BINSEC:
Binary-level Semantic Analysis to the Rescue

Sébastien Bardin
joint work with
Richard Bonichon, Robin David, Adel Djoudi, Benjamin Farinier, Josselin Feist, Laurent Mounier, Marie-Laure Potet, Thanh Dihn Ta, Franck Védrine

CEA LIST (Paris-Saclay, France)

About the BINSEC project

A research project :

- funded by ANR (2013-2017)
- axis 1 (security) and 2 (software engineering)
- formal techniques for binary-level security analysis

Partners : CEA (coordinator), Airbus Group, INRIA Bretagne Atlantique, Université Grenoble Alpes, Université de Lorraine

People : Sébastien Bardin, Frédéric Besson, Sandrine Blazy, Guillaume Bonfante, Richard Bonichon, Robin David, Adel Djoudi, Benjamin Farinier, Josselin Feist, Colas Le Guernic, Jean-Yves Marion, Laurent Mounier, Marie-Laure Potet, Than Dihnh Ta, Franck Védrine, Pierre Wilke, Sara Zennou

Platform : CEA, Université Grenoble Alpes

Takeaway

Binary-level security analysis

- many applications, many challenges
- syntactic and dynamic methods are not sufficient

Semantic approaches can help !

- semantic exploration, semantic disassembly
- yet, still hard to design

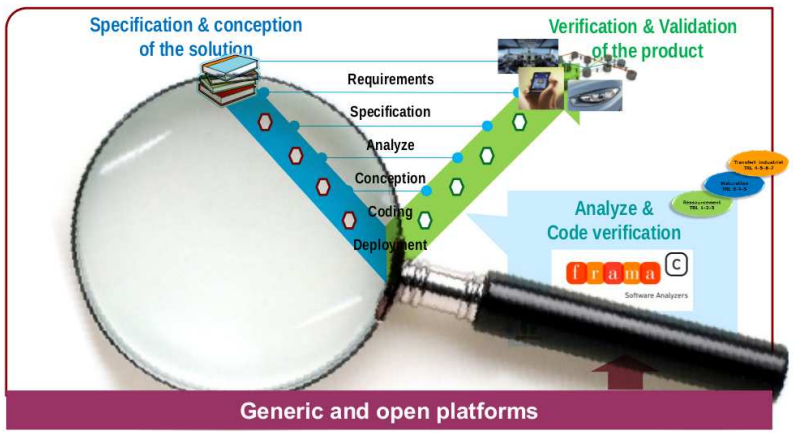
The BINSEC Platform [CEA & Uni. Grenoble Alpes]

- open source, dual goal :
  - help design new binary-level analyzers (basic building blocks)
  - provide innovative analyzers
- current : multi-architecture support, semantic exploration & semantic disassembly, poc on vulnerability analysis and deobfuscation
- still young : beta-version just released [http ://binsec.gforge.inria.fr/]

About my lab @CEA

CEA LIST, Software Safety & Security Lab

- rigorous tools for building high-level quality software
- 2nd part of V-cycle
- automatic software analysis
- mostly source code



frama C Software Analyzers

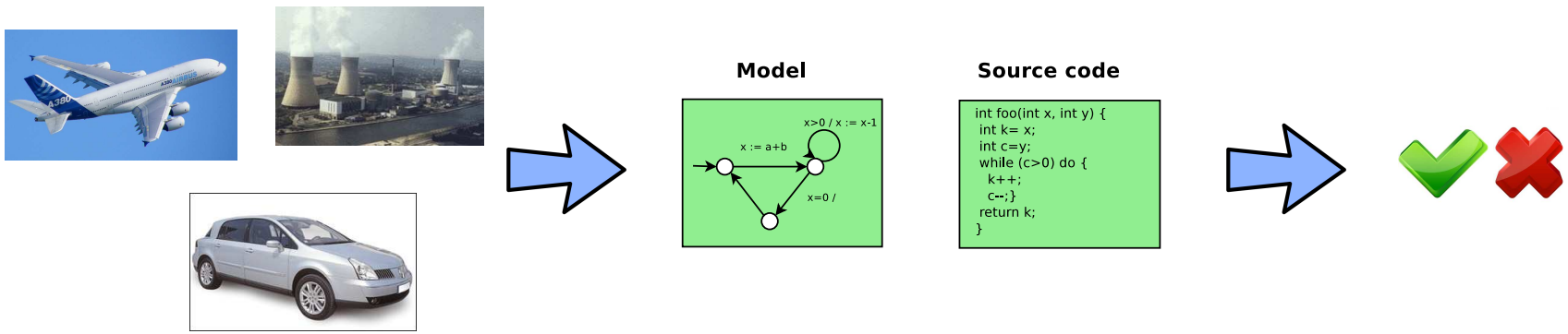
GATeL

PathCrawler

POLAR SSL

About formal verification

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



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

From (a logician's) dream to reality

Industrial reality in some key areas, especially safety-critical domains

- hardware, aeronautics [airbus], railroad [metro 14], smartcards, drivers [Windows], certified compilers [CompCert] and OS [Sel4], etc.

Ex : Airbus

Verification of

- runtime errors [Astrée]
- functional correctness [Frama-C *]
- numerical precision [Fluctuat *]
- source-binary conformance [CompCert]
- ressource usage [Absint]

* : by CEA DILS/LSL

From (a logician's) dream to reality

Industrial reality in some key areas, especially safety-critical domains

- hardware, aeronautics [airbus], railroad [metro 14], smartcards, drivers [Windows], certified compilers [CompCert] and OS [Sel4], etc.

Ex : Microsoft

Verification of drivers [SDV]

- conformance to MS driver policy
- home developers
- and third-party developers

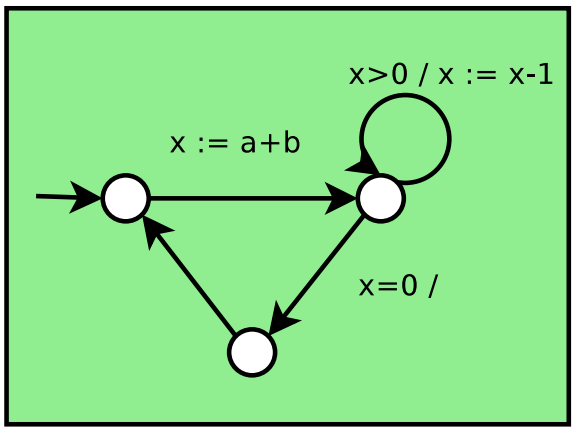
Things like **even software verification**, *this has been the Holy Grail of computer science for many decades but now in* **some very key areas**, *for example, driver verification we're building* **tools that can do actual proof about the software** *and how it works in order to guarantee the reliability.*

- Bill Gates (2002)

Outline

Binary-level software analysis

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
```

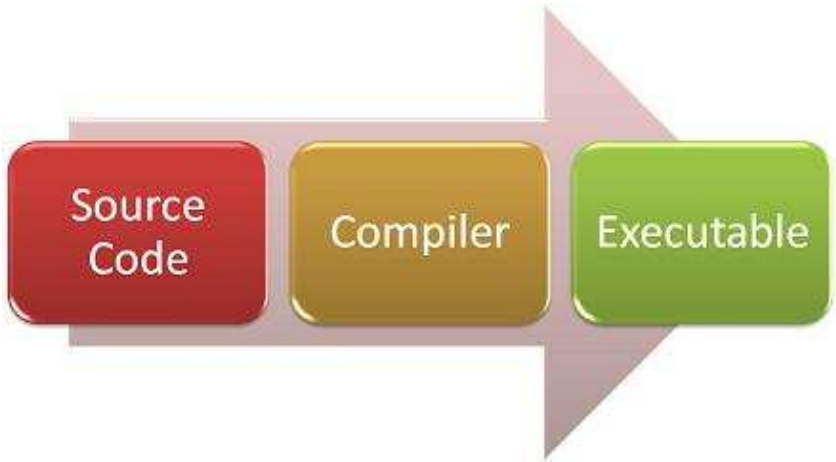
Executable

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

What for ? (1)

How much do you trust your external components ?

What for ? (2)



How much do you trust your compiler ?

What for ? (2)

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));
}
```

- 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

Mentioned in OpenSSH CVE-2016-0777

What for ? (3)

Is it Stuxnet ?

Outline

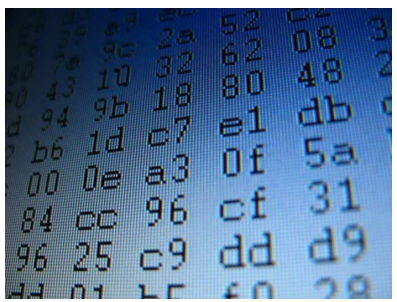
Binary-level security analysis

Several major security analyses are performed at byte-level

- vulnerability analysis [exploit finding]
- malware dissection and detection [deobfuscation]

State-of-the-technique

- very skilled experts, many efforts and **basic tools**
- **dynamic analysis** : gdb, fuzzing [easy to miss behaviours]
- **syntactic analysis** : objdump, IDA Pro [easy to get fooled]



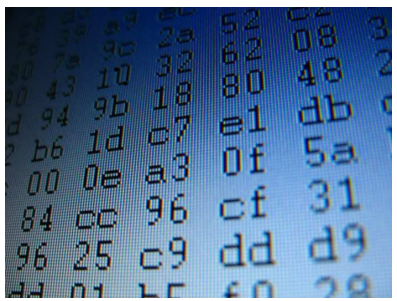
Binary-level security analysis

Several major security analyses are performed at byte-level

- vulnerability analysis [exploit finding]
- malware dissection and detection [deobfuscation]

State-of-the-technique

- very skilled experts, many efforts and **basic tools**
- **dynamic analysis** : gdb, fuzzing [easy to miss behaviours]
- **syntactic analysis** : objdump, IDA Pro [easy to get fooled]

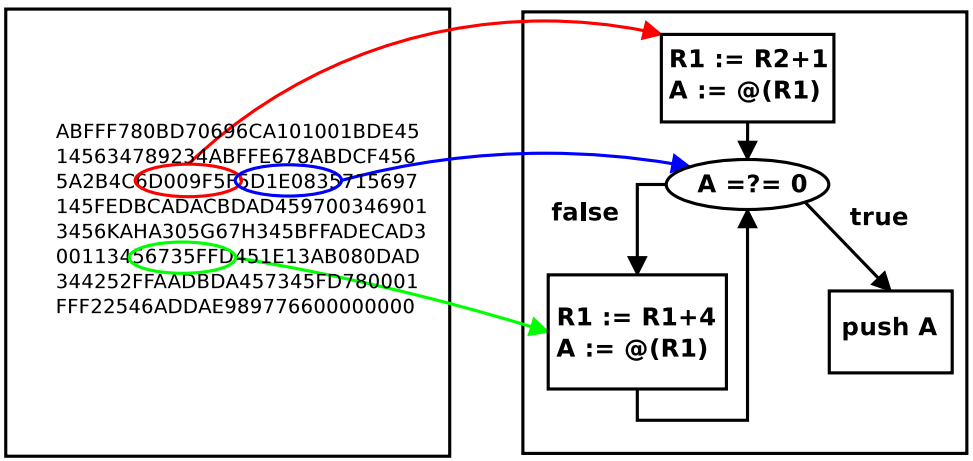


- state-of-the-art tools are not enough !

