BINARY-LEVEL SECURITY: SEMANTIC ANALYSIS TO THE RESCUE

Sébastien Bardin (CEA LIST)

Joint work with
Richard Bonichon, Robin David, Adel Djoudi & many other people

IN A NUTSHELL

- **Binary-level security analysis**
  - Many applications, many challenges
  - Current syntactic and dynamic methods are not enough

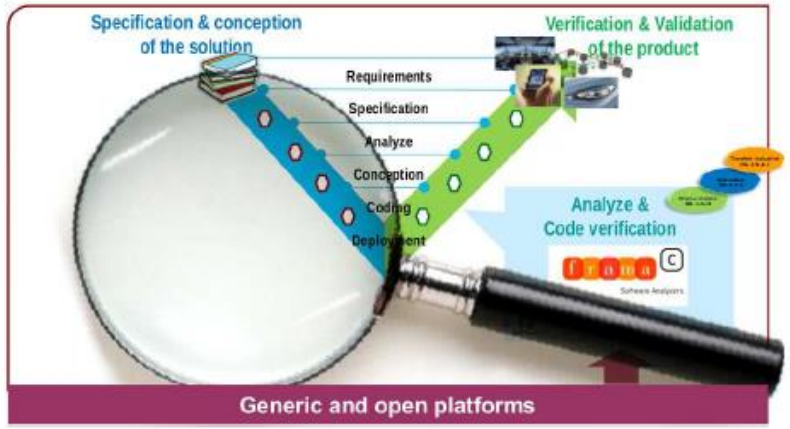
- **Formal methods can change the game … but must be strongly adapted**
  - Complement existing methods
  - Need robustness and scalability!
  - Acceptable to lose both correctness & completeness – in a controlled way
  - **New challenges and variations, many things to do!**

- **Achievements**
  - Vulnerability detection [with Josselin Feist, Laurent Mounier and Marie-Laure Potet]
  - Malware deobfuscation [with Jean-Yves Marion]
  - BINSEC platform: Open-source, still in its infancy, on heavy refactoring

ABOUT MY LAB @CEA

CEA LIST, Software Safety & Security Lab

- rigorous tools for building high-level quality software
- second part of V-cycle
- automatic software analysis
- mostly source code



frama C Software Analyzers

GATeL

PathCrawler

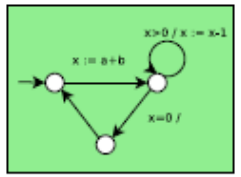
POLAR SSL

ABOUT FORMAL METHODS

- Between Software Engineering and Theoretical Computer Science
- Goal = proves correctness in a mathematical way

Success in safety-critical

Model



Source code

```
int foo(int x, int y) {
 int k= x;
 int c=y;
 while (c>0) do {
  k++;
  c--;}
 return k;
}
```

Key concepts : $M \models \varphi$

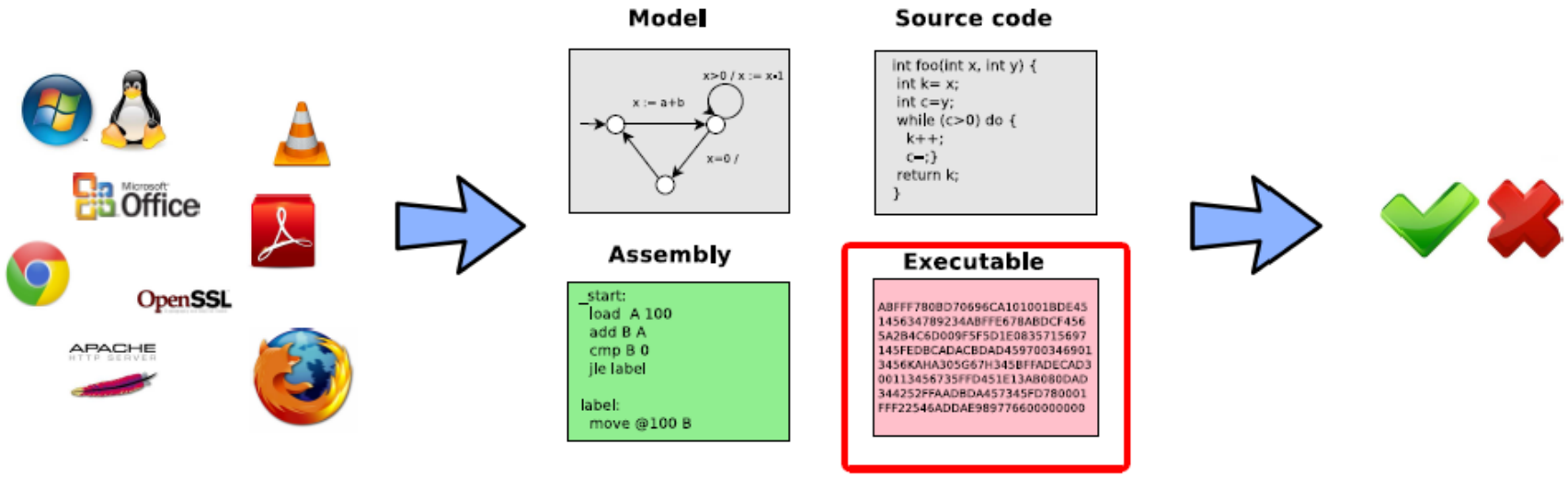
- M : semantic of the program
- φ : property to be checked
- $\models$: algorithmic check

Kind of properties

- absence of runtime error
- pre/post-conditions
- temporal properties

NEXT BIG CHALLENGE

- Apply formal methods to less-critical software
- Very different context : no formal spec, less developer involvement, etc.



Some difficulties

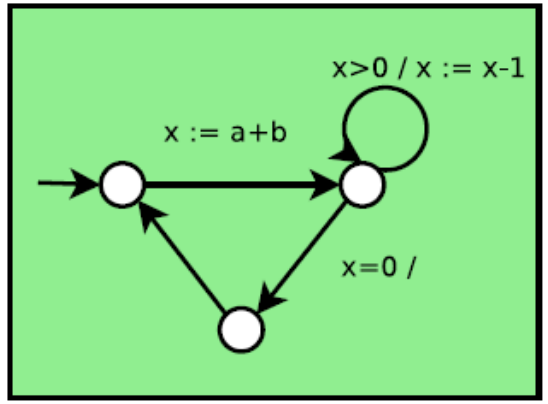
- robustness [w.r.t. software constructs]
- no place for false alarms
- scale
- no model, sometimes no source

Guidelines & trends

- find sweetspots [drivers]
- manage abstractions
- reduction to logic

NOW: BINARY-LEVEL SECURITY

Model



Source code

```
int foo(int x, int y) {
 int k= x;
 int c=y;
 while (c>0) do {
   k++;
   c--;}
 return k;
}
```

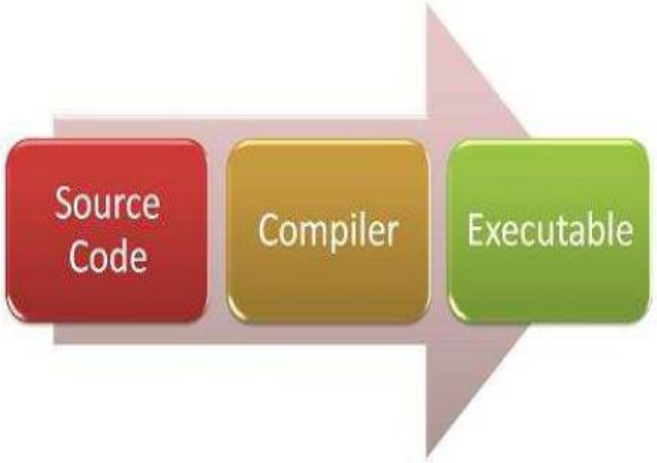
Assembly

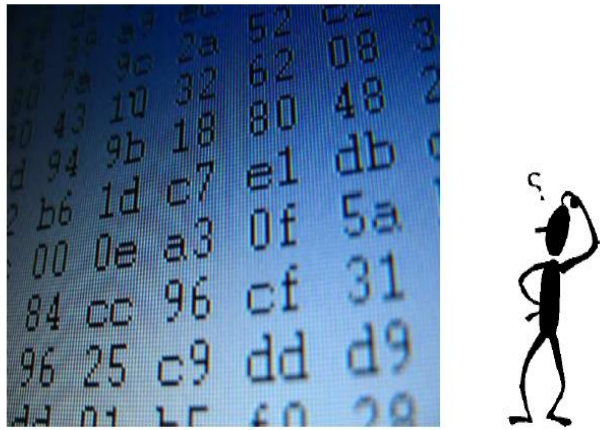
```
_start:
  load  A 100
  add B A
  cmp B 0
  jle label

label:
  move @100 B
```

Executable

```
ABFFF780BD70696CA101001BDE45
145634789234ABFFE678ABDCF456
5A2B4C6D009F5F5D1E0835715697
145FEDBCADACBDAD459700346901
3456KAHA305G67H345BFFADECAD3
00113456735FFD451E13AB080DAD
344252FFAADBDA457345FD780001
FFF22546ADDAE989776600000000
```

BENEFITS





Is it Stuxnet ?

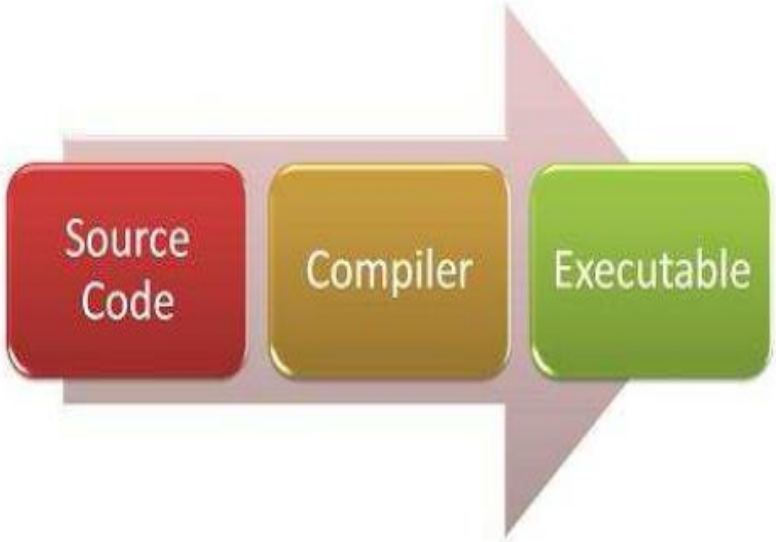
No source code

More precise analysis

Malware

What for: vulnerabilities, reverse (malware, legacy), protection evaluation, etc.

