VOT4CS: A Virtualization Obfuscation Tool for C#

Sebastian Banescu, Ciprian Lucaci, Benjamin Kraemer,
Alexander Pretschner

Technical University of Munich, Germany

2nd International Workshop on Software Protection
28 Oct 2016 – Vienna, Austria

Introduction

Problems for programs written in C#:

– Intellectual property (IP) theft

– Code lifting attacks

Solution:

– Use multiple obfuscation to raise the bar against attackers

  - ➢ **Problem:** No free obfuscation tool with **virtualization obfuscation** as a feature

Contributions:

– Design and implementation of virtualization obfuscator for C# programs

  – An open source alternative to commercial obfuscators

– Survey and implementation of attacks

  – Survey of popular attacks against virtualization obfuscation

  – Implemented automated dynamic analysis attack

– Evaluation based on a case-study

  – Performance

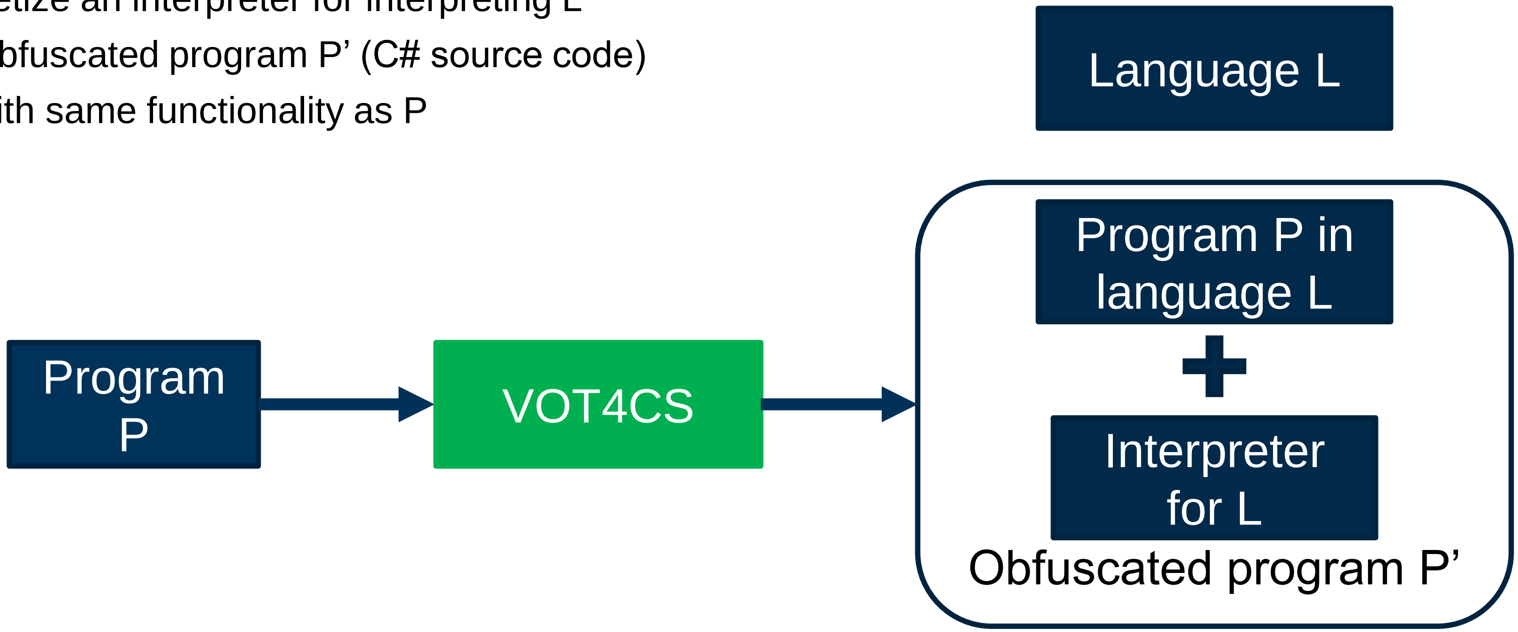
  – Security (resilience against implemented attack)

Overview of Virtualization Obfuscation

Input: Program P (C# source code)

