# Obfuscation

## Presentation

Adrien Guinet, Pierrick Brunet, Juan Manuel Martinez and Béatrice Creusillet,
Serge Guelton

# Table of Contents

# Table of Contents

# What is obfuscation?

## What's to protect?

- Code and data of an application
- Especially secrets within a compiled binary (*disruptive* algorithms, key materials...)

## Attack model

- The attacker has **full read/write access** to the binary
- The attacker has full control over the operating system and the hardware where the binary is running
- The application runs with the less possible privileges, and in *user-land* (on systems where it makes sens (e.g.: any modern x86 OS))

The worst situation possible: the attacker has full control over the hardware, kernel and application

# Principles of obfuscation

## **Goals** of obfuscations

- **Protect** data/code from being recovered/tampered with, in the described attack model
- At **reasonable cost** (performance/memory) for the **defender**

## What (we hope) to gain

## What we (probably) pay

# Principles of obfuscation

## **Goals** of obfuscations

### What (we hope) to gain

- **Slow down** reverse engineering (have it cost a lot)
- **Protect** intellectual properties (code, algorithms, protocols...)
- **Protect** data (secret keys, constants...)

### What we (probably) pay

# Principles of obfuscation

## **Goals** of obfuscations

### What (we hope) to gain

- **Slow down** reverse engineering (have it cost a lot)
- **Protect** intellectual properties (code, algorithms, protocols. . . )
- **Protect** data (secret keys, constants. . . )

### What we (probably) pay

- Slower execution
- Bigger binary
- Biggest memory consumption

# Principles of obfuscation

## **Goals** of obfuscations

### What (we hope) to gain

- **Slow down** reverse engineering (have it cost a lot)
- **Protect** intellectual properties (code, algorithms, protocols. . . )
- **Protect** data (secret keys, constants. . . )

### What we (probably) pay

- Slower execution
- Bigger binary
- Biggest memory consumption

Make the **attacker pay much more** than the defender!

# Table of Contents

# Concrete example (1)

## Protocol protection

### Skype

- On-the-fly code decryption
- Anti-debug
- Integrity protection
- *Time checking*
- Obfuscations: *junk code*, *exceptions redirections*, *indirect calls computations*. . .

*Silver Needle in the Skype* by Philippe Biondi et Fabrice Desclaux, BlackHat 2006

# Concrete example (2)

## Authentication keys protection

### Dropbox

- Packed Python application
- Ciphered bytecode
- Opcode permutation
- Modified Python runtime

*Looking inside the (Drop)Box* by Dhiru Kholia et Przemyslaw W egrzyn, WOOT 2013

# Concrete example (3)

## Protocol protection

### iMessage

- The goal is to protect the iMessage protocol
- Heavily obfuscated application
- Uses a home-made Apple obfuscator
- No known third-party client

# Table of Contents

# Talks

## Talks

- Building a Virtual Machine obfuscation + Questions
- Gaining fine-grain control over pass management + Questions