Challenge : correct disassembly

Input

- an executable code (array of bytes)
- an initial address
- a basic decoder : file $\times$ address $\mapsto$ instruction $\times$ size

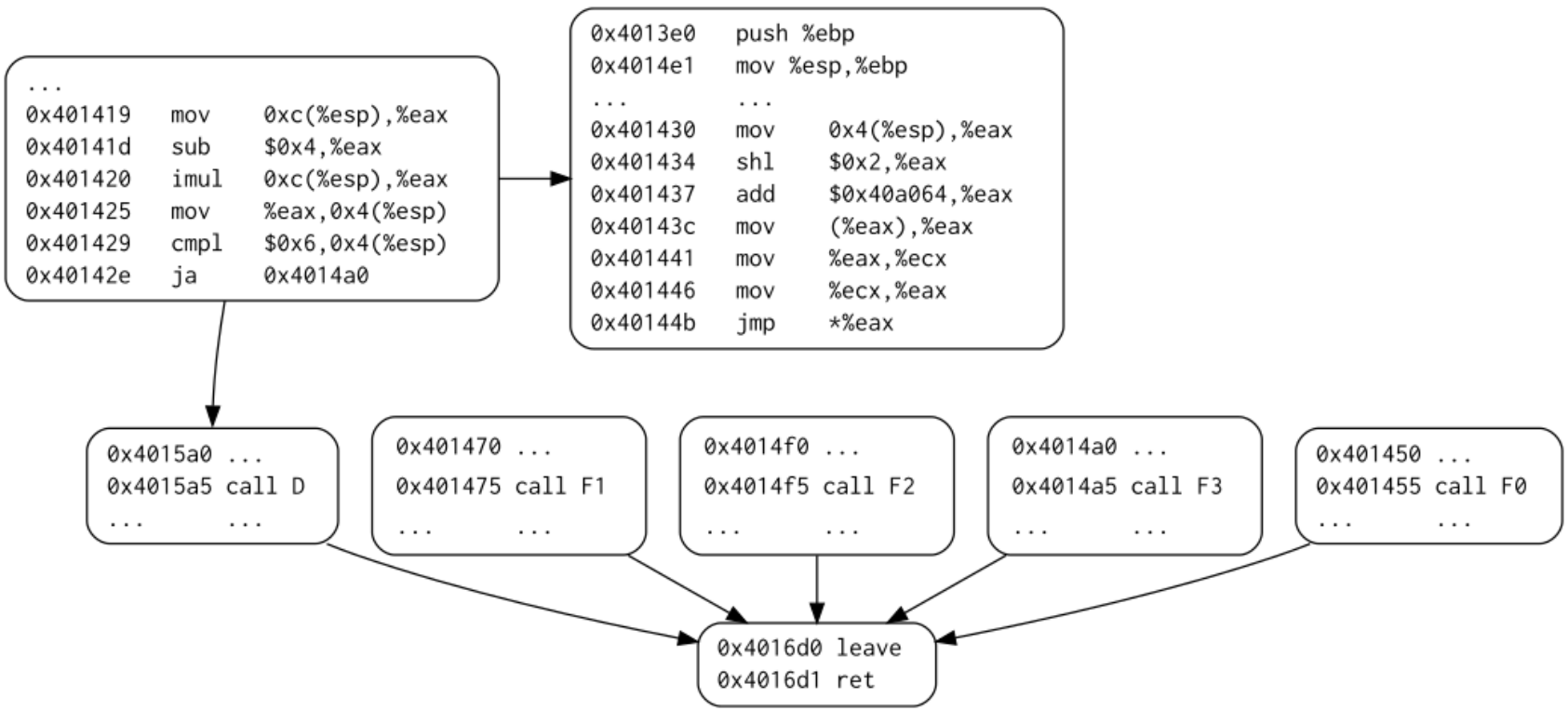


Output : (surapproximation of) the program Control-Flow Graph

- problem : successors of `jmp eax` ?

Limits of syntactic approaches

Ex : IDA is fooled by simple syntactic tricks



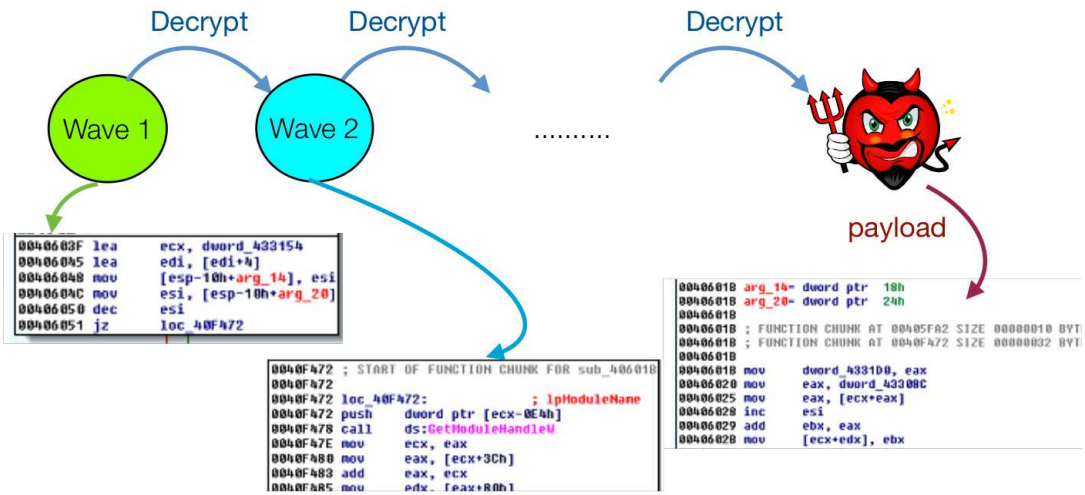
With IDA

Even worse : obfuscated code

Understand or recognize malware despite obfuscation

- self-modifying code, virtual machines
- opaque predicates, stack tampering, etc.

Context : x86-malware



A common protection scheme for malware
a SillyFDC run

Self-modifying program schema

Challenges : vulnerabilities

Use-after-free (UaF) – CWE-416

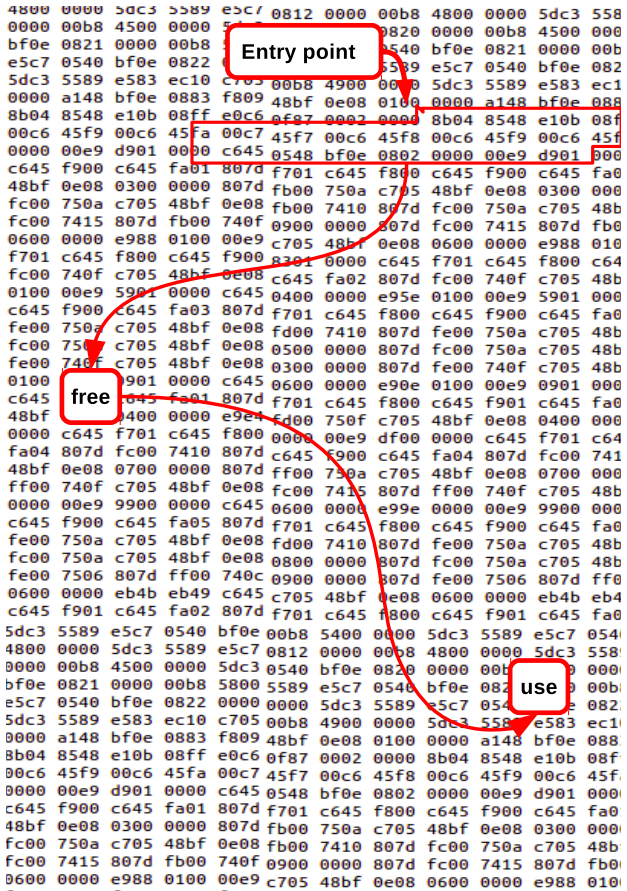
- *dangling pointer* on deallocated-then-reallocated memory
- may lead to arbitrary data/code read, write or execution
- standard vulnerability in C/C++ applications (e.g. web browsers) .
  firefox (CVE-2014-1512), chrome (CVE-2014-1713)

```
1 char *login, *passwords;
  login=(char *) malloc(...);
3 [...]
  free(login);  // login is now a dangling pointer
5 [...]
  passwords=(char *) malloc(...);  // may re-allocate memory of *login
7 [...]
  printf("%s\n", login);  // security threat : may print the passwords !
```

Limits of dynamic analysis

Find a needle in the heap !