1. Generate a random new language L
2. Translate P to the new language
3. Synthetize an interpreter for interpreting L

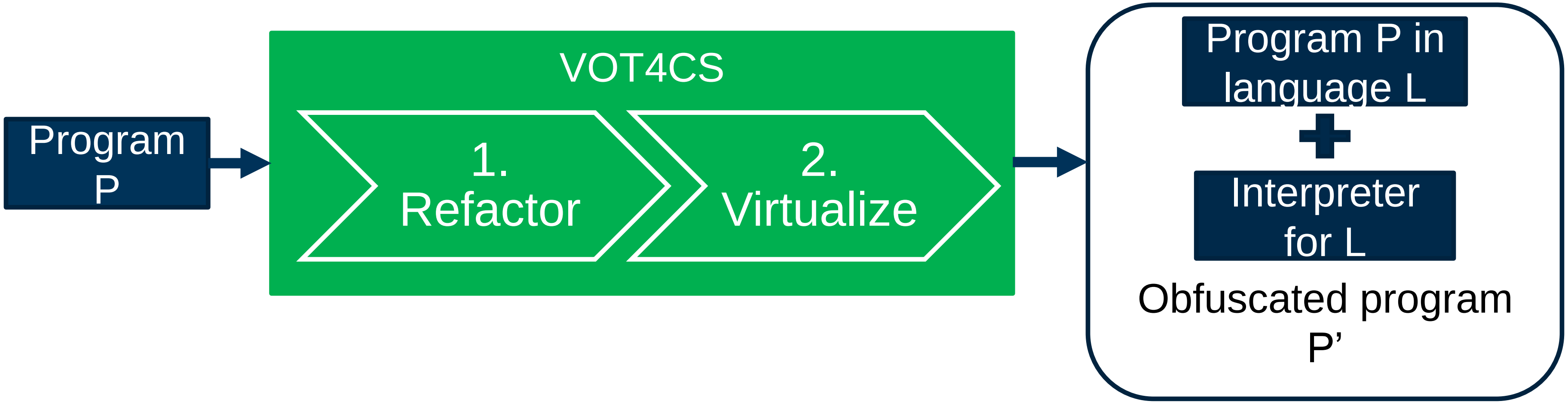
Output: Obfuscated program P' (C# source code) with same functionality as P



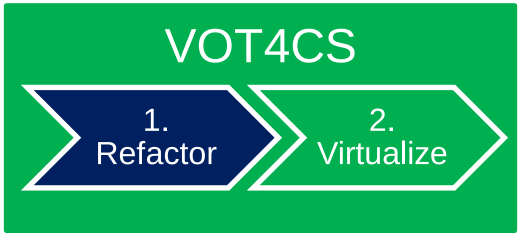
Design and Implementation

VOT4CS consists of**:**

- Refactoring Phase (bring code in canonical form)
- Virtualization Phase (code translation and program generation)



Refactoring Phase



1. Refactoring if-statements and switch statements

```
if (a > b)
```
→
```
cond = a > b;
if (cond)
```

2. Refactoring loops: same as if-statements + jump back if **`cond`** is TRUE

3. Refactoring unary and binary operators

```
a += b;
```
→
```
a = a + b;
```

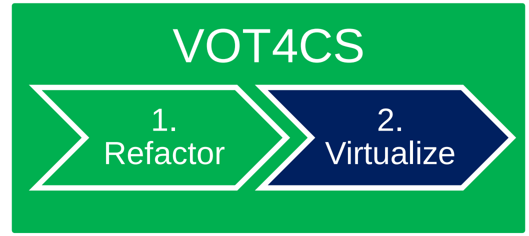
4. Refactoring statements with multiple operands (tunable)

```
a = b + c + d;
```
→
```
tmp = b + c;
a = tmp + d;
```

5. Refactoring statements with multiple method invocations (tunable)

```
a.b().c()
```
→
```
x = a.b();
x.c();
```

Virtualization Phase



1. Map all data items to **`data`** array
   - variables (uninitialized → initialize w. random value)
   - constants (shared)
   - method parameters
2. Map all instructions to **`code`** array
   - generate new random ISA
   - translate each statement in code to new ISA
   - inject random values in code array between: instructions, opcodes & operands
3. Create interpreter

```
void obfuscated_method() {
  object[] data = … //variables, constants
  int[] code = … //bytecode
  int vpc = 0; //virtual program counter
  while (true) {//interpreter
    switch (code[vpc]) {
    case 1023: // assignment opcode
      data[code[vpc + 2]] = data[code[vpc + 3]];
```

