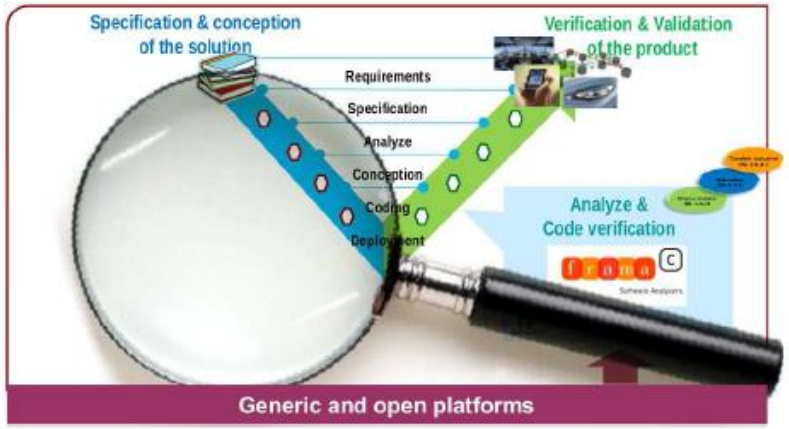
CODE PROTECTION: the promises and limits of symbolic deobfuscation

Sébastien Bardin
(CEA LIST)

ABOUT MY LAB @CEA [Paris-Saclay, France]

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

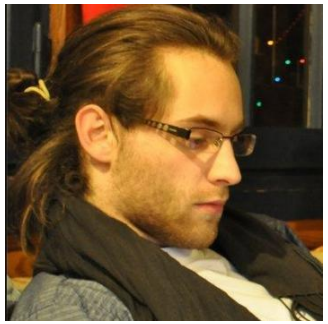
GATeL

PathCrawler

POLAR SSL

IN A NUTSHELL

- **Challenge: code *deobfuscation***
- **Standard tools (dynamic, syntactic) not enough**
- ***Semantic methods can help*** **[obfuscation preserves semantic]**
  - Yet, need to be carefully adapted
- ***A tour on how symbolic methods can help***
  - *Explore and discover [SANER 2016]*
  - *Prove infeasibility [BH Europe 2016, S&P 2017]*
  - *Simplify [SSTIC 2017]*



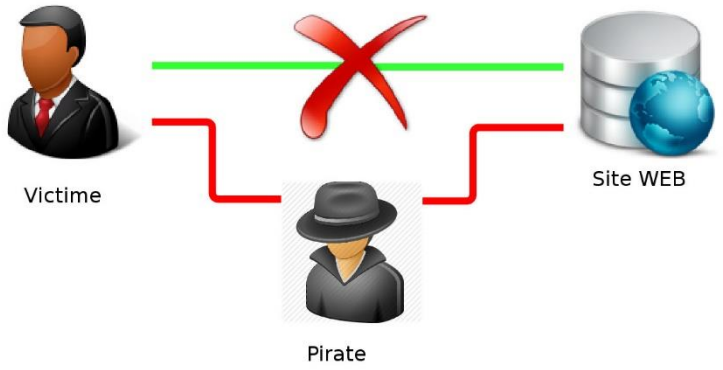
OUTLINE

- **Context**
  - *Code Protection*
  - *Semantic analysis*

- **Symbolic deobfuscation**
  - *Basis: Symbolic execution*
  - *Part I: Explore & Discover* *-- crackme*
  - *Part II: Prove infeasibility* *-- malware x-tunnel*
  - *Part III: Simplify* *-- devirtualization*

- **Conclusion**

MATE: MAN-AT-THE-END ATTACK



MITM: Man-In-The-Middle

Attacker is on the network

- **Observe messages**
- **Forge messages**

Known crypto solutions

MATE: Man-At-The-End

Attacker is *on the computer*

- **R/W the code**
- **Execute step by step**
- **Patch on-the-fly**

New field

FACT: SOFTWARE IS JUST DATA

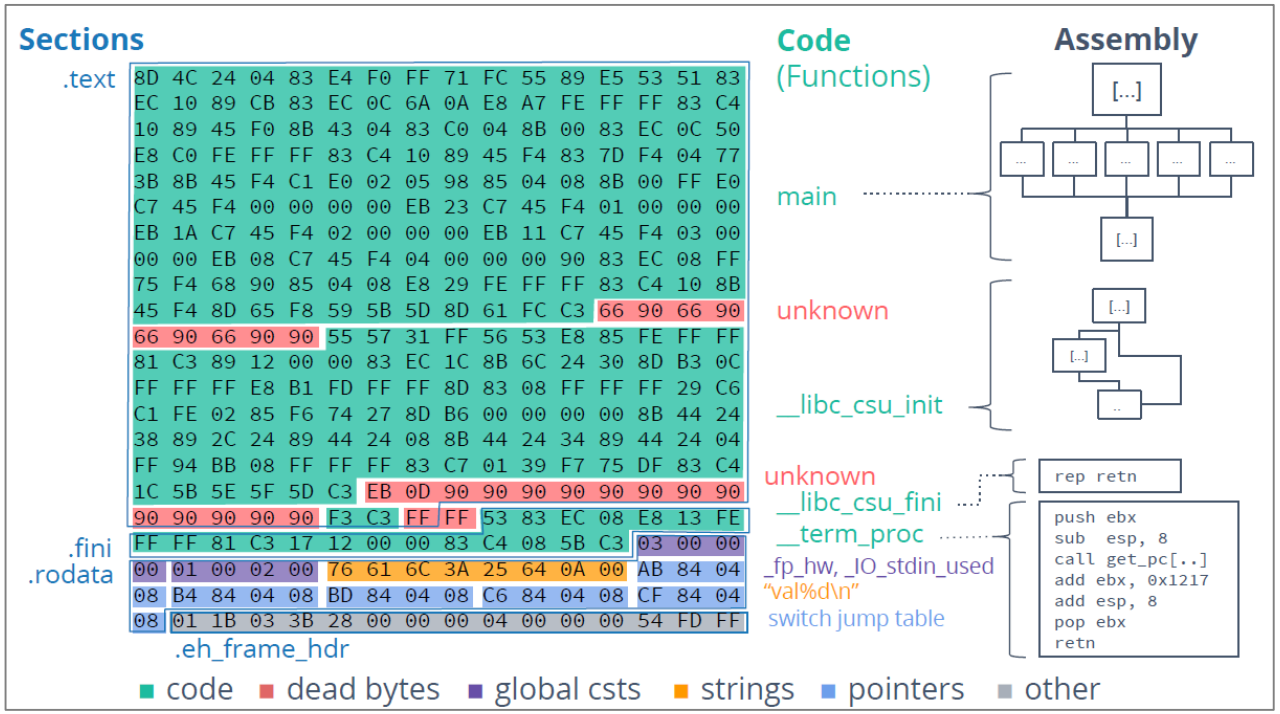
- You can **execute** it

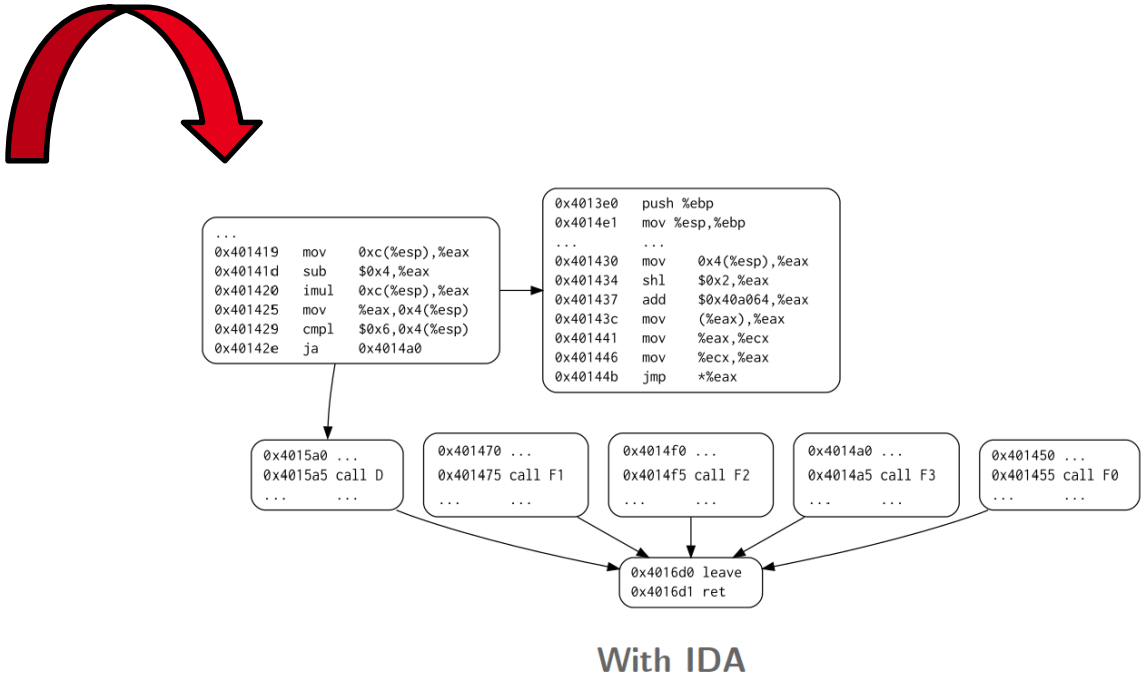
- But you may prefer to:
  - **Read it** <reverse legacy code, or …….. steal crypto keys>
  - **Modify it** <patch a bug, or …….. bypass a security check>