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



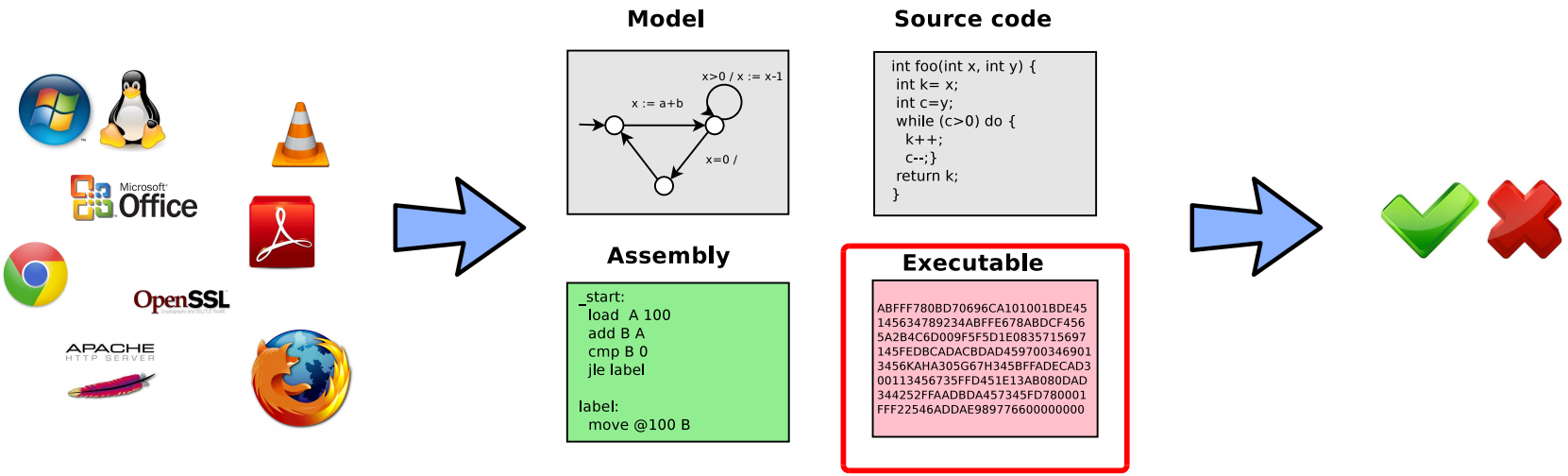
Outline

- Preambule
- Benefits of binary-level analysis
- Challenges of binary-level analysis
- Semantic approaches
- BINSEC platform
- Achievements
- Conclusion

Our proposal : binary-level semantic analysis

Semantic tools help make sense of binary

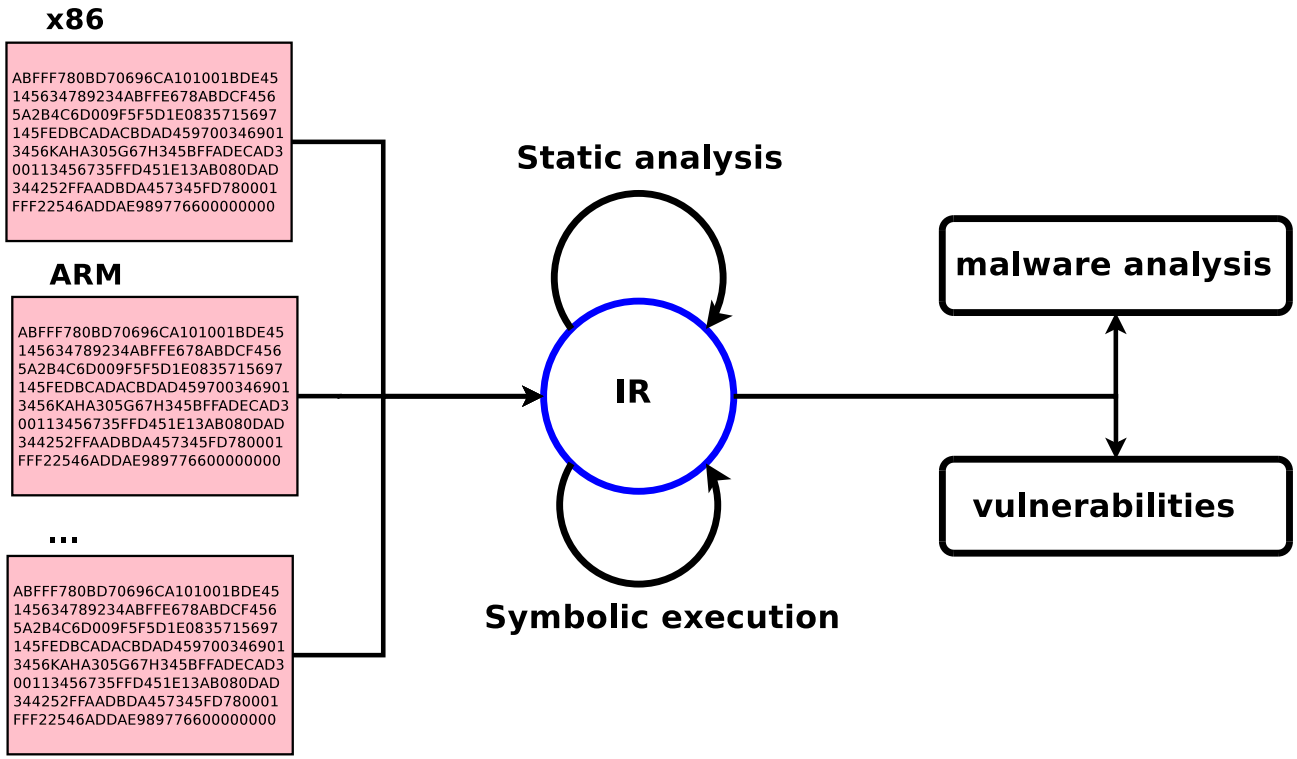
- Develop the next generation of binary-level tools !
- motto : leverage formal methods from safety critical systems



Challenges

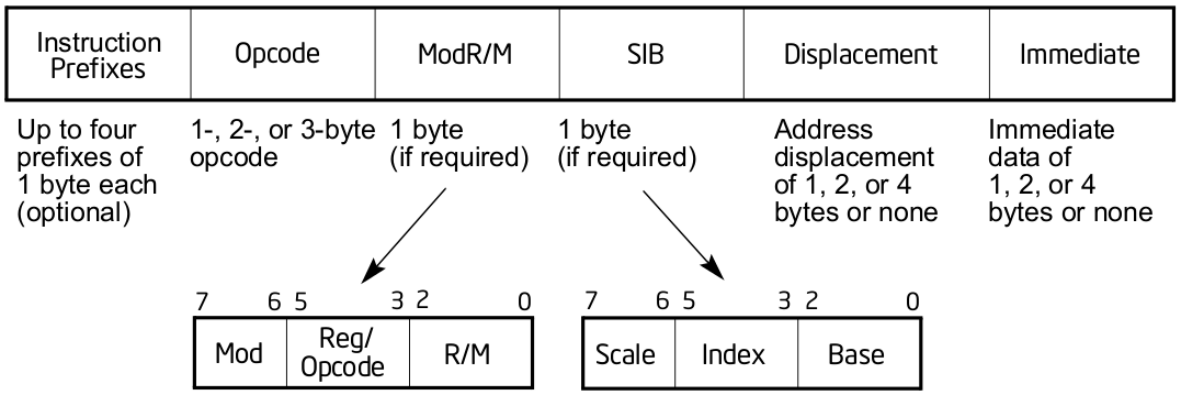
- source-level $\mapsto$ binary-level
- safety $\mapsto$ security
- many (complex) architectures

BINSEC approach



- leverage powerful methods from formal software analysis
- *pragmatic* formal methods (combination, tradeoffs, etc.)
- common basic analysis + dedicated analysis (vuln., malware)

Focus : modelling

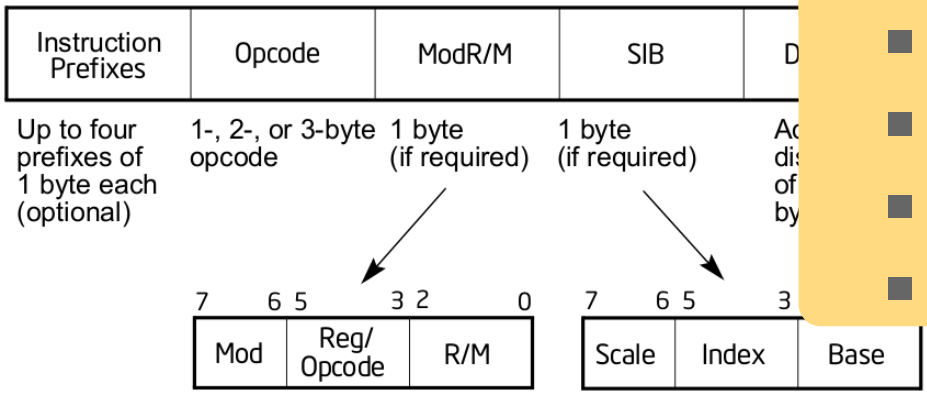


Example of x86

- 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
eee			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
DH/SI/ESI/ST6/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
BH/DI/EDI/ST7/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

Focus : modelling



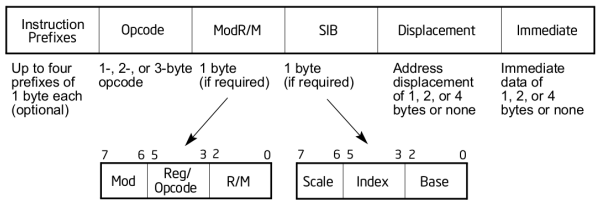
- `lhs := rhs`
- `goto addr, goto expr`
- `ite(cond)? goto addr : goto addr'`
- `assume, assert, nondet, malloc, free`

Intermediate Representation [cav11]

- architecture independent
- (really) reduced set of instructions
  - . 9 instructions, less than 30 operators
- simple, clear semantic, no side-effect

