black hat EUROPE 2016

Code Deobfuscation:

Intertwining Dynamic, Static and Symbolic Approaches

Robin David & Sébastien Bardin
CEA LIST

Who are we ?

#Robin David

- PhD Student at CEA LIST

#Sébastien Bardin

- Full-time researcher at CEA LIST

Where are we ?

Atomic Energy Commission (CEA LIST), Paris Saclay

- Software Safety & Security Lab
  - frama C
  - BINSEC

FROM RESEARCH TO INDUSTRY

cea tech

list

Context & Goal

- Analysis of obfuscated binaries and malware (potentially self-modifying)
- Recovering high-level view of the program (e.g CFG)
- Locating and removing obfuscation if any

Challenges ?

- Static, dynamic and symbolic analyses are not enough used alone
- Scalability, robustness, *"infeasibility queries"*

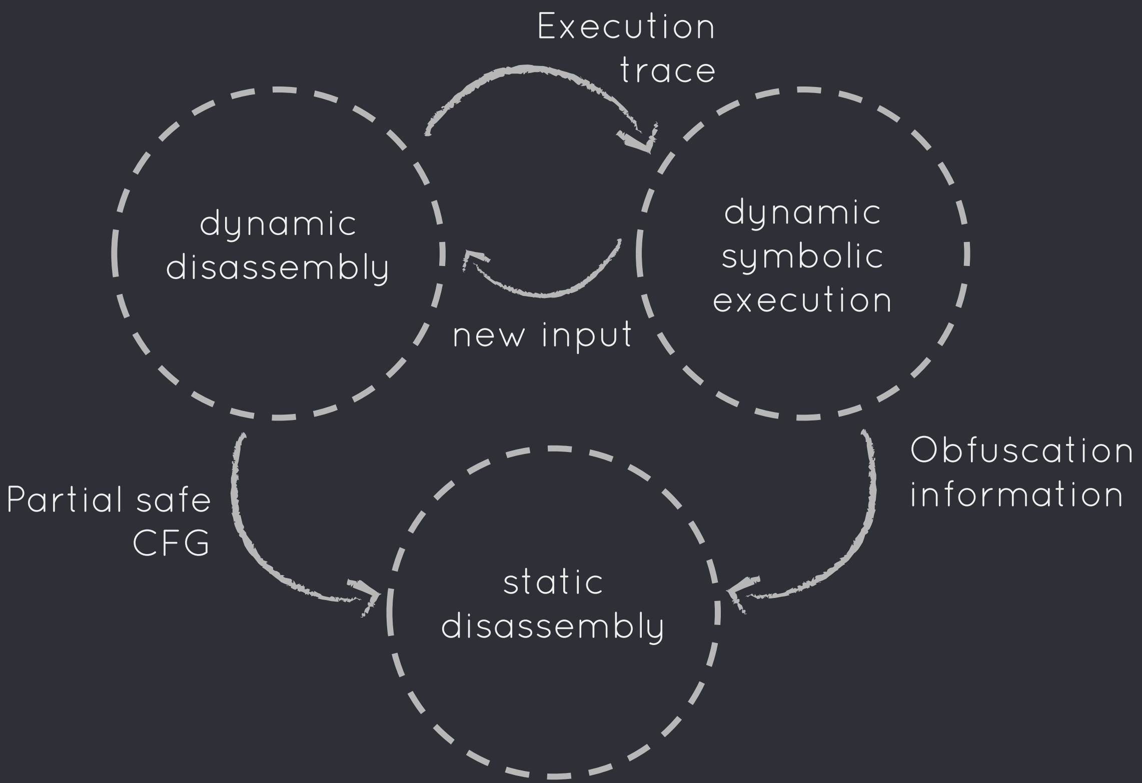
Our proposal

- A new symbolic method for infeasiblity-based obfuscation problems
- A combination of approaches to handle obfuscations impeding different kind of analyses

Achievements

- A set of tool to analyse binaries (instrumentation, binary analysis and IDA integration)
- Detection of several obfuscations in packers
- Deobfuscation of the X-Tunnel malware (for which obfuscation is stripped)

Long term objectives



Takeaway message

- disassembling highly obfuscated codes is challenging
- combining static, dynamic and symbolic is promising (accurate and efficient)

Agenda

Background

1. Disassembling obfuscated codes
2. Dynamic Symbolic Execution

Our proposal

3. Backward-Bounded DSE
4. Analysis combination

Binsec

5. The Binsec platform

Case-studies

6. Packers
7. X-Tunnel

1

Disassembling obfuscated codes

Getting an exploitable representation of the program

An essential task before in-depth analysis is the CFG disassembly recovery of the program

Disassembly issues

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

*segmentation proposed in Binary Code is Not Easy, Xiaozhu Meng, Barton P. Miller

Obfuscation

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

Obfuscation diversity

Control Vs **Data**

function calls, edges — strings, constants..

	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	●		●	

and so many others....

Opaque predicates

Definition: Predicate always evaluating to true (resp. false). (but for which this property is difficult to deduce)