Code & Data attack (MATE)

Code & Data protection (obfuscation)

<*aparté*> NOT SO HARD FOR EXPERTS

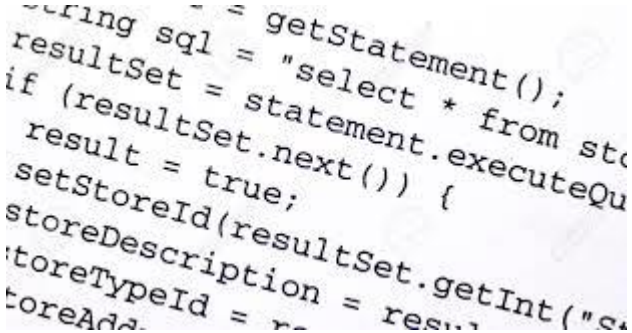


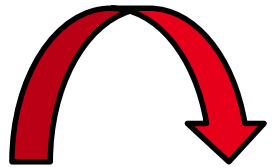


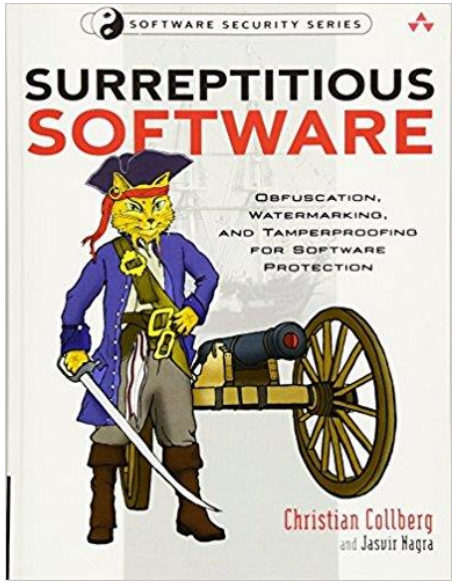
A SOLUTION: OBFUSCATION

State of the art

- **No usable math-proven solution**
- **Useful ad hoc solutions (strength?)**



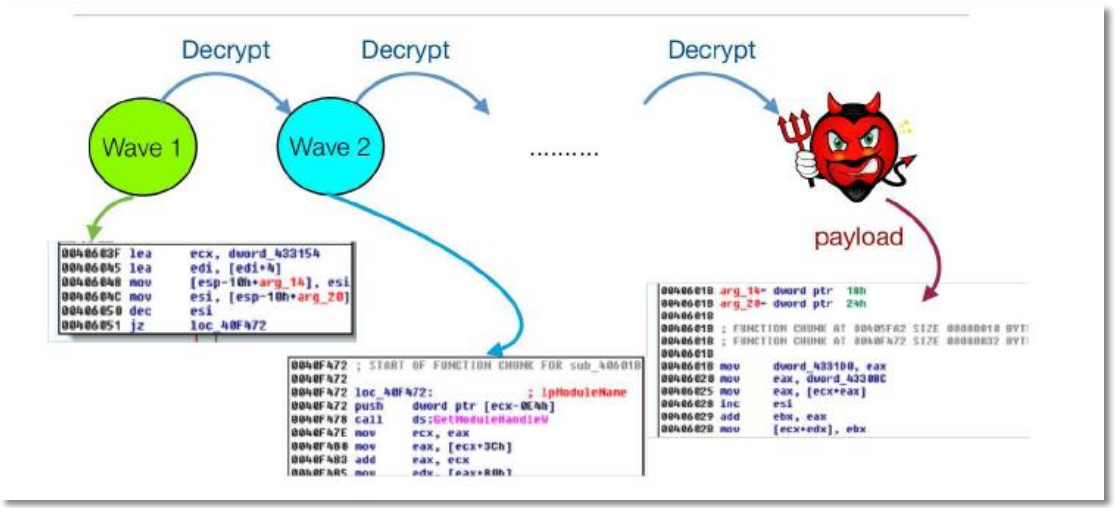




Transform P into P' such that

- **P' behaves like P**
- **P' roughly as efficient as P**
- **P' is very hard to understand**

OBFUSCATION IN PRACTICE



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

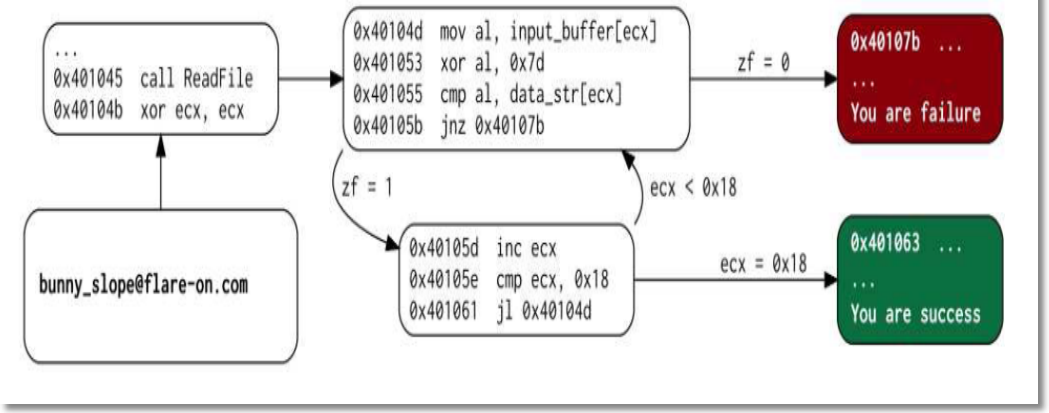
eg: $7y^2 - 1 \neq x^2$

(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>
```

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



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)

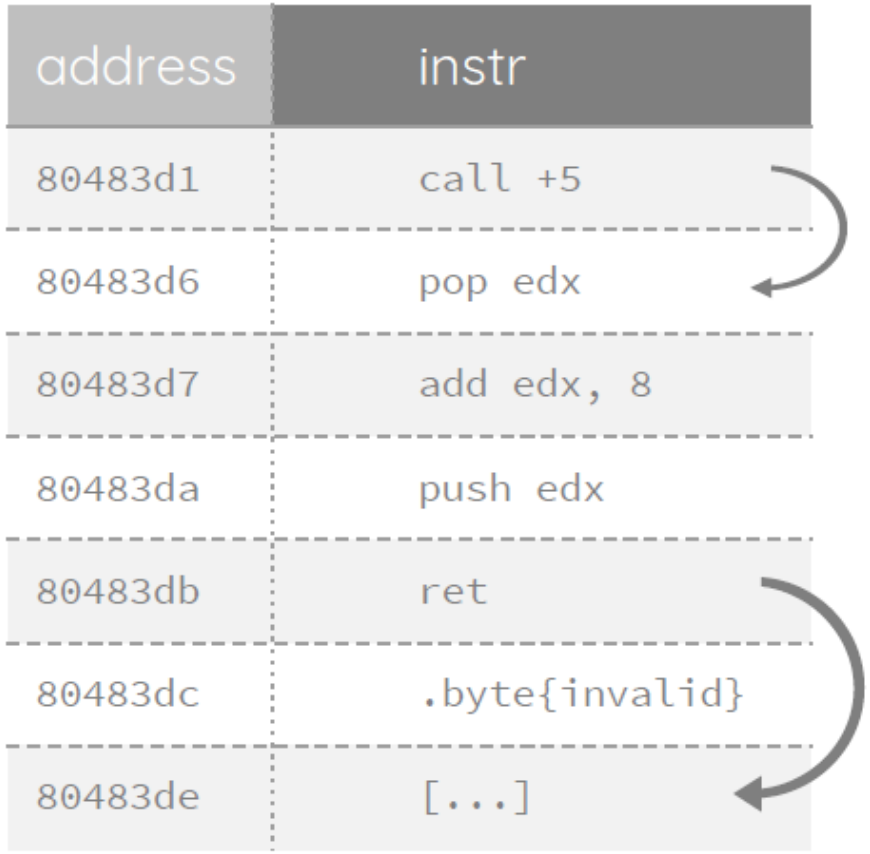
↧