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
eee			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
DH/SI/ESI/ST6/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
BH/DI/EDI/ST7/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

x86 front-end

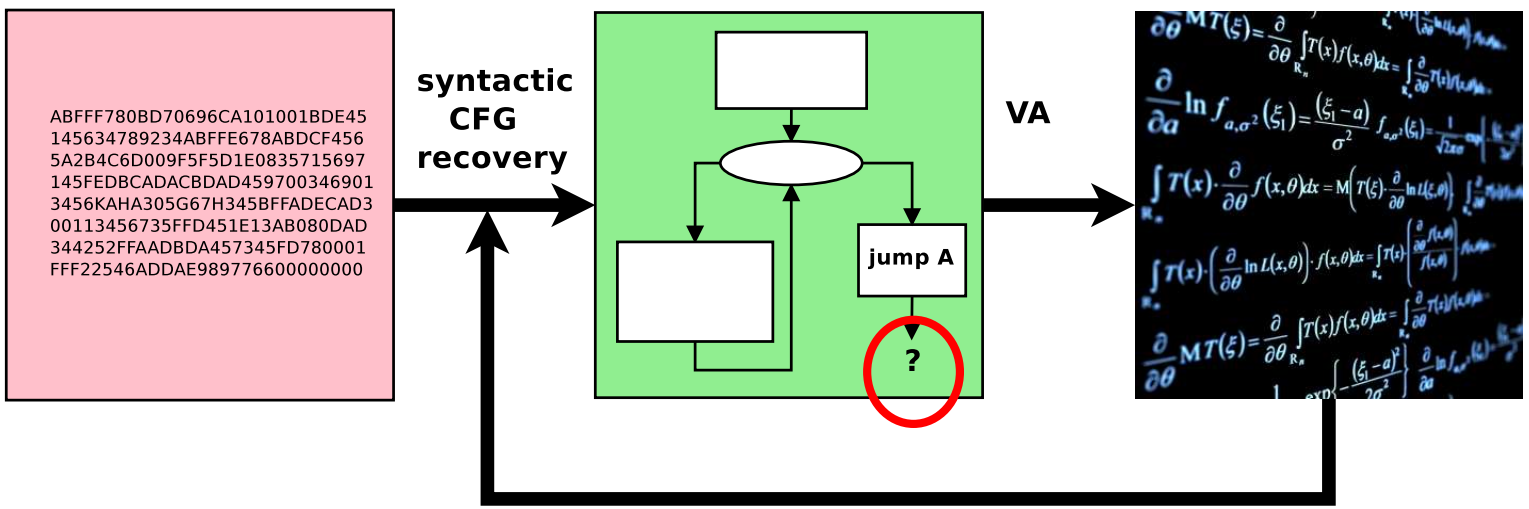


81 c3 57 1d 00 00 $\overset{x86 reference}{\Rightarrow}$ ADD EBX 1d57

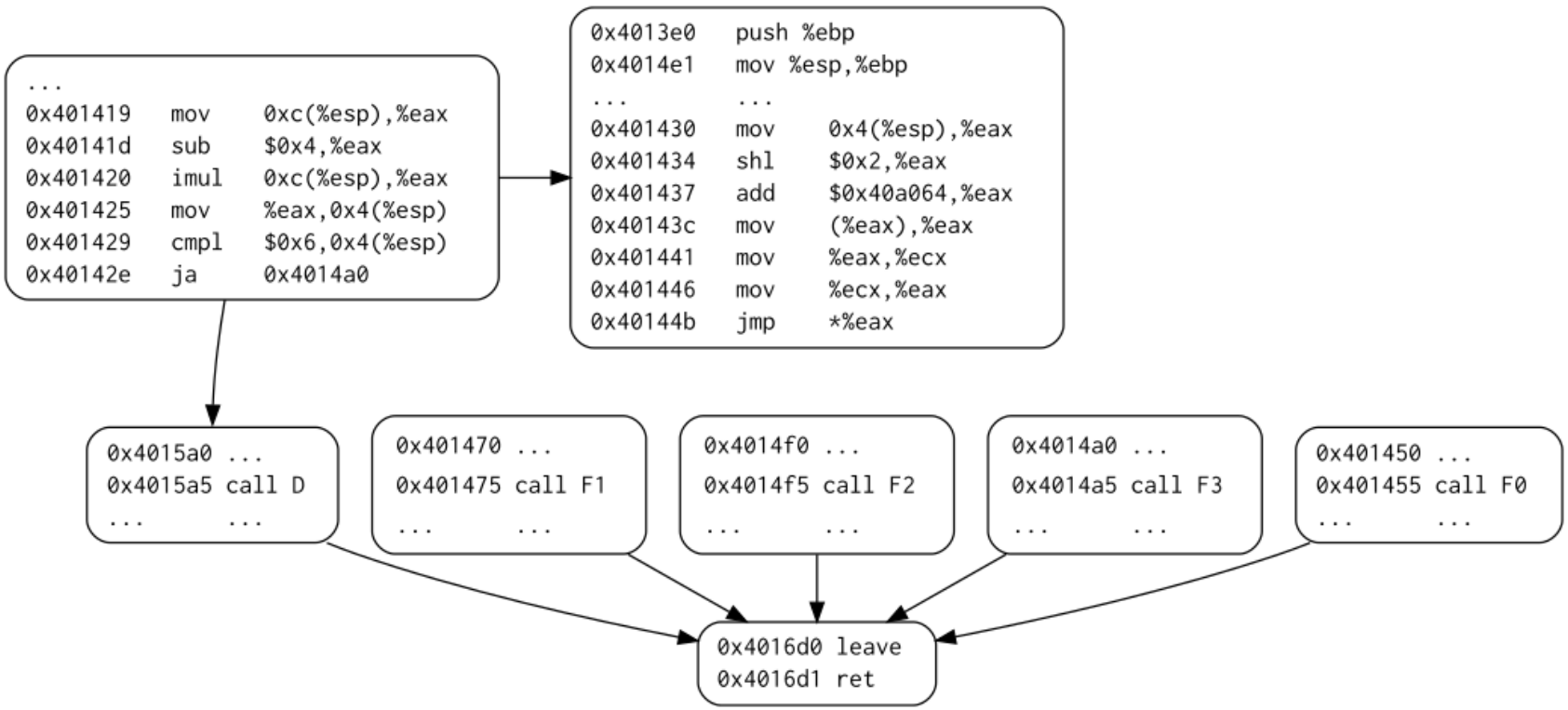
```
(0x29e,0) tmp  := EBX + 7511;
(0x29e,1) OF := (EBX{31,31}=7511{31,31}) && (EBX{31,31}<>tmp{31,31});
(0x29e,2) SF := tmp{31,31};
(0x29e,3) ZF := (tmp = 0);
(0x28e,4) AF := ((extu (EBX{0,7}) 9) + (extu 7511{0,7} 9)){8,8};
(0x29e,6) CF := ((extu EBX 33) + (extu 7511 33)){32,32};
(0x29e,7) EBX  := tmp;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

Semantic disassembly

- simple obfuscation confuses soa disassemblers such as IDA
- ... because they rely on **syntax**
- **semantic** techniques complement and strengthen these approaches

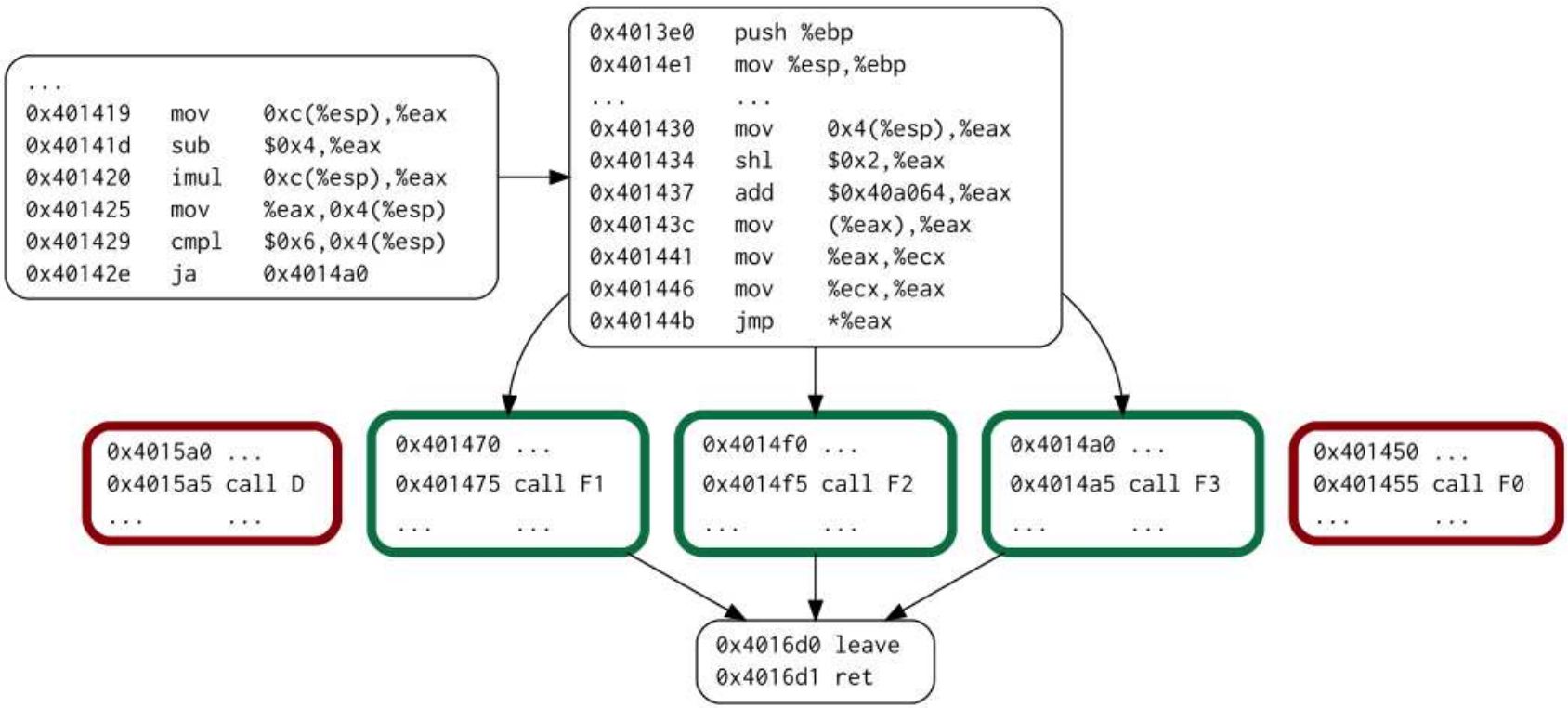


Semantic disassembly (2)



With IDA

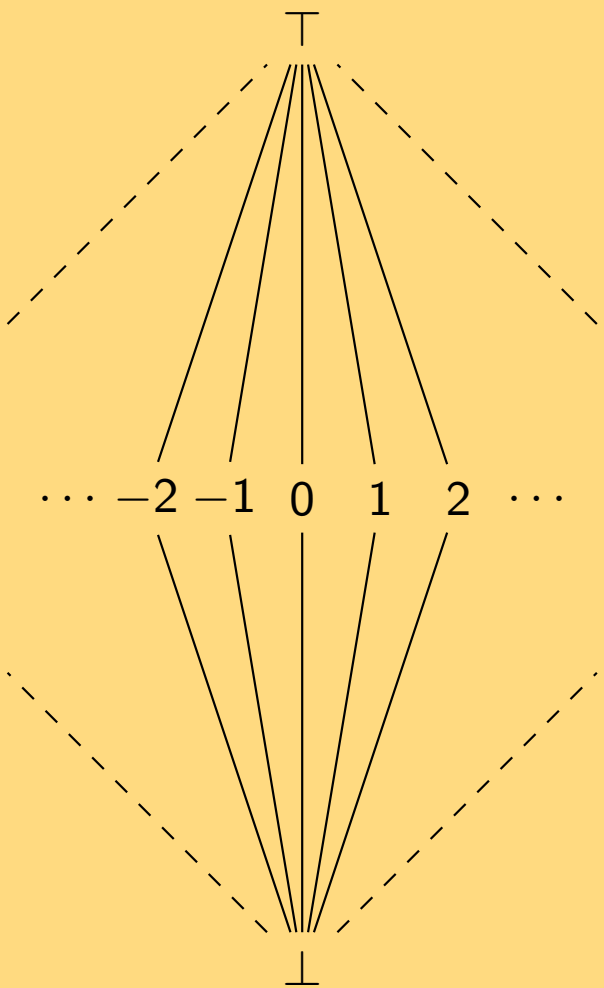
Semantic disassembly (2)



With IDA + BINSEC

Semantic disassembly : keys

Generalize constant propag



Framework : abstract interpretation

- notion of abstract domain

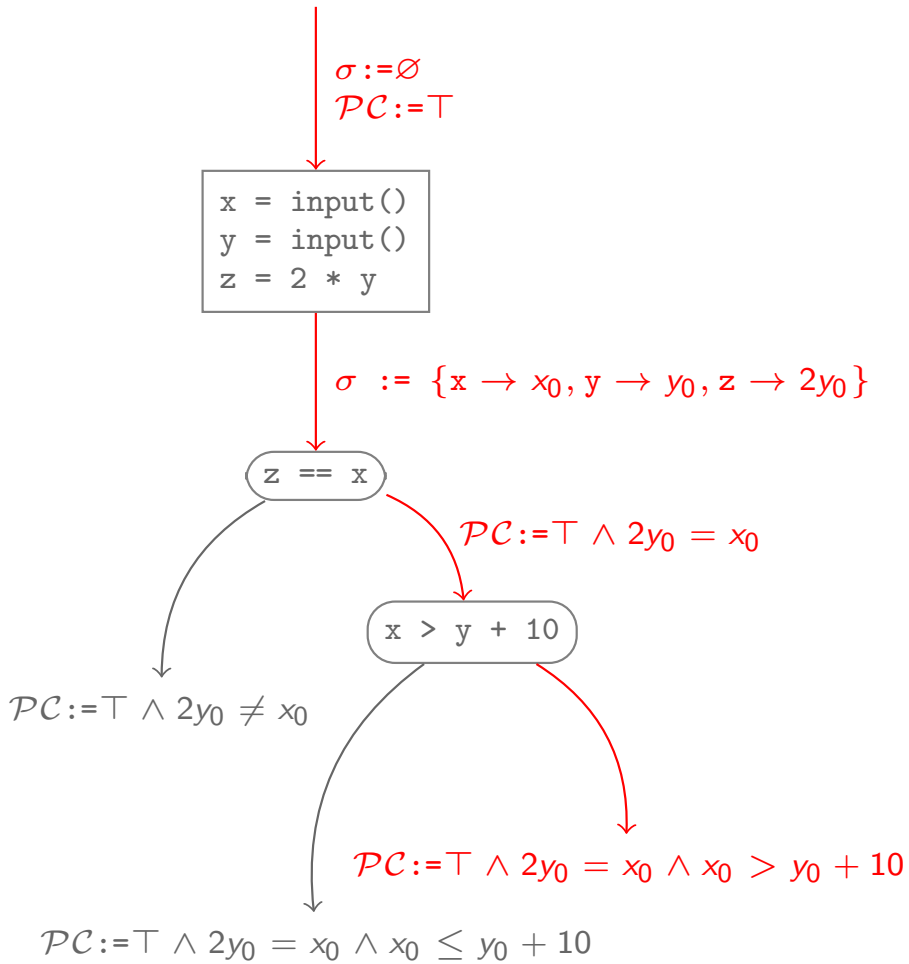
  $\bot, \top, \sqcup, \sqcap, \sqsubseteq$, $\text{eval}^{\#}$
- more or less precise domains

  . intervals, polyhedra, etc.
- fixpoint until stabilization

Semantic exploration