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     <trap_addr>
```

Taxonomy:

- Arithmetic based
- Data-structure based
- Pointer based
- Concurrence based
- Environment based

Corollary:

- the dead branch allow to
  - growing the code (artificially)
  - drowning the genuine code

Call stack tampering

Definition: Alter the standard compilation scheme of calls 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	[...]

In addition, able to characterize the tampering with alignment and multiplicity

Need to handle the tail call optimization..

Deobfuscation

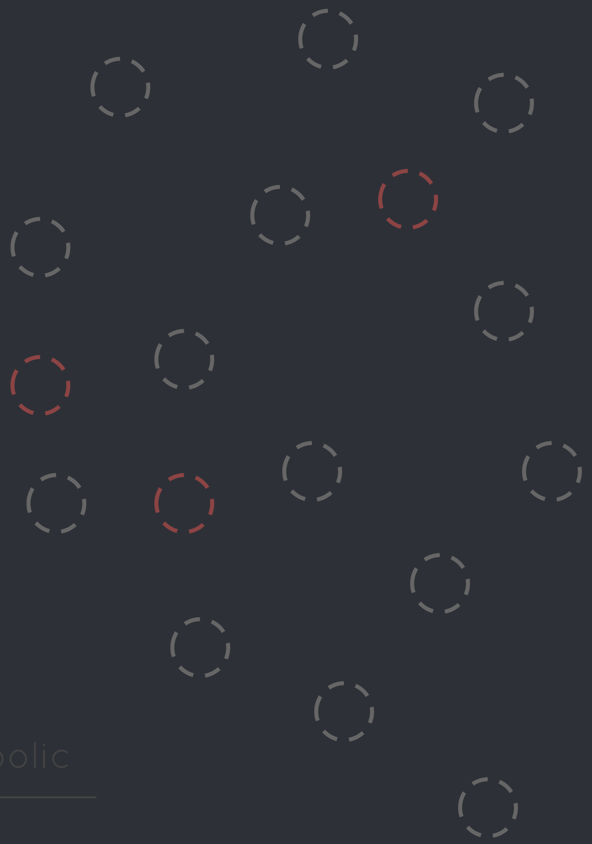
- Revert the transformation (sometimes impossible)
- Simplify the code to facilitate later analyses

Disassembly

Notations

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

Standard approaches

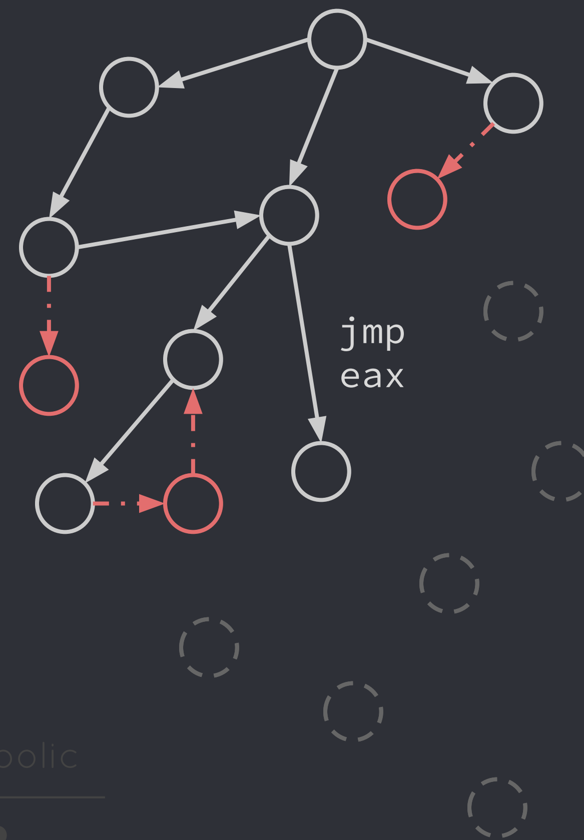


	static	dynamic	symbolic
scale	•	•	•
robust *(obfuscation)*	•	•	•
correct	•	•	•
complete	•	•	•

Disassembly

Notations

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



Standard approaches

- Static disassembly

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

dynamic jump

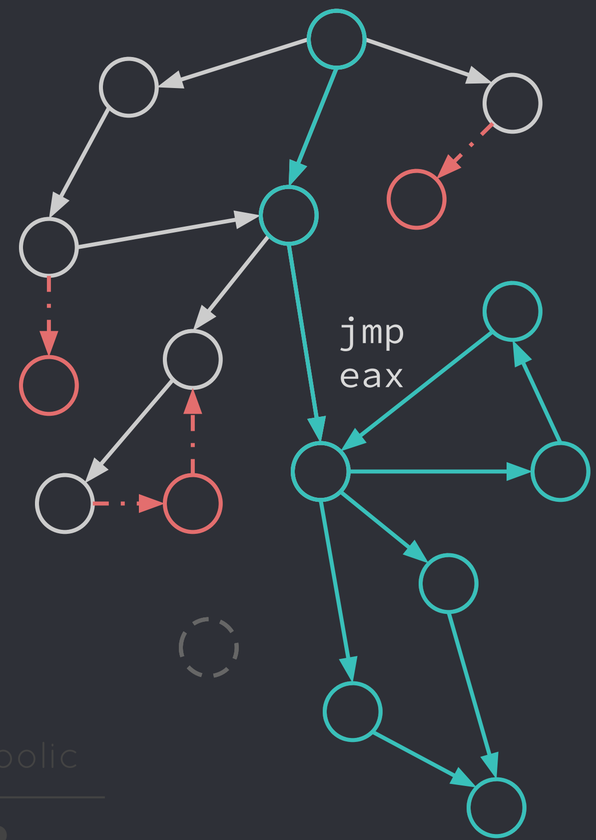
Disassembly

Notations

