BINSEC

BINary-level SECurity analaysis

Sébastien Bardin (coordinator)
Airbus group, CEA, IRISA, LORIA, Uni. Grenoble Alpes

Panorama de BINSEC

Cadre de travail

- Projet ANR INS, appel 2012
- Axes 1 (sécurité) et 2 (génie logiciel)
- Projet de recherche fondamentale sur 4 ans (2013-2017)

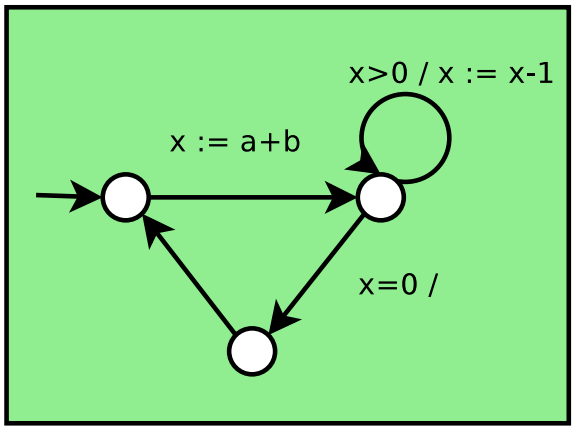
Sujet : Techniques formelles d'analyse de sécurité au niveau binaire

Partenaires : CEA (coordinateur), Airbus Group, INRIA Bretagne Atlantique, Université Grenoble Alpes - VERIMAG, Université de Lorraine - LORIA,

Participants : 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

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
```

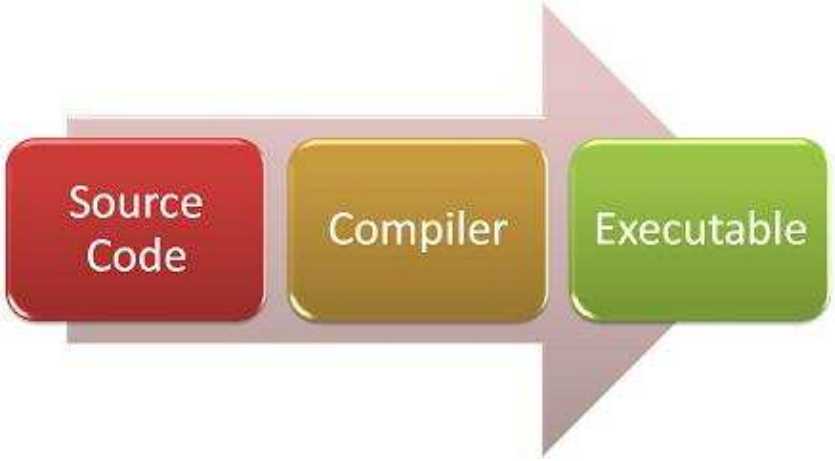
Executable

```
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)

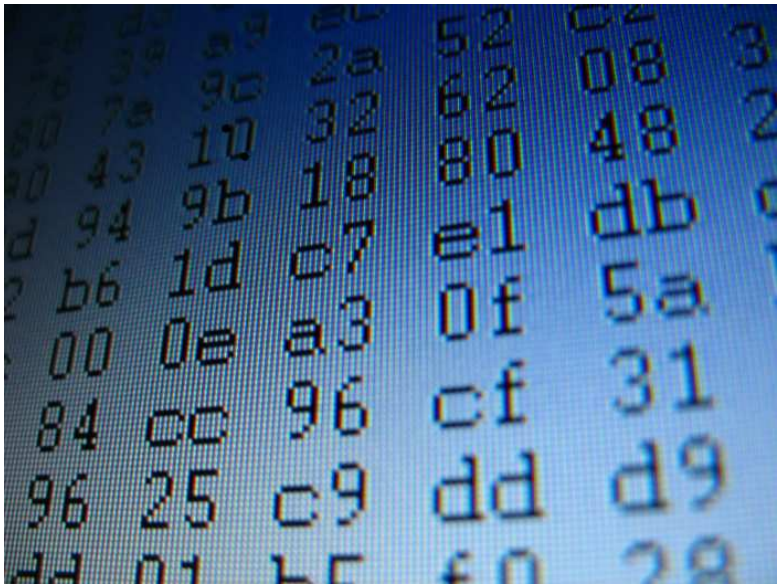
A perfectly legitimate compiler-introduced bug

```
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 ?