```
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



Path predicate computation

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) (branche True)
4	if (x < z) (branche False)

Path predicate computation

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) (branche True)
4	if (x < z) (branche False)

let $W_1 \triangleq Y_0 + 1$ in

Path predicate computation

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) (branche True)
4	if (x < z) (branche False)

let $W_1 \triangleq Y_0 + 1$ in
  let $X_2 \triangleq W_1 + 3$ in

Path predicate computation

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) (branche True)
4	if (x < z) (branche False)

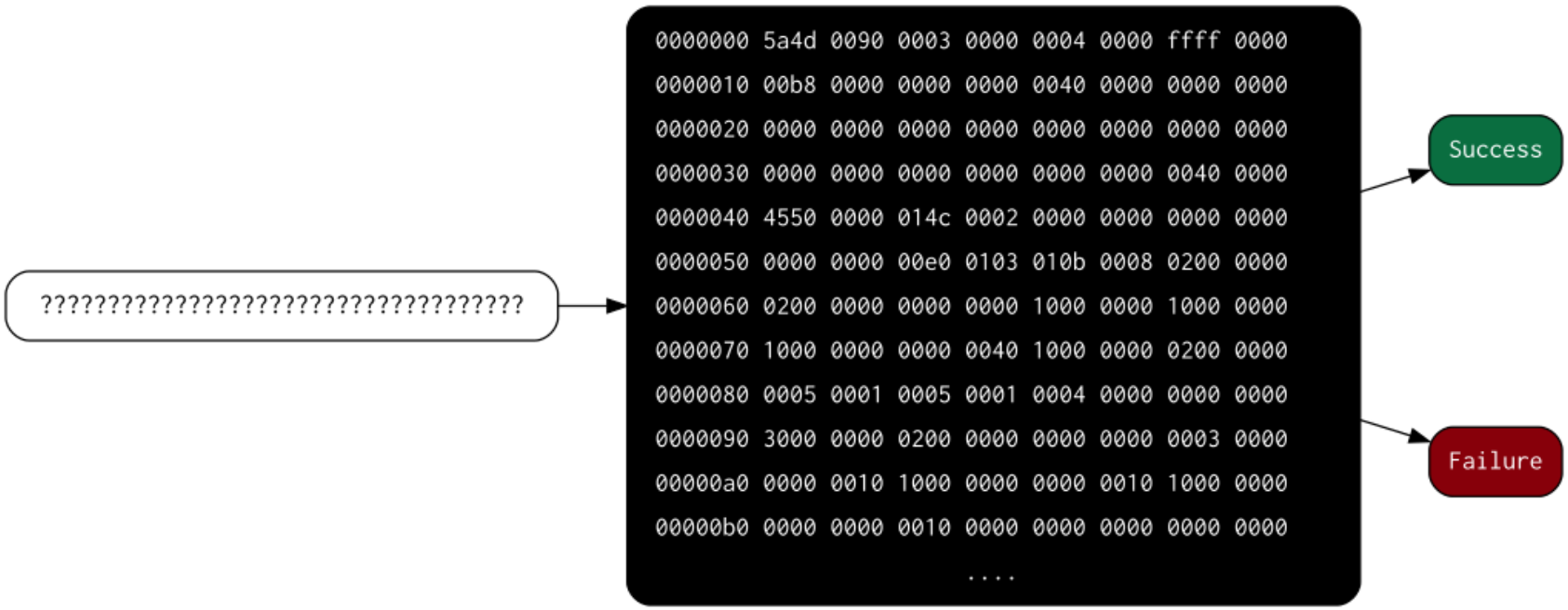
let $W_1 \triangleq Y_0 + 1$ in
  let $X_2 \triangleq W_1 + 3$ in
    $X_2 < 2 \times Z_0$

Path predicate computation

Loc	Instruction
0	input(y,z)
1	w := y+1
2	x := w + 3
3	if (x < 2 * z) (branche True)
4	if (x < z) (branche False)

let $W_1 \triangleq Y_0 + 1$ in
  let $X_2 \triangleq W_1 + 3$ in
    $X_2 < 2 \times Z_0 \wedge X_2 \geq Z_0$

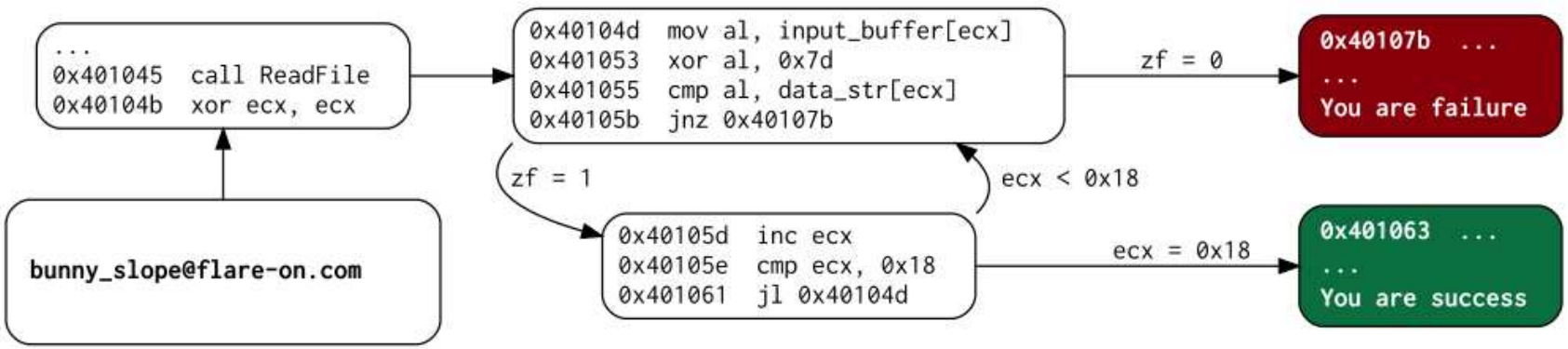
Semantic exploration (2)