EXAMPLE: COMPILER BUG



- Optimizing compilers may remove dead code
- pwd never accessed after memset
- Thus can be safely removed
- And allows the password to stay longer in memory

Security bug introduced by a non-buggy compiler

```
void  getPassword(void) {
char pwd [64];
if (GetPassword(pwd,sizeof(pwd))) {
/* checkpassword */
}
memset(pwd,0,sizeof(pwd));
}
```

OpenSSH CVE-2016-0777

Our goal here:
- **Check the code after compilation**

EXAMPLE: MALWARE COMPREHENSION

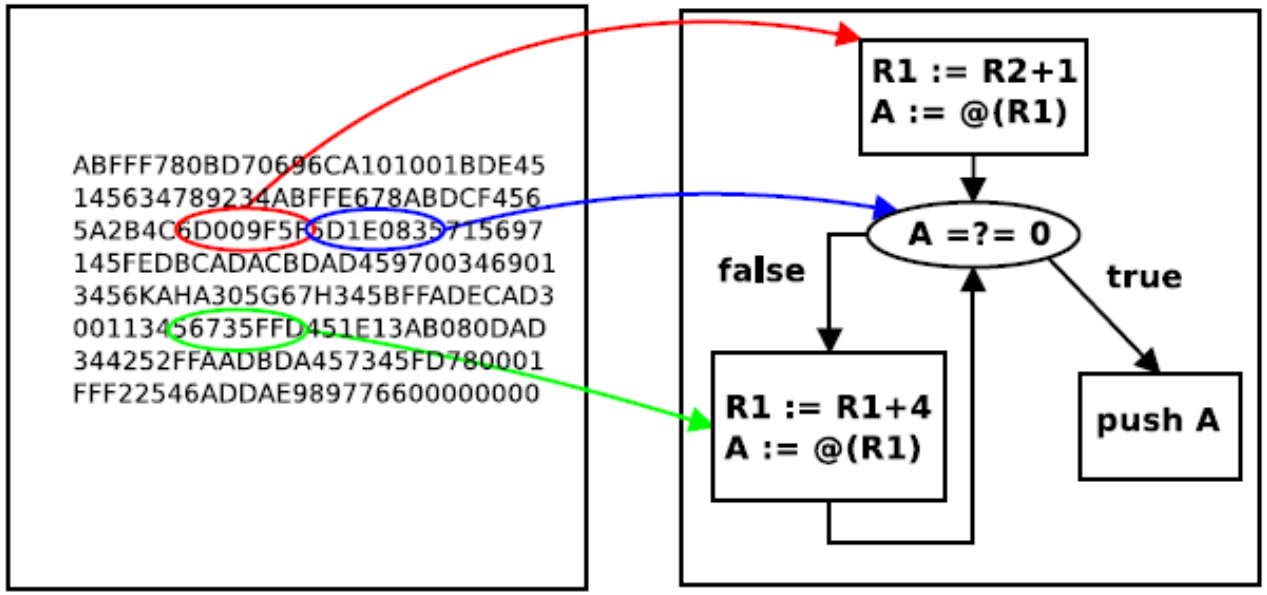
APT: highly sophisticated attacks
- **Targeted malware**
- **Written by experts**
- Attack: 0-days
- Defense: stealth, **obfuscation**
- **Sponsored by states or mafia**

The day after: **malware comprehension**
- understand what has been going on
- mitigate, fix and clean
- improve defense

USA elections: DNC Hack

Highly challenging [obfuscation]

CHALLENGE: CORRECT DISASSEMBLY

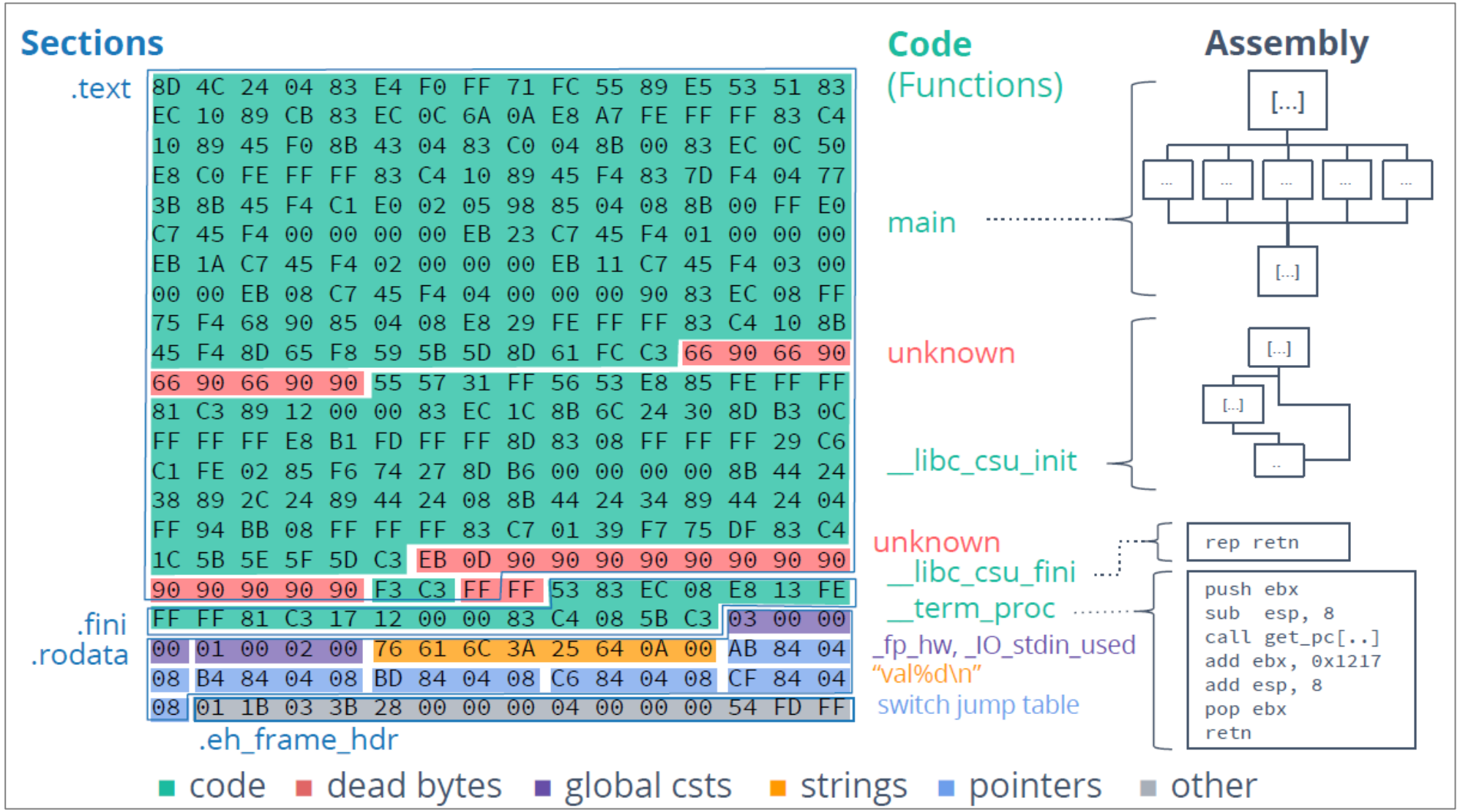


Basic reverse problem

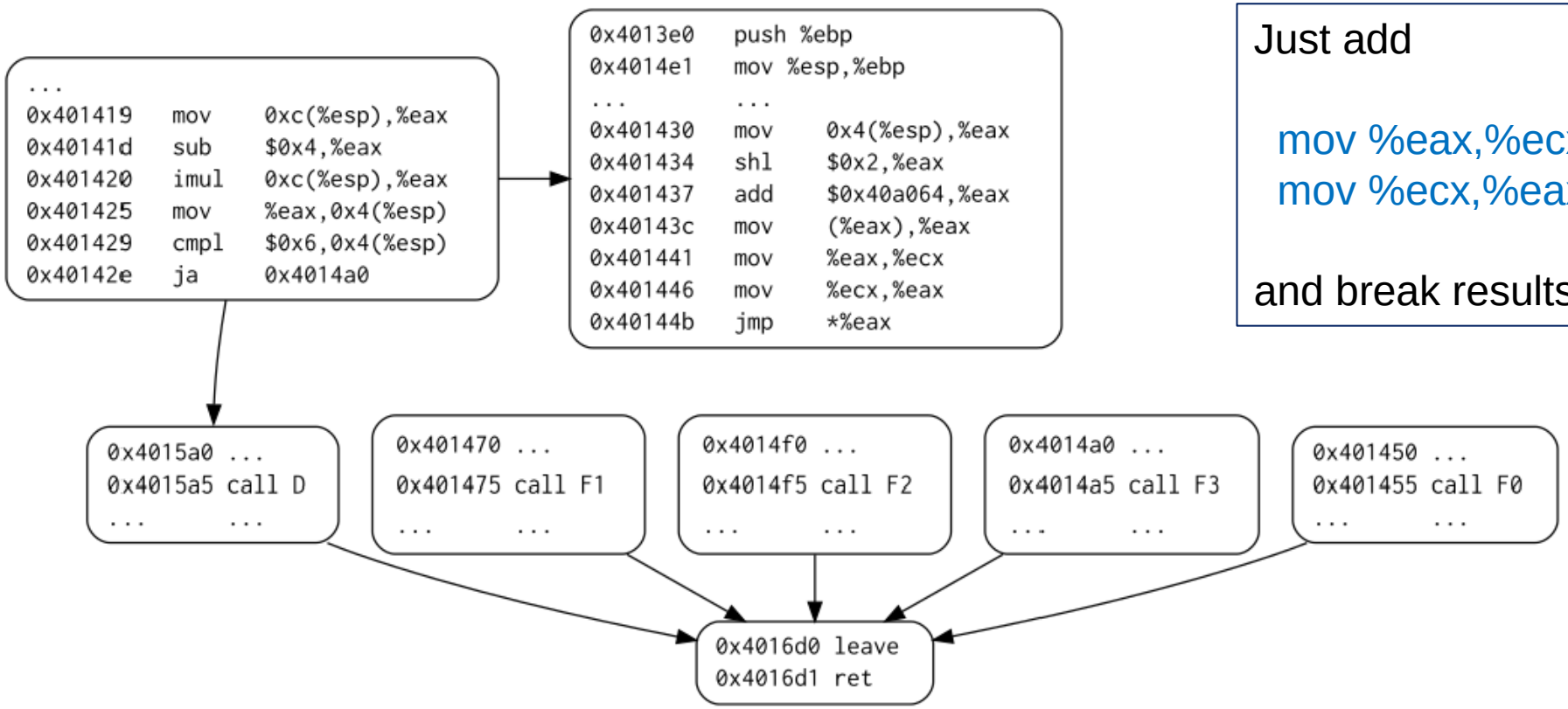
- **aka model recovery**
- **aka CFG recovery**

CAN BE TRICKY!

- code – data
- dynamic jumps (jmp eax)



STATE-OF-THE-ART TOOLS ARE NOT ENOUGH



Just add

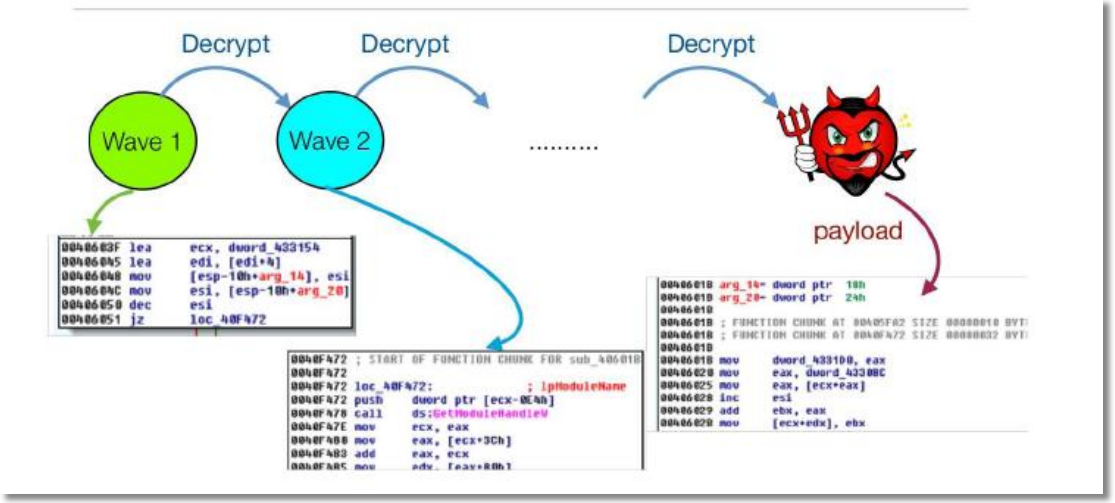
mov %eax,%ecx
mov %ecx,%eax

and break results

With IDA

- **Static (syntactic): too fragile**
- **Dynamic: too incomplete**

[See later] CAN BECOME A NIGHTMARE



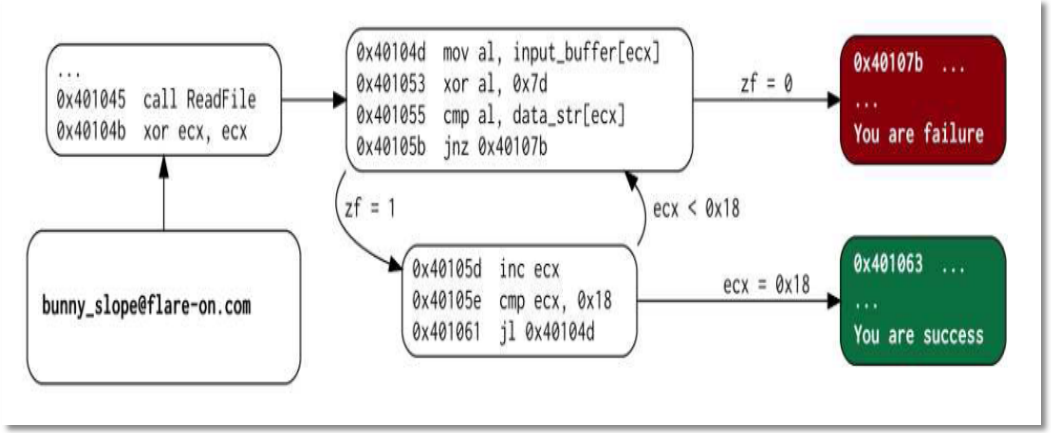
address	instr
80483d1	call +5
80483d6	pop edx
80483d7	add edx, 8
80483da	push edx
80483db	ret
80483dc	.byte{invalid}
80483de	[...]