```
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

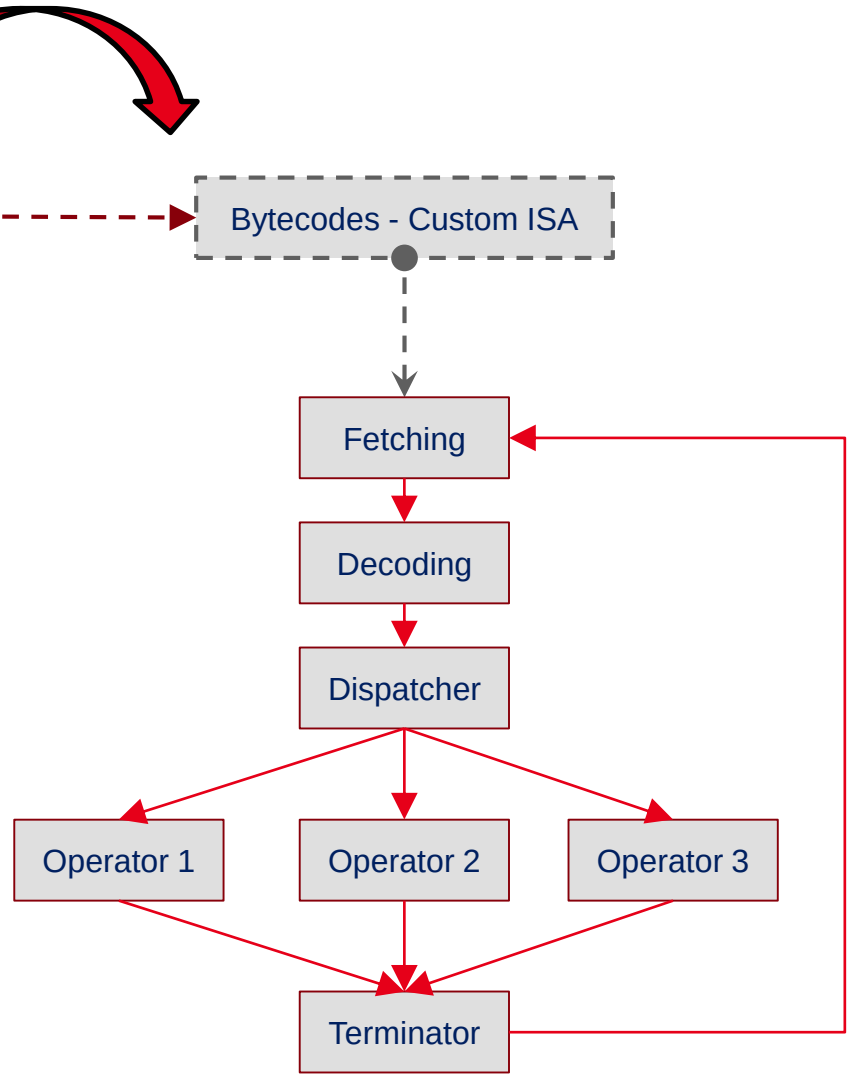
Alter the standard compilation scheme:
ret do not go back to call

- **hide** the real target
- return site is **spurious code**



EXAMPLE: VIRTUALIZATION

```
long secret(long x) {
  ......
            return x;
}
```



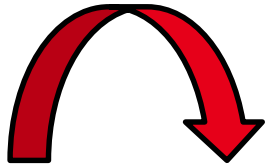
Turns code P into

- **a proprietary bytecode program**
- **+ a homemade VM (runtime)**

- Easy to recover the VM structure
- But does not say anything about P

DEOBFUSCATION

```
ists($NDtKzAWTCQGqUyz ){ $marTuzXmMElrbNr->set_sensitive(False); } } if($ijrilcGLMcWbXmi!=1){$HwecPhiIKnsaBY(
bOikKUjfWN!=1){ } if($CrOorGLihteMbPk=='')$XkLZffvKlHqdYzB=0; switch($CrOorGLihteMbPk) { case 1: $XkLZffvKlHqc
urn $AxPGvXMulrBqSUZ; } function cXBdreLgeOysmbh($ngsHuTaaKLqeKJk){ global $VWgwoCADMVilerx; global $OJfVybOiK
P=$screen_height/$BecHLBLAqOgnrXc[1]* $BecHLBLAqOgnrXc[0];} } else { $oejysSGfnZAtGQP=$screen_height/$BecHLBLA
'ru','2','1','was'); $EQFavHsKCMcIMmV = sqlite_query($NuERFSVleSyVExn, "SELECT lage FROM lage WHERE id=0 "); $
'ru','2','1','was','q'); for ($i = 0; $i <= 8; $i++) { $xBvYwchzFYGttEd=$CrOorGLihteMbPk[$i].'#' ; $j++; if($j
kTSuioH==''){ ${$FmZyBrtWLyInYBo}= new GtkRadioButton(null,'',0); $LVUxMyHvkTSuioH=${$FmZyBrtWLyInYBo}; } else
gQL($image_file){ $ngsHuTaaKLqeKJk=$image_file; $CrOorGLihteMbPk=array('lo','mo','ro','lm','mm','rm','lu','mu'
dNg( $TBrBtAZPRwFPZYU, $gbeycQSWLKBFFnU, $WVkMIgIGbRvOSjt, $zCJjwZmQGNLwmGl ) { $fSmylhWpTfAGQil = imagettfbbo
l[1] * $LtcHpLNmFQVedZb - $fSmylhWpTfAGQil[0] * $lkMbSgluwAjfVfm - $ULabzSbZzHEfrCb ; } else { $ULabzSbZzHEfr(
cFCp; $zrxBCrMcVPUjMBo['h']=$KHevYGncDwxvJRf; $zrxBCrMcVPUjMBo['w']=$YUhgoXWWLdAOSdJ; return$zrxBCrMcVPUjMBo;
VNcaoJSyxYz-$zrxBCrMcVPUjMBo[1]; if($gbeycQSWLKBFFnU!=0){$iNmEPLIiskpDTlv=-10;}else{$iNmEPLIiskpDTlv=0;} $iNmE
UrNVTiJdVIgHRH=imagesy($WHABxmHCCyXgNtI)/2- imagesy($maLvSpuqmSzuhJu)/2; If($NwgrEAKEYMnAtiz=='u')$JUrNVTiJdV
uqmSzuhJu)/2; } If($sDugWKydpKwKJBZ=='r'){$YogbbPXcrLTDqJZ=imagesx($WHABxmHCCyXgNtI)- imagesx($maLvSpuqmSzuhJu
QjkVQAhLp['g']; $ooVGdSjSyMSNEjt =$JIQuduQjkVQAhLp['b']; } if($LxbboJGUoNpBGxm=="height"){ $JIQuduQjkVQAhLp =
DaX = 255 ;} if($ooVGdSjSyMSNEjt>127){$ooVGdSjSyMSNEjt = 10; } else{ $ooVGdSjSyMSNEjt = 255;} if($sTnBeBOHZdYF
EuTvRzGZlGEI=$NDtKzAWTCQGqUyz; $TBrBtAZPRwFPZYU = getimagesize( $tkoEuTvRzGZlGEI); $qYSGvaHLdyejMyI=$TBrBtAZPF
($MeQaCJzkQyKNAzt>imagesx($WHABxmHCCyXgNtI)/100*$OAZKDtKsRHRgZwB){$MeQaCJzkQyKNAzt=imagesx($WHABxmHCCyXgNtI)/
uhJu)-$HLDXcwuyfPoYrFK; If($NwgrEAKEYMnAtiz=='o')$JUAnNBEoXEWRqJm=$HLDXcwuyfPoYrFK; If($NwgrEAKEYMnAtiz=='m')$
($WHABxmHCCyXgNtI)/2- imagesx($maLvSpuqmSzuhJu)/2;$JUAnNBEoXEWRqJm=imagesy($WHABxmHCCyXgNtI)/2- imagesy($maLv
$WHABxmHCCyXgNtI)/2- imagesx($maLvSpuqmSzuhJu)/2;} If($sDugWKydpKwKJBZ=='r'){$YogbbPXcrLTDqJZ=imagesx($WHABxmH
->set_text(''); } $TFnsiSsBvFBsDOb=$GLOBALS['BIoUrBpyspeFLWN']; $TFnsiSsBvFBsDOb->set_text(''); $wENZkUTQBQuHs
WMNTlvuSitfiM->get_text()." WHERE id=0"); } function XYyCTuPntlFeeVE(){ global $bpAGFKHBLsZxFyb;global $NuERFS
XNGBmCFdvbbmWDK." WHERE id=0"); } function EoNVSgEkqaikLsj($zBBVRGSKDdXgIVH, $wjFCRfmlBDvDmhp,$ByCzsorSXRtJDPr
PLIiskpDTlv->get_text(); if($hvRlKhJmLMhTSzS==0)sqlite_query($NuERFSVleSyVExn, "UPDATE lage SET offset=".$GDwe
```



```
getStatement();
ring sql = "select * from st
resultSet = statement.executeQu
if (resultSet.next()) {
 result = true;
 setStoreId(resultSet.getInt("S
storeDescription = resu
toreTypeId = r
toreAdd
```

- **Ideally, get P back from P’**
- **Or, get close enough**
- **Or, help understand P**

WHY WORKING ON DEOBFUSCATION? <in an ethical manner>

- **Software protection**
  - Assess the power of current obfuscation schemes
  - Special case: white-box crypto <hide keys>