Raising the Bar for Attackers

- Software diversification options
  - un-initialized variables are initialized with random values
  - random junk inserted in **`code`** array
  - order of opcode and operands
  - size of each instruction

- Most frequent opcode is assigned to **`default`** branch of **`switch`**
  - opcode value in code array is replaced with random (non-opcode) values
  - harder to identify this instruction in the **`code`** array

- Interpreter level
  - method level
  - class level (multiple methods share the same interpreter)

MATE Attacks on Virtualization Obfuscation

	Authors	Attack Type	Attacker Goal	Drawbacks
1	[Rolles2009]	**manual static analysis**	**extract original code**	time consuming not scalable
2	[Sharif2009]	automated static & dynamic analysis	control flow graph	strong assumptions on interpreter structure
3	[Coogan2011]	**automated dynamic analysis**	approximation of original code **significant trace**	difficult to process large traces
4	[Kinder2012]	automated static analysis	approximated data values	strong assumptions on interpreter structure
5	[Yadegari2015]	**automated dynamic analysis**	**control flow graph**	large input space leads to many traces

Implemented Attack

Assumption: attacker fully aware of VOT4CS design & implementation

1. Trace the program at CIL level
   - implemented CIL traced by instrumenting .NET assembly
   - logs value of current opcode and VPC value

2. Simplify the trace
   - filter out instructions belonging to **`switch`** or **`if-else-if`** statements
   - filter out instructions that increment the VPC
   - replace instructions accessing data array, with variable accesses (new variable names based on index in data array)

Attack Example

Obfuscated CIL

```
#1#0##7697
96#IL_01BD# ldarg data
97#IL_01C1# ldarg code
98#IL_01C5# ldarg vpc
99#IL_01C9# ldc.i4 -17
100#IL_01CE# add
101#IL_01CF# ldelem.i4
102#IL_01D0# ldarg data
103#IL_01D4# ldarg code
104#IL_01D8# ldarg vpc
105#IL_01DC# ldc.i4 14
106#IL_01E1# add
107#IL_01E2# ldelem.i4
108#IL_01E3# ldelem.ref
109#IL_01E4# castclass System.String
110#IL_01E9# ldarg data
111#IL_01ED# ldarg code
112#IL_01F1# ldarg vpc
113#IL_01F5# ldc.i4 3
114#IL_01FA# add
115#IL_01FB# ldelem.i4
116#IL_01FC# ldelem.ref
117#IL_01FD# unbox.any System.Int32
118#IL_0202# box System.Int32
119#IL_0207# call System.String
System.String::Concat(System.Object,System.Object)
120#IL_020C# stelem.ref
```

Original C#

```
string sum = "" + 3 + 4 + "";
```

Simplified CIL

```
ldloc var_3410
ldloc var_3245
call System.String System.String::Concat(System.Object,Syste
stloc var_2165
```

Evaluation: Resilience Against Attack

Original Code:

```
private string f(int b) {
  string sum = "" + 3 + 4 + "";
  sum += car.GetEngine().GetPiston(car.GetEngine().GetPistons().Count - 1).ToString();
  string r = "";
  string[] dst = new string[b];
  for (int i = 0; i < b; i++) { // b iterations
    sum += "_" + i + "_";
    sum += "~";
    r += sum + "#";
    var p1 = car.GetEngine().GetPistons().First().GetSize();
    r += "[" + p1 + "]";
    sum += r.Length;
    dst[i] = sum;
  }
  sum += "#" + dst.Length;
  return sum;
}
```

Evaluation: Resilience Against Attack

1. Obfuscated toy program with various configurations of VOT4CS
2. Recorded traces using our CIL tracer tool
3. Simplified traces

		Original	Refactored-only					Refactored & Virtualized				
VOT4CS settings		none	op2 in1	op3 in1	op4 in1	op4 in4	op4 in5	op2 in1	op3 in1	op4 in1	op4 in4	op4 in5
Recorded trace	5	289	284	259	325	313	313	2498	2030	2092	1824	1687
	25	1149	1104	999	1285	1233	1233	9538	7450	7972	7224	6587
	125	5449	5204	4699	6085	5833	5833	44738	34550	37372	34224	31087
Simplified trace	5	280	275	250	316	304	304	295	234	331	307	289
	25	1120	1075	970	1256	1204	1204	1095	814	1231	1167	1069
	125	5320	5075	4570	5956	5704	5704	5095	3714	5731	5467	4969