# Building a Virtual Machine obfuscation

Manuel Carrasco

implementing a VM obfuscation

preparing a reverse engineering challenge

challenge results

studying attacker tools

implementing countermeasures

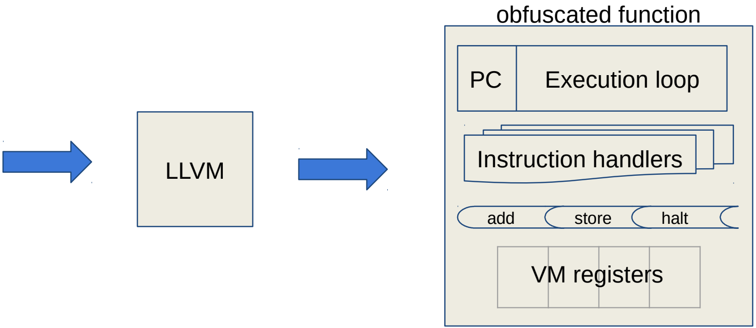
# The virtual machine obfuscation

# A virtual machine as an obfuscation technique

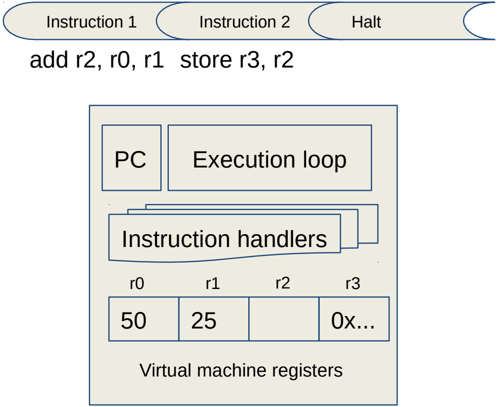
- A virtual machine is an interpreter of certain set of custom instructions (bitcodes).

function to be obfuscated

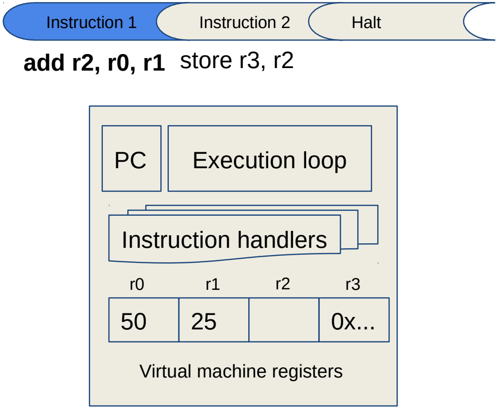
```
// assume there is no
optimization of any kind!
void top_secret(){
      uint8_t* p = 0xABCD;
      uint8_t a = 50;
      uint8_t b = 25;


      uint8_t sum = a + b;
      *p = sum;
}
```