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

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

↓

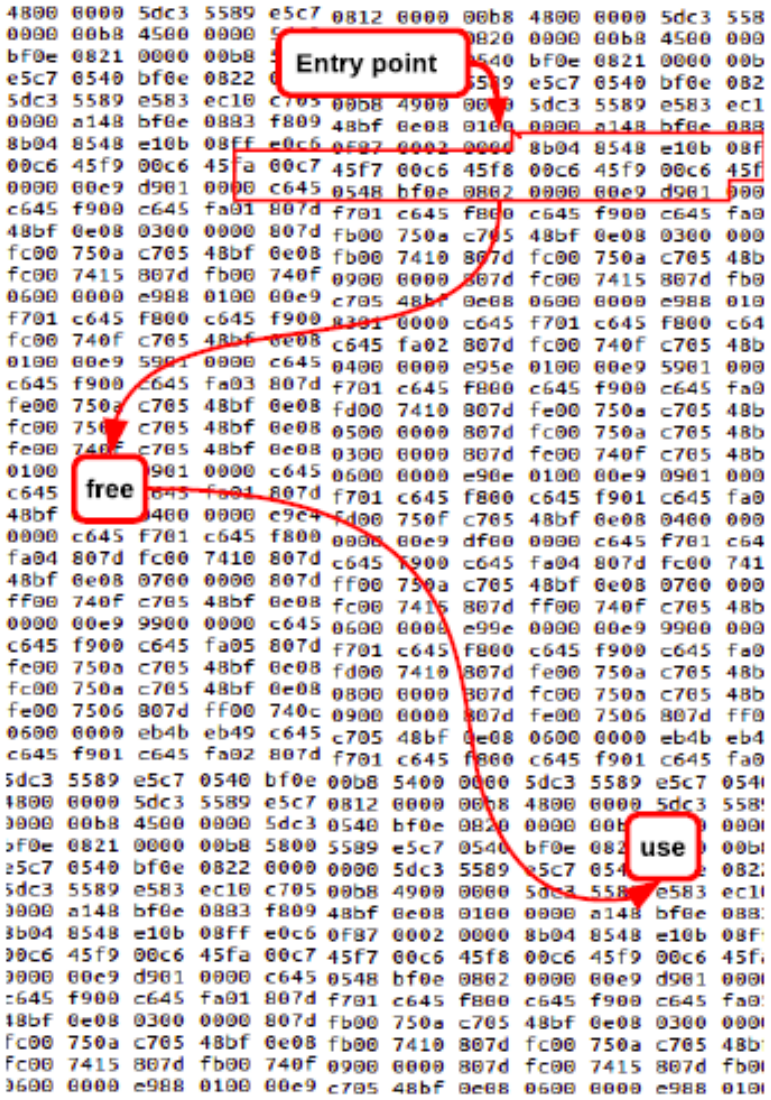
```
mov  eax, ds:X
mov  ecx, ds:Y
imul ecx, ecx
imul ecx, 7
sub  ecx, 1
imul eax, eax
cmp  ecx, eax
jz   <dead_addr>
```



EXAMPLE: VULNERABILITY DETECTION

Find vulnerabilities before the bad guys

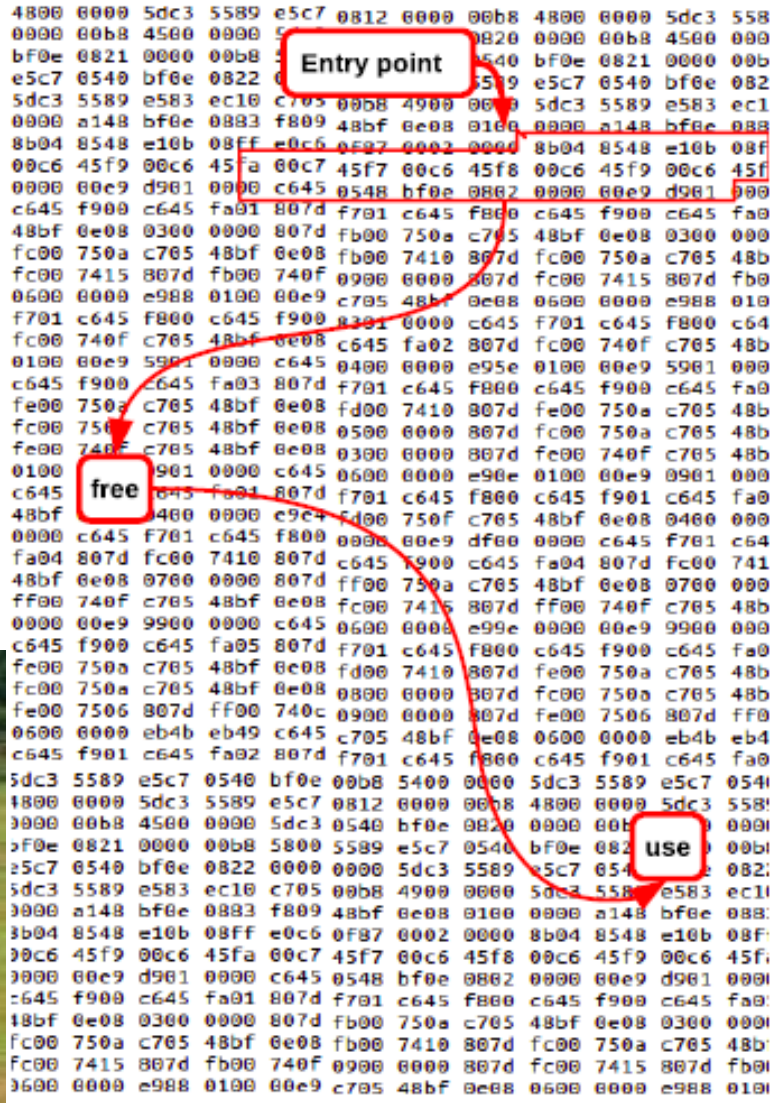
- **On the whole program**
- **At binary-level**
- **Know only the entry point and program input format**



CHALLENGE: In-depth exploration (example: use after free)

Find a needle in the heap !

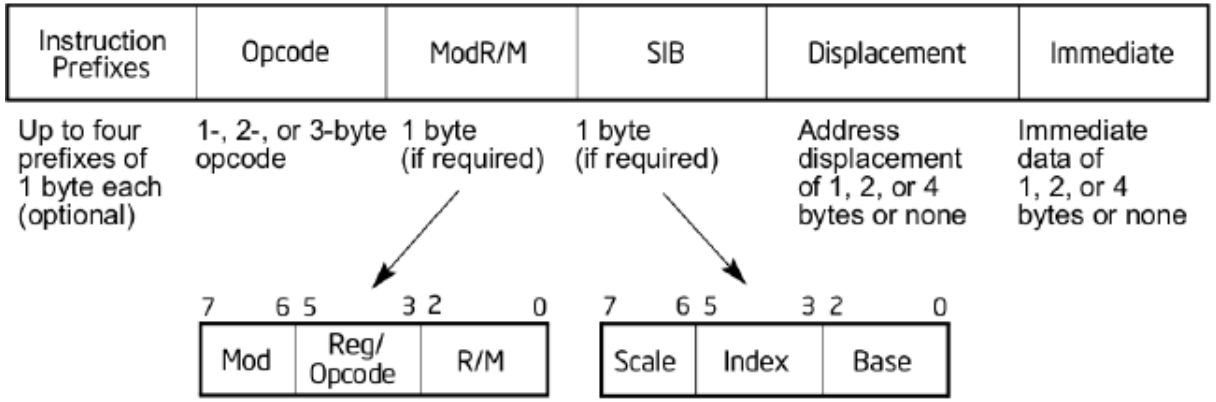
- sequence of events, importance of aliasing
- strongly depend on implem of `malloc` and `free`



Dynamic analysis

- **Too incomplete**

BONUS: (MULTI-)ARCHITECTURE SUPPORT



Example of x86

- more than 1,000 instructions
  - . ≈ 400 basic
  - . + float, interrupts, mmx
- many side-effects
- error-prone decoding
  - . addressing modes, prefixes, ...

r8(/r)			AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)			AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
sreg			ES	CS	SS	DS	FS	GS	res.	res.
eee			CR0	invd	CR2	CR3	CR4	invd	invd	invd
ooo			DR0	DR1	DR2	DR3	DR4[1]	DR5[1]	DR6	DR7
(In decimal) /digit (Opcode)			0	1	2	3	4	5	6	7
(In binary) REG =			000	001	010	011	100	101	110	111
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hex)							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[sib]		100	04	0C	14	1C	24	2C	34	3C
disp32		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[sib]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[sib]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
AL/AX/EAX/ST0/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
CL/CX/ECX/ST1/MM1/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
DL/DX/EDX/ST2/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
BL/BX/EBX/ST3/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
AH/SP/ESP/ST4/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
CH/BP/EBP/ST5/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD

THE SITUATION

- **Binary-level security analysis is necessary**
- **Binary-level security analysis is highly challenging (*)**
- **Standard tools are not enough**

(*) i.e., more challenging than source code analysis

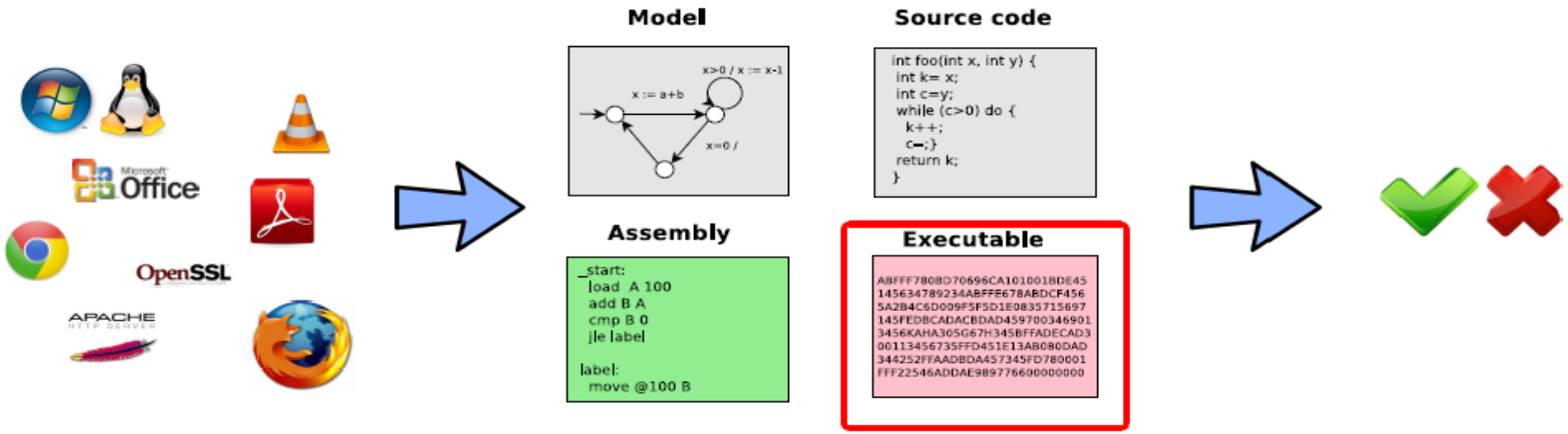
Some challenges

- **correct disassembly (code identification)**
- **multi-architecture support**
- **+ all problems from source-level analysis**