Observation: Some simplified traces shorter than original → missing instructions

Levenshtein Distance

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j) + 1 \\ \operatorname{lev}_{a,b}(i,j-1) + 1 \\ \operatorname{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

Source: Wikipedia

Example:

$a =$ "potato"

$b =$ "tomato"

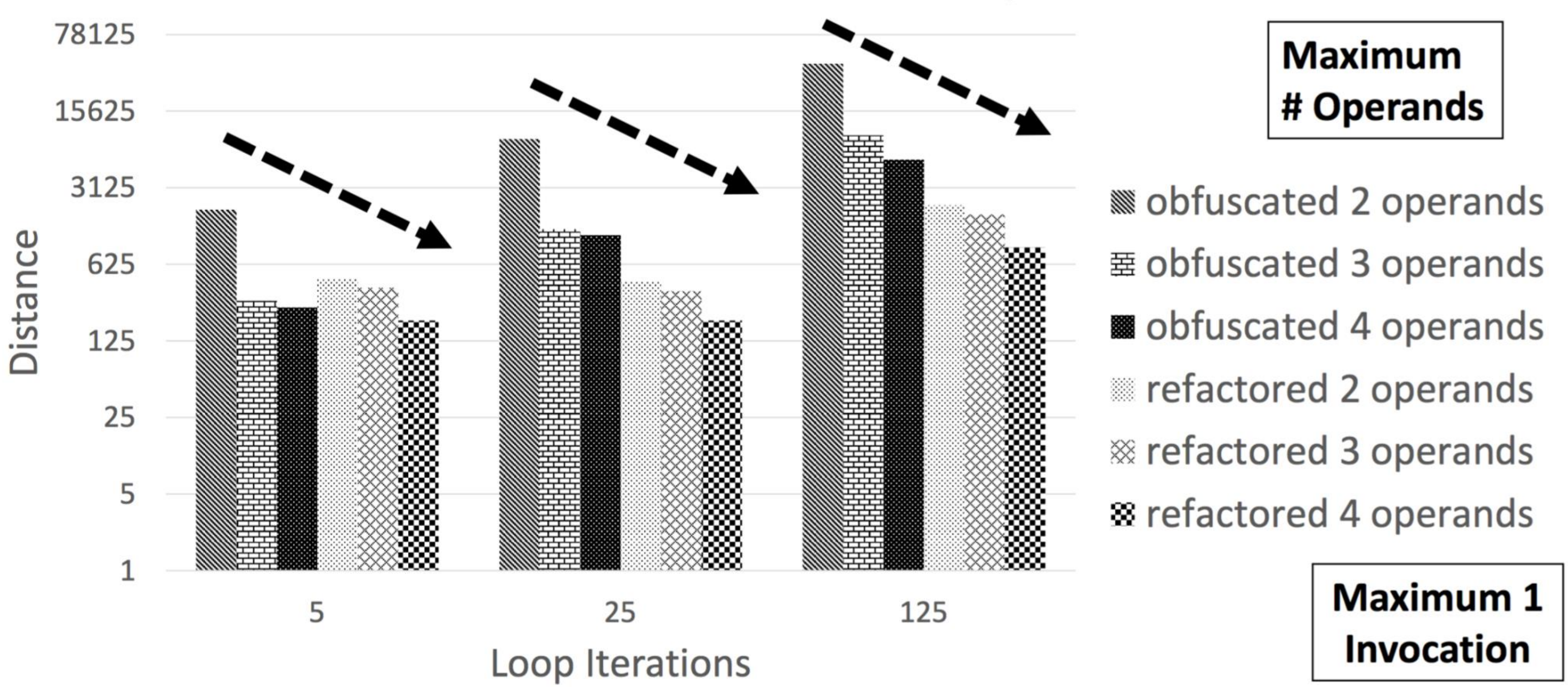
$\operatorname{lev}_{a,b}(|a|, |b|) = 2$