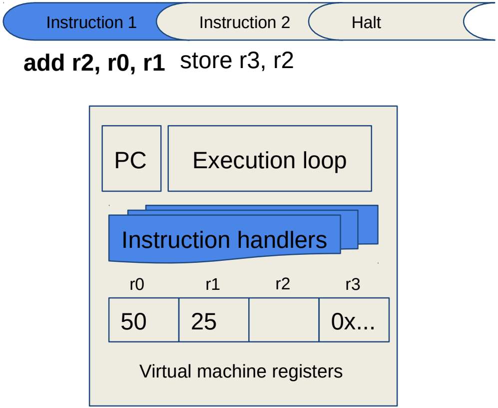
# The VM during execution time



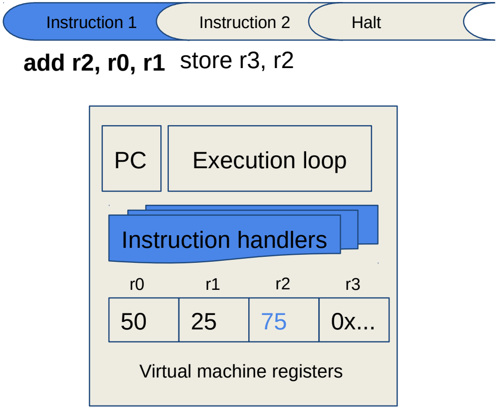
# The VM during execution time



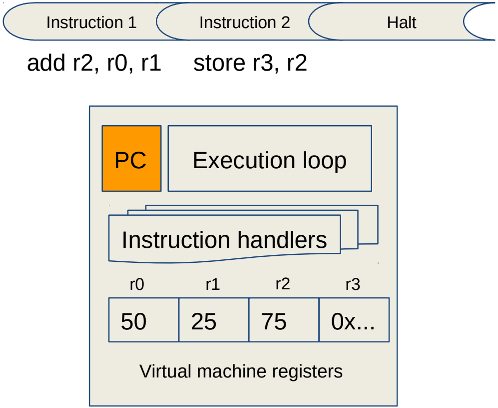
# The VM during execution time



# The VM during execution time

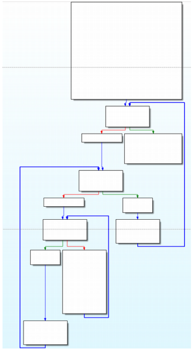


# The VM during execution time

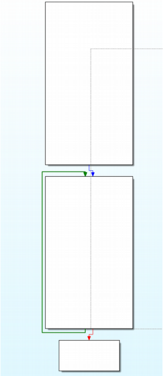


# Pros & cons

# Usefulness of the virtualization technique



Matrix multiplication



Obfuscated code

# Drawbacks of the virtualization technique

- Performance penalties
  - not directly executing the code
  - anyway every obfuscation hurts the performance

- Once a reverser gets a considerable understanding of our virtual architecture the obfuscation becomes pointless.

# Testing the obfuscation

# Internal challenge

Challenge code:

```
#define MAGIC_NUMBER 0123456789

bool OBFUSCATE level0(char const* key)
{

  if (hash(key) == MAGIC_NUMBER)
      return true;

  return false;
}
```

# Feedback

Traditional attack

*manual procedure*

IDA disassembler

only identified parts of the VM

Traditional attack

*manual procedure*

IDA disassembler

only identified parts of the VM

Alternative attack #1

*semi-automatic procedure*

TRITON

Dynamic Binary Analysis

Dynamic Symbolic Execution

solved the challenge

Traditional attack

*manual procedure*

IDA disassembler

only identified parts of the VM

Alternative attack #1

*semi-automatic procedure*

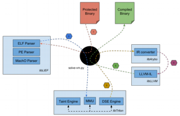
TRITON Dynamic Binary Analysis

Dynamic Symbolic Execution

solved the challenge

Alternative attack #2

*automatic procedure*




Devirtualization technique on top of Triton

new binary without virtual machine obfuscation

# Countermeasures

# New compiler transformation

lookup tables

*new obfuscation*

breaks

Triton's memory accesses modelling

domain divider

*new obfuscation*

breaks

Devirtualization's reachable path exploration

# Lookup tables

# Lookup tables

```c
int a = b & c;
```

```c
int a = and_table[b][c];
```

- and_table is an array generated at compilation time
- memory access based on input is hard for Triton’s DSE ✓
- ‘&’ op is not done during executing time ✓
- reverser could need to understand meaning of the constants ✓

# Lookup tables

```
int a = b & c;
```

```
int a = and_table[b][c];
```

- table's size is **huge** for 32 bits values: aprox. 36893488 terabytes!
- & op table is easily understandable

# ‘&’ table possible sizes

| operand’s size (bits) | table size |
|---|---|
| 32 | 36893488 terabytes |
| 16 | 4,3 gigabytes |
| 8 | 32,8 kilobytes |
| 4 | 64 bytes |
| 2 | 2 bytes |

Can we use the 4 bit table to compute operations in 32 bits? Yes

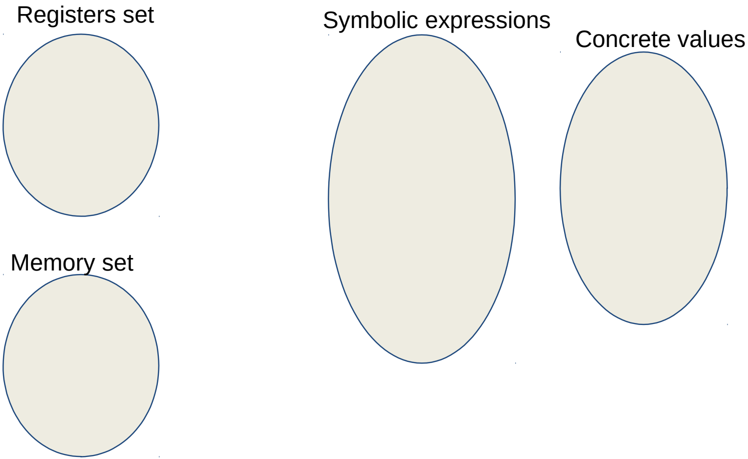
# Folding an instruction chain

```
int A = P0 | P1;
int B = P2 ^ A;
int C = B ^ P3;
```

```
char chain_2_bits[][][][] = {...};
```

