Obfuscation: where are we in anti-DSE protections ?

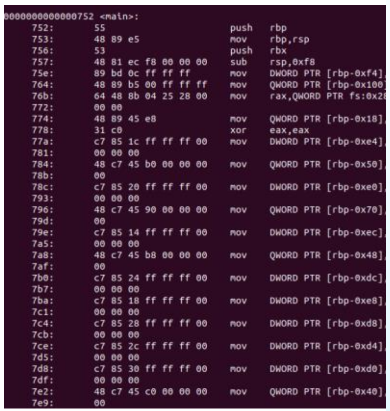
Mathilde OLLIVIER — **CEA LIST, France**

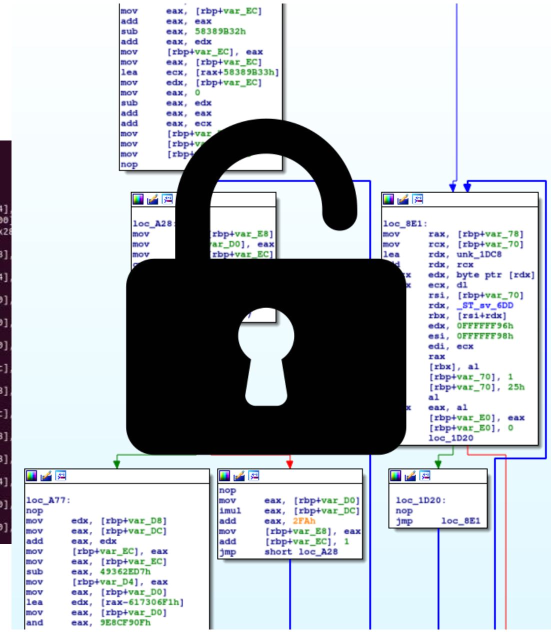
Sébastien BARDIN — CEA LIST

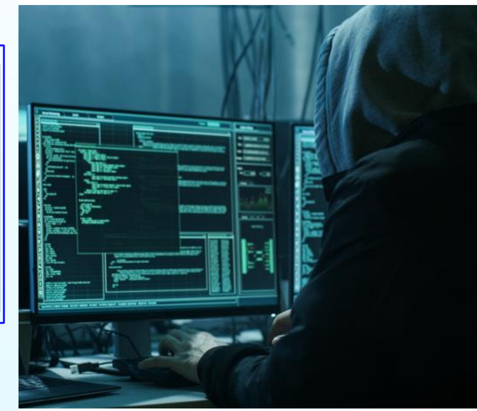
Jean-Yves MARION — Université de Lorraine, CNRS, LORIA