How to Compute Levenshtein Distance on Traces

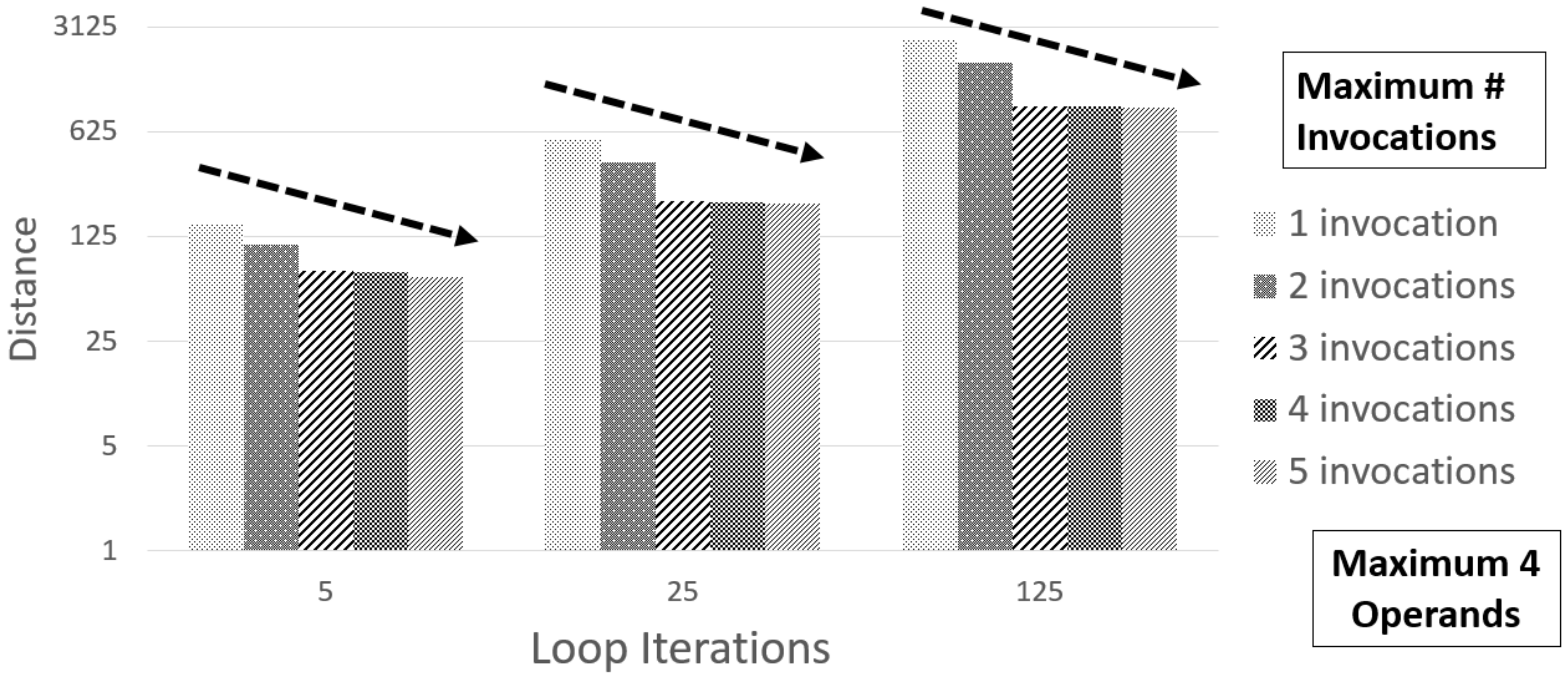
- Problems comparing original & obfuscated traces:
  - Different variable, argument and constant names
  - Functions have a different number of arguments (due to refactoring)

- Solution: **Abstract traces**
  - Loading a variable, argument or constant considered the same
  - Storing a variable, argument or constant considered the same
  - Only function names are compared (not arguments)