Crackme challenges

- input $==$ secret $\mapsto$ success
- input $\neq$ secret $\mapsto$ failure

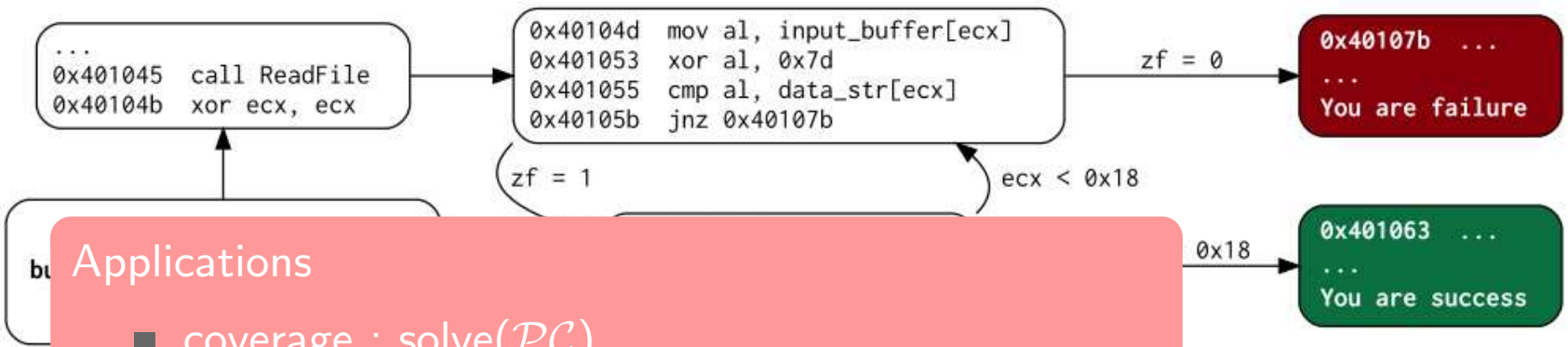
Semantic exploration (2)



With BINSEC [https ://youtu.be/0xUc2jbpjQo]

- find the path leading to success
- "invert" the conditions, find the secret : `bunny_slope@flare.com`
- check : it works !

Semantic exploration (2)



Applications

- coverage : solve($\mathcal{PC}$)
- bug finding : solve($\mathcal{PC} \wedge$ Error)
- exploit finding : solve($\mathcal{PC} \wedge$ Error $\wedge$ Hijack $\wedge$ Payload)

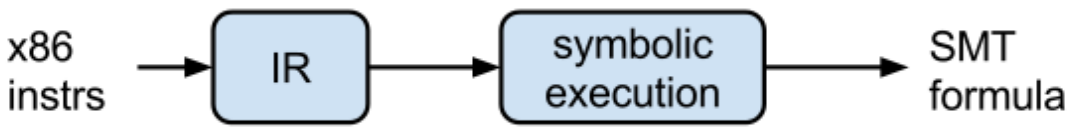
Wi

- "invert" the conditions, find the secret : `bunny_slope@flare.com`
- check : it works !

Semantic exploration : keys

Symbolic Execution

- path predicate computation
- formula preprocessing + SMT solver
- sound execution of the program [path necessarily feasible]



Dynamic Symbolic Execution [DSE]

- combine dynamic and symbolic reasoning
- much more robust [missing code, self-modification, etc.]

Outline

- Preambule
- Benefits of binary-level analysis
- Challenges of binary-level analysis
- Semantic approaches
- BINSEC platform
- Achievements
- Conclusion

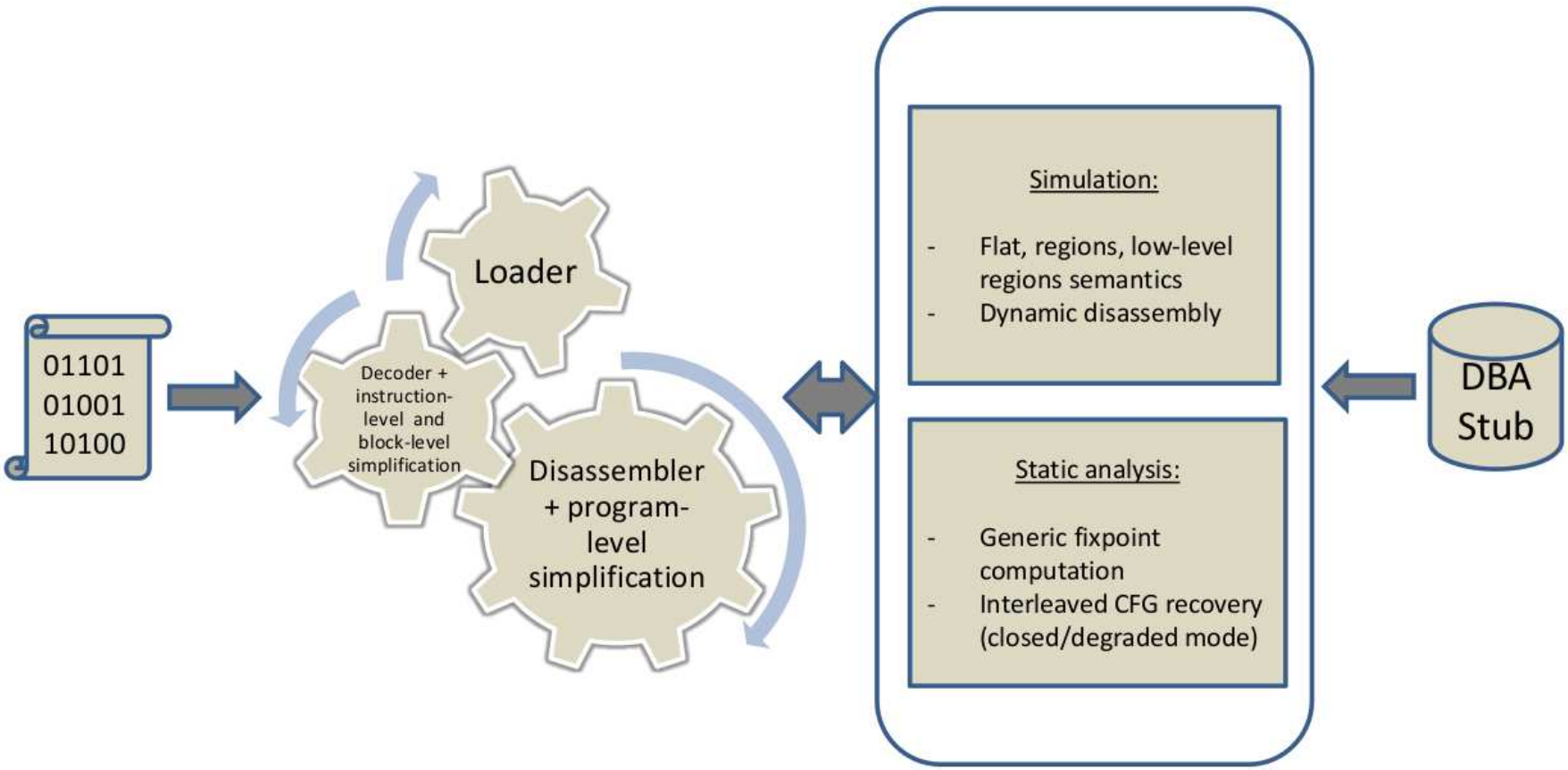
BINSEC platform

The BINSEC Platform [CEA & Uni. Grenoble Alpes]

- open source, lgpl v2.1
- mostly OCaml, 30 kloc (and pintool in C++)
- dual goal
  - help design new binary-level analyzers (basic building blocks)
  - provide innovative analyzers
- allows for combination of techniques
- current : multi-architecture support, semantic exploration & semantic disassembly, poc on vulnerabilities and deobfuscation
- still young : beta-version just released [http ://binsec.gforge.inria.fr/]

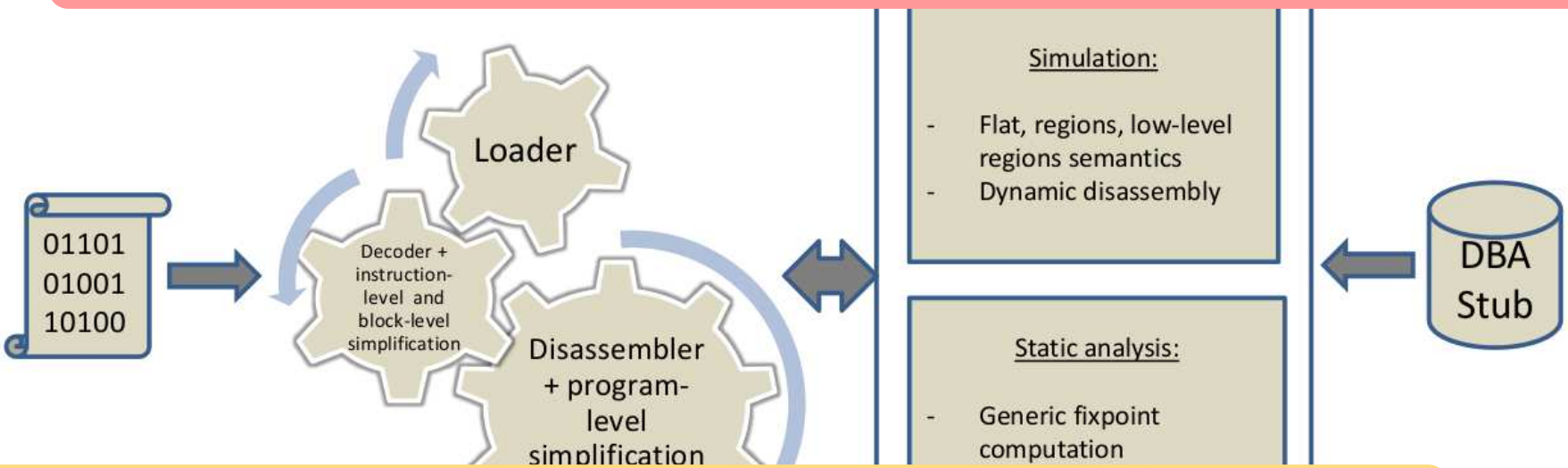
Thx to a bunch of enthusiastic students : Robin David, Adel Djoudi, Josselin Feist, Than Dihn Ta, Benjamin Farinier

BINSEC platform (2)



BINSEC platform (2)

