

# Ассамблея

Мультиязычный обфускатор программ и компилируемая библиотека непрозрачных предикатов

---

Антон Баглий, Борис Штейнберг, Елена Алымова, Константин Гуфан

4 апреля 2017 г.

ЮФУ, ФГАНУ Спецвузавтоматика

# Первоначальные цели и ограничения

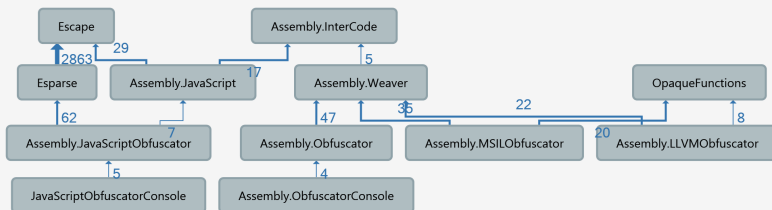
- **Предназначение:** Пооператорное запутывание байт-кода MSIL, LLVM и исходного текста JavaScript
- **Функции:**
  - Описание преобразований операторов общее для всех входных языков
  - Поиск подходящих операторов и применение преобразований к ним
  - Вставка вызовов или inlining непрозрачных функций, указанных в описании преобразований
- **Ограничения:**
  - Обработка каждой инструкции (оператора) отдельно
  - Одно представление для всех входных языков

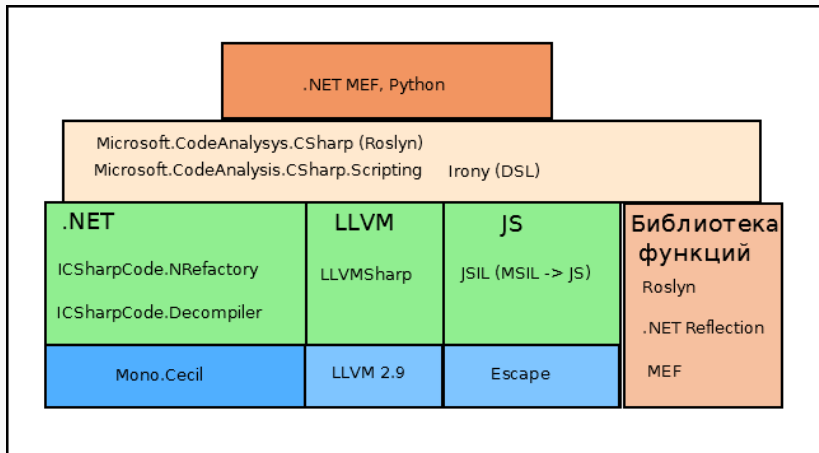
- Сложно создать новое универсальное представление для очень разных входных языков
- Структура исходной программы должна представляться одинаково для всех языков

## решения

- Используется готовое AST с аннотациями произвольного типа, в которые помещаются части исходной программы и другая информация (например, типы)
- Структура графа потока управления с вершинами-простыми блоками подходит больше всего

- Описание преобразований - InterCode
- Применение преобразований к арифметическим операторам в MSIL, LLVM, JavaScript
- Проверка типов при применении преобразований в MSIL и LLVM
- Вставка непрозрачных функций, если они указаны в преобразовании
- Библиотека непрозрачных функций





- Исходная программа обрабатывается в несколько проходов
- В каждом проходе программа разбивается на блоки операторов, например на простые блоки-вершины управляющего графа.
- Посещаются операторы каждого блока, проверяется применимость преобразований к ним.
- Для операторов, к которым применимы преобразования, они применяются.
- Записывается полученный байт-код или исходный код модифицированной программы.