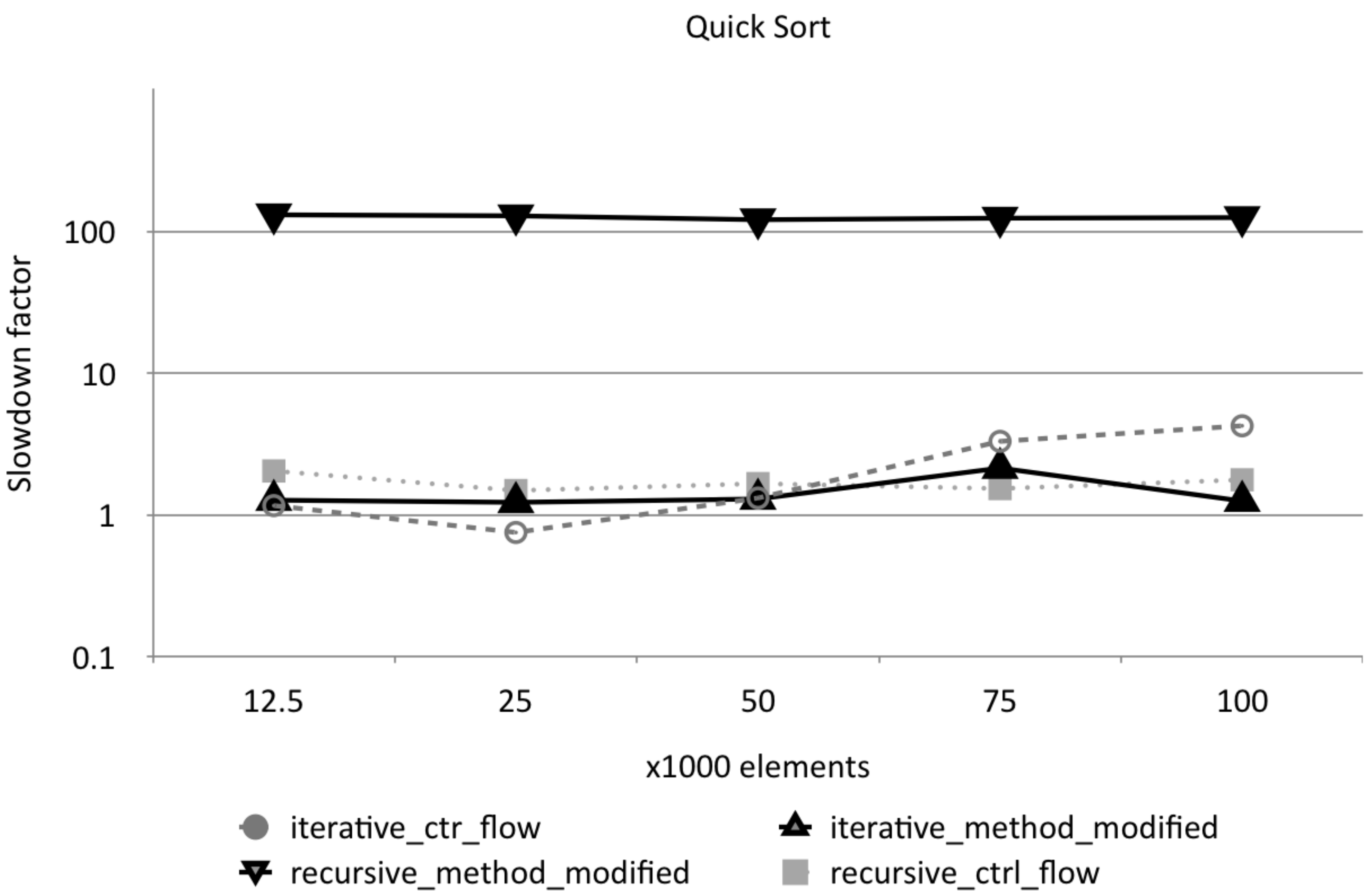
Evaluation: Resilience Against Attack (#operands)