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

Standard approaches

- Static disassembly
- Dynamic disassembly



	static	dynamic	symbolic
scale	🟢	🟢	⚫
robust *(obfuscation)*	🔴	🟢	⚫
correct	🔴	🟢	⚫
complete	🟠	🔴	⚫

dynamic jump

input dependent

2

Dynamic Symbolic Execution

a.k.a Concolic Execution

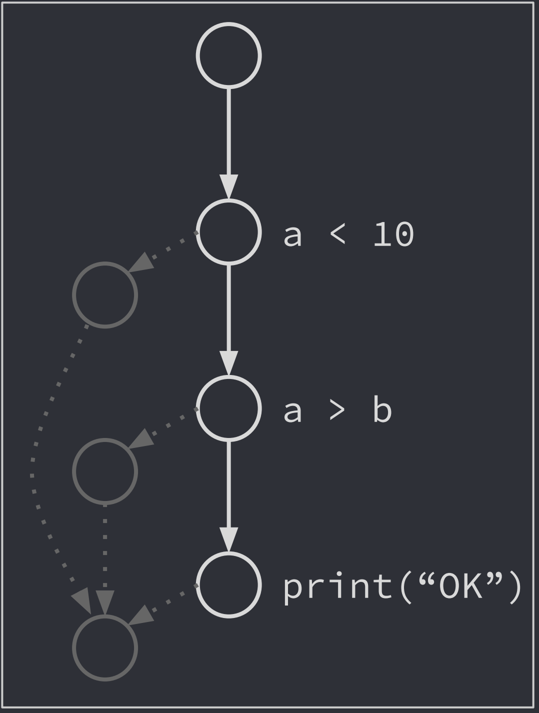
Dynamic Symbolic Execution

Definition: **Symbolic Execution** is the mean of executing a program using symbolic values (logical symbols) rather than actual values (bitvectors) in order to obtain 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 ∧ a > b

Solution:
a=5, b=1

Why using DSE ?

More difficult to hide the semantic of the program than its syntactical form.

Intermediate Representation (IR)

→ Encode the semantic of a machine instruction

Advantages:

- 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

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

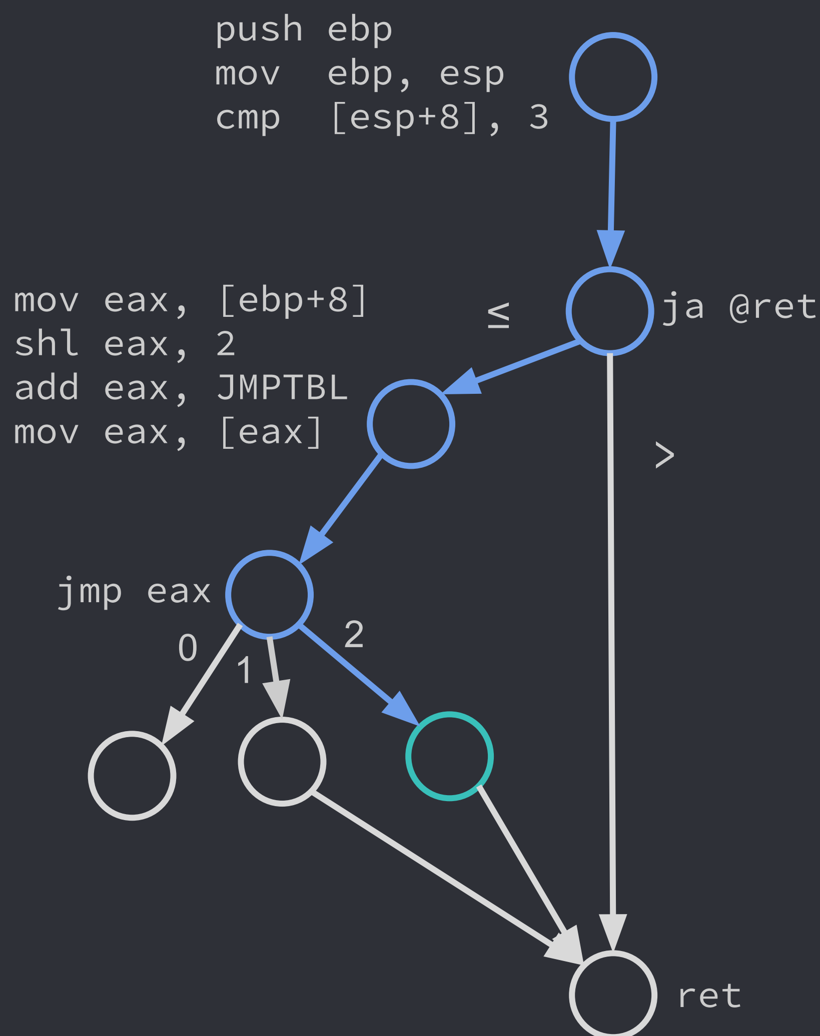
res32	:=	@[esi$_{(32)}$ + 0x14$_{(32)}$] * 7$_{(32)}$
temp64	:=	(exts @[esi$_{(32)}$ + 0x14$_{(32)}$] 64) * (exts 7$_{(32)}$ 64)
OF	:=	(temp64$_{(64)}$ ≠ (exts res32$_{(32)}$ 64))
SF	:=	⊥
ZF	:=	⊥
CF	:=	OF$_{(1)}$
eax	:=	res32$_{(32)}$

DSE on a switch

Source Code (C)

