How to kill symbolic deobfuscation for free

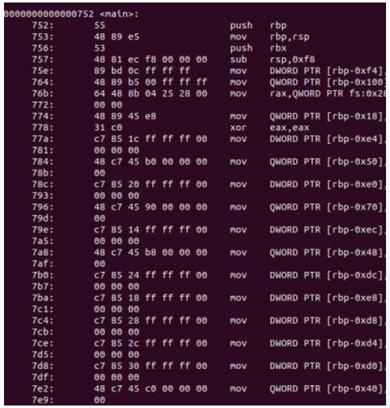
Mathilde OLLIVIER — **CEA LIST, France**

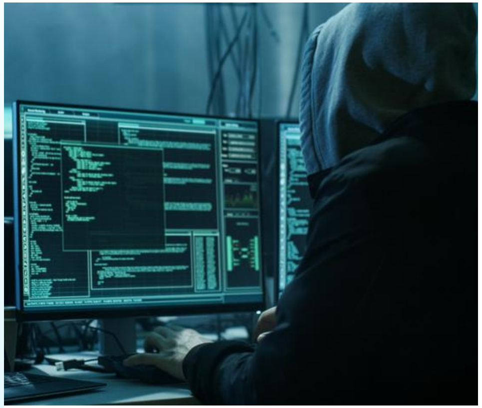
Sébastien BARDIN — CEA LIST

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

Richard BONICHON — Tweag I/O

Reverse engineering is a threat to IP



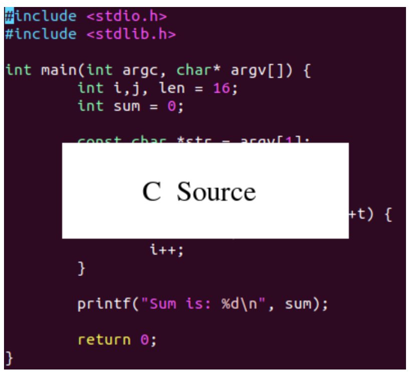