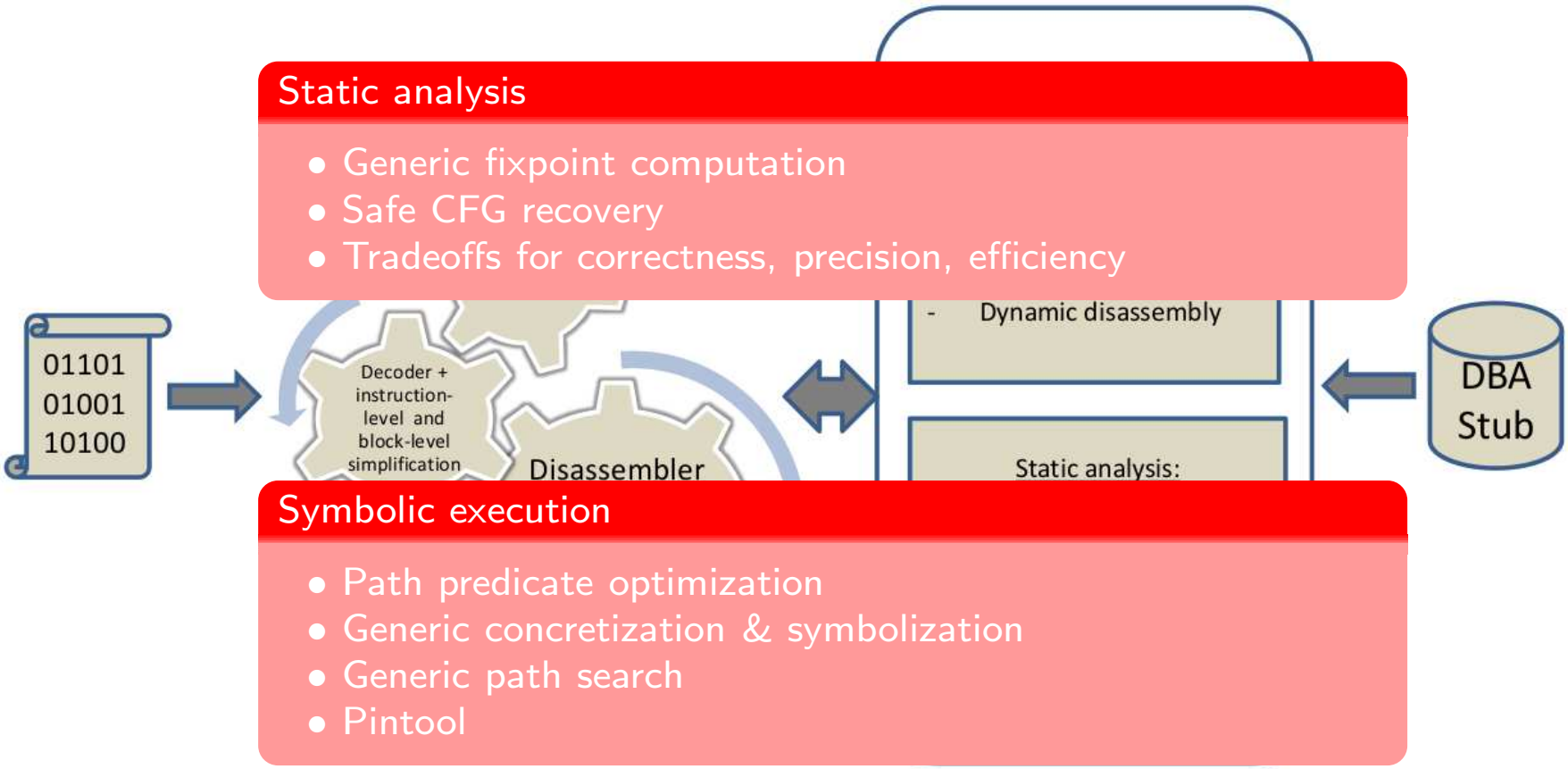
- loader ELF/PE
- decoder (x86) + IR simplification
- 460/500 instructions : 380/380 "basic", 80/120 SIMD, no float/system
- prefixes : op size, addr size, repetition
- standard syntactic disassembly techniques : recursive, linear, combination



Basic services to build analysis on :

- Simulation
- Static analysis [semantic disassembly] [Adel Djoudi – tacas15, sub. fm16]
- Symbolic execution [semantic exploration] [Robin David – saner16,issta16]
- Combinations

BINSEC platform (2)



DBA simplifications

- Instruction level simplifications
  - Idiom simplifications [local rewriting rules]
- Block level simplifications
  - Constants propagation
  - Remove redundant assigns
- Program level simplifications
  - Flag slicing (remove must-killed variables)
  - granularity : function level+automatic summary of callees

DBA simplifications

- Instruction level simplifications
  - Idiom simplifications [local rewriting rules]
- Block level simplifications
  - Constants propagation
  - Remove redundant assigns
- Program level simplifications
  - Flag slicing (remove must-killed variables)
  - granularity : function level+automatic summary of callees

Approach

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

DBA simplifications : Experiments

program	native	DBA	opt (DBA)		
	loc	loc	time	loc	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%	90.00%	67.04%

BINSEC platform (3)

What can be reused ?

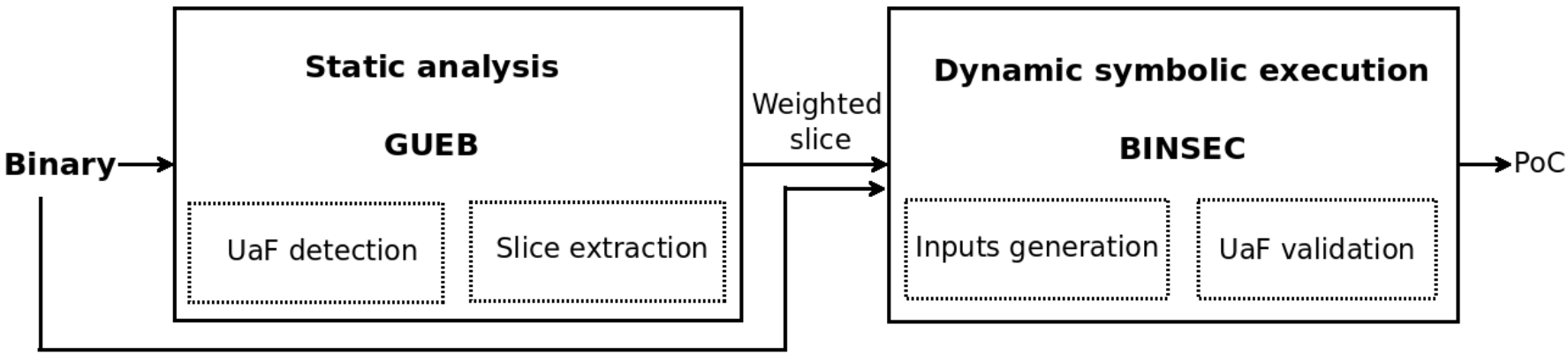
- whole analyses
  - semantic exploration
  - semantic disassembly
- basic blocks [need cleaner APIs]
  - decoding
  - disassembly (cfg, call graph)
  - abstract fixpoint computation
  - path predicate, formula simplification & solving
  - generic path exploration
  - pintool

Outline

Finding *use-after-free* vulnerabilities

A *pragmatic two-step* approach implemented within the BINSEC plateform :

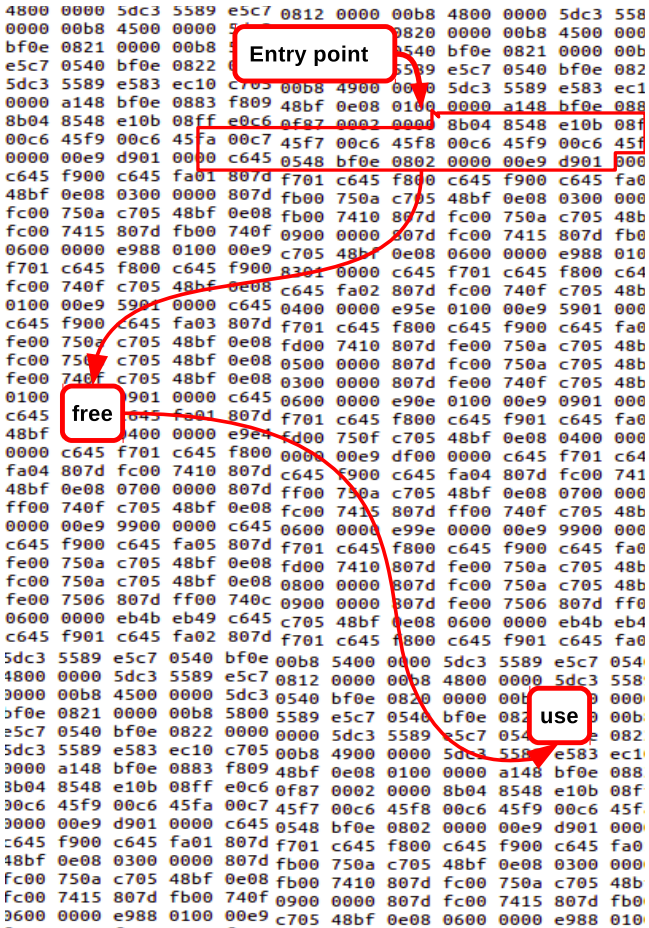
- not complete, but scalable and correct in some cases



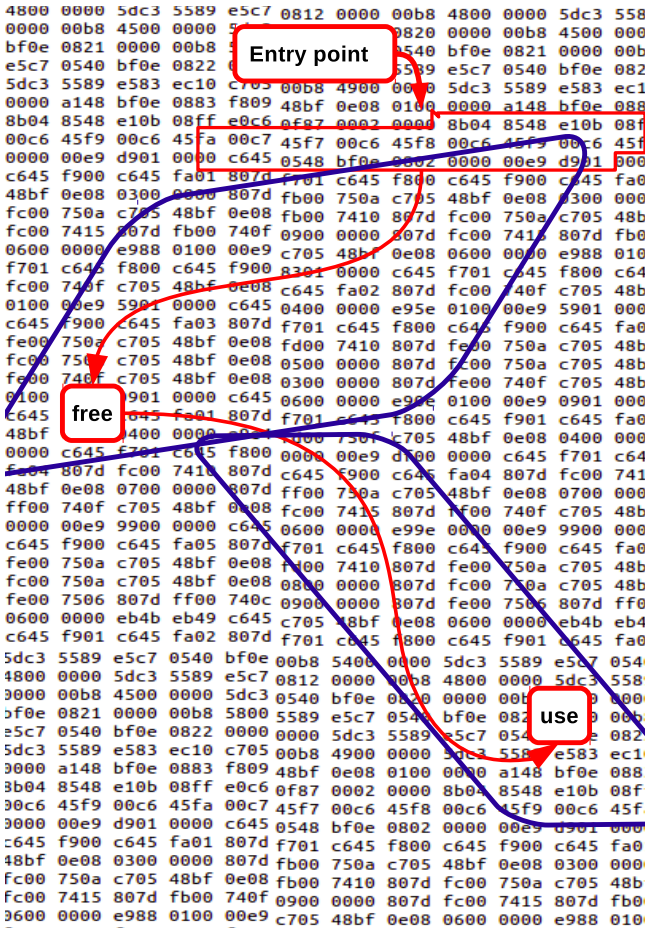
- GUEB : *scalable* lightweight static analysis (not sound, not complete)
  → produces a set of CFGs slices containing **potential** UaF

- BINSEC/SE : guided symbolic execution
  → *confirm* the UaF by finding **concrete** program inputs

Help to find the needle in the heap



Help to find the needle in the heap



Results

Combination of techniques is fruitful !