```c
enum E = {A, B, C}
int myfun(int x) {
    switch(x) {
        case A: x+=0; break;
        case B: x+=1; break;
        case C: x+=2; break;
}   }
```

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 (C)
[...]	
ret	



Path predicate **φ** :

@[ebp1+8] < 3 ∧ eax4 == 2

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

DSE Vs Static & Dynamic approaches

Advantages:

- sound program execution (thanks to dynamic)
- path sure to be feasible (unlike static)
- next instruction always known (unlike static)
- loops are unrolled by design (unlike static)
- can generate new inputs (unlike dynamic)
- guided new paths discovery (unlike dynamic)
- thwart basic tricks (cover-overlapping etc)

	static	dynamic	symbolic
scale	🟢	🟢	🟠
robust *(obfuscation)*	🔴	🟢	🟢
correct	🔴	🟢	🟢
complete	🟠	🔴	🟠

The challenge for DSE is to make it scale on huge path length and to cover all paths...

3

Backward-Bounded DSE

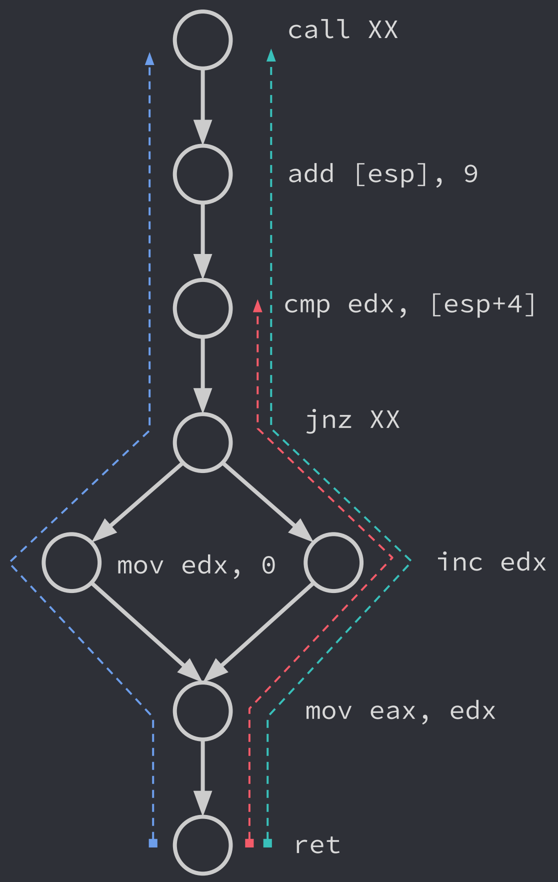
Complementary approach for infeasibility-based problems

BB-DSE: Example of a call stack tampering

Goal

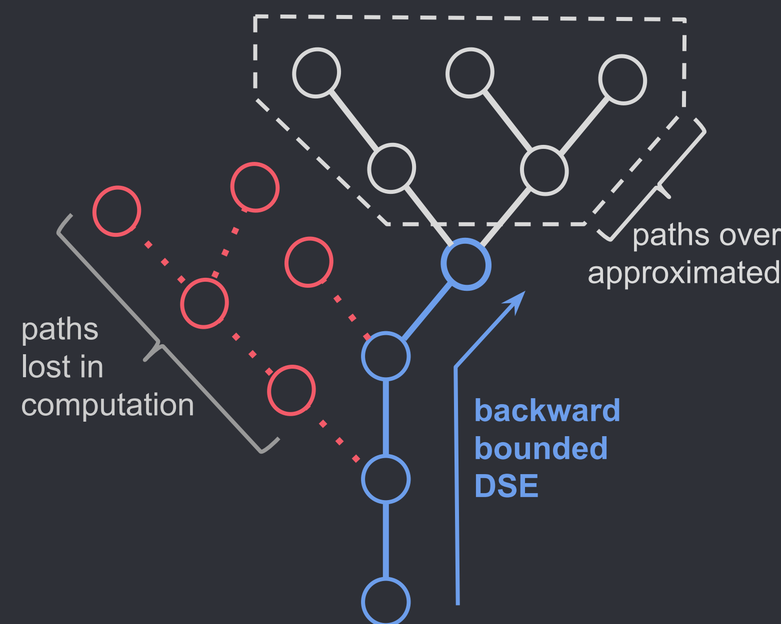
Checking that the return address cannot be tampered by the function

- false negative: miss the tampering (too small bound)
- correct: find the tampering
- + complete: validate the tampering for all paths



Backward-Bounded DSE (new)

Infeasibility query: Query aiming at proving the infeasibility of some events or configuration.
(while traditional SE performs feasibility requests (paths, values) to generate satisfying inputs)



Properties:

- backward approach
- solve infeasibility queries
- goal-oriented computation
- bounded reasoning
- bound modulable for the need

	(forward) DSE	bb-DSE
feasibility queries	🟢	🔴
infeasibility queries	🔴	🟢
scale	🟠	🟢

Not FP/FN free, but very low rates

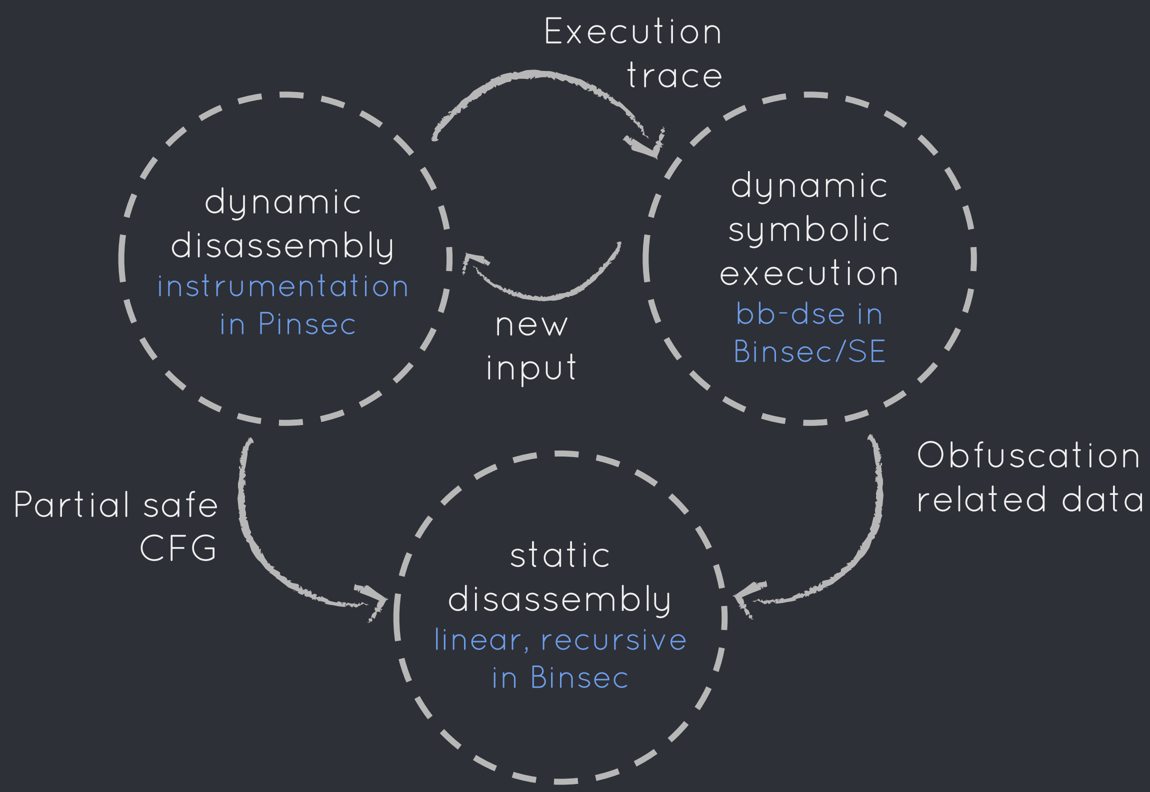
4

Combination

Intertwining Dynamic, Static and Symbolic

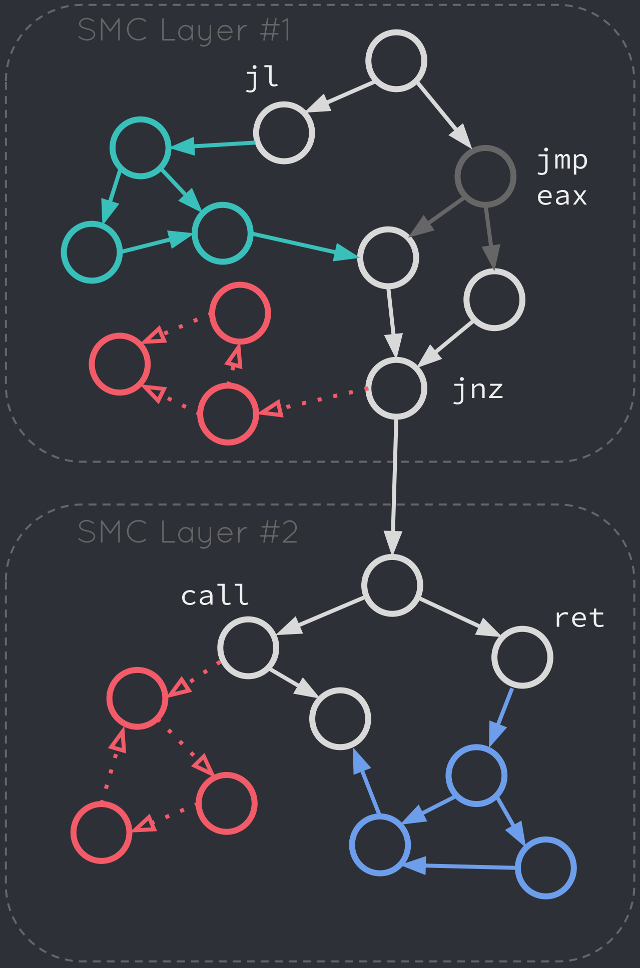
Combination: Principles

- Goal: Enlarging a safe dynamic CFG by static disassembly guided by DSE to ensure a safer and more precise disassembly handling some obfuscation constructs.



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

Combination: Principles