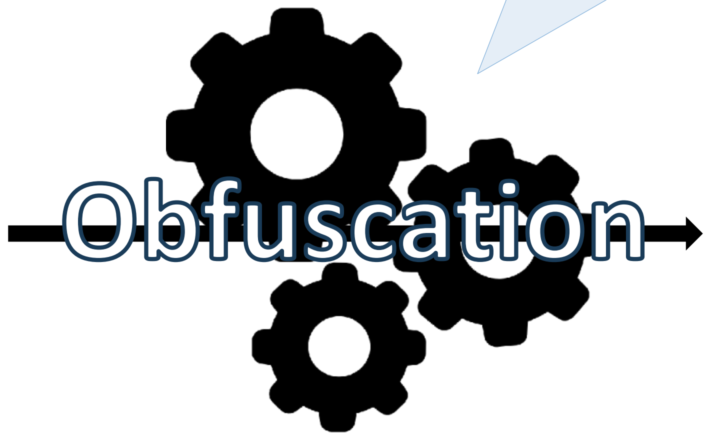
Easy with unprotected code

tools

Then we use obfuscation

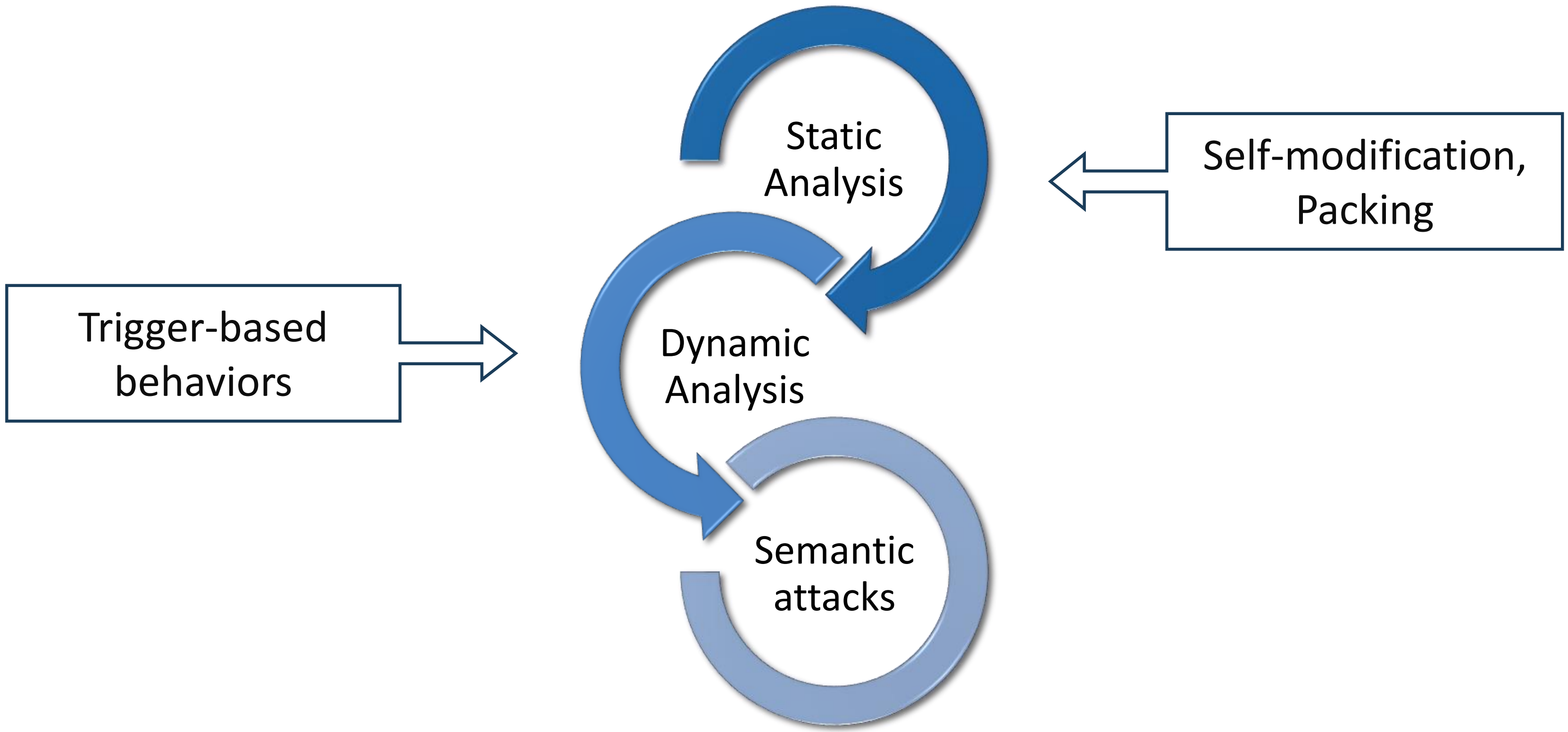
- Functional equivalence
- Efficient
- "Harder " to analyze





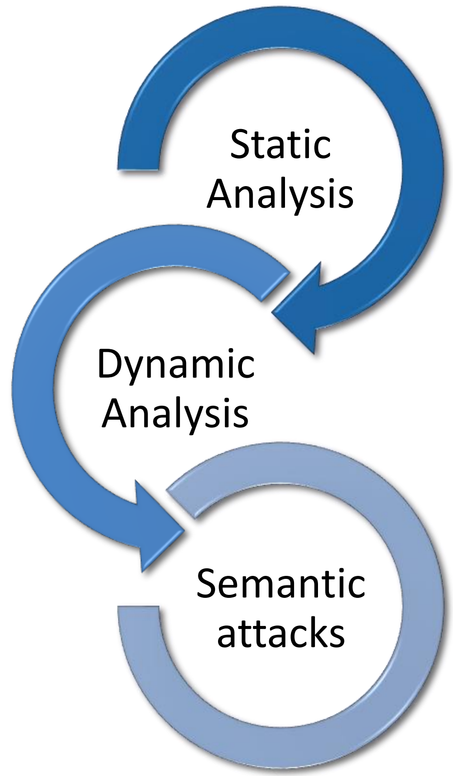
tools

Arm race


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

Arm race

Trigger-based behaviors



Self-modification, Packing

What now ?