## пример

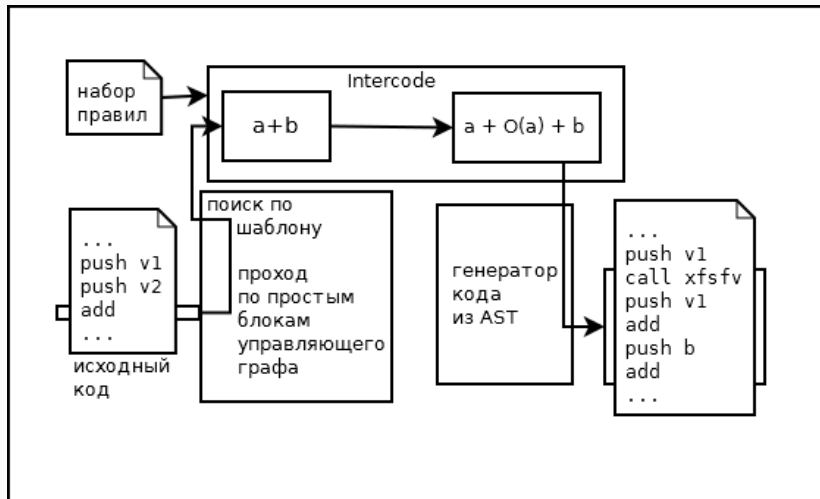
```
SumTransform: a + b => a + 1 + b - 1 + c { var c =  
FormIntOpaqueValue(0);}
```

здесь

- SumTransform, DivTransform - имена преобразований
- выражение1 => выражение2 означает замену оператора (или его части) эквивалентного левой части на правую
- текст в скобках { } задает значения новых переменных, в примере это вызов непрозрачной функции



# Применение правил



## Пример запуска

```
o = PyAssembly.Manager(..., "LLVM")
p = o.GetPass("all_cfg")
o.AddParameter("useCodeGenerator", "true")
o.AddParameter("constant1", "Opaque1LoopInt")
p.CreateRule(name = "1", fromExpr = "a+b",
to = "a+b", action = "␣")
p.CreateRule(name = "2", fromExpr = "a-b",
to = "a-b+1-1", action = "␣")
p.CreateRule(name = "3", fromExpr = "a*b",
to = "a*b*c",
action = "var␣c=FormIntOpaqueValue(1);␣")
o.AddPass(cfgPass)
o.Open(path); o.Run()
o.Save(path)
```

```
c = System.Func[IAttributable, bool](  
    lambda function: function.HasAttribute("ObfuscateThis"))  
extraConditions = List[System.Object]()  
extraConditions.Add(extraCondition)  
p.CreateRule("1", "a+b", "a+b+0", "", extraConditions)  
p.CreateRule("2", "a-b", "a-b+0", "", extraConditions)  
p.CreateRule("3", "a*b", "a*b+0", "", extraConditions)  
o.AddPass(cfgPass)
```

Это применит замены только к функциям  
с атрибутом [ObfuscateThis]

пример

SumTransform:  $a + b \Rightarrow a + 1 + b - 1$  { int a; int b;}

- можно указать типы для всех участвующих переменных
- типы проверяются без неявных преобразований
- расширить проверки типов достаточно просто

## пример

```
1: _FunctionStart=>hub{var hub=AddControlHub();}  
2: _Break=>entry{var entry=AddHubEntryAfter(current);}  
3: _FunctionEnd=>nop{var nop=FinilizeHub();}
```

- можно перенаправлять операторы ветвлений
- нужны проверки корректности
- не достаточно удобно добавлять правила

# Пример MSIL

правило:  $a-b \Rightarrow a - 24 - b + 24$

sub =>

результат

```
stloc a_backup  
ldloc a_backup  
ldc.i4 24  
sub  
ldloc b_backup  
sub  
ldc.i4 24  
add
```

# Пример вставки функций MSIL

правило:  $a+b \Rightarrow a + 3 + b - 3 + f()$ ,  $f = \sin(\pi/3) + \sin(-\pi/3)$

результат

add  
=>

stloc a_backup	ldc.r8 1,0471975511966
ldloc a_backup	call Double Sin(Double)
ldc.i4 3	ldc.r8 -1,0471975511966
add	call Double Sin(Double)
ldloc b_backup	add
add	stloc V_22
ldc.i4 3	ldloc V_22
sub	add