Richard BONICHON — Tweag I/O

Reverse engineering is a threat to IP





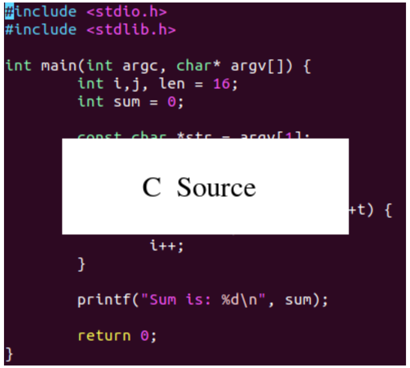


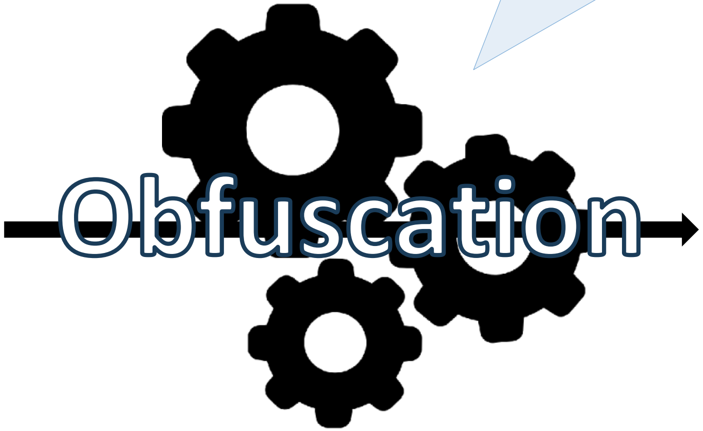
Easy with unprotected code

tools

Then we use obfuscation

- Functional equivalence
- Efficient
- "Harder " to analyze



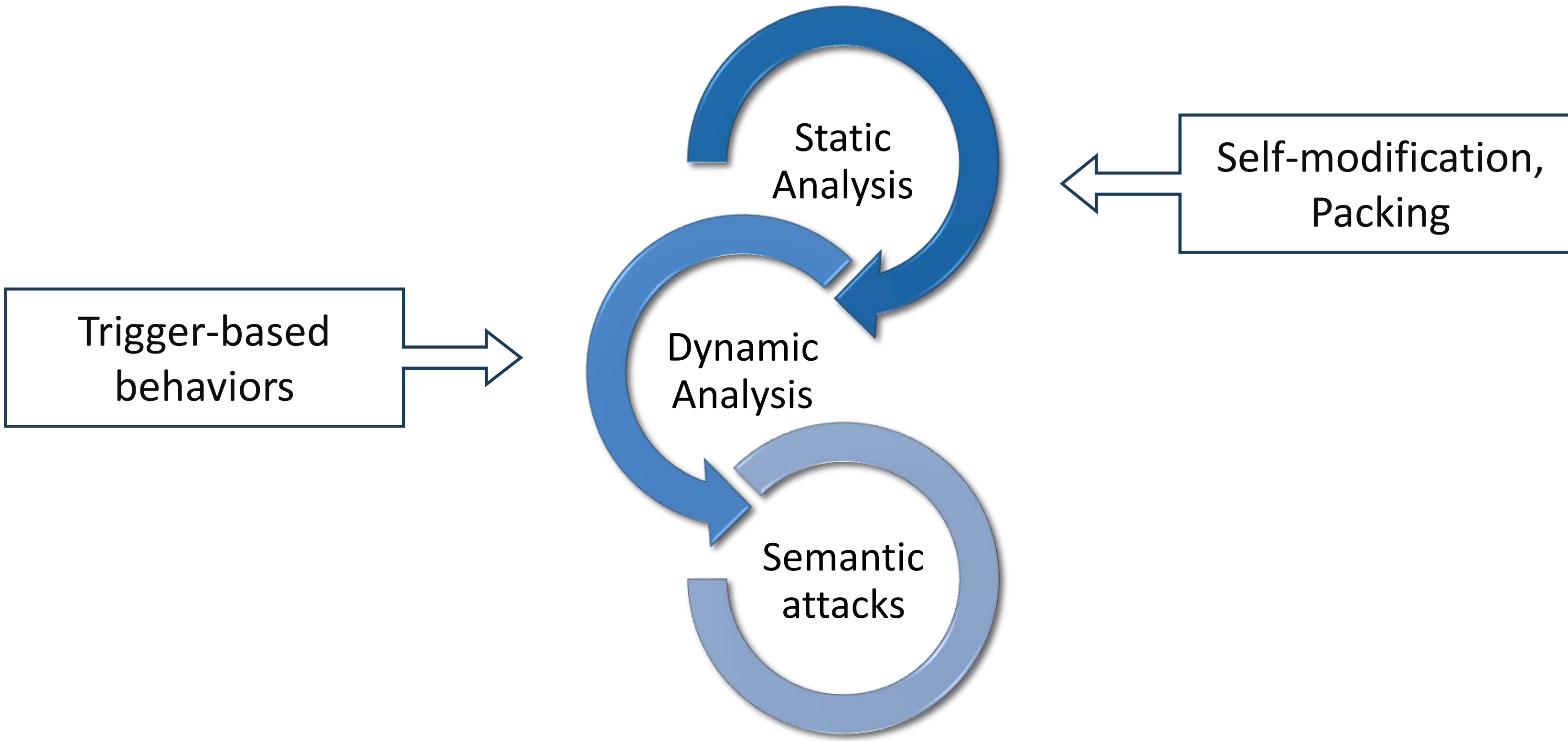


VMProtect software

THEMIDA

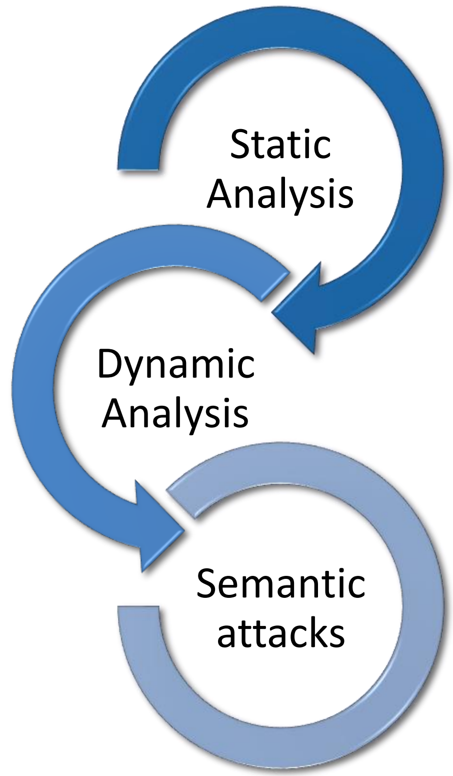
tools

Arm race


Trigger-based behaviors
Static Analysis
Dynamic Analysis
Semantic attacks
Self-modification, Packing

Arm race

Trigger-based behaviors



Self-modification, Packing

What now ?

Semantic attacks

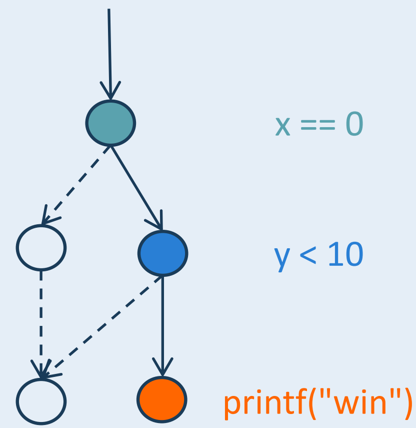