Semantic attacks

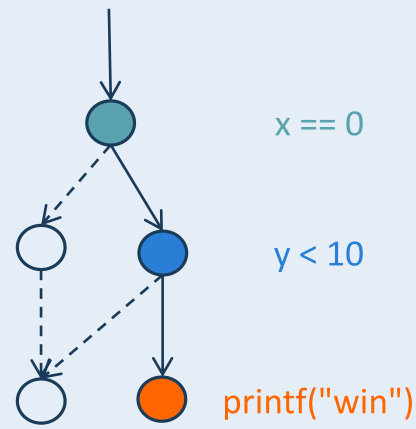
- **Dynamic Symbolic Execution** (DSE), Abstract Interpretation
- Bit-level taint analysis and DSE (Yadegari 2015)
- Backward Bounding DSE (David 2017)
- Banescu 2016

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 Λ y<10

Concrete solution using SMT solvers

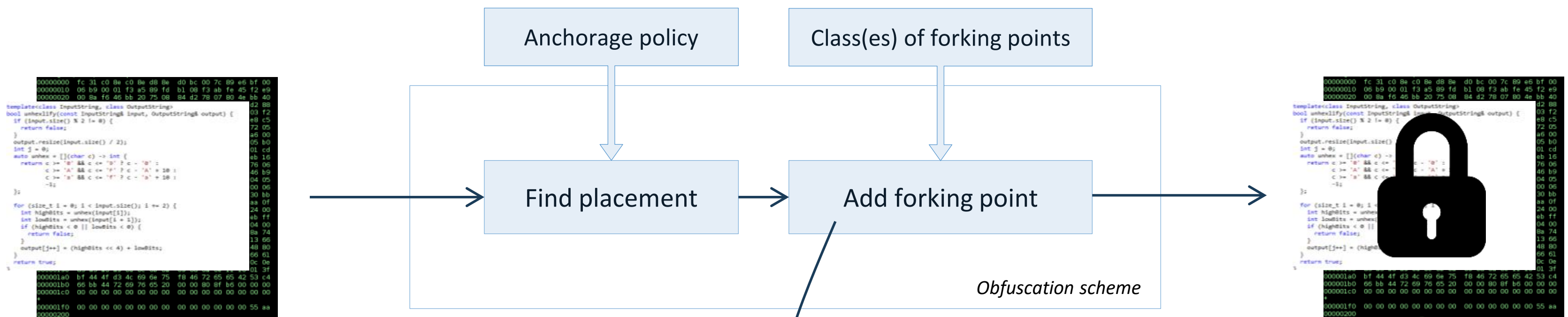
- As robust as dynamic analysis
- Infers triggers
- Simplifies code

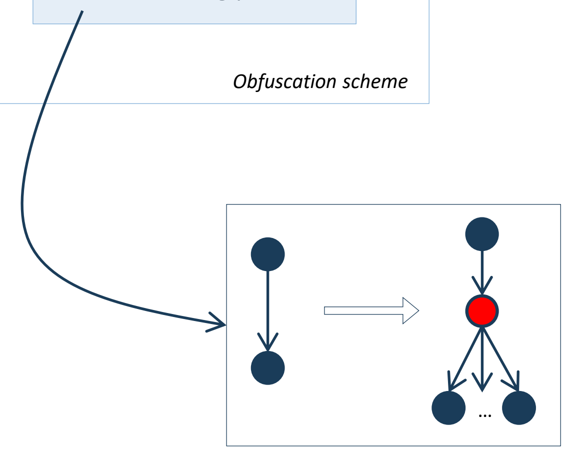
What can be done against semantic attacks ?