# Пример LLVM

правило:  $a * b \Rightarrow a * b * 3 / 3$

```
%tmp2 = mul  
i32 %1,  
%tmp1
```

результат

```
%tmp0012 = alloca i32  
store i32 3, i32* %tmp0012  
%tmp0023 = load i32, i32* %tmp0012  
%tmp0034 = mul i32 %tmp0023, %tmp0001  
%tmp0045 = alloca i32  
store i32 3, i32* %tmp0045  
%tmp0056 = load i32, i32* %tmp0045  
%tmp0067 = sdiv i32 %tmp0056, %tmp0034
```



# Пример вставки функций LLVM

правило:  $a * b \Rightarrow a * b * c, c = \text{Opaque1LoopInt}()$

результат

```
%tmp2 = mul  
i32 %1,  
%tmp1
```

```
%tmp1 = mul i32 %tmp1, %1  
%tmp2 = call i32 @Opaque1LoopInt()  
%tmp3 = mul i32 %tmp2, %tmp1  
%tmp4 = mul i32 %1, %tmp0  
...  
define i32 @Opaque1LoopInt() {  
  "3":  
    %loc0 = alloca double  
    ...  
  "4": ...
```

## Пример JavaScript

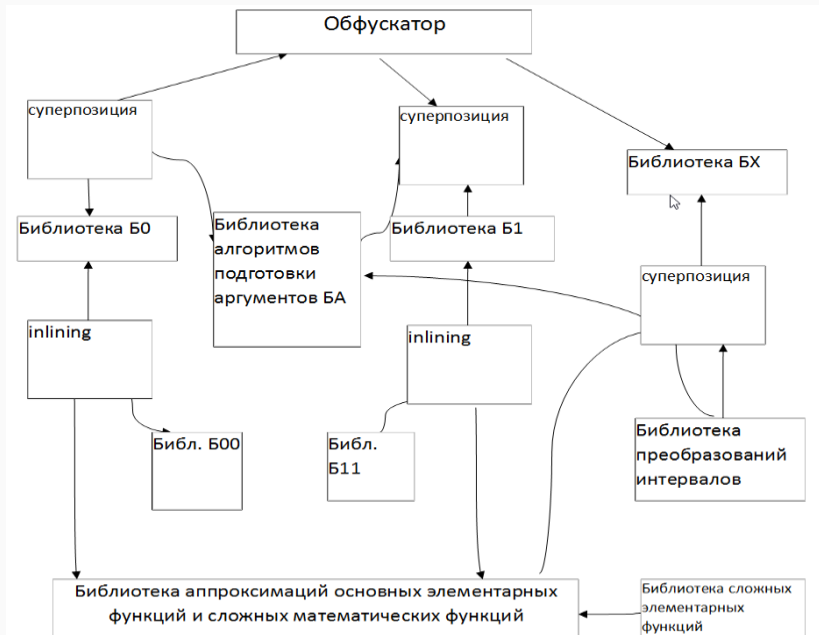
### ВХОД

```
var res1 = (a - b) | 0; // result = 3  
var res2 = (a * b) | 0; // result = 18  
var res3 = (a / b) | 0; // result = 2  
var res4 = (((a + b) | 0) - 1) | 0;
```

### результат

```
var res0 = (((((a + 2) + b) - 2)) | 0;  
var res1 = (((((a - 42) - b) + 42)) | 0;  
var res2 = (((((a * b) * 2) / 2)) | 0;  
var res3 = (((a * 1) / b)) | 0; // result = 2  
var res4 = (((((((a + 3) + b) - 3)) | 0) - 1) | 0;
```

# Структура библиотеки непрозрачных функций



## Пример простой функции

$$\ln(1+x) = \frac{2x}{2+x} - \frac{x^2}{3(2+x)} - \frac{4x^2}{5(2+x)} - \dots$$
$$\dots - \frac{m^2 x^2}{(2m+1)(2+x)} - \dots;$$