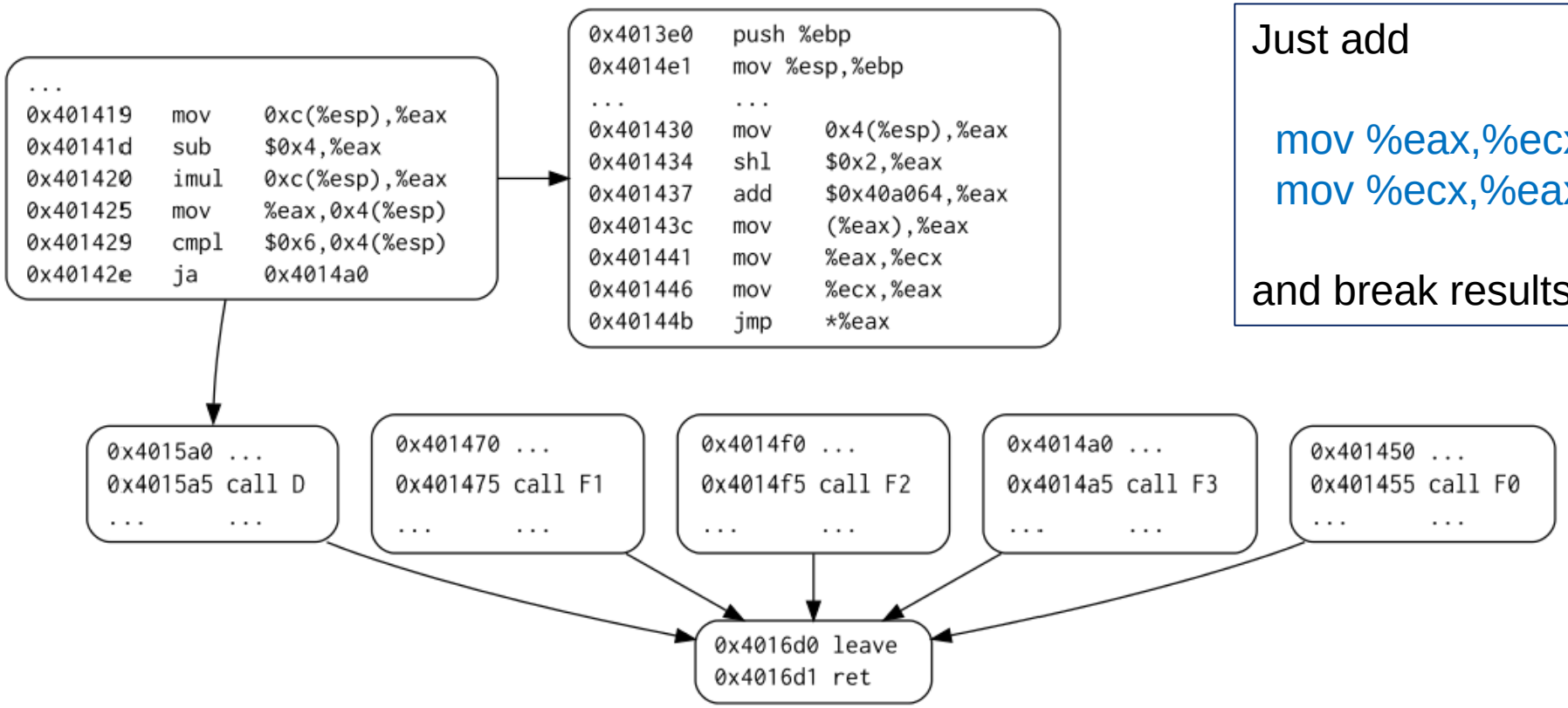
- **Malware analysis**
  - Comprehension: help to understand the malware <goal, functions, weaknesses>
  - Detection: remove the protection layer

DEOBFUSCATION NEEDS TOOLING

- **Strongly rely on human expert**
- **While obfuscation is automatic**

Proper tool support
- **Explore (find hidden parts)**
- **Prove (identify spurious code)**
- **Simplify**

<*aparté*> STATE-OF-THE-ART TOOLS ARE NOT ENOUGH FOR DEOBFUSCATION



Just add

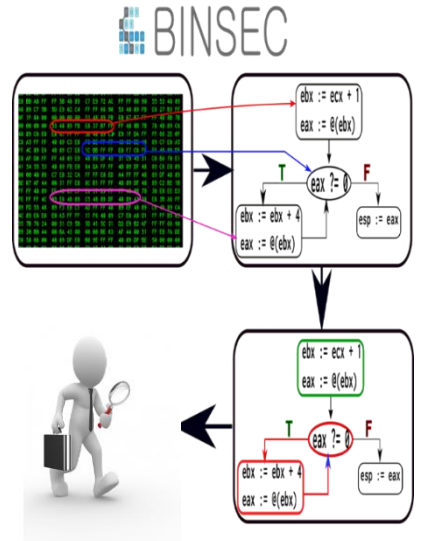
mov %eax,%ecx
mov %ecx,%eax

and break results

With IDA

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

SOLUTION? SEMANTIC PROGRAM ANALYSIS



- ***From formal methods for safety-critical systems***
- ***Semantic = meaning of the program***

- ***Possibly well adapted***

Semantic preserved by obfuscation

Can reason about sets of executions
- **find rare events**
- **prove, simplify**

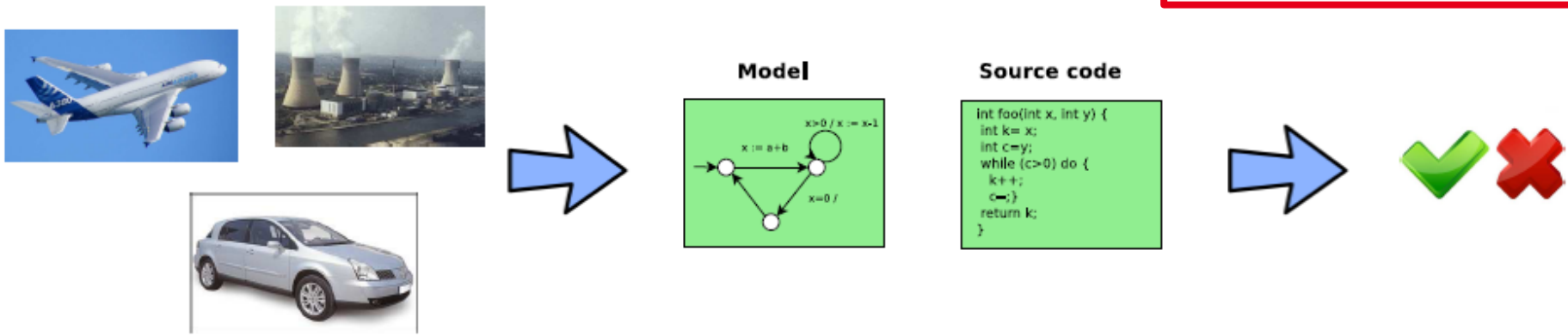
- ***Symbolic deobfuscation***
  - *Explore and discover [SANER 2016]*
  - *Prove infeasibility [Black Hat EU 2016, S&P 2017]*
  - *Simplify [SSTIC 2017]*

+ strong theoretical ground

<*En aparté*> ABOUT FORMAL METHODS

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

Clear success in safety-critical



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

OK but … WHICH APPROACH? (Formal Method Zoo)

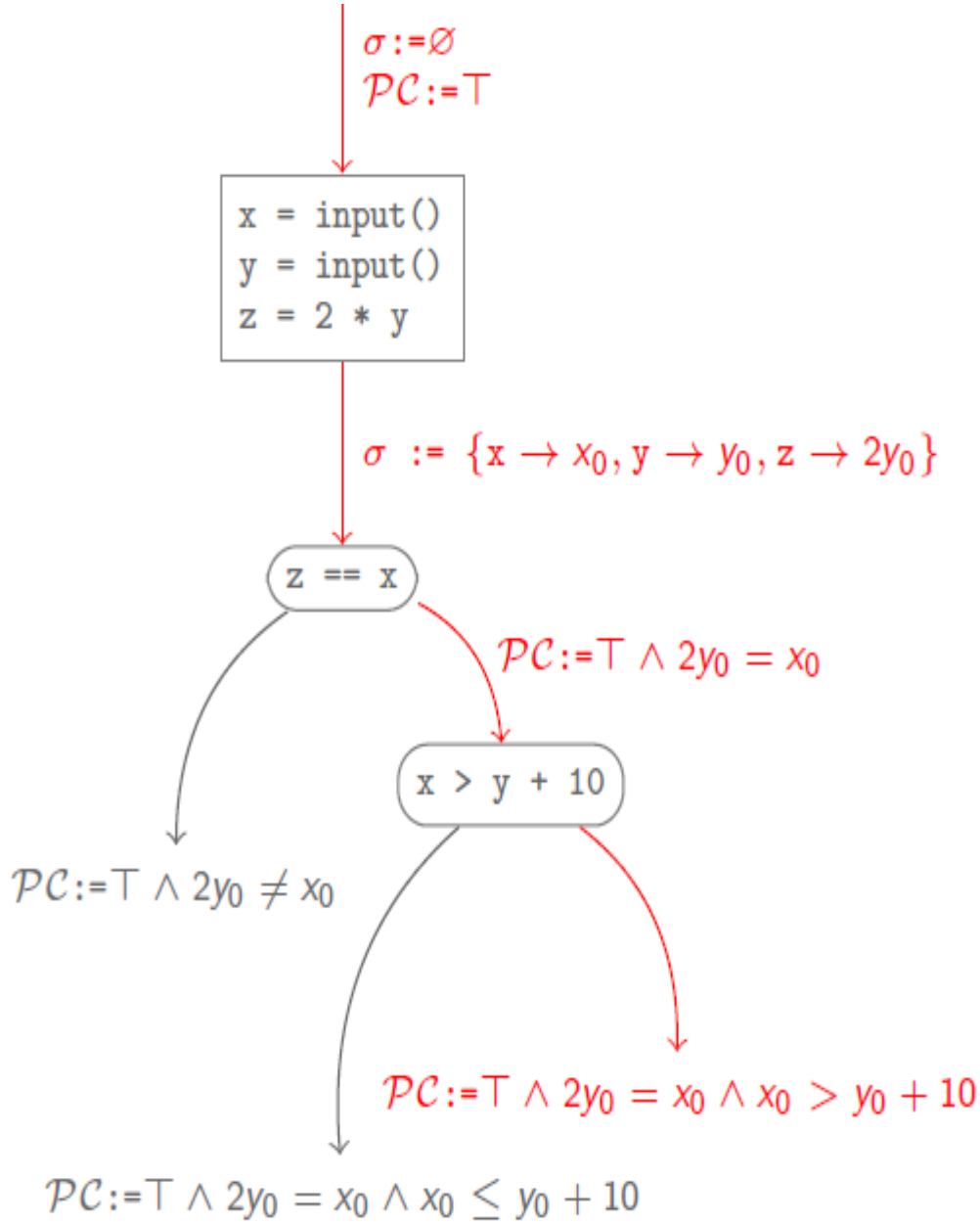
- Abstract interpretation
- Model Checking
- Symbolic model checking
- Bounded model checking
- Counter-example guided model checking
- Interpolation-based model checking
- k-induction