- **Dynamic Symbolic Execution** (DSE), Abstract Interpretation
- Bit-level taint analysis and DSE (Yadegari 2015)
- Backward Bounding DSE (David 2017)
- Banescu 2016

Dynamic Symbolic Execution (DSE)

C source code

```
int func (int x, int y) {
   if (x == 0) {
      if (y < 10) {
         printf("win");
      }
   }
}
```

Paths tree

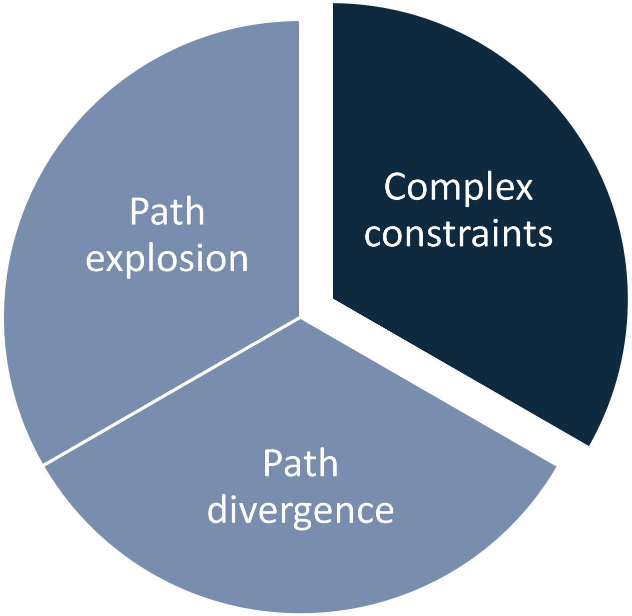


Path constraint to print "win":
$X=0 \wedge y<10$

Concrete solution using SMT solvers

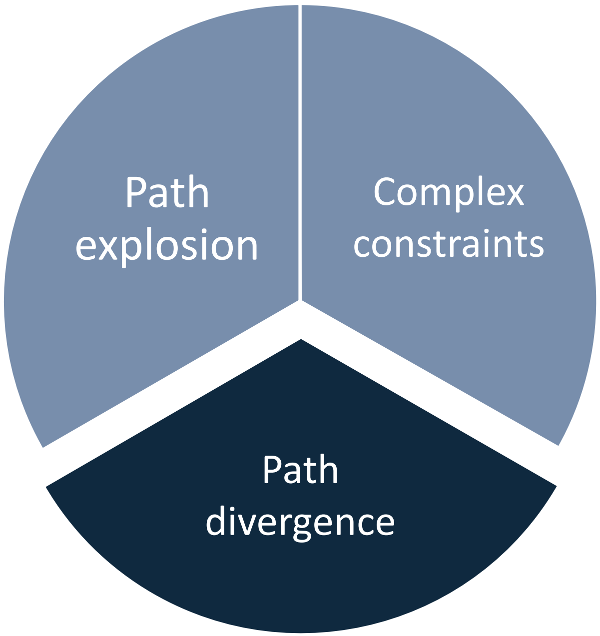
- As robust as dynamic analysis
- Infers triggers
- Simplifies code

What can we do against DSE ?