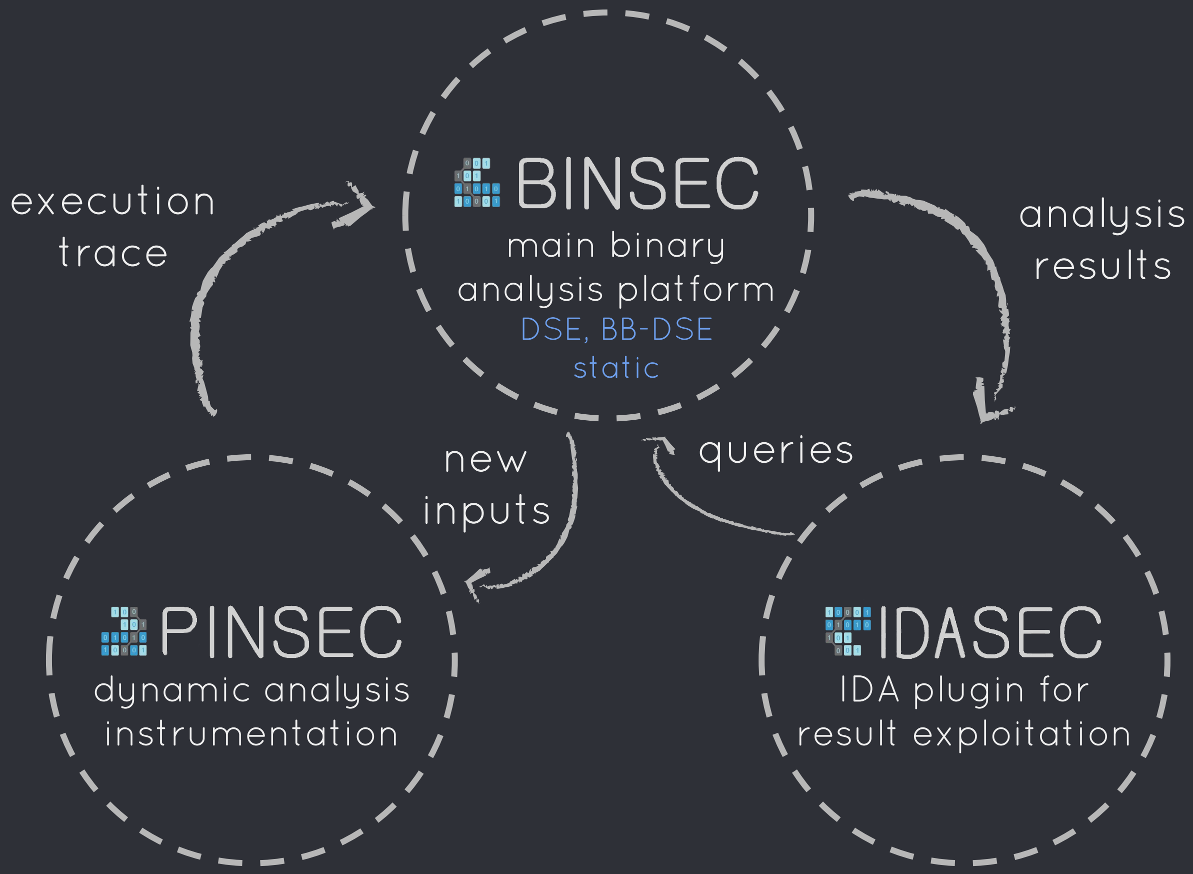
Features:

- ■ enlarge partial CFG on genuine conditional jump
- ■ use dynamic jumps found in the dynamic trace
- ■ do not disassemble dead branch of opaque predicate
- ■ disassemble the target of tampered ret
- ■ do not disassemble the return site of tampered ret

Promising results 10 to 32% less instructions in obfuscated programs (with opaque predicates, call stack tampering).

5 BINSEC

Binsec platform architecture



Open source and available at:

- Binsec+Pinsec: http://binsec.gforge.inria.fr
- IDASec: https://github.com/RobinDavid/idasec

PINSEC

Pintool based on Pin 2.14-71313

Features:

- Generate a protobuf execution trace (with all runtime values)
- Can limitate the instrumentation time / space
- Working on **Linux / Windows**
- Configurable via JSON files
- Allow on-the-fly value patching
- Retrieve some function parameters on known library functions
- Remote control *(prototype)*
- **Self-modification layer tracking**

Still lacks many anti-debug countermeasures..

BINSEC

Binsec (main platform)

Features:

- Front-end: x86 (+simplification)
- Disassembly: linear, recursive, linear+recursive
- **Static analysis**: abstract interpretation

Binsec/SE (symbolic execution engine)

Features:

- **generic C/S policy engine**
- path selection for coverage (thanks Josselin 😊)
- configurable via JSON file
- (basic) stub engine for library calls (+cdecl, stdcall)
- analysis implementation
- **path predicate optimizations**
- SMT solvers supported: Z3, boolector, Yices, CVC4

Many other DSE engines: Mayhem (ForAllSecure), Triton (QuarksLab), S2E, and all DARPA CGC challengers ....

IDASEC

Python plugin for IDA (from 6.4)

Goal:

- triggering analyses remotly from IDA and retrieving the results for post-processing
- leveraging Binsec features into IDA

Features:

- DBA decoding of an instruction
- reading an execution trace
- colorizing path taken
- dynamic disassembly (following the execution trace)
- triggering analyses via **remote connection to Binsec**
- **exploiting the results** depending of the analysis triggered

6

Packers study

Packers & X-Tunnel

Packer: deobfuscation evaluation

Evaluation of 33 packers