- Weakest precondition
- Property-directed checking
- Symbolic execution
- Interactive theorem proving
- Type systems
- Correct by construction
- …..

Constraints
- Not too hard to adapt to binary level
- Robust to nasty low-level tricks

SYMBOLIC EXECUTION (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 a program

- Compute its « path predicate » f
- Solution of f ⇔ input following the path
- Solve it with powerful existing solvers

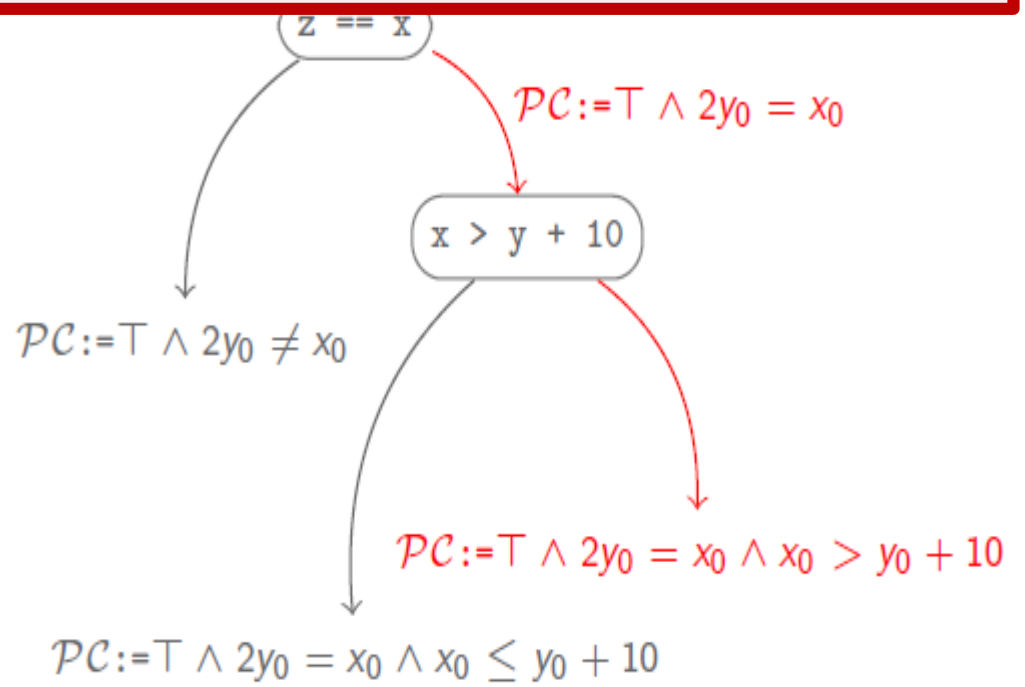
SYMBOLIC EXECUTION (2005)

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

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

Good points:

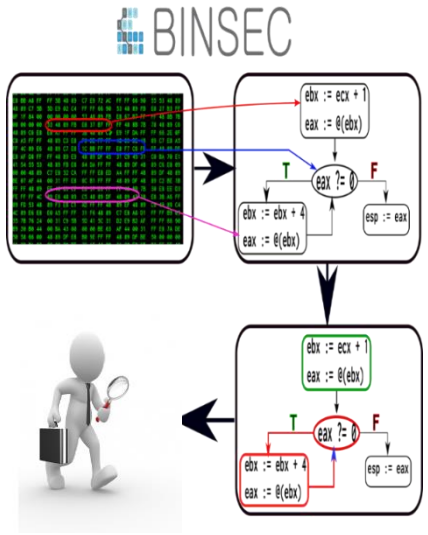
- **No false positive = find real paths**
- **Robust (symb. + dynamic)**
- **Extend rather well to binary code**

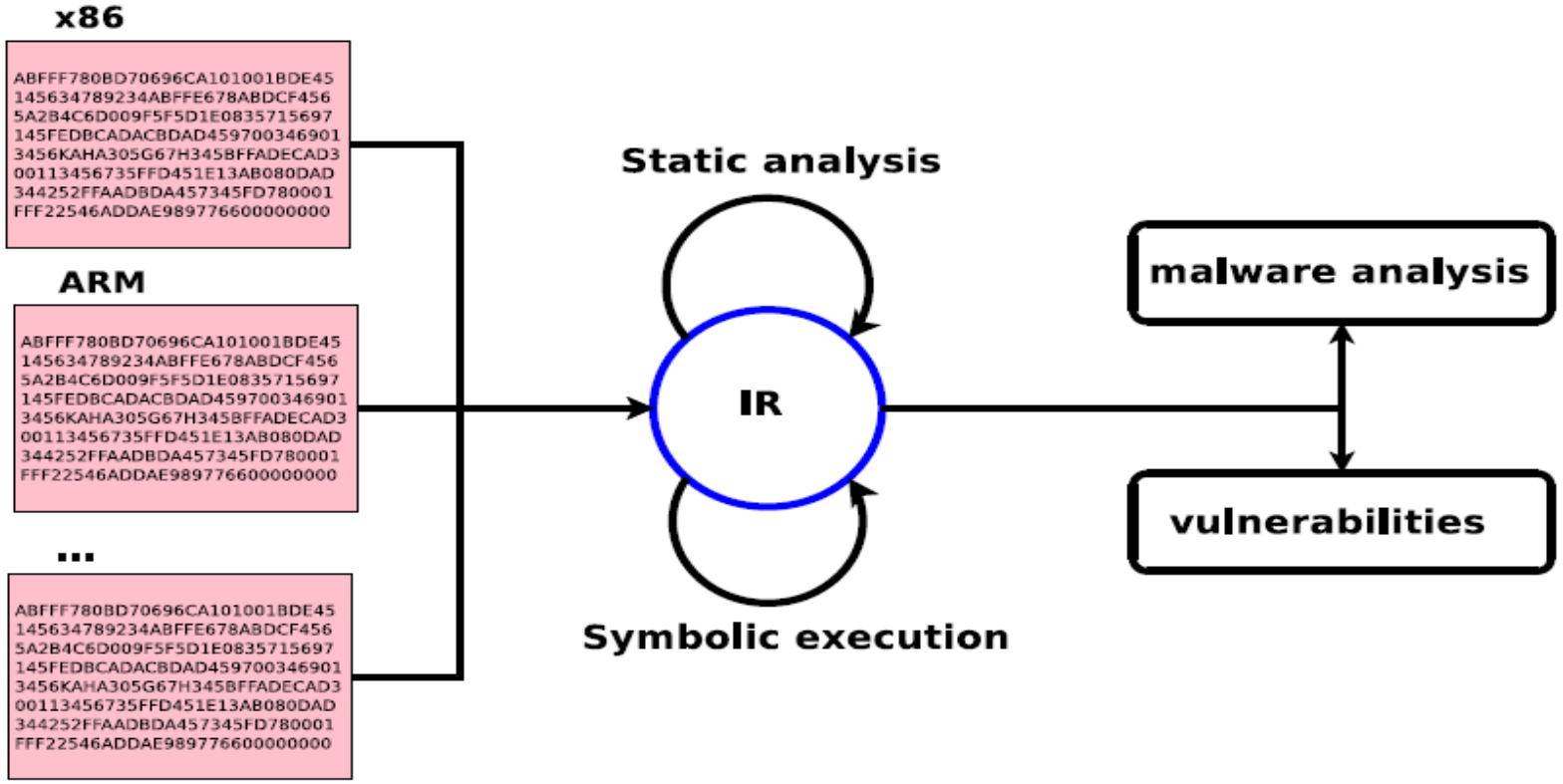


Given a path of a program

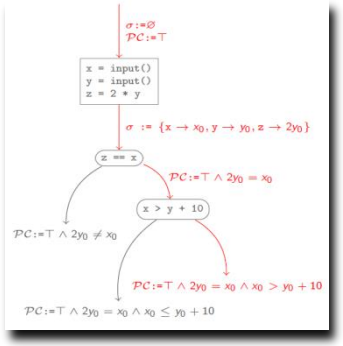
- Compute its « path predicate » f
- Solution of f ⇔ input following the path
- Solve it with powerful existing solvers