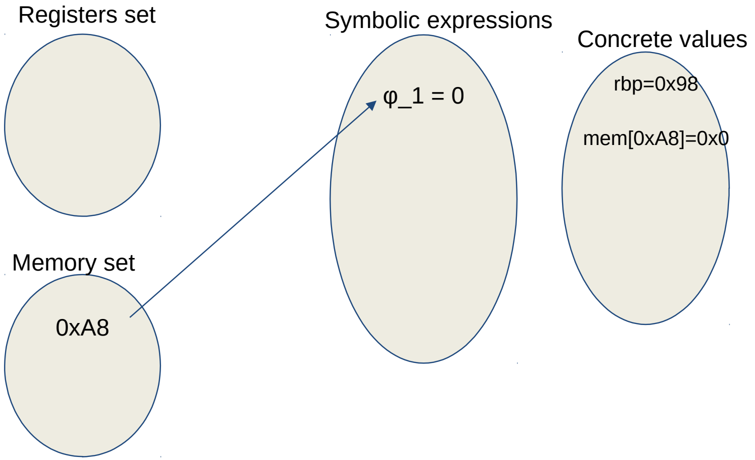
# Triton’s DSE

# Dynamic symbolic execution in Triton



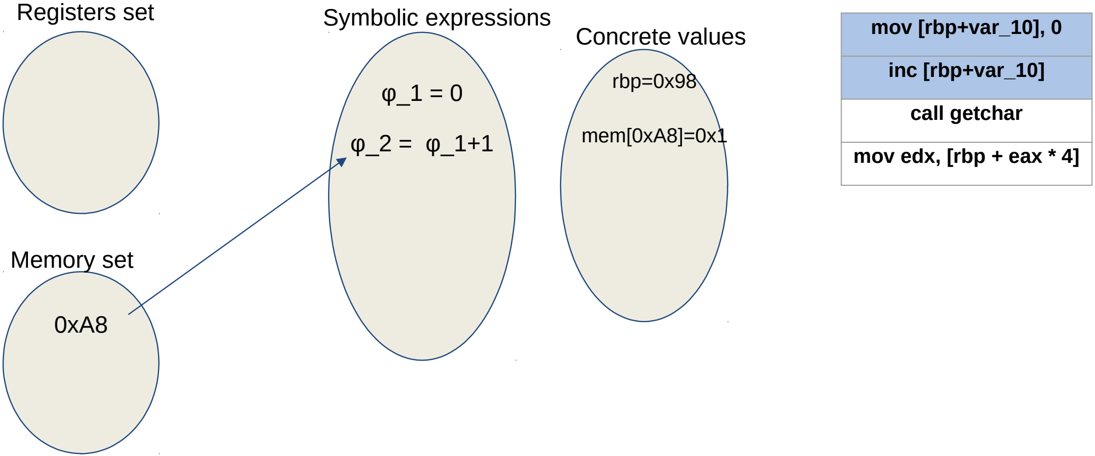

| |
|---|
| **mov [rbp+var_10], 0** |
| **inc [rbp+var_10]** |
| **call getchar** |
| **mov edx, [rbp + eax * 4]** |