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{4x^3}{3(4+3x)} - \frac{27x^3}{5(12+7x)} - \dots$$
$$\dots - \frac{(m^2-1)^3 x^2}{(2m+1)(2m(m+1) + (m^2+m+1)x)} - \dots;$$

$$\ln(1+x) = x - \frac{3x^2}{6+4x} - \frac{10x^2}{30+16x} - \frac{126x^2}{70+36x} - \dots$$
$$\dots - \frac{m(m-1)(2m-3)(2m+1)x^2}{2(4m^2-1)+4m^2x} - \dots;$$

# Рабочий пример MSIL - ГОСТ Р 34.12-2015.

```
IL_0038: nop
IL_0039: ldloc.s V_4
IL_003b: ldc.i4.s 24
IL_003d: blt.s IL_0047

IL_003f: ldc.i4.7
IL_0040: ldloc.s V_4
IL_0042: ldc.i4.8
IL_0043: rem
IL_0044: sub
IL_0045: br.s IL_004b

IL_0047: ldloc.s V_4
IL_0049: ldc.i4.8
IL_004a: rem
IL_004b: stloc.s V_5
IL_004d: ldloc.2
IL_004e: ldarg.0
IL_004f: ldloc.1
IL_0050: ldarg.0
IL_0051: ldfld uint32[] GostPlugin.Magma::_subKeys
IL_0056: ldloc.s V_5
IL_0058: ldelem.u4
IL_0059: call instance uint32 GostPlugin.Magma::funcG(uint32,
```

uint32)

```
IL_005e: xor
IL_005f: stloc.s V_6
IL_0061: ldloc.1
IL_0062: stloc.s V_6
```

```
IL_0038: nop
IL_0039: ldloc.s V_4
IL_003b: ldc.i4.s 24
IL_003d: blt.s IL_005d

IL_003f: ldc.i4.7
IL_0040: ldloc.s V_4
IL_0042: ldc.i4.8
IL_0043: rem
IL_0044: stloc V_11
IL_0048: stloc V_12
IL_004c: ldloc V_12
IL_0050: ldloc V_11
IL_0054: sub
IL_0055: ldc.i4 0x0
IL_005a: add
IL_005b: br.s IL_0061
```

```
IL_005d: ldloc.s V_4
IL_005f: ldc.i4.8
IL_0060: rem
IL_0061: stloc.s V_5
IL_0063: ldloc.2
IL_0064: ldarg.0
IL_0065: ldloc.1
IL_0066: ldarg.0
IL_0067: ldfld uint32[] GostPlugin.Magma::_subKeys
IL_006c: ldloc.s V_5
IL_006e: ldelem.u4
IL_006f: call instance uint32 GostPlugin.Magma::funcG(uint32,
```

uint32)

```
IL_0074: xor
IL_0075: stloc.s V_6
IL_0077: ldloc.1
IL_0078: stloc.s V_6
```