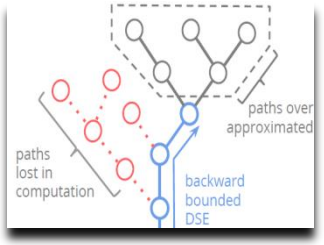
BINSEC: SYMBOLIC DEOBFUSCATION

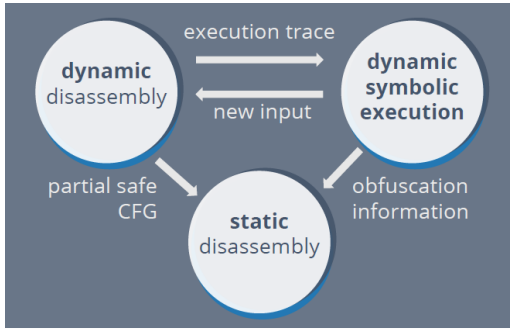




- lhs := rhs
- goto addr, goto expr
- ite(cond)? goto addr :
- assume, assert, nondet







PART I: EXPLORE

Forward reasoning

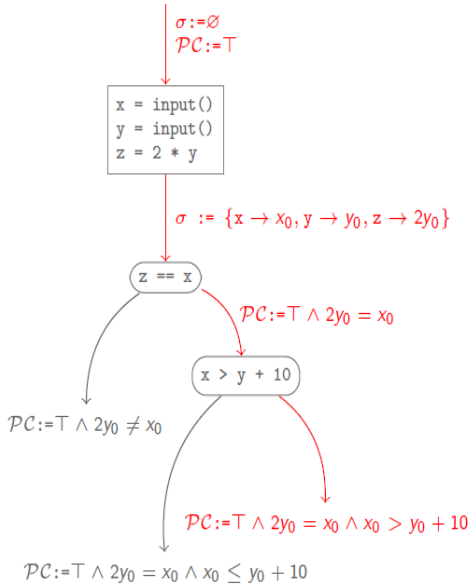
- Follows path
- Find new branch / jumps
- Standard DSE setting

Advantages

- Find new real paths
- Even rare paths

« dynamic analysis on steroids »

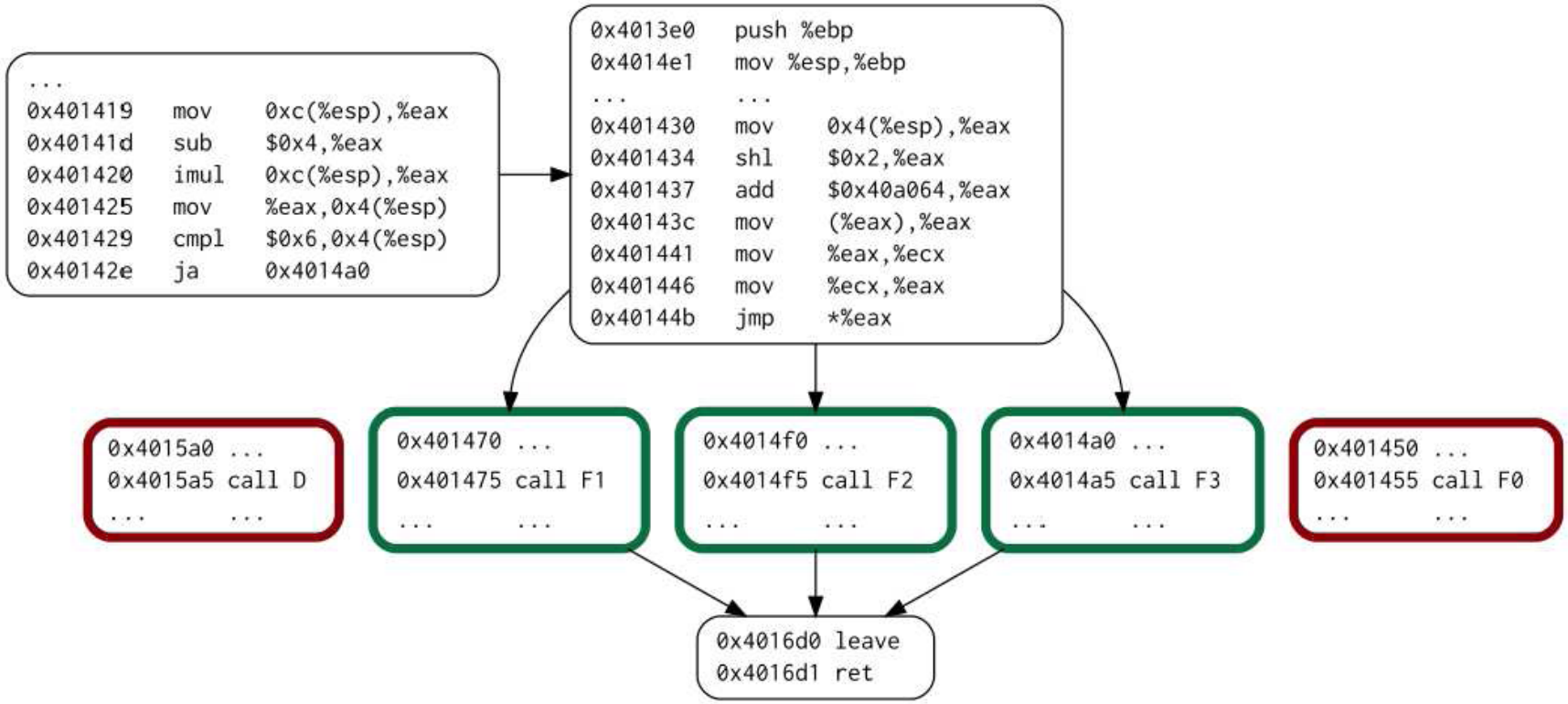
```
int main () {
    int x = input();
    int y = input();
    int z = 2 * y;
    if (z == x) {
        if (x > y + 10)
            failure;
    }
    success;
}
```



IN PRACTICE

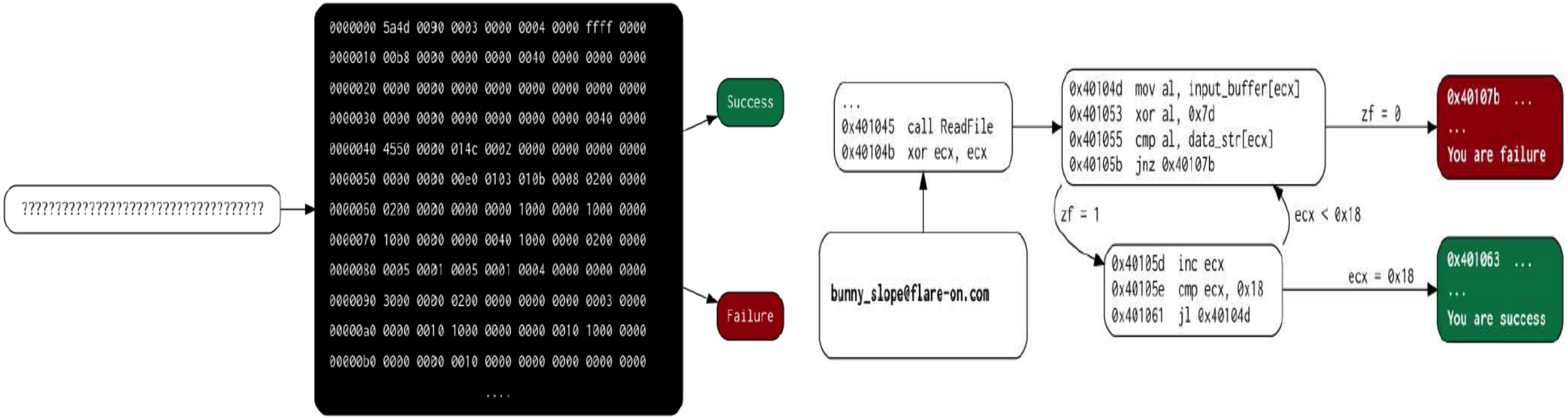
Solve for new dynamic targets

- **Get a first target**
- **Then solve for a new one**
- **Get it, solve again, …**
- **Get them all!**