Prior work is not enough

Protections	Strength	Cost	Correctness	Stealth
		Standard		
Virtualization	×	~	✓	~
Virtualization ×3	~	×	✓	~
Virtualization ×5	✓	××	✓	~
		Anti DSE protections		
MBA	× / ?	?	✓	××
Cryptographic hash functions	✓✓	?	✓	××
Covert channels	✓	?	×	✓

Our proposition: path-oriented protections





Focus weakness of DSE: path explosion
→ Protection ? Cost ? Stealth ?

Forking point: *Location in code where a path split into two or more paths*

Attacker model

Men-at-the-end attacks

Access to **state-of-the-art off-the-shelf** tools

→ *No crafted dedicated tools*

Focus on **symbolic execution** and trace-based semantic attacks

We abstract two goals for attacks:

→ **Secret finding**

→ **Exhaustive path exploration**

What can PO protections really do ?

Protection	Slowdown		Runtime overhead
	Coverage	*Secret*	
Virt	××	××	×1.1 ✓
Virt ×2	×	××	×1.3 ✓
Virt ×3	✓	×	×40 ×
SPLIT (k=11)	××	××	×1.0 ✓
SPLIT (k=19)	✓	××	×1.0 ✓
FOR (k=1)	✓	×	×1.0 ✓
FOR (k=3)	✓	✓	×1.0 ✓

Banescu *et. al.* 2016

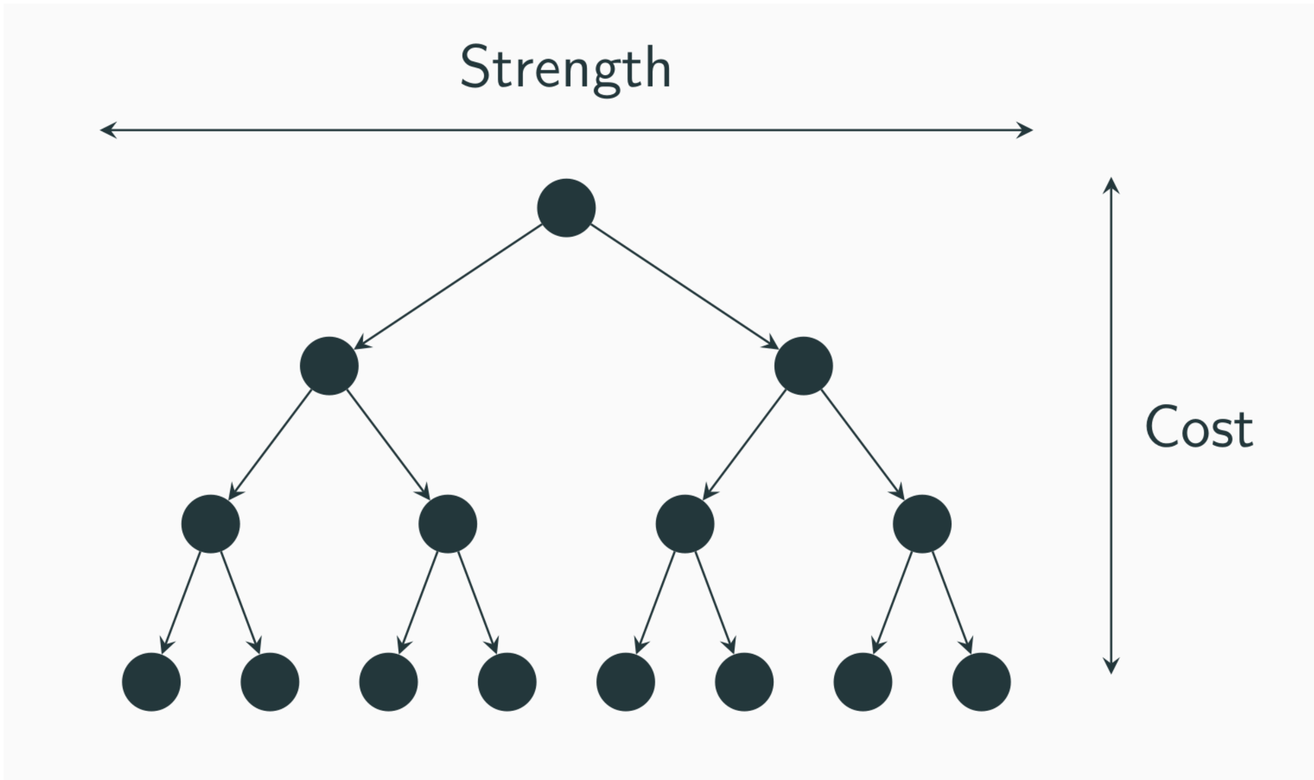
×× t≤1s   × 30s<t<5min   ✓ Time out (≥1h30)