- Specific operations hard to solve
- Reduces the subset of paths that can be explored

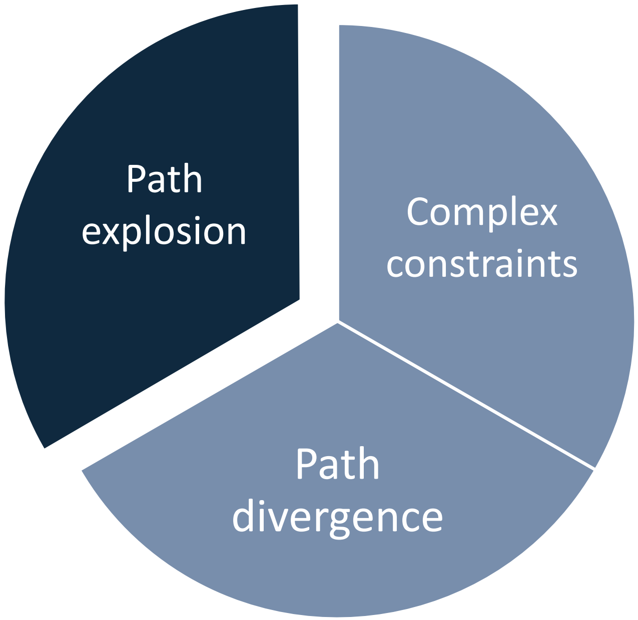
What can we do against DSE ?



- Computing correct path constraint can be hard
- DSE misses feasible paths and take unfeasible paths

What can we do against DSE ?

- DSE need to solve all constraints and store all pending states
- Realistically DSE can explore a reduced amount of paths in a limited amount of time



We need clear classification

Need clear classification and comparison

Today's contributions:

→ **Classify** existing protections

→ **Compare** protections using key parameters (strength, cost, stealth, implementation availability, etc.)

→ **Point out defficiencies** in the current state-of-the-art

Hard constraints

Mixed-boolean arithmetic – ZHOU et. al. 2007

original code

```
int func (int x, int y) {
    int var = x + y;
    if (var > 10) {
        // code
    }
}
```

obfuscated code

```
int func (int x, int y) {
    int var = (((x ⊕ y) + 2 × (x ∧ y)) × 39 + 23) × 151 + 111;
    if (var > 10) {
        // code
    }
}
```

$$var = \left(\left((x \oplus y) + 2 \times (x \wedge y)\right) \times 39 + 23\right) \times 151 + 111$$

Mixed-boolean arithmetic - CHARACTERISTICS

Strength & Cost

→ Solving constraints: NP-hard problem

→ No general results indicating that MBA are significantly harder to solve

→ Hard against **simplification** queries

→ No cost results for large and efficient MBA protections

Stealth & Mitigation

→ Specific use of uncommon operators

→ Mitigation using arithmetic simplification coupled with MBA expressions equivalence

Eyrolles et. al. 2016

Cryptographic hash functions – SHARIF et. al. 2008

original code

```
int func (int x) {
   if (x == TRIGGER) {
      // code
   }
}
```

obfuscated code

```
int func (int x) {
   if (hash(x) ==  HASHED_TRIGGER) {
      // encrypted_code
   }
}
```

Cryptographic hash functions- CHARACTERISTICS

Strength & Cost

- Irreversible functions by definition
- Relevant part of the code encrypted
- Encryption is not cheap

Stealth & Mitigation

- Cryptographic routines easy to spot
- Limited scope (trigger-based behaviors)

Path divergence

Self-modification

original code

```
      call    get_input()
L1:   call    func1()
```

obfuscated code

```
      call    get_input()
      mov     [L1], nop
L1:   call    spurious_func()
L2:   call    func1()
```

Self-modification - CHARACTERISTICS

Strength & Cost

→ Theoretically not an issue for DSE

→ Current symbolic engines cannot cope with this obfuscation

Moslty engineering effort

→ Full program unpacking has a high runtime cost

Stealth & Mitigation

→ Mitigations proposed but not implemented

Yadegari et. al. 2015
Brumley et. al. 2013
Bonfante et. al. 2015

→ Self-modification easy to spot

Symbolic code – YADEGARI *et. al.* 2015

original code

```
     call    get_input()
     cmp     eax, TRIGGER
     jz      L
     call    abort()
L:   call    payload()
```

obfuscated code

```
     call    get_input()
     sub     eax, TRIGGER
     // operations on eax
     mov     [L1], eax
L1:  nop
L2:  call    abort()
L3:  call    payload()
```