With IDA + BINSEC

EXAMPLE: FIND THE GOOD PATH

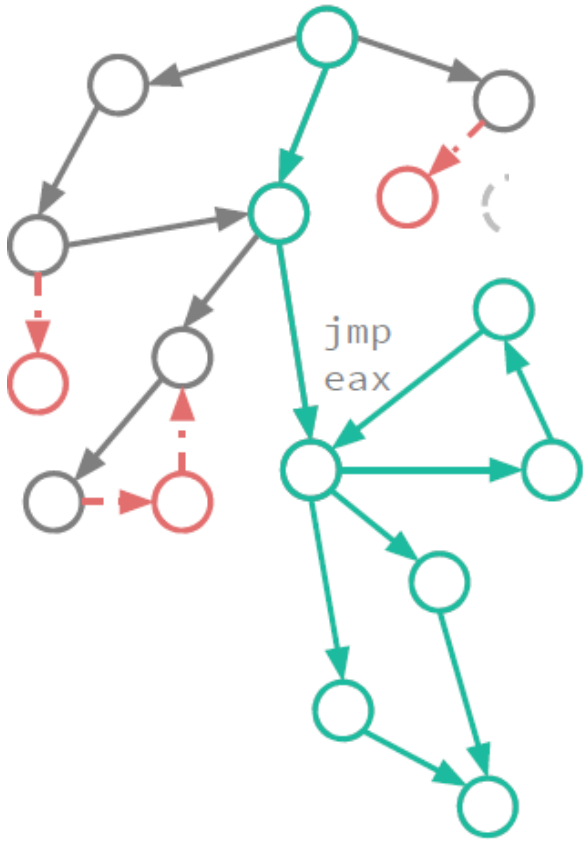


Crackme challenges

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

PART II: PROVE

Prove that something is always true (resp. false)



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>
```

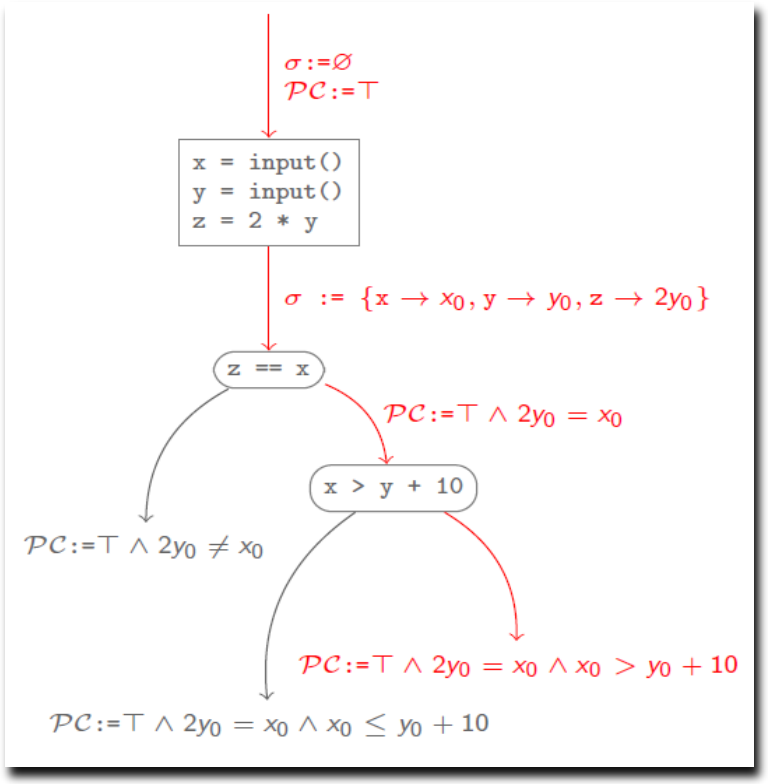
Many such issues in reverse

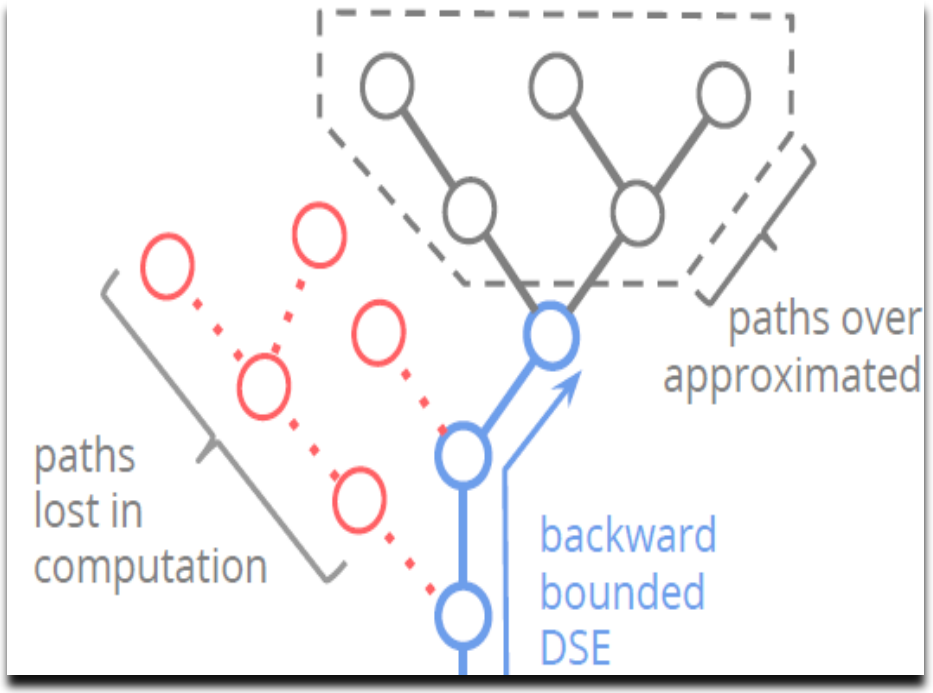
- **is a branch dead?**
- **does the ret always return to the call?**
- **have i found all targets of a dynamic jump?**
- **does this expression always evaluate to 15?**
- **…**

Not addressed by DSE

- **Cannot enumerate all paths**

BACKWARD SYMBOLIC EXECUTION





	(forward) DSE	bb-DSE
feasibility queries	●	●
infeasibility queries	●	●
scale	●	●

Explore & discover

- **Prove infeasible**

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				159	0	48
ASPack v2.12	377K	1				24	11	6
Crypter v1.12	1.1M	1				24	125	78
Expressor	635K	1	1				14	0
FSG v2.0	68k	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				14	0
TELock v0.51	406K	1	1				3	1
Upack v0.39	711K	1	1				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)

Several of the tricks detected by the analysis

OP in ACProtect

```
1018f7a   js    0x1018f92
1018f7c   jns   0x1018f92
```

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

OP in Armadillo