Path-oriented protections are promising → **How do we make it work ?**

Contributions

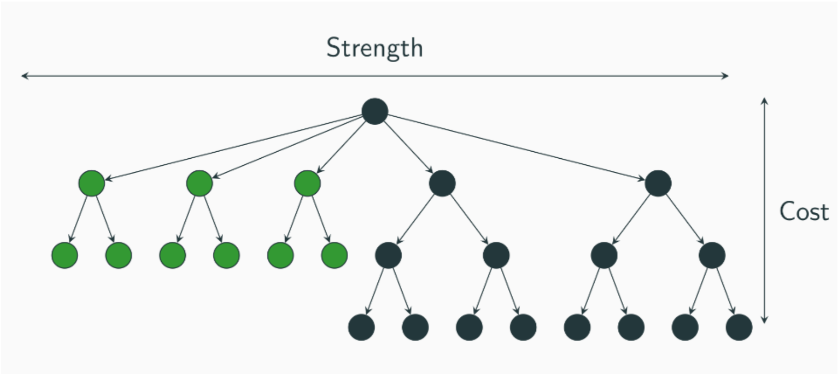
1. Formal definition and predictive characterization
   - → Encompass prior work and key notion of single value path
2. New obfuscation schemes
   - → First tractable SVP schemes
3. Optimal composition properties and resistance **by-design** to taint and slice
4. Extensive experiments
   - → Including robustness and cost

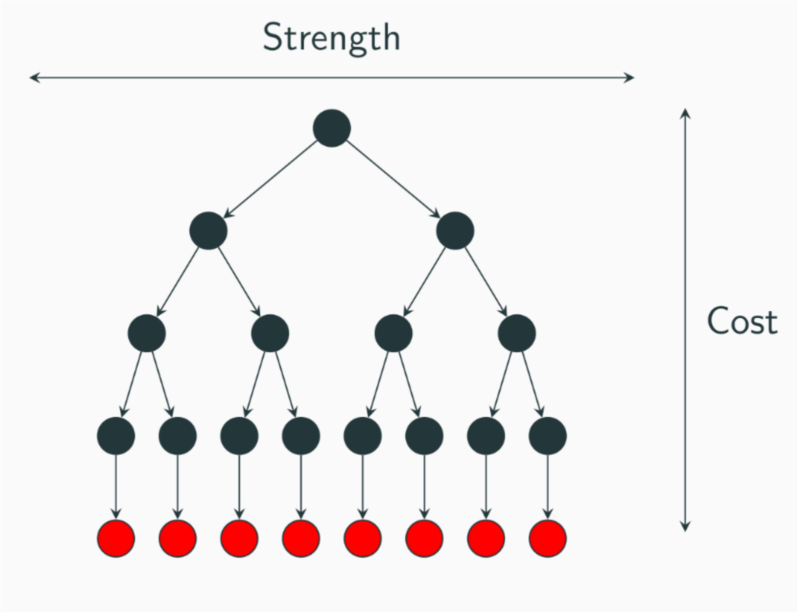
Strong and tractable



Strong and tractable

Formal definitions in paper





Single Value Path

Banescu et. al. (2016)