SOLUTION? BINARY-LEVEL SEMANTIC ANALYSIS

Semantic tools help make sense of binary

- Develop the next generation of binary-level tools !
- motto : leverage formal methods from safety critical systems



Advantages

- more robust than syntactic
- more thorough than dynamic

Challenges

- source-level $\mapsto$ binary-level
- safety $\mapsto$ security
- many (complex) architectures

THE HARD JOURNEY FROM SOURCE TO BINARY

Low-level semantics of data

- machine arithmetic, bit-level operations, untyped memory
- ▶ difficult for any state-of-the-art formal technique

Low-level semantics of control

- no distinction data / instructions, dynamic jumps (`jmp eax`)
- no (easy) syntactic recovery of Control-Flow Graph (CFG)
- ▶ violate an implicit prerequisite for most formal techniques

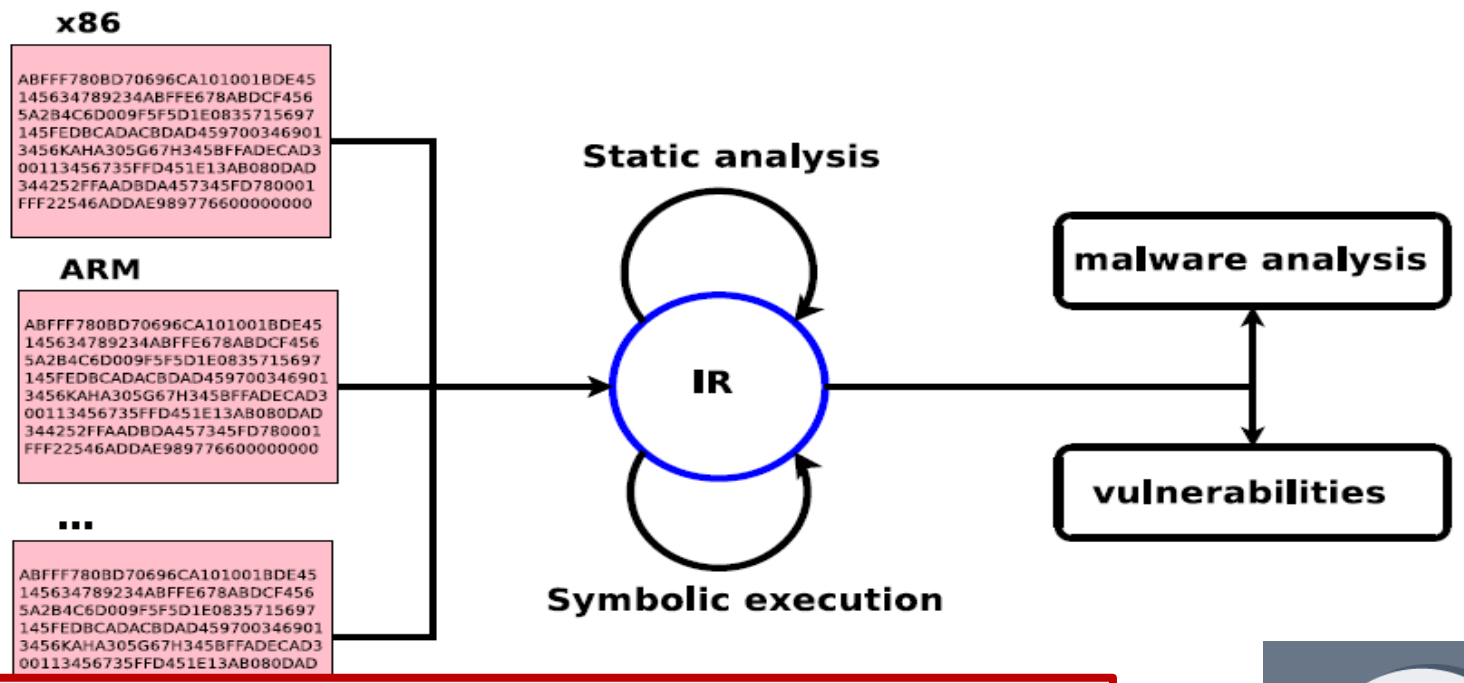
Diversity of architectures and instruction sets

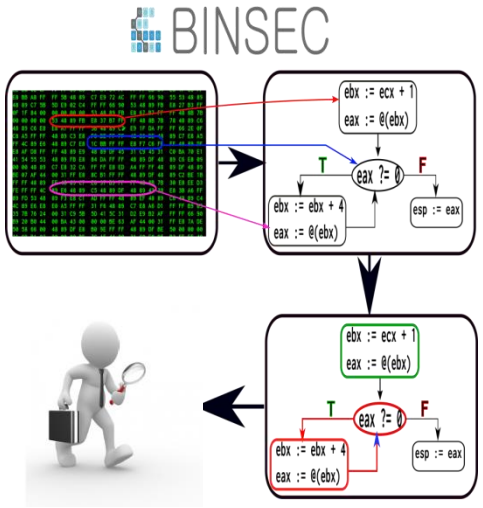
- support for many instructions, modelling issues
- ▶ tedious, time consuming and error prone

Wanted

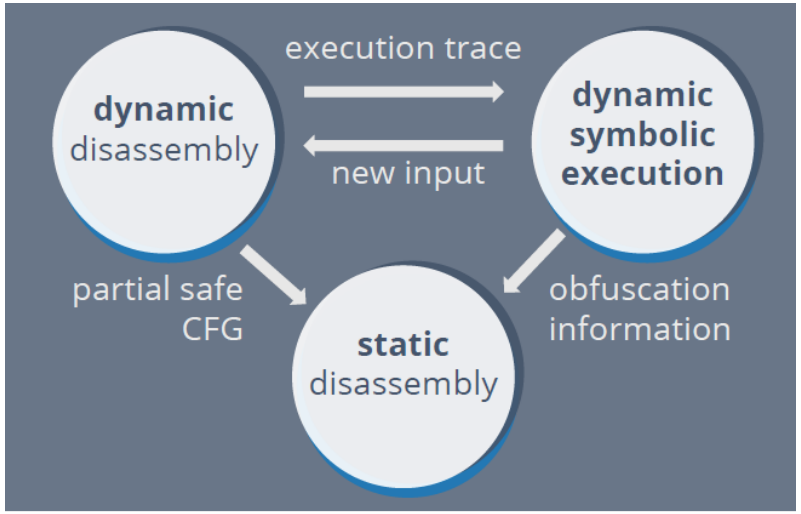
- robustness
- precision
- scale

OUR APPROACH

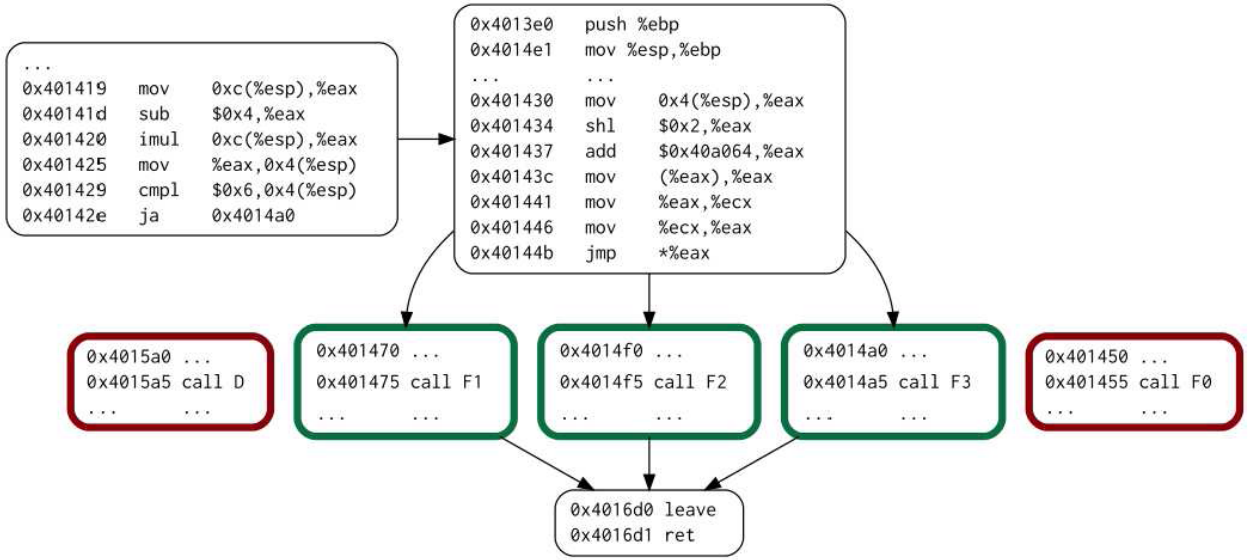




- **A simple Intermediate Representation**
- **A set of « standard » analysis**
  - **Adapted to binary**
  - **Robustness in mind**
  - **Keep correct/complete as much as possible**
- **Domain-dedicated analysis [combination]**



IN PRACTICE




```
cmp eax ebx
cmc
jae ...
```

$CF := (eax <_u ebx)$

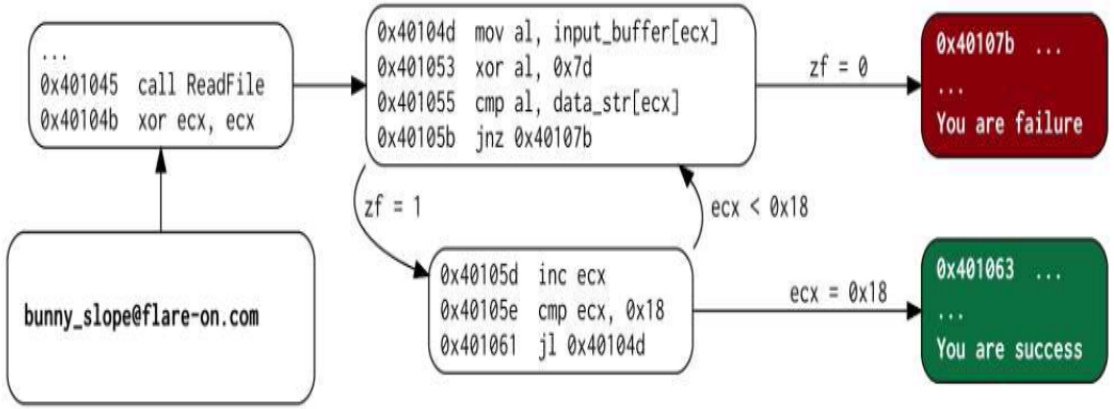
$CF := \neg CF$

$\text{if } (\neg CF) \text{ goto } \ldots$

With IDA + BINSEC

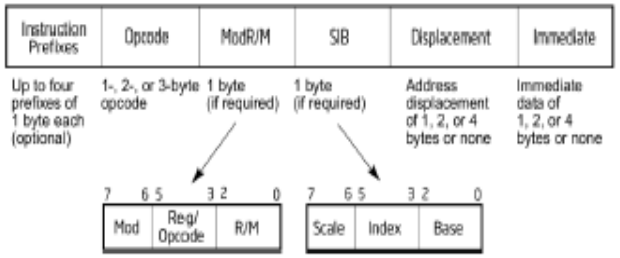
Can recover useful semantic information

- **More precise disassembly**
- **Exact semantic of instructions**
- **Input of interest**
- **…**