# Dynamic symbolic execution in Triton

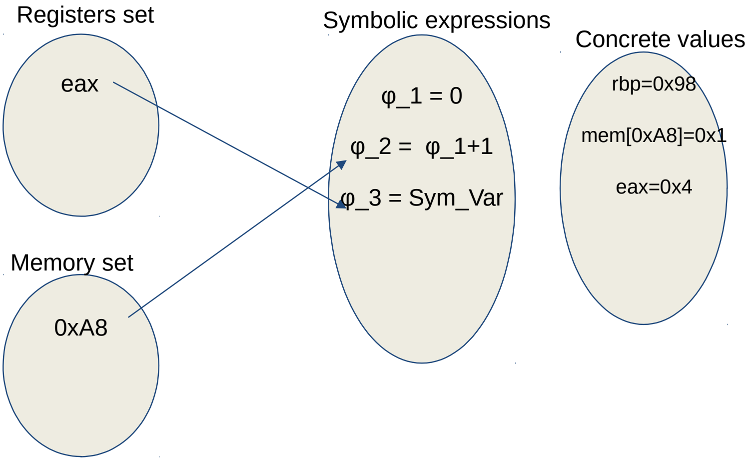



| |
|---|
| **mov [rbp+var_10], 0** |
| **inc [rbp+var_10]** |
| **call getchar** |
| **mov edx, [rbp + eax * 4]** |

# Dynamic symbolic execution in Triton



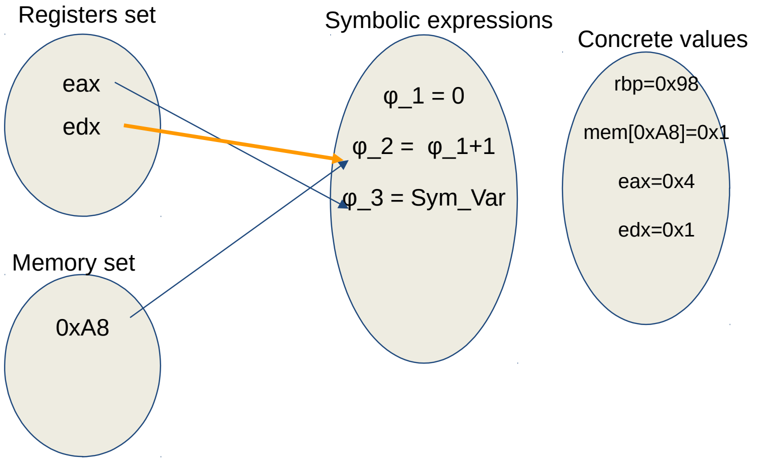
# Dynamic symbolic execution in Triton




| |
|---|
| **mov [rbp+var_10], 0** |
| **inc [rbp+var_10]** |
| **call getchar** |
| **mov edx, [rbp + eax * 4]** |

assume getchar returns 0x4 in eax

# Dynamic symbolic execution in Triton




| |
|---|
| **mov [rbp+var_10], 0** |
| **inc [rbp+var_10]** |
| **call getchar** |
| **mov edx, [rbp + eax * 4]** |

rbp + eax * 4 = 0xA8

# Domain divider

# Domain divider: adding reachable paths

1. Split the domain of a partial computation of the result
2. On each path recompute the partial computation

```
uint32 interm_computation = a + b;
```

Intermediate computation in our program

Why is it effective against the devirtualization attack?

# Why is it effective against the devirtualization attack?

- The devirtualization must explore every reachable path by generating concrete input using an SMT solver.

# Symbolic deobfuscation: from virtualized code back to the original*
# (long version)

Jonathan Salwan[1], Sébastien Bardin[2], and Marie-Laure Potet[3]

# Testing the new paths

```
unsigned long SECRET(unsigned long input) {

   unsigned char *data = (unsigned char*)&input;
   size_t len = sizeof(input);
   uint32_t a = 1, b = 0;
   size_t index;

   /* Process each byte of the data in order */
   for (index = 0; index < len; ++index) {
      a = (a + data[index]) % MOD_ADLER;
      b = add(b, a) % MOD_ADLER;
   }

   return (b << 16) | a;
}
```

| | *not obfuscated* | *obfuscated* |
|---|---|---|
| finished? | yes | timeout (10 minutes) |