Range Divider - SPLIT

```
int func (char x) {
   char var = 0;
   if (x > 0) {
         var = x+10;
    else {
       //obfuscated version of "var=x+10"
   }
   return(var);
}
```

2 paths
var has **128** possible values

FOR - word

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

256 paths
var has **1** possible value

Single Value Path

Banescu et. al. (2016)

Range Divider - SPLIT

```
int func (char x) {
   char var = 0;
   if (x > 0) {
         var = x+10;
    else {
       //obfuscated version of "var=x+10"
   }
   return(var);
}
```

2 paths
var has **128** possible values

FOR - word

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

256 paths
var has **1** possible value

Protection	**Slowdown**	
	Coverage	*Secret*
SPLIT (k=11)	××	××
SPLIT (k=19)	✓	××
FOR (k=1)	✓	×
FOR (k=3)	✓	✓

×× t≤1s   × 30s<t<5min   ✓ Time out (≥1h30)

Empirically SVP protection performs better!

FOR- WORD : beware!

original code

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

obfuscated code

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

Paths: 2^{32}

Loop iterations: $\leq \mathbf{2^{32}}$

properties

Tractable (space)	✓
Tractable (time)	×
SVP	✓

FOR- BYTE

original code

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

obfuscated code

```
int func (int x) {
   char tmp[4] = (char *)(&x);
   char res[4] = 0;
   for (int i=0; i<tmp[0]; i++) {
      res[0]++;
   }
   for (i=0; i<tmp[1]; i++) {
      res[1]++;
   }
   for (i=0; i<tmp[2]; i++) {
      res[2]++;
   }
   for (i=0; i<tmp[3]; i++) {
      res[3]++;
   }
   int var = (int)(*res);
 return(var);
}
```

Paths: 2^{32}

Loop iterations: $\leq 4 \times 2^8$

properties

- Tractable (space) ✓
- Tractable (time) ✓
- SVP ✓

Obfuscation schemes

		New ?	Tractable		SVP	Stealth
			Time	Space		
Range divider	SWITCH	No	✓	✗	✓	✗
Split	IF	No	✓	✓	✗	✓
For	Word	Yes	✗	✓	✓	✓
	Byte	Yes	✓	✓	✓	✓
Write		Yes	✓	✓	✓	✗

Threats and robustness

Slice and taint attacks

→ Resistance **by-design** (more details in paper)
→ Robustness against sound attacks

Pattern attacks

→ Code diversity: several schemes
→ Algorithm diversity: several implementation for each scheme

What else ?

→ Optimal composition

Experiments

- Strength
- Cost
- Robustness

Datasets

Two datasets:

- #1: 46 small programs of Banescu *et. al.*
- #2: 7 "real world" programs

  Hash functions; DES; AES; Grub

Entry size	#LOC		KLEE exec. (s)	
	avevarge	*stdDev.*	*average*	*maximum*
1 byte	21	1.9	2.6	17.8
16 bytes	17	2.2	1.0	23.4

Dataset #1

Program	#LOC	KLEE exec. (s)
City hash	547	7.41
Fast hash	934	7.74
Spooky hash	625	7.12
MD5 hash	157	33.31
AES	571	1.42
DES	424	0.15
Grub	101	0.06