KEY 1: INTERMEDIATE REPRESENTATION

[cav'11]



- lhs := rhs
- goto addr, goto expr
- ite(cond)? goto addr

81 c3 57 1d 00 00 $\overset{x86reference}{\Rightarrow}$ ADD EBX 1d57

- **Concise**
- **Well-defined**
- **Clear, side-effect free**

```
(0x29e ,0) tmp  := EBX + 7511;
(0x29e ,1) OF := (EBX{31,31}=7511{31,31}) && (EBX{31,31}<>tmp{31,31});
(0x29e ,2) SF := tmp{31,31};
(0x29e ,3) ZF := (tmp = 0);
(0x28e ,4) AF := ((extu (EBX{0,7}) 9) + (extu 7511{0,7} 9)){8,8};
(0x29e ,6) CF := ((extu EBX 33) + (extu 7511 33)){32,32};
(0x29e ,7) EBX  := tmp; goto (0x2a4 ,0)
```

KEY 1: INTERMEDIATE REPRESENTATION (2)

-- simplifications [tacas'15]

program	native loc	DBA loc	opt (DBA) time	opt (DBA) loc	opt (DBA) red
bash	166K	559K	673.61s	389K	30.45%
cat	8K	23K	18.54s	18K	23.02%
echo	4K	10K	6.96s	8K	24.26%
less	23K	80K	69.99s	55K	30.96%
ls	19K	63K	65.69s	44K	30.58%
mkdir	8K	24K	19.74s	17K	29.50%
netstat	17K	50K	52.59s	40K	20.05%
ps	12K	36K	36.99s	27K	23.98%
pwd	4K	11K	7.69s	9K	23.56%
rm	10K	30K	24.93s	22K	25.24%
sed	10K	32K	28.85s	23K	26.20%
tar	64K	213K	242.96s	154K	27.48%
touch	8K	26K	24.28s	18K	27.88%
uname	3K	10K	6.99s	8K	23.62%

	reduction			
	time	dba instr	tmp assigns	flag assigns
BINSEC	1279.81s	28.64%	88.00%	75.04%

Approach

- Inspired from standard compiler optim
- Targets : flags & temp
- Sound : w.r.t. incomplete CFG
- Inter-procedural (summaries)

- DBA level
- Instruction level
- Program level

KEY 2: PRECISE & ROBUST SYMBOLIC REASONNING (DSE, Godefroid 2005)

```
int main () {
    int x = input();
    int y = input();
    int z = 2 * y;
    if (z == x) {
        if (x > y + 10)
            failure;
    }
    success;
}
```

- given a path of the program
- automatically find input that follows the path
- then, iterate over all paths

$\sigma := \emptyset$
$\mathcal{PC} := \top$

Perfect for intensive testing

- **Correct, relatively complete**
- **No false alarm**
- **Robust**
- **Scale in some ways**

// incomplete

$\mathcal{PC} := \top \wedge 2y_0 \neq x_0$

$\mathcal{PC} := \top \wedge 2y_0 = x_0 \wedge x_0 > y_0 + 10$

$\mathcal{PC} := \top \wedge 2y_0 = x_0 \wedge x_0 \leq y_0 + 10$

KEY 2: DSE, about ROBUSTNESS (imo, the major advantage)

Goal = find input leading to ERROR

(assume we have only a solver for linear integer arith.)

```
g(int x) {return x*x; }
f(int x, int y) {z=g(x); if (y == z) ERROR; else OK }
```

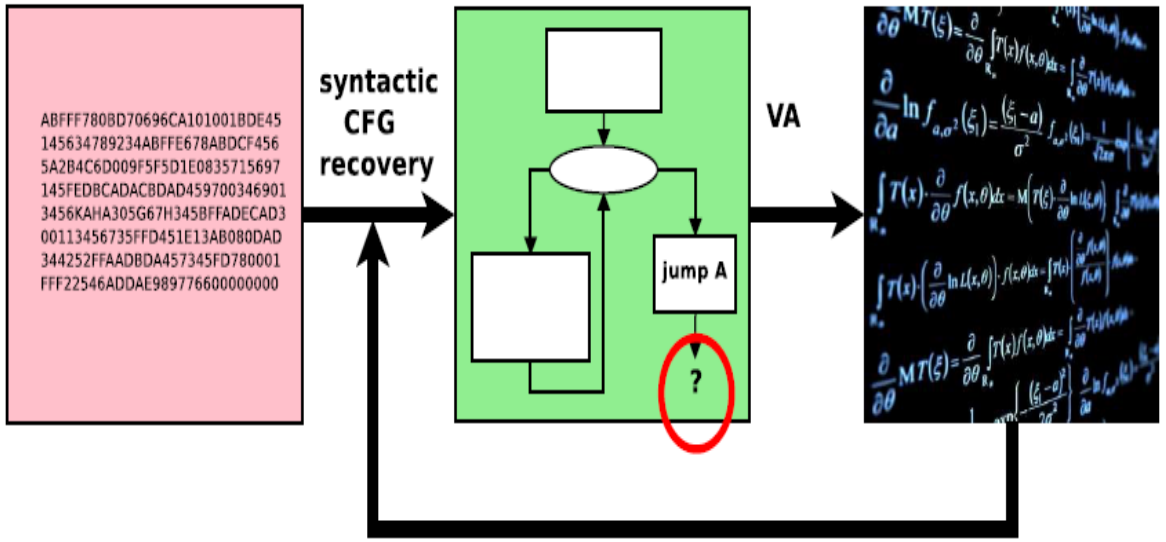
Symbolic Execution

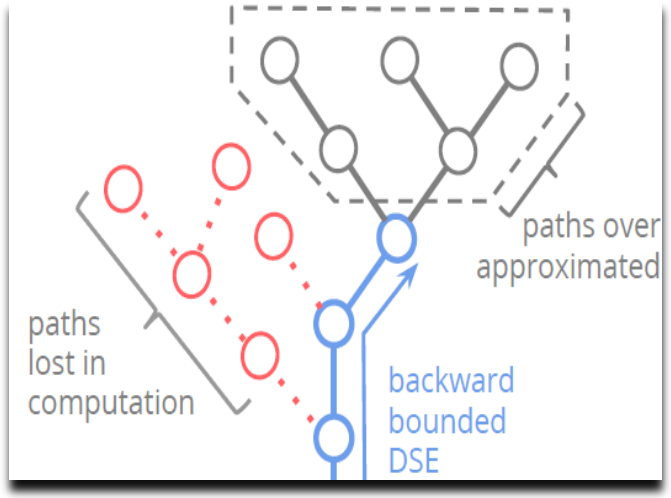
- create a subformula $z = x * x$, out of theory [FAIL]

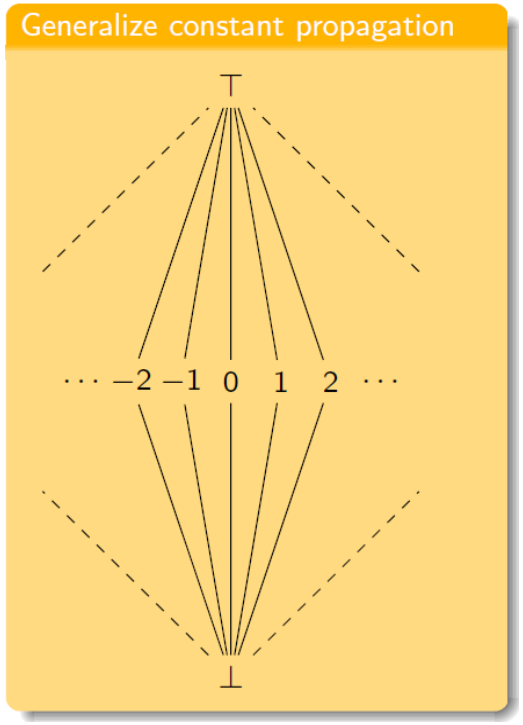
Dynamic Symbolic Execution

- first concrete execution with x=3, y=5 [goto OK]
- during path predicate computation, $x * x$ not supported
  . x is concretized to 3 and z is forced to 9
- resulting path predicate : $x = 3 \wedge z = 9 \wedge y = z$
- a solution is found : x=3, y=9 [goto ERROR] [SUCCESS]

And more … (more or less experimental)