сделано замен: 39, всего инструкций 1235

# Рабочий пример LLVM - ГОСТ Р 34.12-2015.

```
%mul16 = mul nsw i32 %conv15, 2
%add = add nsw i32 %mul16, 1
%mul17 = mul nsw i32 %add, 16
%add.ptr18 = getelementptr inbounds i8, i8* %18, i32 %mul17
```

```
call void @llvm.memcpy.p0i8.p0i8.i32(i8* %arraydecay14, i8* %add.ptr18, i32 16, i32 1, i8* %i, align 1)
br label %for.cond19
```

```
for.cond19:                                ; preds = %for.inc, %for.body
    %20 = load i8, i8* %i, align 1
    %conv20 = zext i8 %20 to i32
    %mul21 = mul nsw i32 %conv20, 8
    br i1 %cmp21, label %for.body23, label %for.end
```

```
for.body23:                                ; preds = %for.cond19
    %21 = load void (i8*, i8*, i32)*, void (i8*, i8*, i32)** @print.addr, align 4
    %arraydecay24 = getelementptr inbounds [16 x i8], [16 x i8]* %C, i32 0, i32 0
    %22 = load i8, i8* %j, align 1
    %conv25 = zext i8 %22 to i32
```

```
%mul26 = mul nsw i32 %conv25, 8
%23 = load i8, i8* %i, align 1
%conv27 = zext i8 %23 to i32
```

```
%tempVar0027 = load i32, i32* %tempVar0016
%tempVar0038 = mul i32 %tempVar0027, %tempVar0005
%mul16 = mul nsw i32 %conv15, 2
%tempVar0009 = add i32 1, %mul16
%add = add nsw i32 %tempVar0038, 1
%tempVar00010 = mul i32 16, %add
%tempVar00111 = alloca i32
store i32 1, i32* %tempVar00111
%tempVar00212 = load i32, i32* %tempVar00111
%tempVar00313 = mul i32 %tempVar00212, %tempVar00010
%mul17 = mul nsw i32 %tempVar0009, 16
%add.ptr18 = getelementptr inbounds i8, i8* %18, i32 %tempVar00313
```

```
call void @llvm.memcpy.p0i8.p0i8.i32(i8* %arraydecay14, i8* %add.ptr18, i32 16, i32 1, i8* %i, align 1)
br label %for.cond19
```

```
for.cond19:                                ; preds = %for.inc, %for.body
    %20 = load i8, i8* %i, align 1
    %conv20 = zext i8 %20 to i32
    %mul21 = mul nsw i32 %conv20, 8
    br i1 %cmp21, label %for.body23, label %for.end
```

```
for.body23:                                ; preds = %for.cond19
    %21 = load void (i8*, i8*, i32)*, void (i8*, i8*, i32)** @print.addr, align 4
    %arraydecay24 = getelementptr inbounds [16 x i8], [16 x i8]* %C, i32 0, i32 0
    %22 = load i8, i8* %j, align 1
    %conv25 = zext i8 %22 to i32
    %tempVar00014 = mul i32 8, %conv25
    %tempVar00115 = alloca i32
    store i32 1, i32* %tempVar00115
    %tempVar00216 = load i32, i32* %tempVar00115
    %tempVar00317 = mul i32 %tempVar00216, %tempVar00014
    %mul26 = mul nsw i32 %conv25, 8
    %23 = load i8, i8* %i, align 1
    %conv27 = zext i8 %23 to i32
```

Стр: 1226 Пото: 37/80 Зв: 37/80

1251

Unix

Стр: 999 Пото: 1/45 Зв: 1/45

1251

Unix

сделано замен: 98, всего инструкций: 5103

## Другие примеры и большие сборки.

1. Linpack и Whetstone для .NET (сборка 20-30кб) сделано замен: 252, всего инструкций: 2628, время работы около минуты
2. MathNet.Numerics (сборка 1.5мб) время работы больше часа, потребляет больше 10Гб памяти...

# Нерешенные проблемы

1. обход управляющего графа в JavaScript и использование общего интерфейса
2. вывод типов в JavaScript - сложная задача в общем случае
3. оптимизация времени работы обфускатора MSIL (использование "тяжелых" инструментов)
4. неоптимальная организация обфускатора LLVM из-за использования C-интерфейсов
5. как не замедлить программу модификациями? Пример для ГОСТ с MSIL может легко замедлить его в 5-10 раз из-за вставки кода во вложенные циклы.
6. разработка эквивалентных преобразований



1. семейство обфускаторов, объединенных общим подходом (описанием преобразований, алгоритмом их выбора)
  - 1.1 улучшенная работа с динамическими языками за счет использования уже имеющихся инструментов
  - 1.2 сложнее обеспечить какой-то общий способ модификации потока управления
2. многоязыковая среда с обработкой нескольких языков через обобщенный метаязык - сильнее ограничивает работу с каждым из языков и сложнее в реализации