Dataset #2

Strength

Transformation	Dataset #1		Dataset #2		
	Secret Finding 1h TO	*Full Coverage 3h TO*	*Secret Finding 3h TO*	*Secret Finding 8h TO*	*Full Coverage 24h TO*
Virtualization	0 / 15	0 / 46	0 / 7	0 / 7	0 / 7
Virtualization ×2	0 / 15	1 / 46	0 / 7	0 / 7	0 / 7
Virtualization ×3	**2** / 15	**5** / 46	**0** / 7	**0** / 7	**1** / 7
SPLIT (k=10)	0 / 15	1 / 46	0 / 7	0 / 7	0 / 7
SPLIT (k=13)	1 / 15	4 / 46	1 / 7	0 / 7	1 / 7
SPLIT (k=17)	**2** / 15	**18** / 46	**2** / 7	**1** / 7	**3** / 7
FOR (k=1)	0 / 15	2 / 46	0 / 7	0 / 7	0 / 7
FOR (k=3)	8 / 15	30 / 46	2 / 7	1 / 7	3 / 7
FOR (k=5)	**15** / 15	**46** / 46	**7** / 7	**7** / 7	**7** / 7

Several **heuristics**:
- BFS
- DFS
- NURS

Several **tools**:
- KLEE
- Binsec
- Triton

Cost

Transformation	Dataset #1		Dataset #2	
	Runtime overhead	*Code size increase*	*Runtime overhead*	*Code size increase*
Virtualization	× 1.5	× 1.5	× 1.5	× 1.5
Virtualization ×2	× 15	× 2.5	× 15	× 15
Virtualization ×3	**× $1.6 \cdot 10^3$**	**× 4**	**× 1.5**	**× 1.5**
SPLIT (k=10)	× 1.2	× 1.0	× 1.0	× 1.0
SPLIT (k=50)	**× 1.2**	**× 1.0**	**× 1.0**	**× 1.0**
FOR (k=1)	× 1.0	× 1.0	× 1.0	× 1.0
FOR (k=5)	× 1.3	× 1.0	× 1.1	× 1.0
FOR (k=50)	**× 1.5**	**× 1.5**	**× 1.2**	**× 1.1**
FOR (k=1) *word*	**× $2.6 \cdot 10^3$**	**× 15**	**× $2.1 \cdot 10^3$**	**× 15**

Robustness

Tool	Robust ?		
	Basic	*Obfuscated*	*Weak*
GCC –Ofast	✓	✓	×
Clang –Ofast	×	✓	×
Frama-C Slice	✓	✓	×
Frama-C Taint	✓	✓	✓
Triton (taint)	✓	✓	✓
KLEE	✓	✓	✓

✓ no protection simplified
× ≥1 protection simplified

Conclusion

Semantic attacks are very powerful against standard obfuscations

Path-oriented protections:

→ Exploit DSE's weakness**, path explosion**
→ Completely hinders DSE
→ Very low to no performance cost
→ **Resistance by-design** to taint and slice attacks
→ Large experiments on strength, cost and robustness

We propose a **hardened benchmark** obfuscated with PO protections

Questions ?

Range divider - IF

original code

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

obfuscated code

```
int func (int x) {
   int var = 0;
   if (x > 0) {
         var = x+10;
    else {
       //obfuscated version of "var=x+10"
   }
   return(var);
}
```

properties

Tractable (space)	✓
Tractable (time)	✓
SVP	✗

Range divider - SWITCH

original code

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

obfuscated code

```
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);
}
```

properties

Tractable (space)	×
Tractable (time)	✓
SVP	✓

Write

original code

```
L:   mov    a, input
```

obfuscated code

```
L1:  mov    L2+off, input
L2:  mov    a, 0
```

Exemple for **input = 100**

Write

original code

```
L:   mov    a, input
```

obfuscated code

```
L1:  mov    L2+off, input
L2:  mov    a, 100
```

Exemple for **input = 100**

properties

- Tractable (space) ✓
- Tractable (time) ✓
- SVP ✓