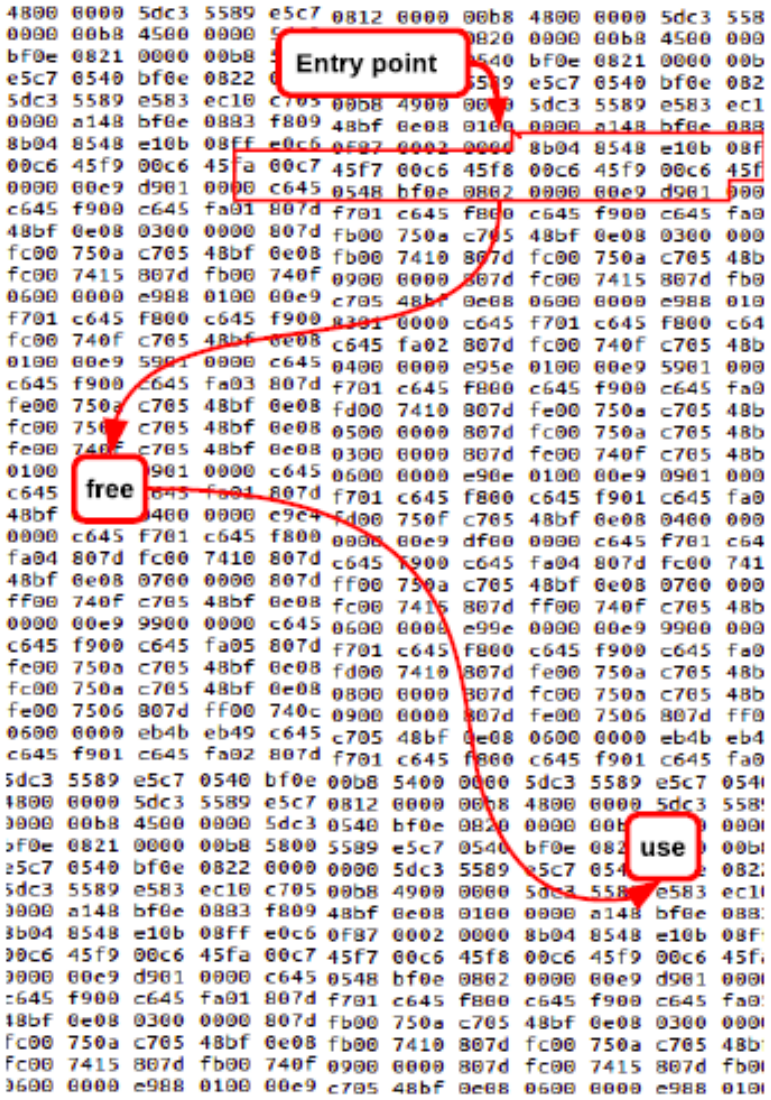




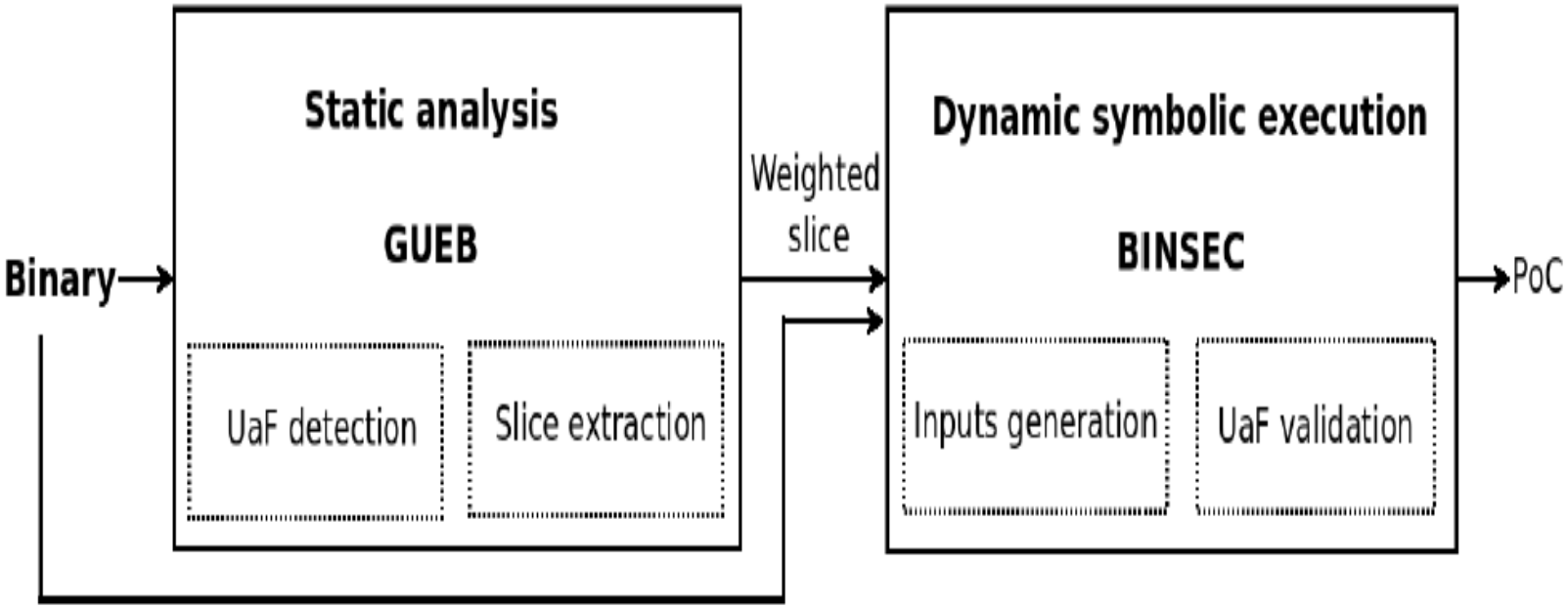
APPLICATION: VULNERABILITY DETECTION [SSPREW 2016, with VERIMAG]

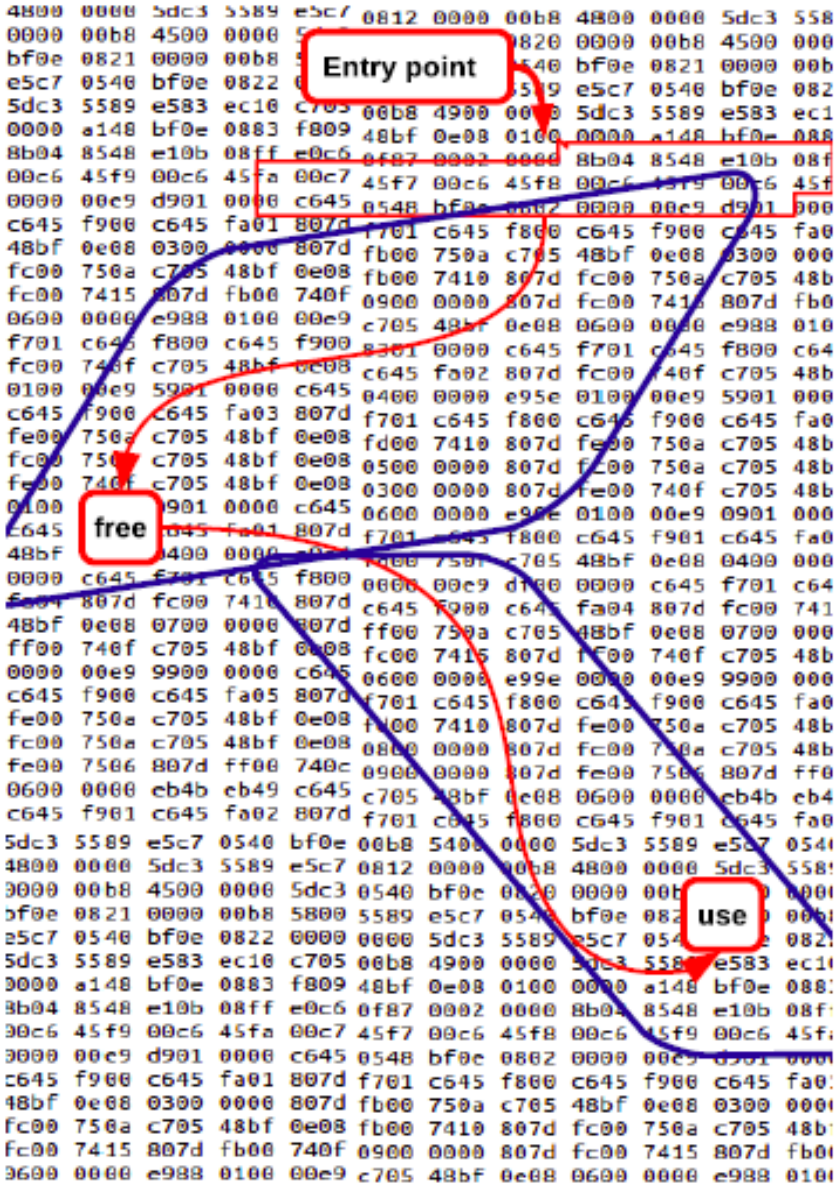
Find vulnerabilities before the bad guys

- **On the whole program**
- **At binary-level**
- **Know only the entry point and program input format**



KEY IDEAS (Josselin Feist)





A Pragmatic 2-step approach

- **Static: scale, not complete, not correct**
- **Symbolic: correct, directed by static**
- **Combination: scalable and correct**

EXPERIMENTAL EVALUATION

- GUEB + manual analysis [j. comp. virology 14]
  - tiff2pdf : CVE-2013-4232
  - openjpeg : CVE-2015-8871
  - gifcolor : CVE-2016-3177
  - accel-ppp
- GUEB + BINSE/SE [ssprew16]
  - Jasper JPEG-2000 : CVE-2015-5221

On these examples:

- **Better than DSE alone**
- **Better than blackbox fuzzing**
- **Better than greybox fuzzing with no seed**

APPLICATION: MALWARE DEOBFUSCATION [S&P 2017, with LORIA]

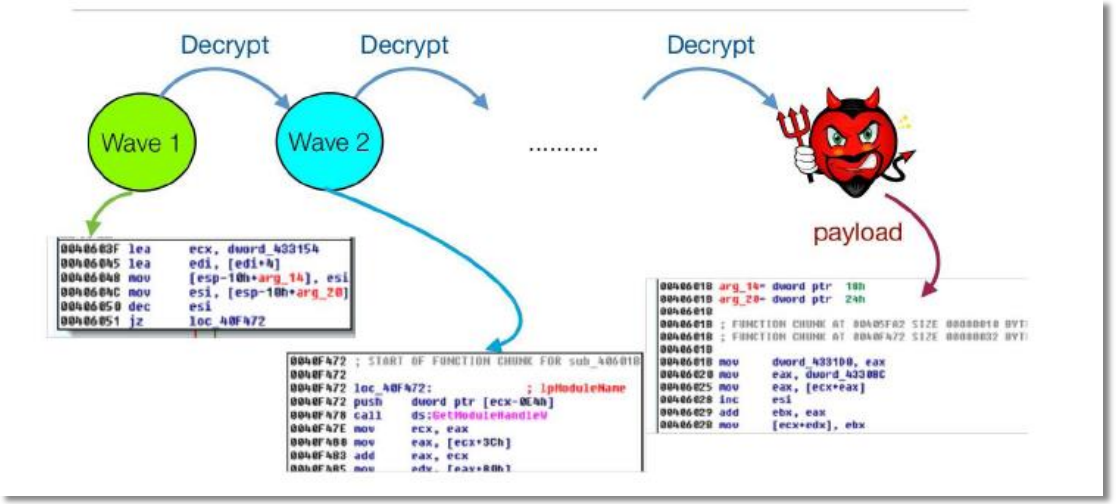
APT: highly sophisticated attacks
- **Targeted malware**
- **Written by experts**
- Attack: 0-days
- Defense: stealth, **obfuscation**
- **Sponsored by states or mafia**

The day after: **malware comprehension**
- understand what has been going on
- mitigate, fix and clean
- improve defense

USA elections: DNC Hack

Goal: help malware comprehension
- **Reverse of heavily obfuscated code**
- **Identify and simplify protections**

REVERSE CAN BECOME A NIGHTMARE (OBFUSCATION)



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

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>
```

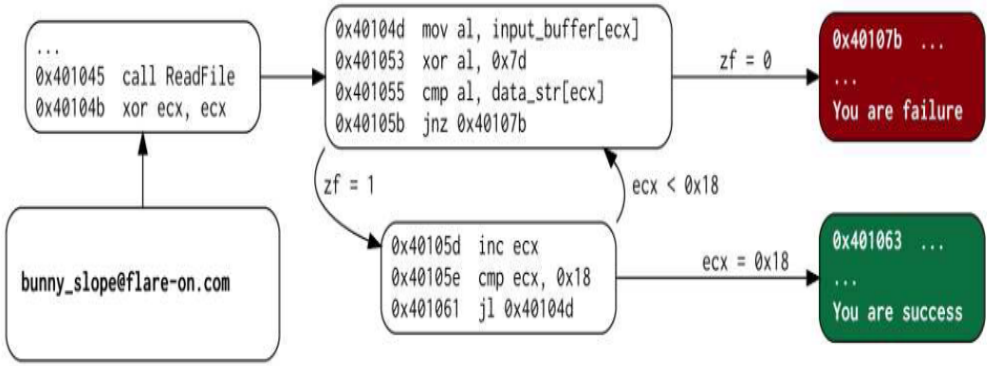
Obfuscatio

hard to reve

- self-modification
- encryption
- virtualization
- code overlapping
- opaque predicates
- callstack tampering
- …

Goal: help malware comprehension

- **Identify and simplify protections**



EXAMPLE: OPAQUE PREDICATE

Constant-value predicates

(always true, always false)

- dead branch points to spurious code
- goal = waste reverser time & efforts

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>
```