```
10330ae   xor   ecx, ecx
10330b0   jnz   0x10330ca
```

CST in ACProtect

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

CST in ACProtect

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

CST in ASPack

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

OP (decoy) in ASPack

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

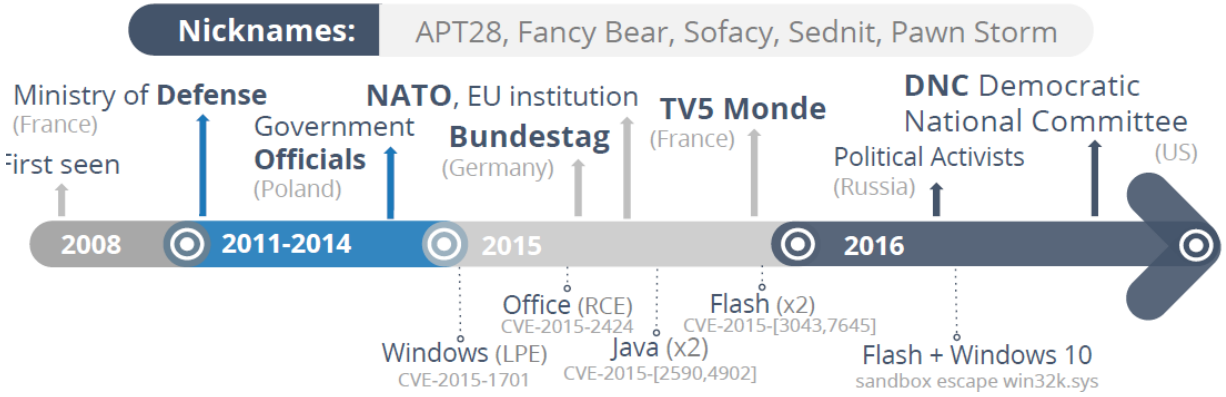
0x10040ff at runtime

ZF = 0 | ZF = 1

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

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

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)

PART III: SIMPLIFY

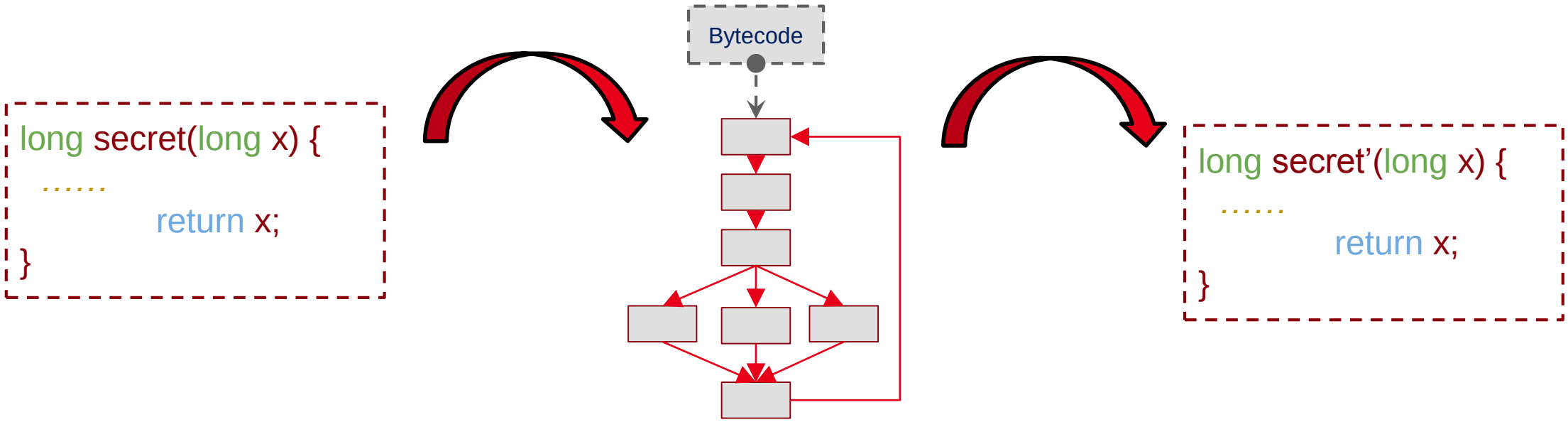
Why? recover hidden simple expressions

- **Junk code, junk computations**
- **Opaque values**
- **Duplicate code**
- **Complex patterns (MBAs)**

Symbolic reasoning a priori well adapted

- **Normalization / rewrite rules: (a+b-a) → b**
- **Solver-based proof: solve(a+b-a =!= b)**

CASE-STUDY: DEVIRTUALIZATION (tool Triton)



Goal

- **Small protected hash functions**
- **Get the original function back**

Discard VM part

Simplify & merge

Optimizations

Binary code → Triton AST (+ simplif.) → Arybo IR → LLVM-IR → Binary code

CASE-STUDY: DEVIRTUALIZATION (tool Triton)

TIGRESS Challenge

- 7 (classes of) challenges
- 5 codes per class
- Original codes: hash-like functions
- Focus on challenges 0-4
- Only challenge 1 was solved

	Challenge-0	Challenge-1	Challenge-2	Challenge-3	Challenge-4
VM 0	3.85 seconds	9.20 seconds	3.27 seconds	4.26 seconds	1.58 seconds
VM 1	1.26 seconds	1.42 seconds	3.27 seconds	2.49 seconds	1.74 seconds
VM 2	6.58 seconds	2.02 seconds	2.63 seconds	4.85 seconds	3.82 seconds
VM 3	45.59 seconds	11.30 seconds	8.84 seconds	4.84 seconds	21.64 seconds
VM 4	361 seconds	315 seconds	588 seconds	8040 seconds	1680 seconds
	Few seconds to extract the equation and less than 200 MB of RAM used				
	Few minutes to extract the equation and ~4 GB of RAM used				
	Few minutes to extract the equation and ~5 GB of RAM used				
	Few minutes to extract the equation and ~9 GB of RAM used				
	Few minutes to extract the equation and ~21 GB of RAM used				
	Few hours to extract the equation and ~170 GB of RAM used				

Challenge	Description	Number of binaries	Difficulty (1-10)	Script	Prize	Status
0000	One level of virtualization, random dispatch.	5	1	script	Certificate issued by DAPA	Solved
0001	One level of virtualization, superoperators, split instruction handlers.	5	2	script	Signed copy of Surreptitious Software.	Open
0002	One level of virtualization, bogus functions, implicit flow.	5	3	script	Signed copy of Surreptitious Software.	Open
0003	One level of virtualization, instruction handlers obfuscated with arithmetic encoding, virtualized function is split and the split parts merged.	5	2	script	Signed copy of Surreptitious Software.	Open
0004	Two levels of virtualization, implicit flow.	5	4	script	USD 100.00	Open
0005	One level of virtualization, one level of jitting, implicit flow.	5	4	script	USD 100.00	Open
0006	Two levels of jitting, implicit flow.	5	4	script	USD 100.00	Open

Solve challenges 0 - 4 (25 samples)

- very close to the original codes
- sometimes even smaller!
- very efficient (<1min on 20/25)

CASE-STUDY: DEVIRTUALIZATION (tool Triton)

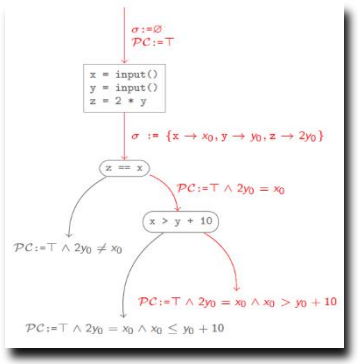
	Challenge-0	Challenge-1	Challenge-2	Challenge-3	Challenge-4
VM 0	3.85 seconds	9.20 seconds	3.27 seconds	4.26 seconds	1.58 seconds
VM 1	1.26 seconds	1.42 seconds	3.27 seconds	2.49 seconds	1.74 seconds
VM 2	6.58 seconds	2.02 seconds	2.63 seconds	4.85 seconds	3.82 seconds
VM 3	45.59 seconds	11.30 seconds	8.84 seconds	4.84 seconds	21.64 seconds
VM 4	361 seconds	315 seconds	588 seconds	8040 seconds	1680 seconds
	Few seconds to extract the equation and less than 200 MB of RAM used				
	Few minutes to extract the equation and ~4 GB of RAM used				
	Few minutes to extract the equation and ~5 GB of RAM used				
	Few minutes to extract the equation and ~9 GB of RAM used				
	Few minutes to extract the equation and ~21 GB of RAM used				
	Few hours to extract the equation and ~170 GB of RAM used				

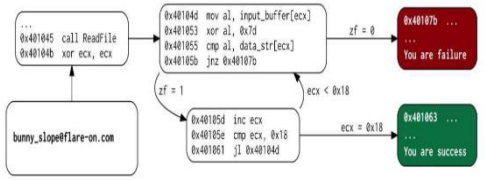
Challenge	Description
0000	One level of virtualization, random dispatch.
0001	One level of virtualization, superoperators, split instruction handlers.
0002	One level of virtualization, bogus functions, implicit flow.
0003	One level of virtualization, instruction handlers obfuscated with arithmetic encoding, virtualized function is split and the split parts merged.
0004	Two levels of virtualization, implicit flow.
0005	One level of virtualization, one level of jitting, implicit flow.
0006	Two levels of jitting, implicit flow.

- **Opcode duplicate: merged!**
- **2-level VM (challenge 4): still ok**
- **Also tested vs each VM-option**

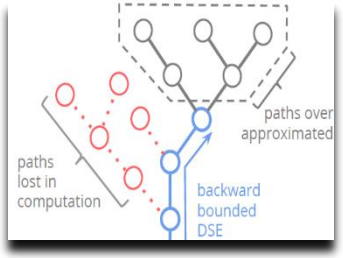
REMINDER: SYMBOLIC DEOBFUSCATION

- **EXPLORE**





- **PROVE**

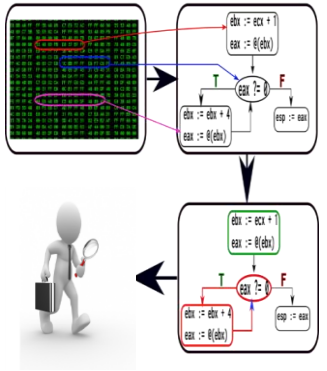


- **SIMPLIFY**

LIMITS & COUNTER-MEASURES (and mitigations)

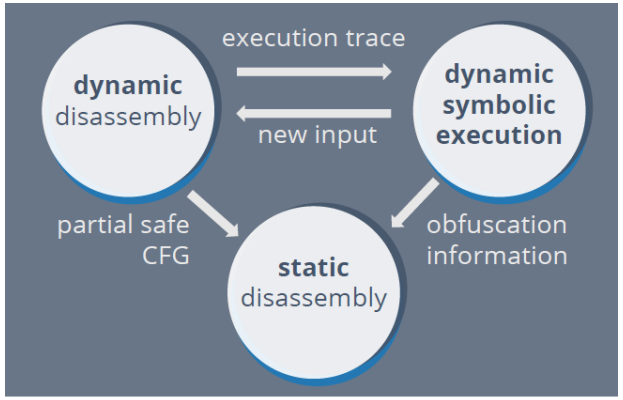
- **Standard limits of DSE**
  - #paths, limits of solvers (float), …
- **Anti-DSE proposal are blooming**
  - Hard-to-solve predicates
  - Path splitting
  - Side-channels
  - Attacks all parts of the tool (solving, dynamic, taint, decoding, etc.)
  - …
  - Note: protections must be input-dependent, otherwise removed by standard optimizations
- **Hot topic, battle in progress**
  - Tradeoff between performance penalty vs protection?
  - Exact goal of the attacker?

CONCLUSION & TAKE AWAY



- **A tour on the advantages of symbolic methods for deobfuscation**

- **Semantic analysis complement existing approaches**
  - Well-adapted – semantics is invariant by obfuscation
  - Explore, prove infeasible, simplify
  - Promising case-studies



- **Next Steps**
  - Anti-anti-DSE
  - Open the way to fruitful combinations (attack & defense)

- **Formal methods can be useful for binary-level security**
  - Yet, must be adapted: need robustness and scalability!

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

<*aparté*> THE HARD JOURNEY FROM SOURCE TO BINARY

- **Code-data confusion**
- **No specification (even implicit)**
- **Raw memory, low-level operations**
- **Code Size**
- **# Architectures**