(packed with a stub binary)

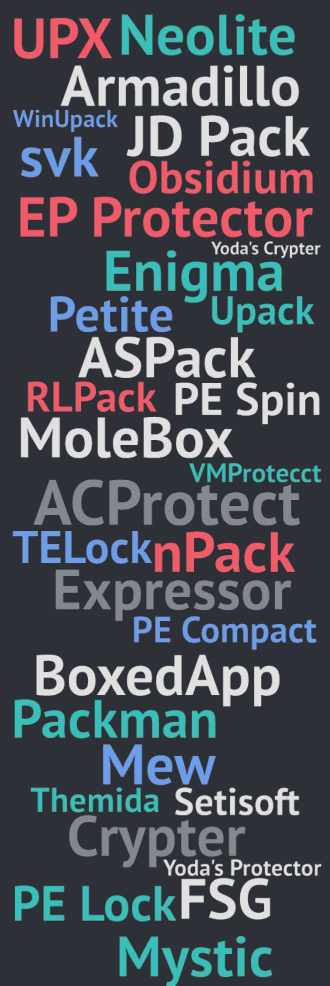
Looking for (with BB-DSE):

- **Opaque predicates**
- **Call stack tampering**
- record of self-modification layers

Settings:

- execution trace limited to 10M instructions

Goal: To perform a systematic and fully automated evaluation of packers



Packer: Analysis results

Packer	Trace len.	#proc	#th	#SMC	opaque predicates (OK)	(OP)	Call/stack tampering (OK)	(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 don't have 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

Packer: Analysis results

Packer	Trace len.	#proc	#th	#SMC	opaque predicates (OK)	(OP)	Call/stack tampering (OK)	(tamper)
ACProtect v2.0	1.8M	1				159	0	48
ASPack v2.12	377K	1				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

The technique scales on significant traces

- Several don't have 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

Packer: Analysis results

Packer	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				159	0	48
ASPack v2.12	377K	1				24	11	6
Crypter v1.12	1.1M	1	1	1	399	24	125	78
Expressor	635K	1	1	1			14	0
FSG v2.0	68k	1	1	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 scales on significant traces

Many true positives. Some packers are using it intensively

- Several don't have 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

Packer: Analysis results

Packer	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				159	0	48
ASPack v2.12	377K	1				24	11	6
Crypter v1.12	1.1M	1	1	1	399	24	125	78
Expressor	635K	1	1	1			14	0
FSG v2.0	68k	1	1	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			3	1
Upack v0.39	711K	1	1	2			7	1

The technique scales on significant traces

Many true positives. Some packers are using it intensively

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

- Several don't have 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

Packer: Tricks and patterns found

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

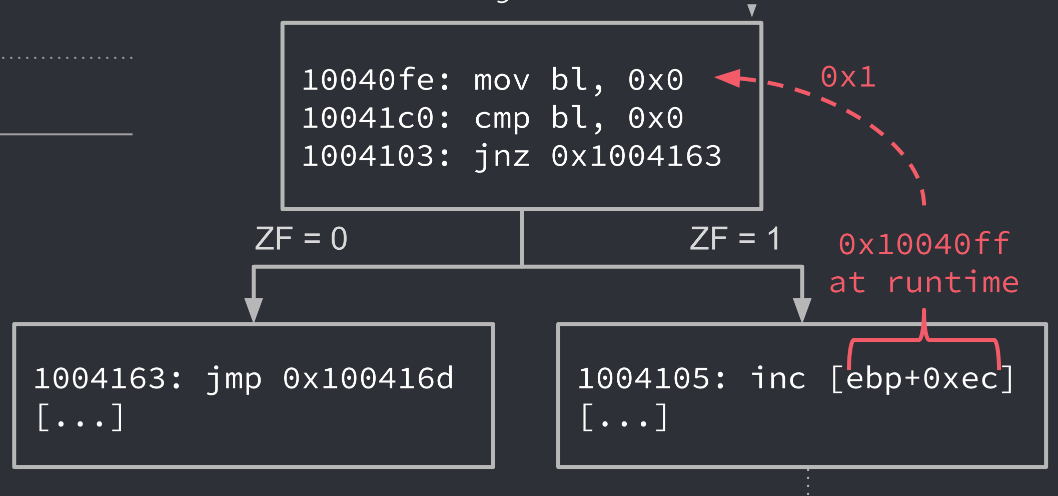
CST in ACProtect

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

CST in ASPack

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

OP (decoy) in ASPack



7

X-Tunnel