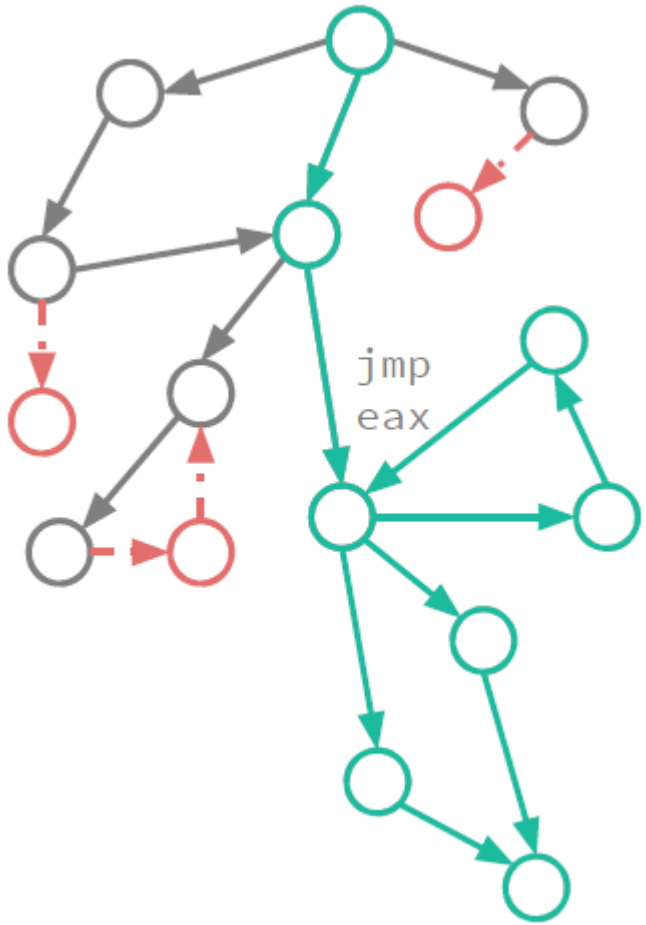
EXAMPLE: STACK TAMPERING

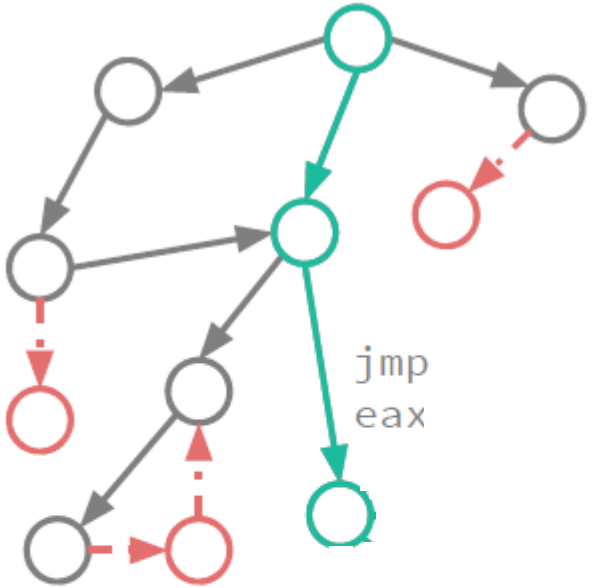
Alter the standard compilation scheme:
ret do not go back to call

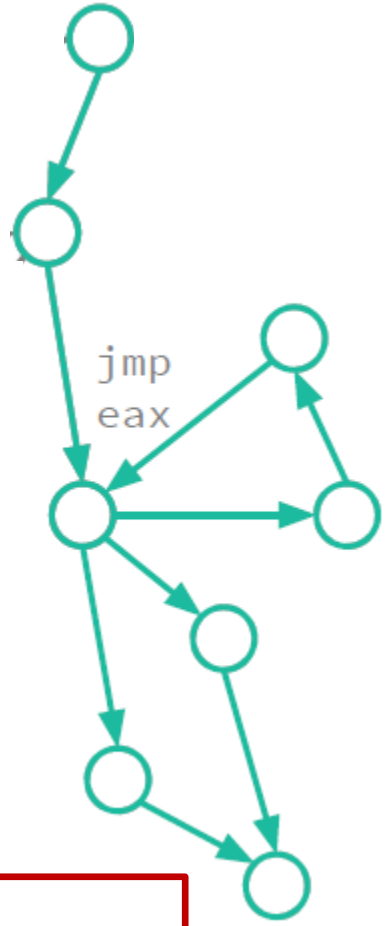
- hide the real target
- return site may be spurious code

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

STANDARD DISASSEMBLY TECHNIQUES ARE NOT ENOUGH







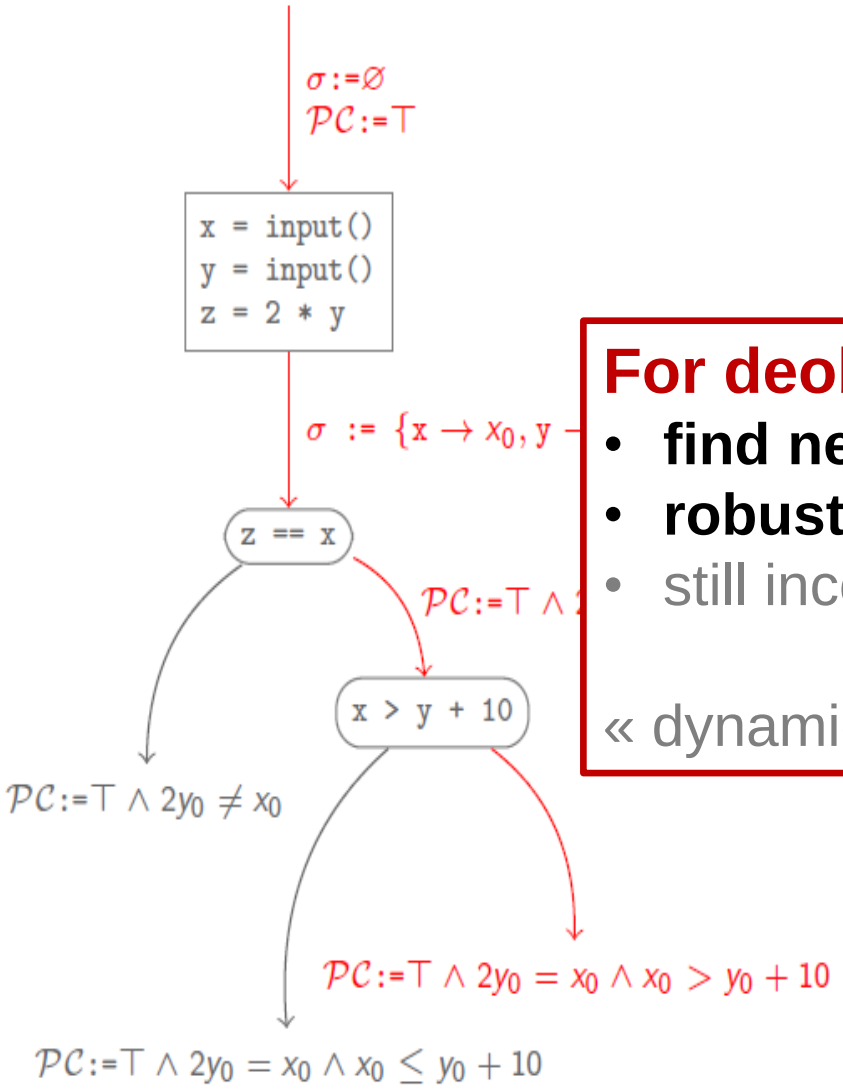
Static analysis
- too fragile vs obfuscation
- junk instr, missed instr.

Dynamic analysis
- robust vs obfuscation
- too incomplete

DYNAMIC SYMBOLIC EXECUTION CAN HELP (Debray, Kruegel, …)

```
int main () {
    int x = input();
    int y = input();
    int z = 2 * y;
    if (z == x) {
        if (x > y + 10)
            failure;
    }
    success;
}
```

- given a path of the program
- automatically find input that follows the path
- then, iterate over all paths



For deobfuscation

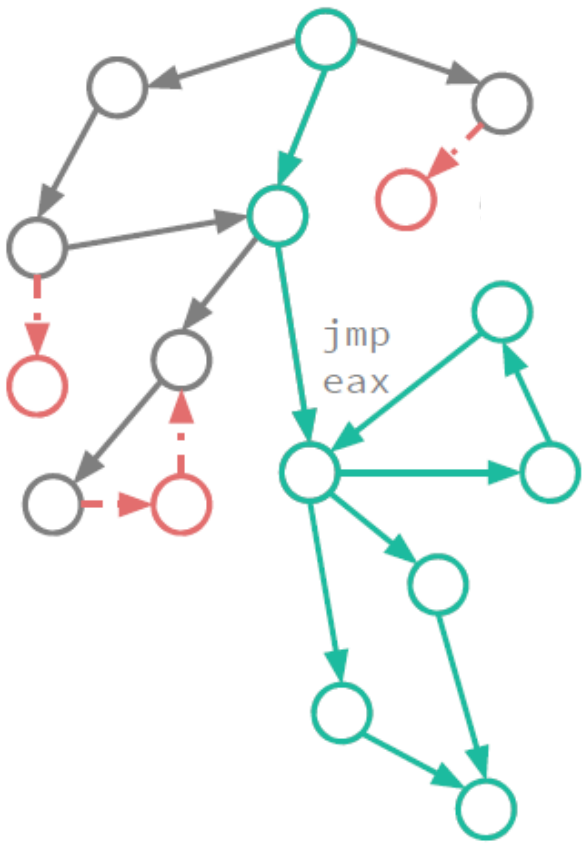
- **find new real paths**
- **robust**
- still incomplete

« dynamic analysis on steroids »

YET … WHAT ABOUT INFEASIBILITY QUESTIONS?

Prove that something is always true (resp. false)

Many such issues in reverse
- **is a branch dead?**
- **does the ret always return to the call?**
- **have i found all targets of a dynamic jump?**



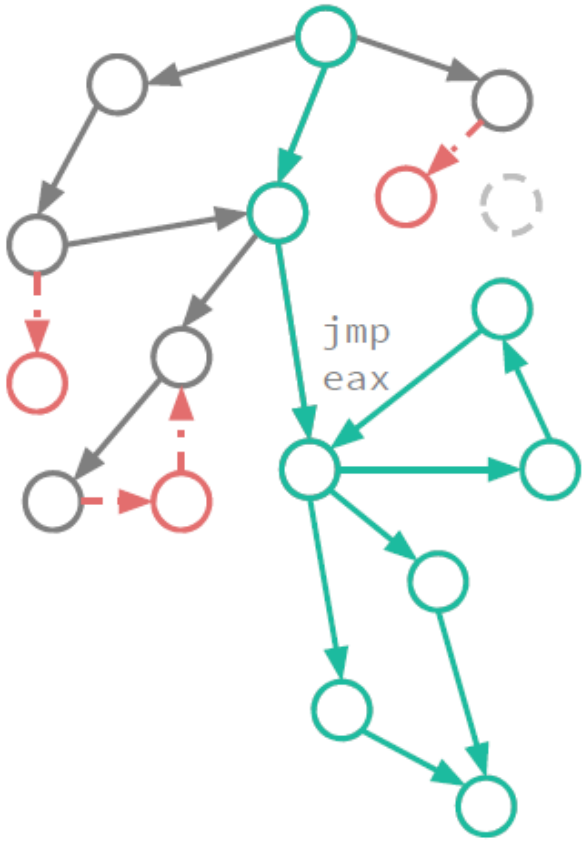
And more
- **does this malicious ret always go there?**
- **does this expression always evaluate to 15?**
- **does this self-modification always write this opcode?**
- **does this self-modification always rewrite this instr.?**
- **…**

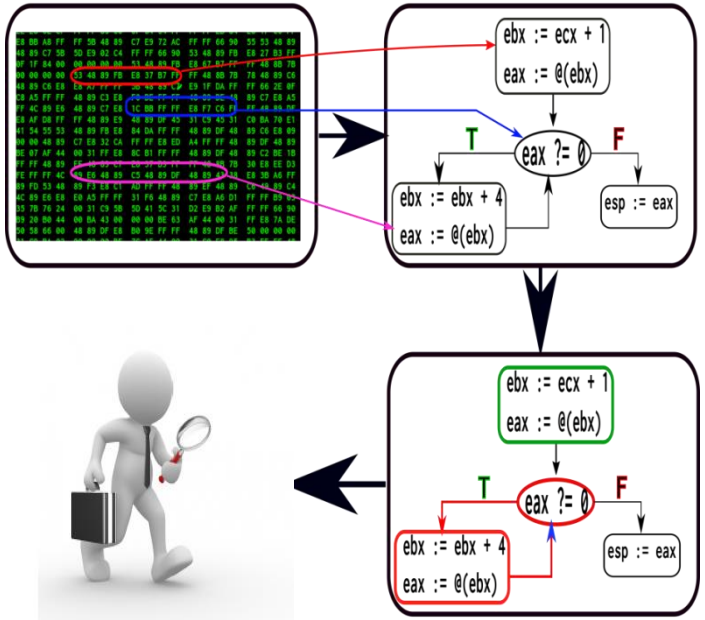
Not addressed by DSE
- **Cannot enumerate all paths**