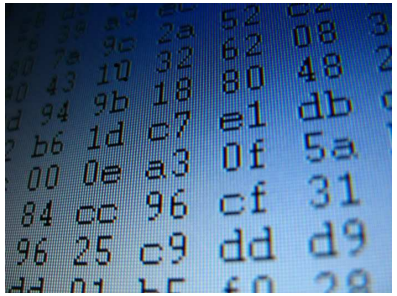
Binary-level security analysis

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]
- static analysis : objdump, IDA Pro [easy to get fooled]



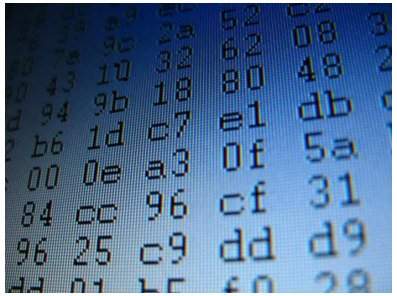
Binary-level security analysis

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]
- static analysis : objdump, IDA Pro [easy to get fooled]



- goal : develop advanced tools for experts
- motto : leverage formal methods from safety critical systems

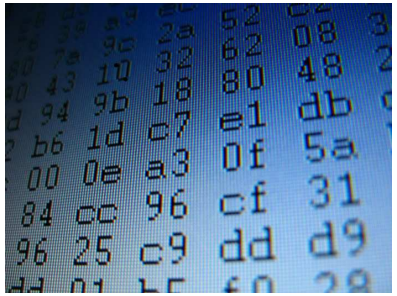
Not so easy!

Source-level ↦ Binary-level

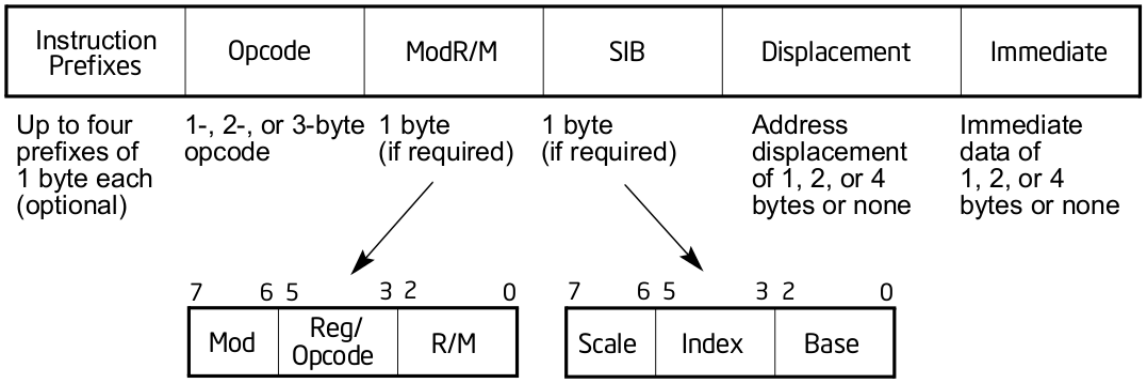
- semantics ?
- how to recover high-level structure ?

Safety ↦ Security

- vulnerabilities : from warnings to exploitable bugs
- malware : from managed code to nasty code



Challenge : modelling



Example of x86

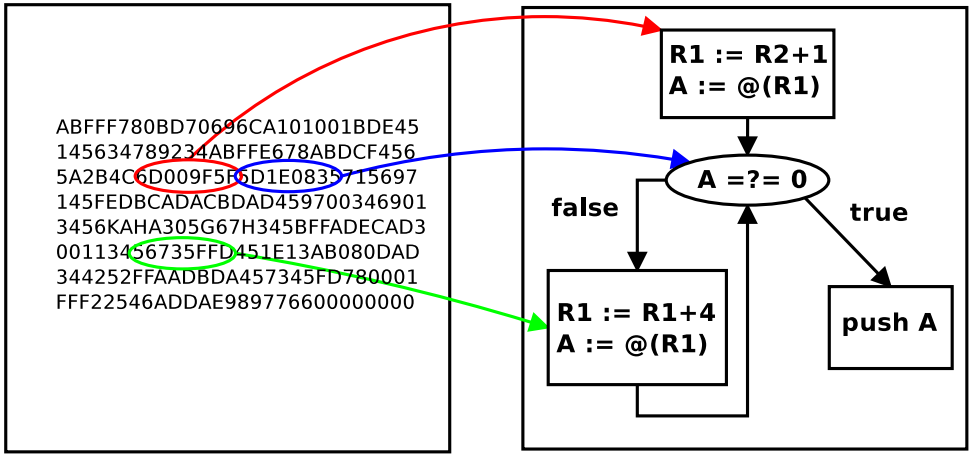
- more than 1,000 instructions
  - . ≈ 400 basic
  - . + float, interrupts, mmx
- many side-effects
- error-prone decoding
  - . addressing modes, prefixes, ...