Several new vulnerabilities found

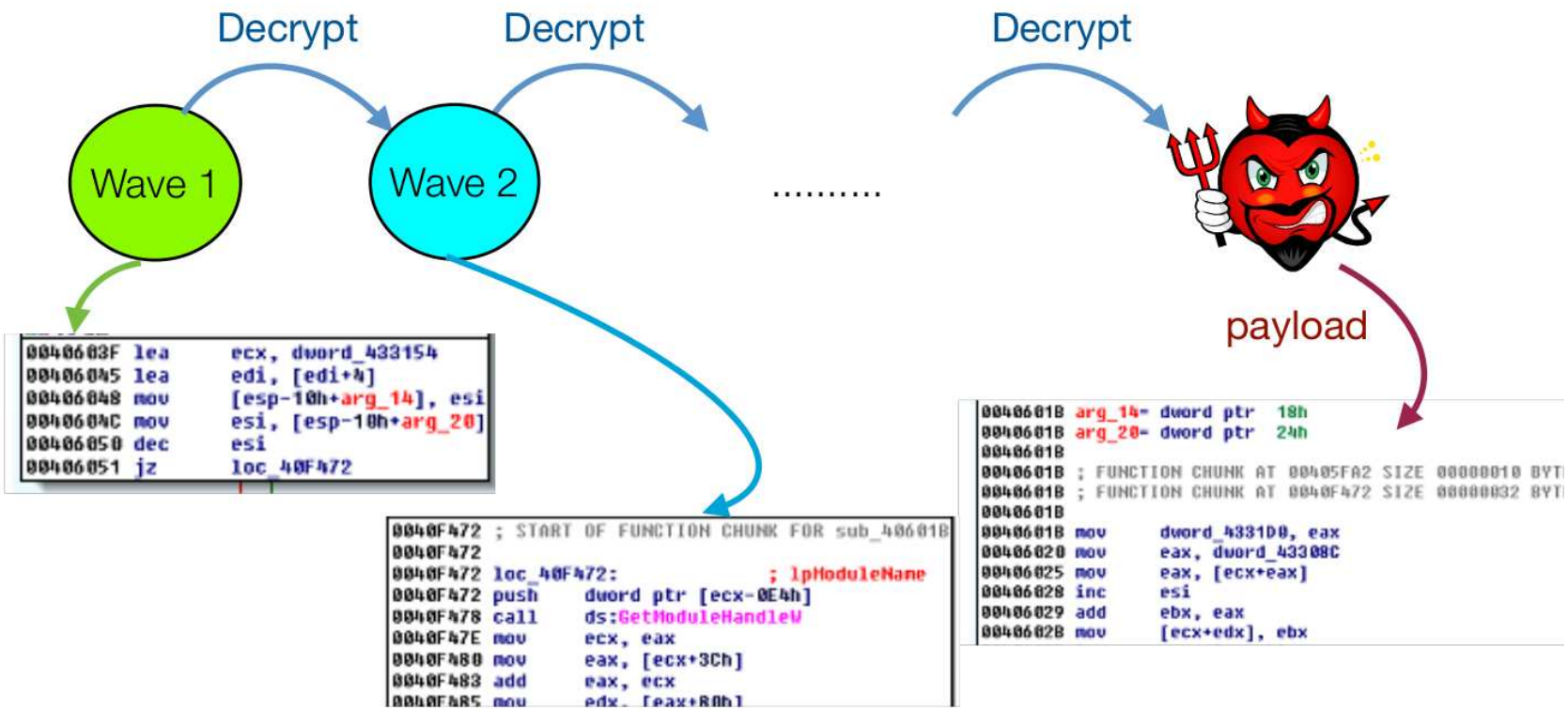
- GUEB + manual analysis [j. comp. virology 14]
  - `tiff2pdf` : CVE-2013-4232
  - `openjpeg` : CVE-2015-8871
  - `gifcolor` : CVE-2016-3177
  - `accel-ppp`
- GUEB + BINSE/SE [sefm16]
  - `Jasper JPEG-2000` : CVE-2015-5221

Gueb [Josselin Feist]

- MIT licence
- Ocaml, 5kloc
- https ://github.com/montyly/gueb

Malware deobfuscation

Context : x86-malware

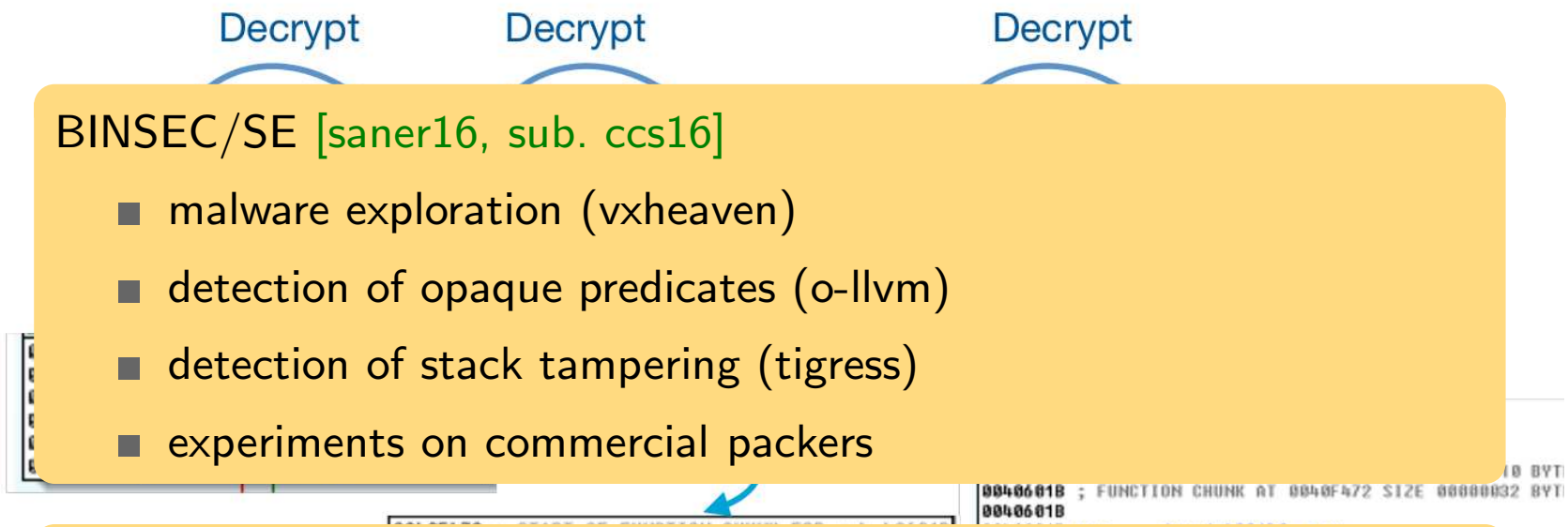


A common protection scheme for malware
a SillyFDC run

Self-modifying program schema

Malware deobfuscation

Context : x86-malware



BINSEC/SE [saner16, sub. ccs16]

- malware exploration (vxheaven)
- detection of opaque predicates (o-llvm)
- detection of stack tampering (tigress)
- experiments on commercial packers

- static analysis : not safe, complete, not robust to obfuscation
- dynamic analysis : safe, not complete, robust to obfuscation
- symbolic execution : best of both world
- + fruitful combination dynamic, static, symbolic

Self-modifying program schema

Outline

- Preambule
- Benefits of binary-level analysis
- Challenges of binary-level analysis
- Semantic approaches
- BINSEC platform
- Achievements
- At last

Conclusion

Binary-level security analysis

- many applications, many challenges
- syntactic and dynamic are not enough

Semantic approaches can help !

- semantic exploration, semantic disassembly
- yet, still hard to design

The BINSEC Platform [CEA & Uni. Grenoble Alpes]

- open source, dual goal
  - help design new binary-level analyzers (basic building blocks)
  - provide innovative analyzers [already a few ones]
- current : multi-architecture support, semantic exploration & semantic disassembly, poc on vulnerability detection and deobfuscation
- still young : beta-version just released [http ://binsec.gforge.inria.fr/]

Conclusion

Binary-level security analysis

- many applications, many challenges
- syntactic and dynamic are not enough

In progress

- tutorials, doc
- code cleaning
- ARM and PowerPC

ntic disassembly

The BINSEC Platform [CEA & Uni. Grenoble Alpes]

- open source, dual goal
  - help design new binary-level analyzers (basic building blocks)
  - provide innovative analyzers [already a few ones]
- current : multi-architecture support, semantic exploration & semantic disassembly, poc on vulnerability detection and deobfuscation
- still young : beta-version just released [http ://binsec.gforge.inria.fr/]

Conclusion

Binary-level security analysis

- many applications, many challenges
- syntactic and dynamic are not enough

In progress

- tutorials, doc
- code cleaning
- ARM and PowerPC

ntic disassembly

The BINSEC Platform [CEA & Uni. Grenoble Alpes]

- open source, dual g
  - help design n
  - provide innov
- current : multi-arch
  disassembly, poc or
- still young : beta-version just released [http ://binsec.gforge.inria.fr/]

Formal methods for software analysis

- lots of effort in proprietary industry
- open source community needs to keep up the pace

s)

ntic