OUR CHALLENGE

Check infeasibility questions in obfuscated codes

- **scale to realistic malware sizes**
- **robust to obfuscation such as self-modification**
- **precise**
- **generic**





Rest of the talk:

- **opaque predicate**
- **stack tampering**

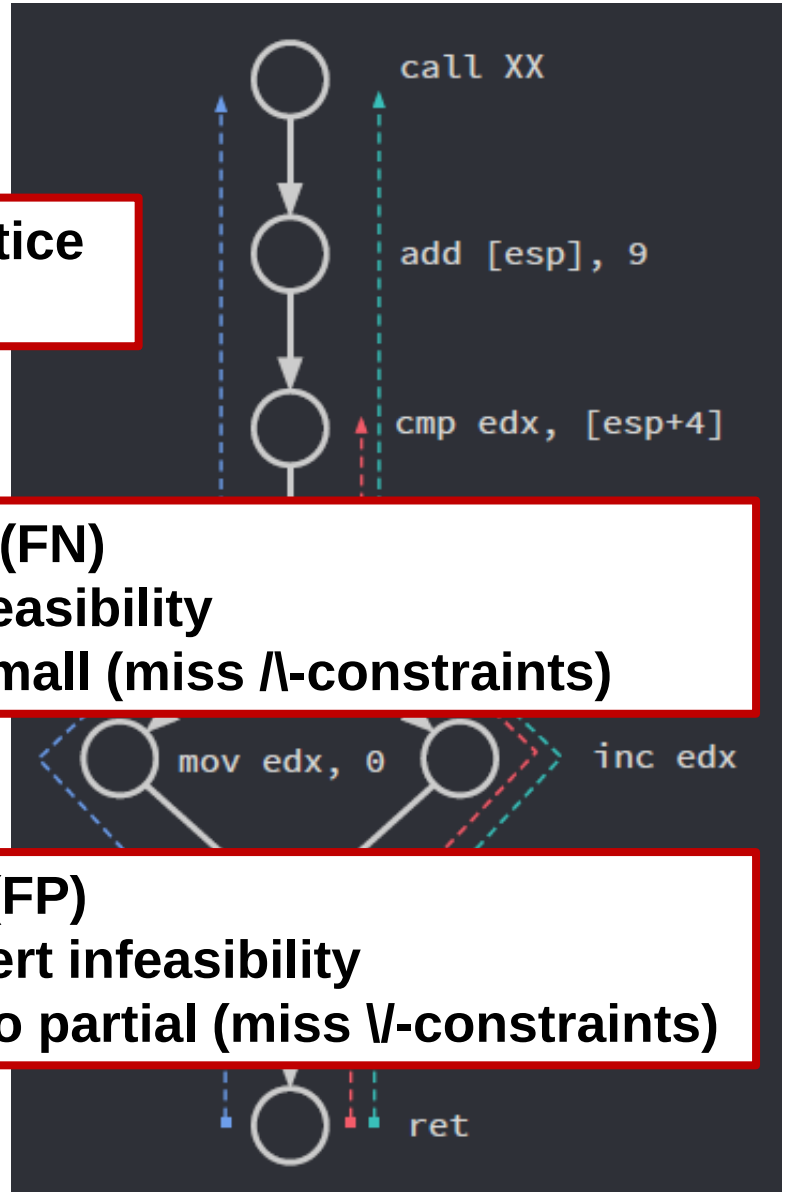
OUR PROPOSAL: BACKWARD-BOUNDED SYMBOLIC EXECUTION

Insight 1: symbolic reasoning
- precision
- But: need finite #paths

Insight 2: backward-bounded
- pre_k(c)=0 => c is infeasible
- finite #paths
- efficient, depends on k
- But: backward on jump eax?

Insight 3: dynamic partial CFG
- solve (partially) dyn. jumps
- robustness

Low FP/FN rates in practice
- ground truth xp



False negative (FN)
- can miss infeasibility
- why: k too small (miss /\-constraints)

False positive (FP)
- wrongly assert infeasibility
- why: CFG too partial (miss \/-constraints)

EXPERIMENTAL EVALUATION

- **Controlled experiments** (ground truth) → **precision**
- **Large-scale experiment: packers** → **scalability, robustness**
- **Case-study: X-tunnel malware** → **usefulness**

CONTROLLED EXPERIMENTS

- **Goal = assess the precision of the technique**
  - ground truth value

- **Experiment 1: opaque predicates (o-llvm)**
  - 100 core utils, 5x20 obfuscated codes
  - k=16: 3.46% error, no false negative
  - robust to k
  - efficient: 0.02s / query

- **Experiment 2: stack tampering (tigress)**
  - 5 obfuscated codes, 5 core utils
  - almost all genuine ret are proved (no false positive)
  - many malicious ret are proved « single-targets »

k	OP (5556)		Genuine (5183)		TO	Error rate	Time	avg/query
	ok	miss (FN)	ok	miss (FP)		(FP+FN)/Tot (%)	(s)	(s)
2	0	5556	5182	1	0	51.75	89	0.008
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	96	0.009
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	120	0.011
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	**152**	**0.014**
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	**197**	**0.018**
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	**272**	**0.025**
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	**384**	**0.036**
32	5552	4	4579	604	25	5.66	699	0.065
40	5548	8	4523	660	39	6.22	1145	0.107
50	5544	12	4458	725	79	6.86	2025	0.189

- **Very precise results**
- **Seems efficient**

Sample	runtime genuine			runtime violation		
	#ret †	proved genuine	proved a/d	#ret †	proved a/d	proved single
obfuscated programs						
simple-if	6	6	6/0	9	0/0	8
bin-search	15	15	15/0	25	0/0	24
bubble-sort	6	6	6/0	15	0/1	13
mat-mult	31	31	31/0	69	0/0	68
huffman	19	19	19/0	23	0/3	19
non-obfuscated programs						
ls	30	30	30/0	0	-	-
dir	35	35	35/0	0	-	-
mktemp	21	20	20/0	0	-	-
od	21	21	21/0	0	-	-
vdir	49	43	43/0	0	-	-

CASE-STUDY: PACKERS

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	[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

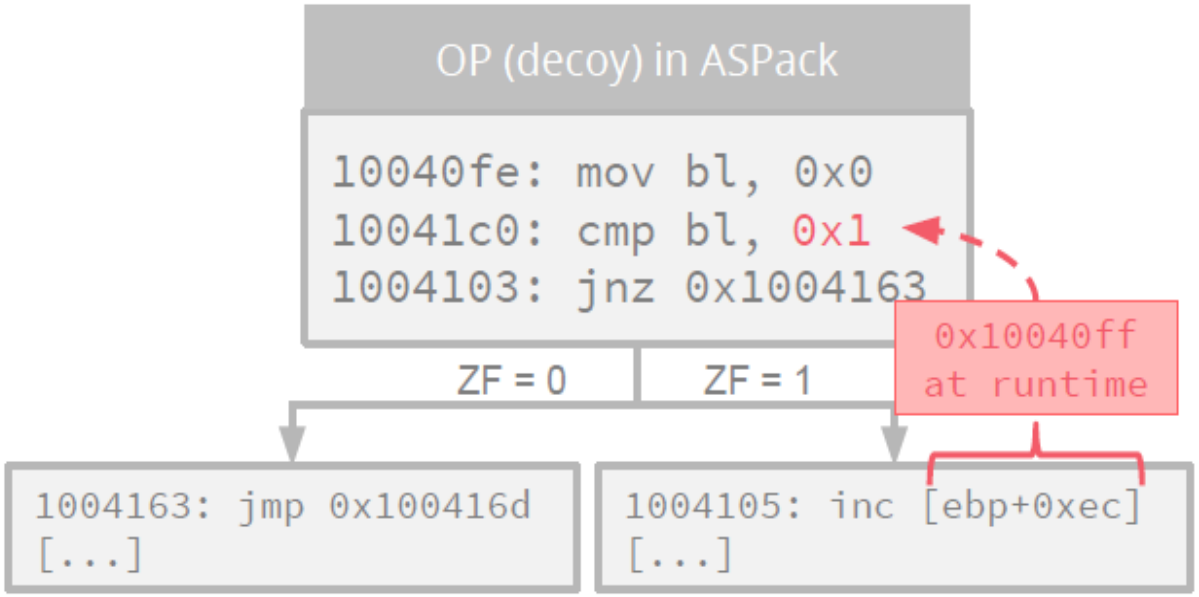
The technique scale on significant traces

Many true positives. Some packers are using it intensively

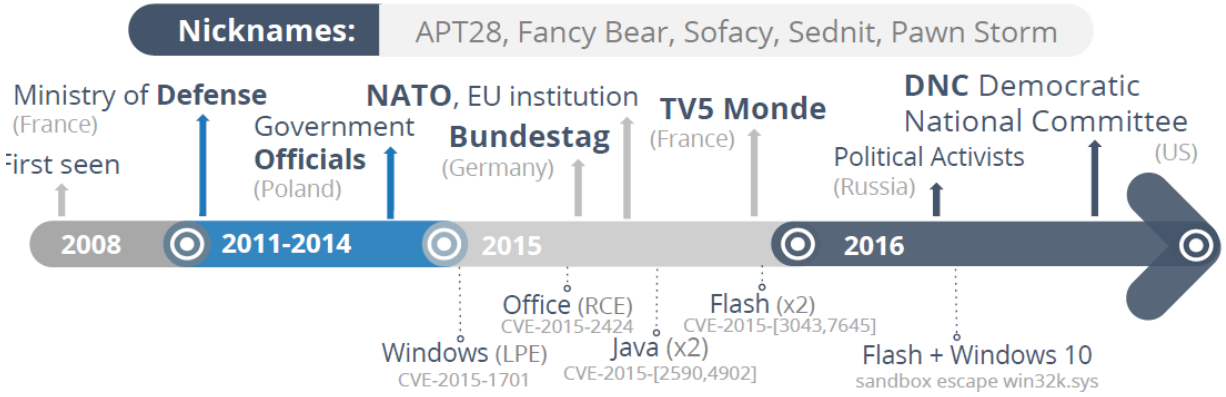
Packers using ret to perform the final tail transition to the entrypoint

Packers: legitimate software protection tools (basic malware: the sole protection)

CASE-STUDY: PACKERS (fun facts)



CASE-STUDY: THE XTUNNEL MALWARE (part of DNC hack)



Two heavily obfuscated samples

- **Many opaque predicates**

Goal: detect & remove protections

- Identify 50% of code as spurious
- Fully automatic, < 3h

	C637 Sample #1	99B4 Sample #2
#total instruction	**505,008**	**434,143**
#alive	+279,483	+241,177

CASE-STUDY: THE XTUNNEL MALWARE (fun facts)

- **Protection seems to rely only on opaque predicates**
- **Only two families of opaque predicates** $7y^2 - 1 \neq x^2$ $\frac{2}{x^2+1} \neq y^2 + 3$
- **Yet, quite sophisticated**
  - original OPs
  - interleaving between payload and OP computation
  - sharing among OP computations
  - possibly long dependencies chains (avg 8.7, upto 230)

CONCLUSION & TAKE AWAY

- **Binary-level security analysis**
  - Many applications, many challenges
  - Current syntactic and dynamic methods are not enough

- **Formal methods can change the game … but must be strongly adapted**
  - Complement existing approaches
  - Need robustness and scalability!
  - Acceptable to lose both correctness & completeness – in a controlled way
  - **New challenges and variations, many things to do!**

- **First results**
  - Advanced pocs and vulnerability detection and malware deobfuscation
  - BINSEC: open-source, still in its infancy, on heavy refactoring

// Advertisement: International Summer School on Software Protection @ Saclay

Commissariat à l'énergie atomique et aux énergies alternatives
Institut List | CEA SACLAY NANO-INNOV | BAT. 861 – PC142
91191 Gif-sur-Yvette Cedex - FRANCE
www-list.cea.fr

Établissement public à caractère industriel et commercial | RCS Paris B 775 685 019