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

Challenge : safe CFG recovery

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 CFG

[basic artifact for verif]

- problem : successors of $\langle$`addr: goto a`$\rangle$?

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)
- challenges :
  - sequence of events, importance of aliasing
  - strongly depend on the implementation of `malloc` and `free`
  - scale, correctness (demonstrate bug), assess exploitability

```
1 char *login, *passwords;
  login=(char *) malloc(...);
3 [...]
  free(login);  // login is now a dangling pointer
5 [...]
  passwords=(char *) malloc(...);  // ansi-C : fresh memory area
7 [...]
  printf("%s\n", login);   // ansi-C : error, yet no particular security threat
```

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)
- challenges
  - sequence of events, importance of aliasing
  - strongly depend on the implementation of `malloc` and `free`
  - scale, correctness (demonstrate bug), assess exploitability

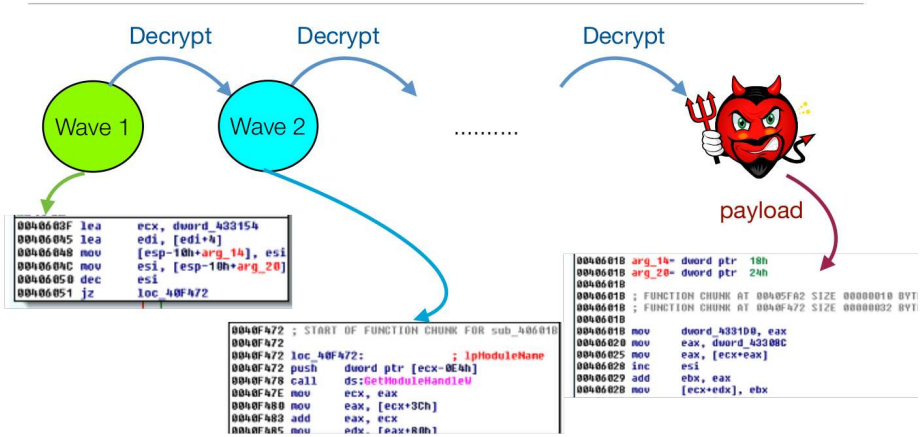
```
char *login, *passwords;
login=(char *) malloc(...);
[...]
free(login);  // login is now a dangling pointer
[...]
passwords=(char *) malloc(...);  // may re-allocate memory of *login
[...]
printf("%s\n", login);  // security threat : may print the passwords !
```

Challenge : malware

From managed code to nasty code

- recognize malware despite polymorphism
  - requires more robust notions of signatures
- recognize malware despite obfuscation
  - self-modifying code, opaque predicates, stack tampering, etc.