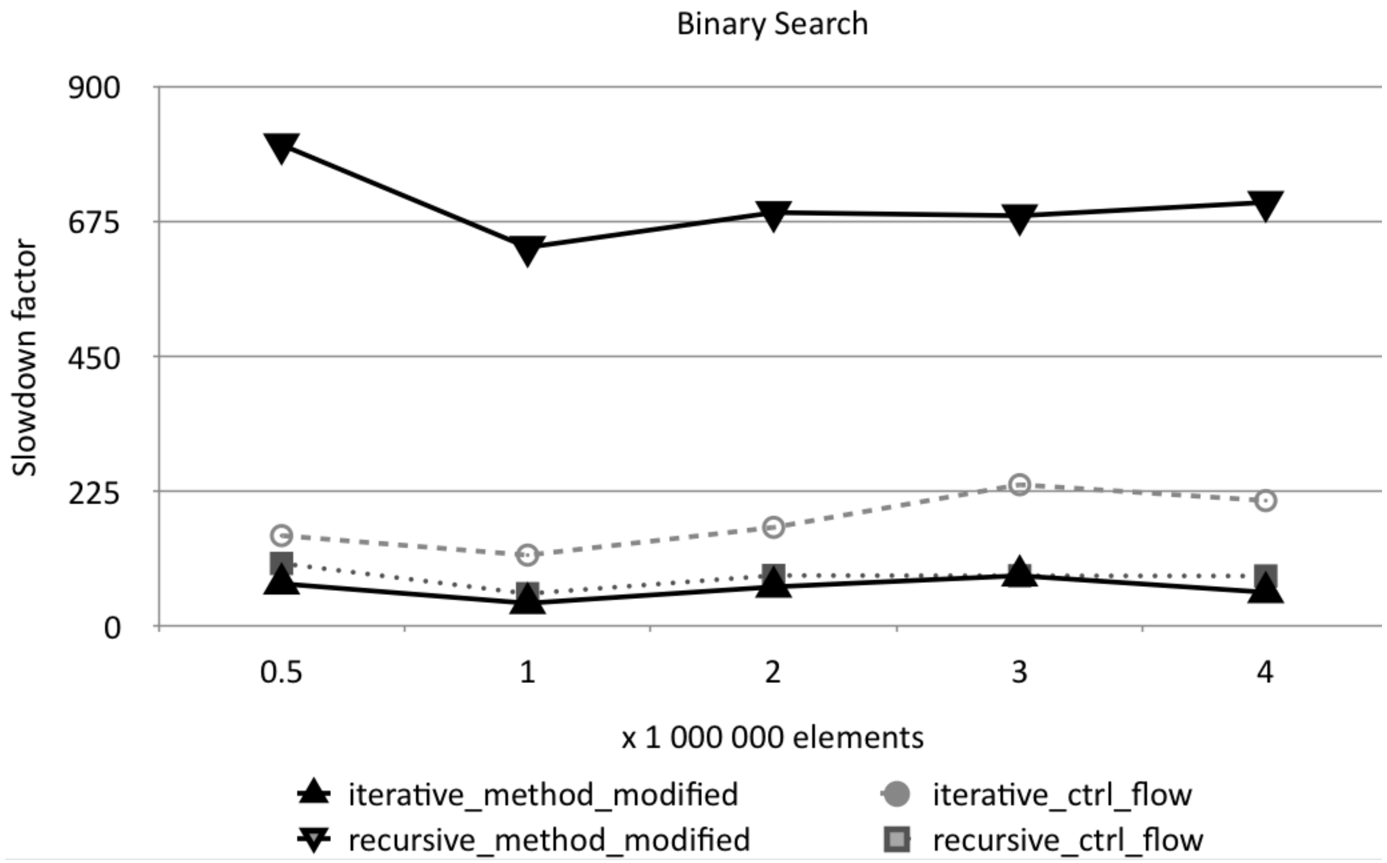
Evaluation: Resilience Against Attack (#invocations)



Evaluation: Run-time overhead (Quick Sort)



Evaluation: Run-time overhead (Binary Search)



Evaluation: File Size

- Small increase (e.g. <3%) for large programs or few functions
- Large increase (e.g. >90%) for small programs or many functions

	Original	Obfuscated	Relative Increase
Quick Sort	479,232	491,520	2.57%
Binary Search	483,328	491,520	1.70%
ResourceLib	75,264	145,920	93.87%

Conclusions

- Design and implementation of VOT4CS
- Implementation of CIL tracer and dynamic analysis attack
- Security evaluation of VOT4CS resilience against attack
  - Measured security using Levenshtein (edit) distance
  - Lower number of operands gives more security
  - Lower number of invocations gives more security
- Performance evaluation of VOT4CS
  - Iterative methods overall faster in VOT4CS than ConfuserEx CFO
  - Recursive methods much slower in VOT4CS than ConfuserEx CFO
- Future work
  - Add more features to VOT4CS
  - Automated equivalence checking of VOT4CS input and output

Thank you for your attention !

Questions?

Source code: https://github.com/tum-i22/vot4cs

Contact:

Sebastian Banescu

banescu@in.tum.de