Symbolic code – YADEGARI *et. al.* 2015

original code

```
     call   get_input()
     cmp    eax, TRIGGER
     jz     L
     call   abort()
L:   call   payload()
```

obfuscated code

```
     call   get_input()
     sub    eax, TRIGGER
     // operations on eax
     mov    [L1], eax
L1:  jmp    L3
L2:  call   abort()
L3:  call   payload()
```

Symbolic code- CHARACTERISTICS

Strength & Cost

- Trigger-based behavior
- Current symbolic engines cannot cope with this protection
- Probably no runtime cost

Stealth & Mitigation

- Mitigation proposed but not implemented
  Yadegari et. al. 2015
- Self-modification easy to spot

Covert channel – STEPHENS *et. al.* 2018

original code

```
int func (int x) {
    int var = x;
    return(var);
}
```

obfuscated code

```
int func (int x) {
   int value=0;
   for( i in [0 . . . (bits in b)-1]) {
      timeT start = time();
      if (( i th bit of b)==1)
         slow_process(param);
      else
         fast_process(param);
      timeT time = time()-start;
      if (time > threshold)
          value |= 1 << i ;
   }
   int var = value;
return(var);
}
```

Covert channels - CHARACTERISTICS

Strength & Cost

- → State-of-the-art symbolic engines does not support covert channels
- → Some primitives hinder runtime performances
- → *Probabilistically* correct

Stealth & Mitigation

- → Sensitive to system call-based anomaly detection
- → No mitigation proposed

Path explosion

Linear obfuscation – WANG et. al. 2011

original code

```
int func (int x) {
   if ( x == 10) {
         // code
   }
}
```

Collatz conjecture

obfuscated code

```
int func (int x) {
   int y = x + 1000;
   while (y > 1) {
      if (y % 2 == 1) {
         y = 3 × y + 1;
      } else {
         y = y / 2;
      }
      if ((x – y > 28) && (x + y < 32)) {
         // code
         break;
      }
   }
}
```

Linear obfuscation- CHARACTERISTICS

Strength & Cost

- Input dependant loop
- Runtime number of loop iterations depends on input value

Stealth & Mitigation

- Common control flow structure
- But unusual arithmetic operators (modulo 3 or 5)
- Pattern attacks

Path-oriented protections

original code

```
int func (int x) {
    int var = x + 10;
    return(var);
}
```

obfucated code
- Range Divider -

```
int func (int x) {
    int var = 0;
    switch(x) {
        case 0:
            var = x+10;
        …
        case INT_MAX:
            //obfuscated version of "var=x+10"
    }
    return(var);
}
```

BANESCU et. al. 2016

obfuscated code
- For -

```
int func (int x) {
    int var = 0;
    for (int i=0; i<x+10; i++) {
        var++;
    }
    return(var);
}
```

OLLIVIER et. al. 2019

Path-oriented protections - CHARACTERISTICS

Strength & Cost

→ Input dependant loop

→ Strength exponential to the size in bits of the input space

→ Some primitives increase the size of the code

Stealth & Mitigation

→ Common control flow structure and no exotic operators

→ Some primitives use large jump tables

→ Pattern attacks (need diversity)

→ Path-merging ? Customized existing tools ?

Anti-DSE protections

Protections	Strength	Cost	Correctness	Stealth	Implementation	Mitigation
Complex Constraints						
MBA		?	✓	✗	✓	✓
Crypto Hash Functions	†	?	✓	✗	✗	✗
Path Divergence						
Self-modification			✓	✗	✓	✓
Symbolic Code			✓		✗	✓
Covert Channel	†		✗		✓	✗
Path Explosion						
Linear Obfuscation		†	✓		✗	✗
Path-oriented	††	††	✓		✗	✗

✗	Bad/No	?	Unknown
	Medium	†	Some experimental evaluation
	Good	††	Large experimental evaluation

Conclusion

State-of-the-art in anti-DSE protections unclear

→ We propose a classification and comparison of existing work

State-of-the-art insufficiency and call for action:

→ Many implementations not available

→ Many studies lack strong enough experimental evaluation

→ Cost and stealth are often overlooked

Questions ?