2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

# Manufacturing Resilient Bi-Opaque Predicates against Symbolic Execution

Hui Xu[*†], Yangfan Zhou[‡§], Yu Kang[‡], Fengzhi Tu[*], Michael R. Lyu[*†]
[*] Shenzhen Research Institute, The Chinese University of Hong Kong
[†] Dept. of Computer Science and Engineering, The Chinese University of Hong Kong
[‡] School of Computer Science, Fudan University
[§] Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education

# Code Obfuscation Against Symbolic Execution Attacks

Sebastian Banescu
Technische Universität München
banescu@in.tum.de

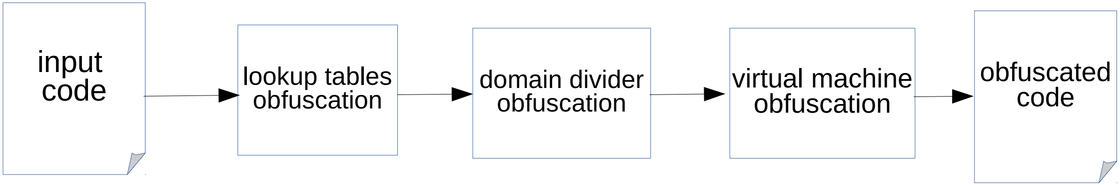
Christian Collberg
University of Arizona
collberg@gmail.com

Vijay Ganesh
University of Waterloo
vganesh@uwaterloo.ca

Zack Newsham
University of Waterloo
znewsham@uwaterloo.ca

Alexander Pretschner
Technische Universität München
pretschn@in.tum.de

# Achievements



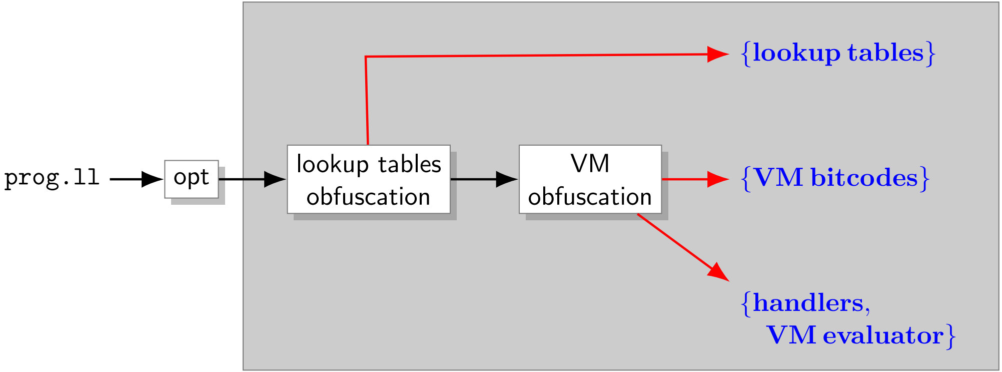

# Future work

Implement countermeasures against other types of dynamic analysis such as dynamic taint analysis

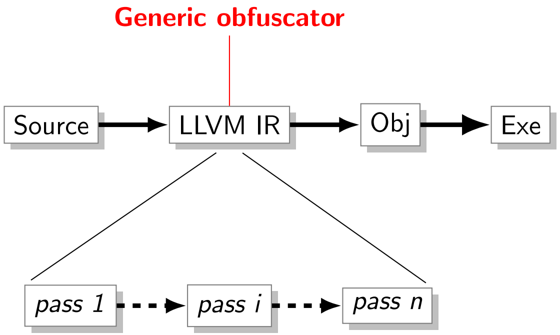
# Gaining Fine-Grain Control over Pass Management

Béatrice Creusillet, Adrien Guinet, Pierrick Brunet,
Juan Manuel Martinez and Serge Guelton