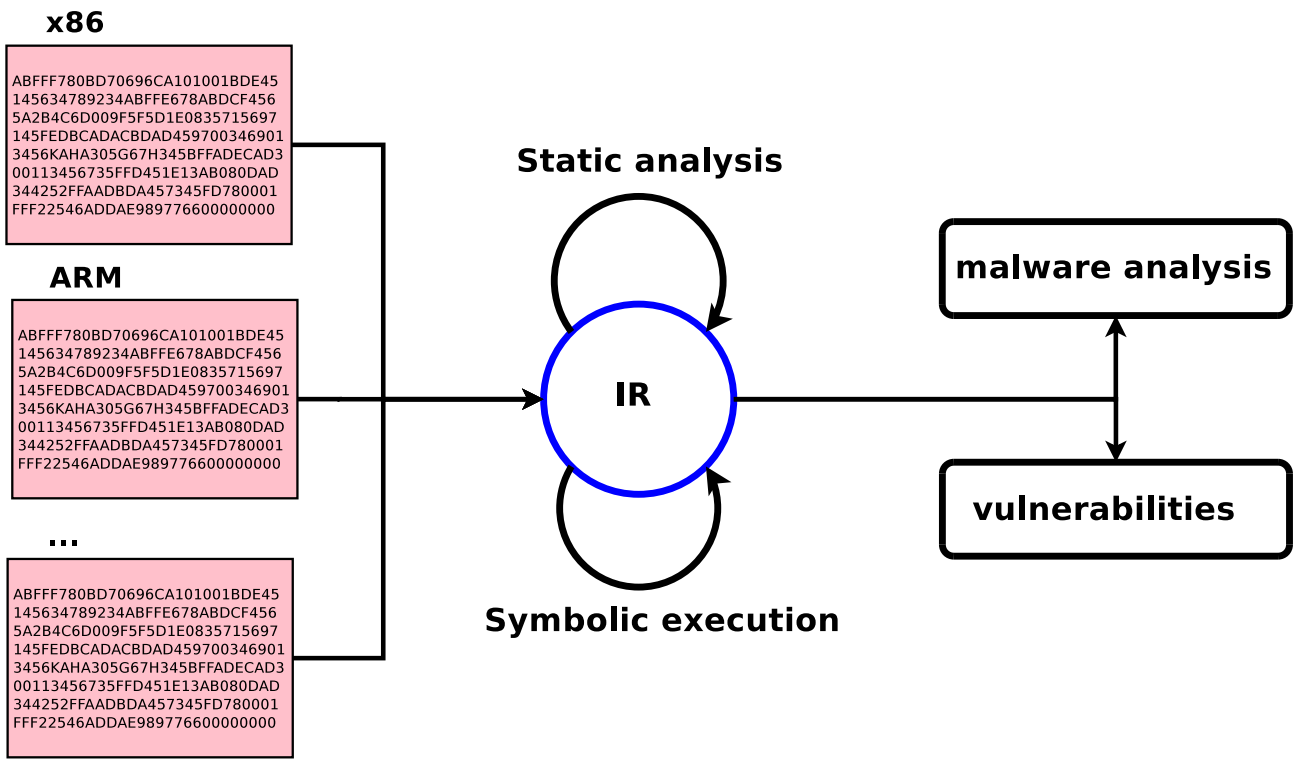
Context : x86-malware



A common protection scheme for malware
a SillyFDC run

Self-modifying program schema

BINSEC approach

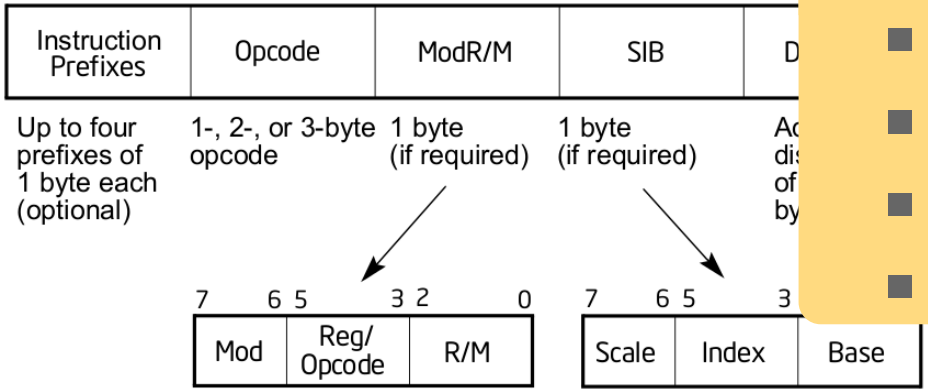


- leverage powerful methods from formal software analysis
- *pragmatic* formal methods (combination, tradeoffs, lose guarantees, etc.)