A dive into the APT28 ciphering proxy

Introduction: Sednit / APT28 / Pawn Storm

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

Alleged attacks:

- NATO, EU institutions [2015]
- German Parliament (Germany) [2015]
- TV5 Monde (France) [2015]
- DNC: Democratic National Committee (US) [2016]
- Political activists (Russia)
- MH17 investigation team (Netherlands) [2015]
- Many more ambassadies and military entities ....

Data collected from: ESET, Trend Micro, CrowdStrike ...

0-days used:

- 2 Flash [CVE-2015-7645] [CVE-2015-3043]
- 1 Office (RCE) [CVE-2015-2424]
- 2 Java [CVE-2015-2590] [CVE-2015-4902]
- 1 Windows (LPE) [CVE-2015-1701]

(delivered via their exploit kit "sedkit" with many existing exploits)

Tools used:

- Droppers / Downloader
- X-Agent / **X-tunnel**
- Rootkit / Bootkit
- Mac OS X trojan (Komplex)
- USB C&C

X-Tunnel

What it is ?

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.

Samples

	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 ESET Montreal and especially to Joan Calvet

Can we remove the obfuscation ?

Are there new functionalities ?

Can we remove the obfuscation ?

spoiler:

Are there new functionalities ?

spoiler:

X-Tunnel: Analysis

Goal: Detecting and removing all opaque predicates to extract a clean CFG of the functions

Analysis context:

- full static analysis (because need to connect C2C, wait clients...)
- perform the backward-bounded DSE combined with IDA
- driven by IDASec

Combination divergence:

- without the dynamic component (ok because no SMC)
- the symbolic disassembly reduction performed “a-posteriori”

Analysis procedure:

1. opaque predicate detection
2. high-level predicate recovery
3. dead and spurious instruction removal
4. reduced CFG extraction

IDASec features used:

1. custom CFG structure to enumerate paths and which support annotation
2. liveness propagation
3. custom SMT formula
4. CFG extraction based on annotations

High-level predicate recovery (synthesis)

Behavior: Computes the dependency for a conditional jump, and recursively replace terms in order to obtain the predicate.

Corollary: The algorithm is able to determine which instructions are used for the computation of a conditional jump.

CFG