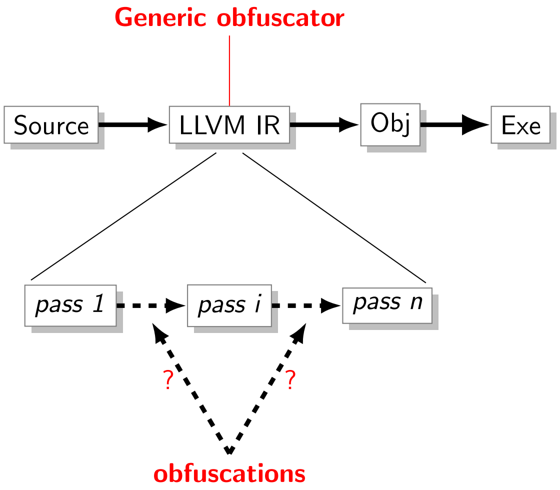
# Beyond VM Obfuscation



# Epona: an LLVM IR obfuscator



# Epona: an LLVM IR obfuscator




performance

vs.

protection

# Obfuscation/optimizations pass scheduling: Challenges

# Obfuscation/optimizations pass scheduling: Challenges

**The code is indeed obfuscated**

- Previous passes don't hinder obfuscations
- Next optimization passes don't *deobfuscate* the generated code

# Obfuscation/optimizations pass scheduling: Challenges

**The code is indeed obfuscated**

- Previous passes don't hinder obfuscations
- Next optimization passes don't *deobfuscate* the generated code

**No invalid code is generated**

- Obfuscations are only applied to inputs meeting certain criteria
- No uncompatible obfuscations are applied to a Function or Module.

# Obfuscation/optimizations pass scheduling: Challenges

**The code is indeed obfuscated**

- Previous passes don't hinder obfuscations
- Next optimization passes don't *deobfuscate* the generated code

**No invalid code is generated**

- Obfuscations are only applied to inputs meeting certain criteria
- No uncompatible obfuscations are applied to a Function or Module.

**Generated code meets performance expectations**

- Performance impact is hard to infer statically
- Obfuscate only when actually necessary

# Obfuscation/optimizations pass scheduling: Challenges

**The code is indeed obfuscated**

- Previous passes don't hinder obfuscations
- Next optimization passes don't *deobfuscate* the generated code

**No invalid code is generated**

- Obfuscations are only applied to inputs meeting certain criteria
- No uncompatible obfuscations are applied to a Function or Module.

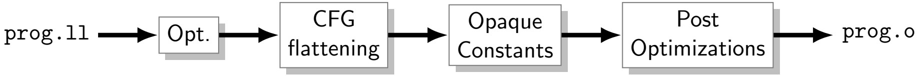
**Generated code meets performance expectations**

- Performance impact is hard to infer statically
- Obfuscate only when actually necessary

**Compilation time remains acceptable**

- Run obfuscations/optimizations only when necessary

# What about opt?




```
> opt -O2 -call-graph-flattening \
      -opaque-constants -opaque-constants-ratio=0.1 \
      -post-optimize -post-optimize-level=2 my_prog.ll
```

- Linear pass chaining
- Passes are applied on all functions
- Pass options apply to every invocations of the pass

# An evolution over opt: optsh

**New features**

- Select functions on which to apply passes
- Set/unset options

```
set   opaque-constant-ratio=0.1
apply opaque-constant on foo
apply cfg-flattening on bar
reset opaque-constant-ratio
apply opaque-constant on bar
set   post-optimize-level=2
apply post-optimize
```

**But still...**

- No control over new resources produced by passes.
- No mean to apply passes conditionaly

# Property-based pass scheduling

- **Properties** are associated to functions
- **Properties aware passes**:
  - wrap raw passes
  - transform only the input functions satifying a predicate checking the propeties
  - also transform the associated properties