- common basic problems (models, analysis) + dedicated methods (vuln., malware)

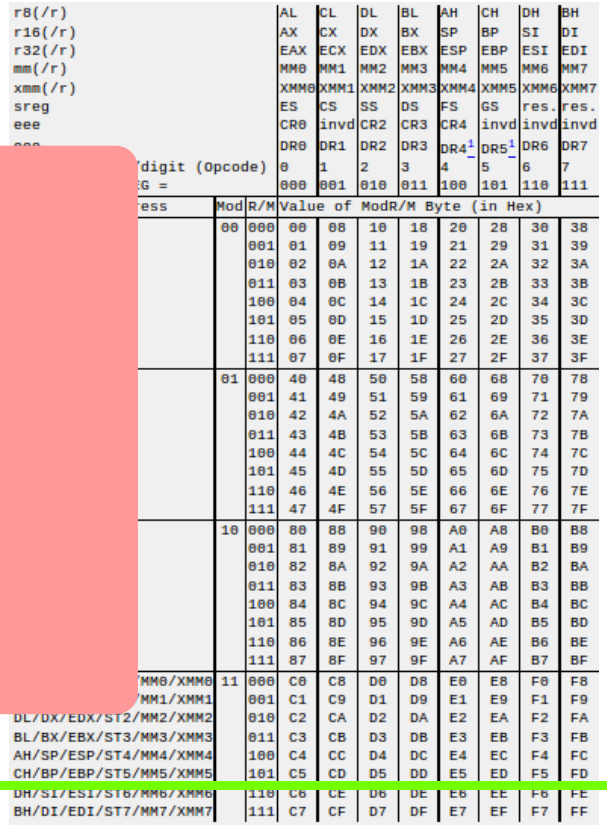
Work and results

- Models and generic analysis [leader : irisa]
  - concise intermediate representation
  - low-level memory model for low-level programs [aplas14,itp15]
  - high-level structure recovery (cfg, conditions) [subm. fm16]
  - semantic exploration via symbolic execution [issta16]
- Dedicated security analysis [leaders : verimag, loria]
  - vulnerability : detection of use-after-free [j. virology 14,sub. woot16]
  - malware : semantic signatures [malware13-15,fps14]
  - deobfuscation : combined symbolic methods [ccs15, sub. ccs16]
- Open-source platform [leader : cea] [tacas15,saner16]
  - IR, frontend (loader, decoder, syntactic disassembly)
  - static analysis, symbolic execution
- Industrial evaluation [leader : airbus group]
  - industrial case-studies (iprint client, proftpd) and challenges (coreutils, vxheavens, samate, etc.)
  - Airbus Group : internal evaluation of GUEB
  - LORIA : startup around Gorille

Focus : modelling



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

