O-LLVM (like)
  - some predicates have far dependencies (use local variable)
  - some computation reuse between opaque predicates
- **Next:**
  - **in-depth graph similarity** (Bindiff) to find new functionalities)
  - integration as an IDA processor module (IDP) ?
- **For more:** Visiting the Bear Den, *Joan Calvet, Jessy Campos, Thomas Dupuy* [RECON 2016][Botconf 2016][CCC 2016]

6.

Conclusion

Conclusion: Contributions

General conclusion about contributions provided by this thesis

Main: Practical approaches for formal analysis-based deobfuscation

#1 DSE for obfuscation

C/S policies, CSML, Backward Bounded DSE

#2 Implementation in Binsec

DSE engine in Binsec/SE, optimizations, IDASec, Pinsec network interaction etc.

#3 Analysis Combinations

Source software Testing, UaF detection, Sparse disassembly

#4 Case-Studies

Large scale-automated analysis of packers, X-Tunnel deobfuscation

Conclusion: Publications

Publications submitted as part of my thesis fulfillment

ICST 2015
Graz, Austria

Sound and Quasi-Complete Detection of Infeasible Test Requirements, Sébastien Bardin, Mickaël Delahaye, Robin David, Nikolai Kosmatov, Mike Papadakis and Yves Le Traon
8th IEEE International Conference on Software Testing

SANER 2016
Osaka, Japan

Binsec/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis, Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet and Jean-Yves Marion
23rd International Conference on Software Analysis, Evolution and Reengineering

ISSTA 2016
Saarbrücken, Germany

Specification of Concretization and Symbolization Policies in Symbolic Execution, Robin David, Sébastien Bardin, Josselin Feist, Marie-Laure Potet, Thanh Dinh Ta, Jean-Yves Marion
25th International Symposium on Software Testing and Analysis

SSPREW 2016
Los Angeles, USA

Finding the Needle in the Heap: Combining Static Analysis and Dynamic Symbolic Execution to Trigger Use-After-Free, Josselin Feist, Sébastien Bardin, Laurent Mounier, Robin David, Marie-Laure Potet
6th Software Security, Protection and Reverse Engineering Workshop

Backward-Bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes, Robin David, Sébastien Bardin and Jean-Yves Marion
(submitted S&P 17)

Conclusion: Perspectives

Near and long term improvements both from research and implementation perspectives

Binary analysis & Deobfuscation futur work:

- more obfuscations: VM, conditional self-modification, DGA etc.. *(with a similar approach)*
- DSE robustness: initial state, taint, path predicate optimizations

Malware analysis:

- exploring tradeoff between **comprehension & detection**
- more **semantic-aware disassembly** *(to get rid of obfuscation)*
- combination with **control-flow (graph-based)** signatures *(Jean-Yves Marion)*
- combination with **data semantic summary** signatures *(Arun Lakhotia)*

Goal : Obtaining more accurate signatures

THANK YOU !

HUGE THANKS to the jury,
CEA co-workers, LORIA, family and friends

Thanks to all the **people** I worked with during my thesis:

Link references

1. IBM Security, 2016 - Ponemon Cost of Data Breach Study, *https://www-03.ibm.com/security/data-breach/*
2. Intel Security, June 2014 - Net Losses: Estimating the Global Cost of Cybercrime, http://www.mcafee.com/us/resources/reports/rp-economic-impact-cybercrime2.pdf
3. Hamilton Place Strategies, Feb 2016 - Cybercrime Costs More Thank You Think, http://www.hamiltonplacestrategies.com/sites/default/files/newsfiles/HPS%20Cybercrime2_0.pdf
4. Panda Security, Jan 2016 - 27% of all Recorded Malware Appeared in 2015, http://www.pandasecurity.com/mediacenter/press-releases/all-recorded-malware-appeared-in-2015/
5. Symantec, April 2016 - Internet Security Threat Report, https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf
6. Netzpolitik, June 2015 - Digital Attack on German Parliament: Investigative Report on the Hack of the Left Party Infrastructure in Bundestag https://netzpolitik.org/2015/digital-attack-on-german-parliament-investigative-report-on-the-hack-of-the-left-party-infrastructure-in-bundestag/
7. CrowdStrike June 2016 - Bears in the Midst: Intrusion into the Democratic National Committee, https://www.crowdstrike.com/blog/bears-midst-intrusion-democratic-national-committee/
8. NCCIC, FBI, Dec 2016 - GRIZZLY STEPPE – Russian Malicious Cyber Activity, https://www.us-cert.gov/sites/default/files/publications/JAR_16-20296A_GRIZZLY%20STEPPE-2016-1229.pdf