```
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
```

SMT Formula

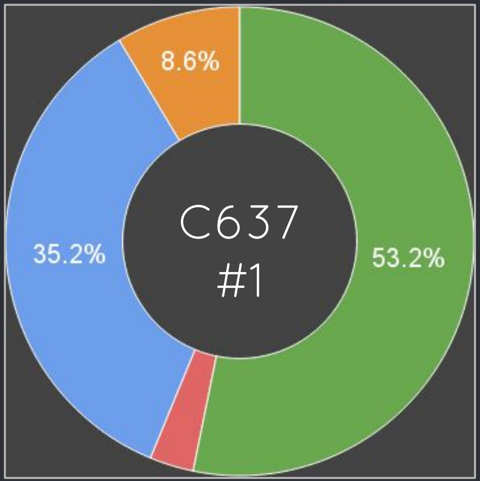
```
(define-fun esi2 (load32_at memory #x005d7a84))
(define-fun edi0 (load32_at memory #x005d7a80))
(assert (not (= ZF2 #b1)))

(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))
```

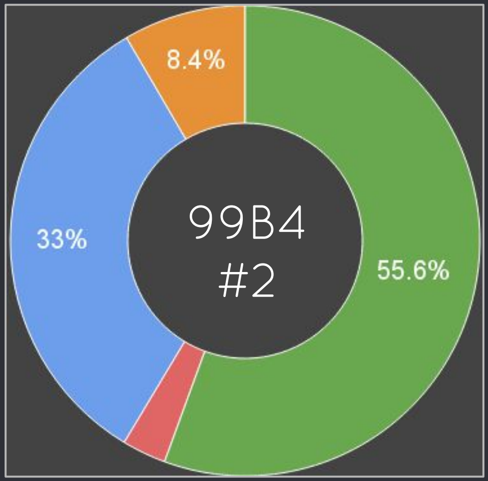
((bvsub (bvmul (bvmul esi2 esi2) #x7) #x1) ≠ (bvmul edi0 edi0) ↦ $7x^2 - 1 \neq y^2$

Analysis: Results

	#cond jmp	bb-DSE	Synthesis	Total
C637 #1	34505	57m36	48m33	1h46m
99B4 #2	30147	50m59	40m54	1h31m

(only one path per conditional jump is analysed)





■ Ok ■ Opaque predicate ■ False positive ■ OP missed

good candidate for signature ?

Only 2 different opaque predicate

$$7x^2 - 1 \neq x^2$$

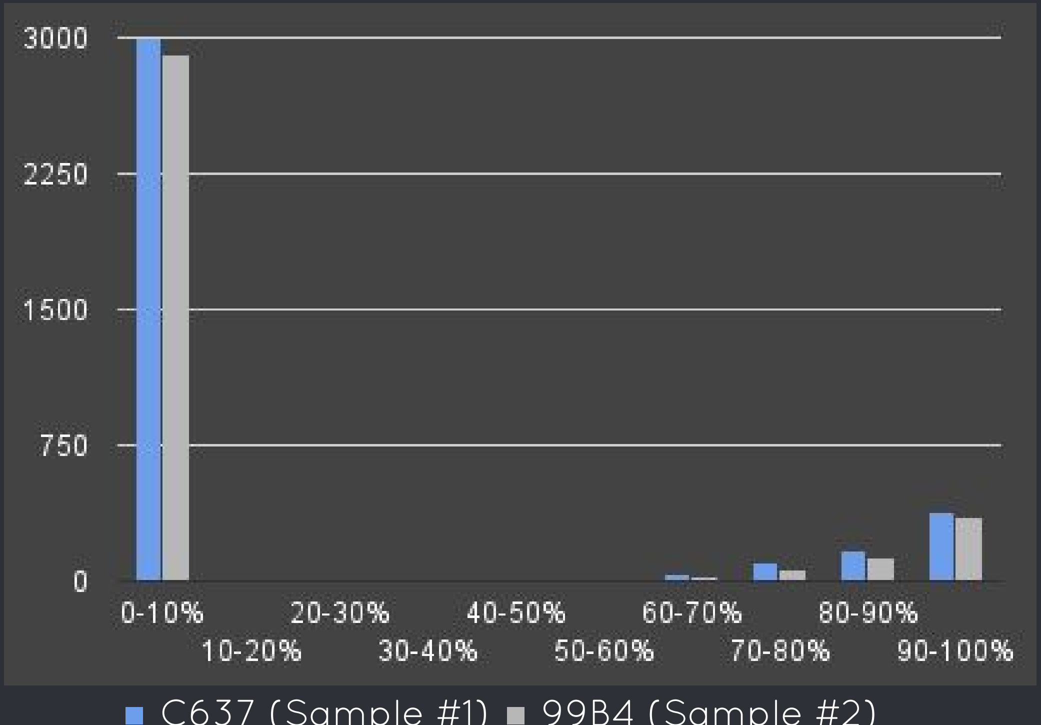
$$\frac{2}{x^2+1} \neq y^2 + 3$$

unseen elsewhere

both present in the same proportions..

Analysis: Obfuscation distribution

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



- Very few function are obfuscated ~500 (due to statically linked library not obfuscated OpenSSL etc..)
- This allow nonetheless to **narrow the post-analysis on these functions** (likely of interest) ...

Analysis: 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, and probably due to some noise

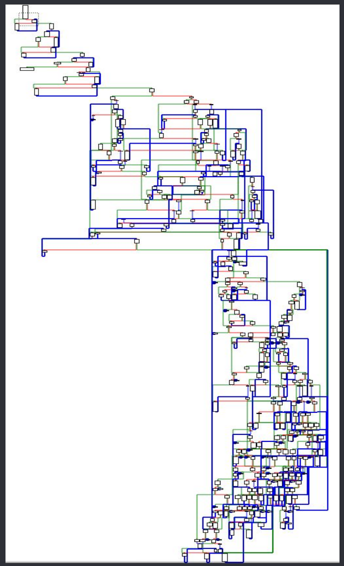
Analysis: Reduced CFG extraction

Goal: Performing a-posteriori the static disassembly sketch in the combined approach

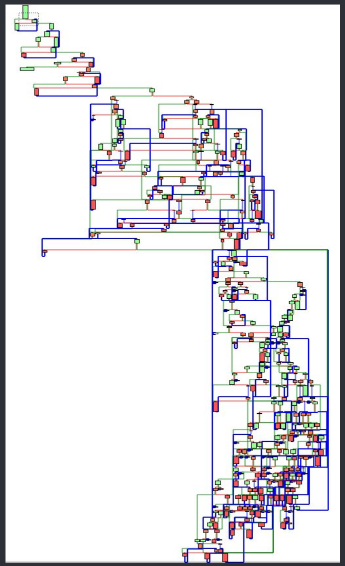
Algorithm:

- remove basic blocks marked dead
- remove spurious instructions (part of the computation of OP)
- recreate the CFG by concatenating instructions with a single predecessor

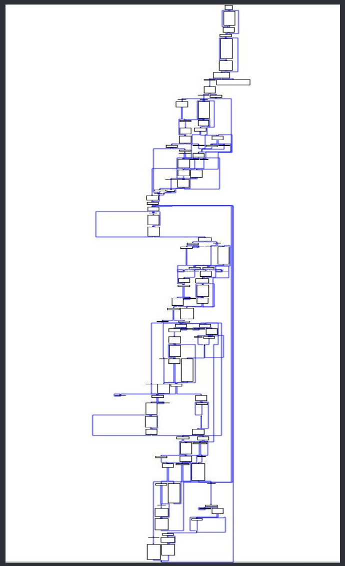
Result:



Original CFG



CFG marked



CFG extracted

Demo !

X-Tunnel deobfuscation

X-Tunnel: Conclusion

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

Obfuscation: Differences with O-LLVM (like)

- some predicates have a great dependency (use local variables)
- some computation reuse between opaque predicates

Technique:

- Combination: Backward Symbolic Execution and "a-posteriori" static disassembly reduction (without the dynamic aspect)
- very few FP / FN refined manually by predicate synthesized (due to the low diversity of predicates)

Next:

- **in-depth graph similarity** (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]

Binsec Takeaways

Tip of what can be done with Binsec

dynamic symbolic execution, abstract interpretation, simulation, optimizations, simplifications, on-the-fly value patching ...

More is yet to come

documentation, ARMv7 support, code flattening and VM deobfuscation...

Still a young platform

under heavy development, API not stabilized,
(considering rewriting IDASec with Binary Ninja)...

Take part !

- Download it, try it, experiment it !
- Don't hesitate contacting us for questions !

Open source and available at:

- Binsec+Pinsec: http://binsec.gforge.inria.fr
- IDASec: https://github.com/RobinDavid/idasec

Takeaways

- More is not always better in terms of disassembly on obfuscated programs
- The backward bounded DSE scale well and allowed to detect obfuscations considered on many packers and X-Tunnel
- The combination yielded very good results on X-Tunnel
- The combination dynamic, static and symbolic is the way to go on obfuscated binaries and helped recovering a clean CFG on X-Tunnel. Still under integration in Binsec with support of different self-modification layers....

Thank you !
Q & A

Robin David
robin.david@riseup.net
@RobinDavid1

Sébastien Bardin
sebastien.bardin@cea.fr