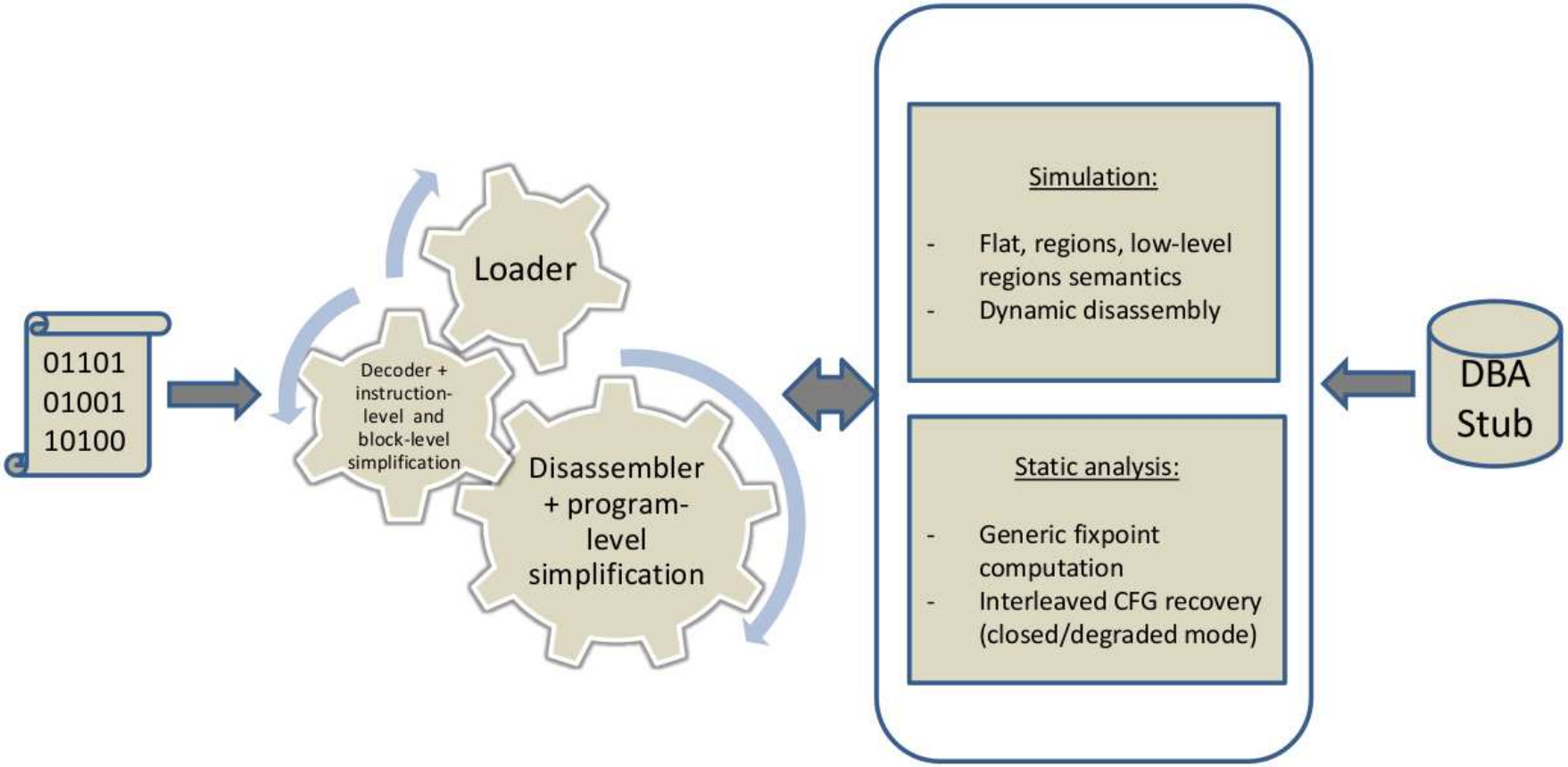


r8(/r)			AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)			AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
sreg			ES	CS	SS	DS	FS	GS	res.	res.
eee			CR0	invd	CR2	CR3	CR4	invd	invd	invd
			DR0	DR1	DR2	DR3	DR4[1]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

Intermediate Representation taken from [cav11]

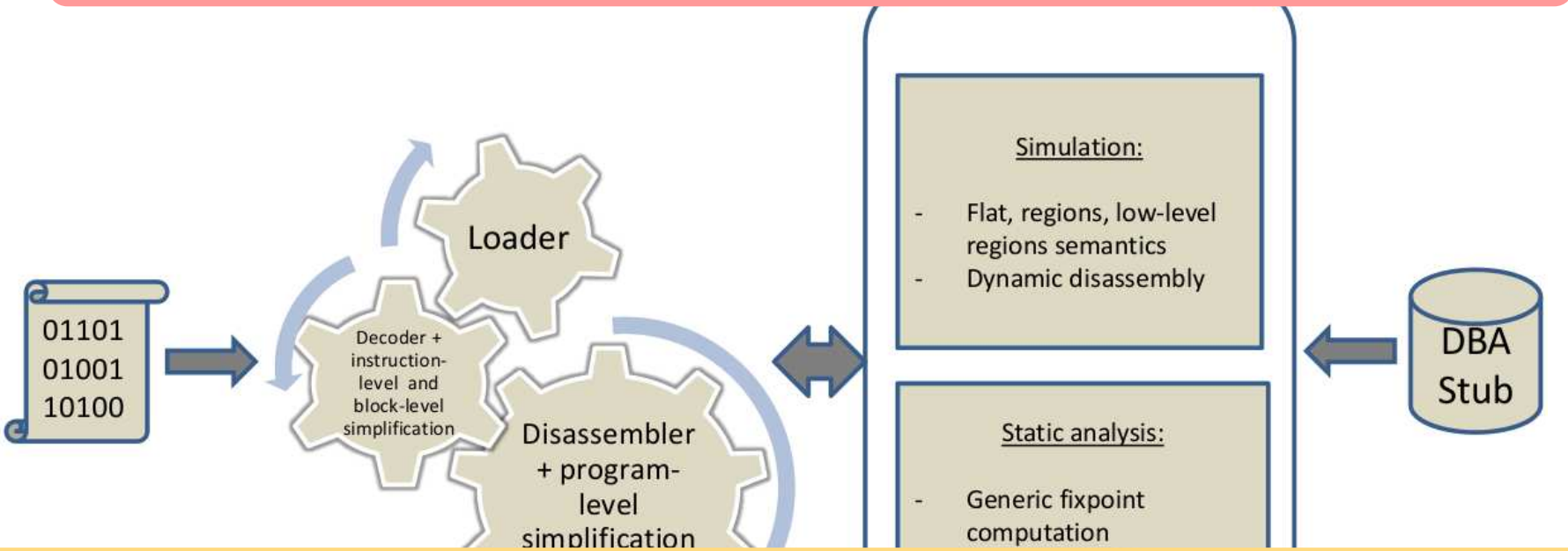
- architecture independent
- really reduced set of instructions
  . 9 instructions, less than 30 operators
- simple
  clear semantic, no side-effect
- yet : no threads, no self-modif, no float

BINSEC platform



BINSEC platform

- loader ELF/PE, decodr x86
- 460/500 instructions : 380/380 "basic", 80/120 SIMD, no float/system
- prefixes : op size, addr size, repetition



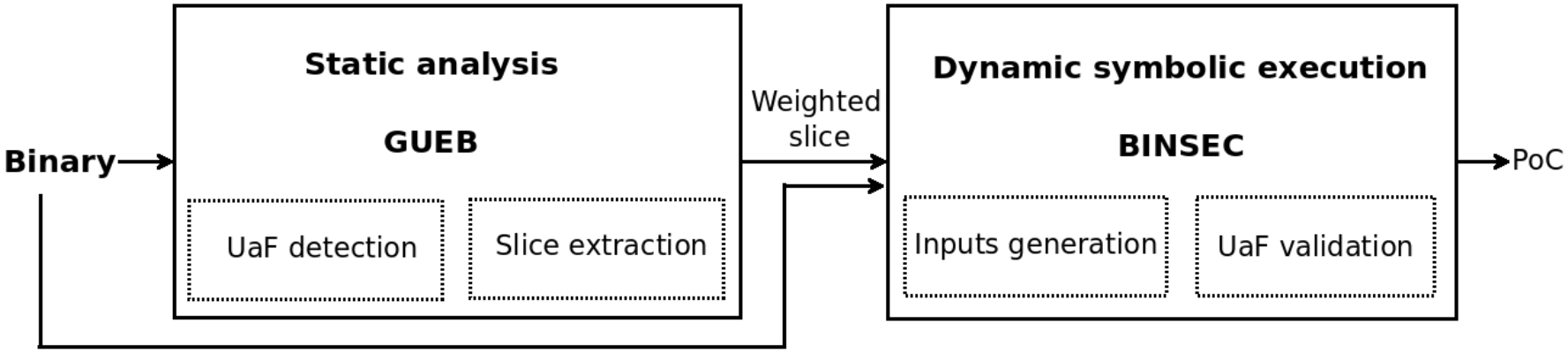
Basic services to build analysis on :

- Simulation
- Static analysis [safe CFG recovery] [tacas15, sub. fm16]
- Symbolic execution [flexible, optimized] [saner16,issta16,sub. fm16]

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

Results

Several **new** vulnerabilities found

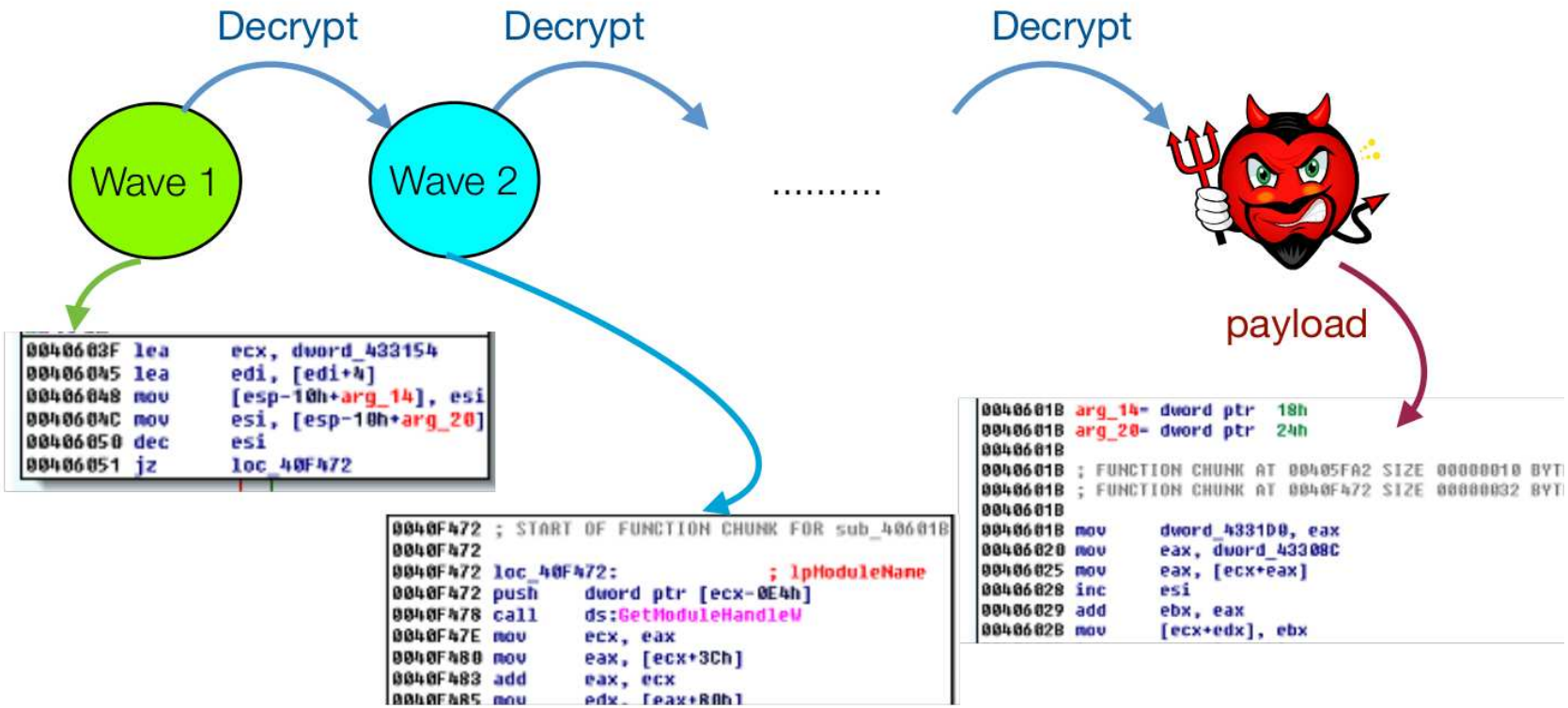
- GUEB + manual analysis [j. comp. virology 14]
  - `tiff2pdf` : CVE-2013-4232
  - `openjpeg` : CVE-2015-8871
  - `gifcolor` : CVE-2016-3177
  - `accel-ppp`
- GUEB + BINSE/SE [sefm16,sub. woot16]
  - `Jasper JPEG-2000` : CVE-2015-5221

In progress

- **third step : exploitability analysis** of UaF
  → SMT-axiomatization of memory allocation strategies
- proFTPD industrial case study

Malware detection and Deobfuscation

Context : x86-malware



A common protection scheme for malware
a SillyFDC run

Self-modifying program schema

Malware detection and Deobfuscation

Context : x86-malware

Decrypt Decrypt Decrypt

Recognize malware despite polymorphism

- morphological analysis [malware13,fps14,malware15]
- experiments on Stuxnet, Flame, Duqu, Gauss, Waledec, Regin
- startup Simorfo since 2016

- standard anti-virus signatures ≈ byte sequences
  . easy to fool with slight rewriting (polymorphism)
- here : signatures = normalized subgraphs
  . much harder to fool !
  . efficient to compute / check
  . in practice : add white lists

Malware detection and Deobfuscation

Context : x86-malware

Decrypt Decrypt Decrypt

Recognize malware despite obfuscation

- Codisasm [ccs15] : self-modifcation, code overlapping, anti-debug
- BINSEC/SE [saner16, sub. ccs16] : malware exploration, opaque predicate, stack tampering

```
00406048 mov     [esp-10h+arg_14], esi
0040604C mov     esi, [esp-10h+arg_20]
00406050 dec     esi
00406051 jz      loc_40F472
```

```
0040601B arg_14= dword ptr  18h
0040601B arg_20= dword ptr  24h
0040601B
0040601B ; FUNCTION CHUNK AT 00405FA2 SIZE 00000010 BYT
0040601B ; FUNCTION CHUNK AT 0040F472 SIZE 00000032 BYT
0040601B
```

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

Self-modifying program schema

Conclusion

Binary-level security analysis

- many applications
- many challenges

Our approach

- leverage powerful methods from formal software analysis
- *pragmatic* formal methods (combination, tradeoffs, etc.)
- common basic solutions (models, analysis) + application-directed methods

A few results

- vulnerability analysis [new CVE discovered]
- malware detection [startup]
- deobfuscation [applications on obfucated malware]
- low-level memory regions [toward refining CompCert memory model]
- open-source platform [release soon]