# Property-based pass scheduling

- **Properties** are associated to functions
- **Properties aware passes**:
  - wrap raw passes
  - transform only the input functions satifying a predicate checking the propeties
  - also transform the associated properties

**Ex: apply LLVM post-optimizations solely on obfuscated functions**

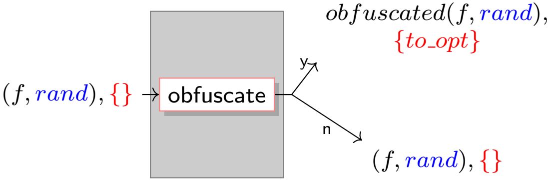
$(f, rand), \{\} \rightarrow$ obfuscate

In: *Combining Obfuscation and Optimizations in the Real World*, Scam 2018, Madrid, Spain
Serge Guelton, Adrien Guinet, Pierrick Brunet, Juan Manuel Martinez, Fabien Dagnat and Nicolas Szlifierski,

# Property-based pass scheduling

- **Properties** are associated to functions
- **Properties aware passes**:
  - wrap raw passes
  - transform only the input functions satifying a predicate checking the propeties
  - also transform the associated properties

**Ex: apply LLVM post-optimizations solely on obfuscated functions**

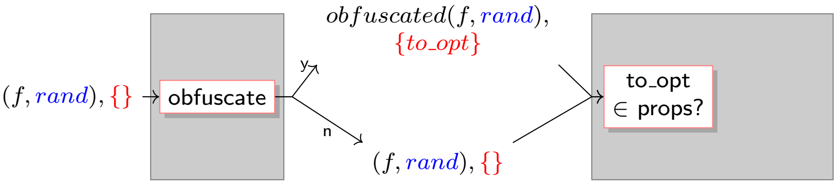



In: *Combining Obfuscation and Optimizations in the Real World*, Scam 2018, Madrid, Spain
Serge Guelton, Adrien Guinet, Pierrick Brunet, Juan Manuel Martinez, Fabien Dagnat and Nicolas Szlifierski,

# Property-based pass scheduling

- **Properties** are associated to functions
- **Properties aware passes**:
  - wrap raw passes
  - transform only the input functions satifying a predicate checking the propeties
  - also transform the associated properties

**Ex: apply LLVM post-optimizations solely on obfuscated functions**

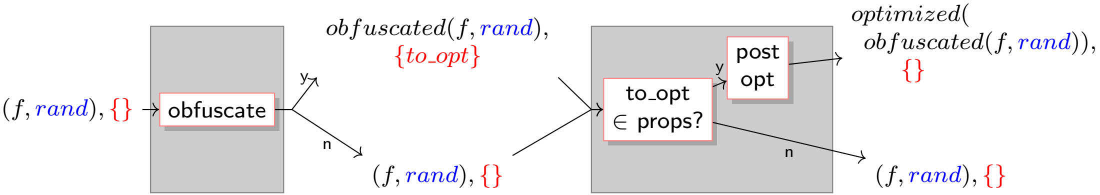



In: *Combining Obfuscation and Optimizations in the Real World*, Scam 2018, Madrid, Spain
Serge Guelton, Adrien Guinet, Pierrick Brunet, Juan Manuel Martinez, Fabien Dagnat and Nicolas Szlifierski,

# Property-based pass scheduling

- **Properties** are associated to functions
- **Properties aware passes**:
  - wrap raw passes
  - transform only the input functions satifying a predicate checking the propeties
  - also transform the associated properties

**Ex: apply LLVM post-optimizations solely on obfuscated functions**




In: *Combining Obfuscation and Optimizations in the Real World*, Scam 2018, Madrid, Spain
Serge Guelton, Adrien Guinet, Pierrick Brunet, Juan Manuel Martinez, Fabien Dagnat and Nicolas Szlifierski,

# Value Views: consistently conveying information between passes

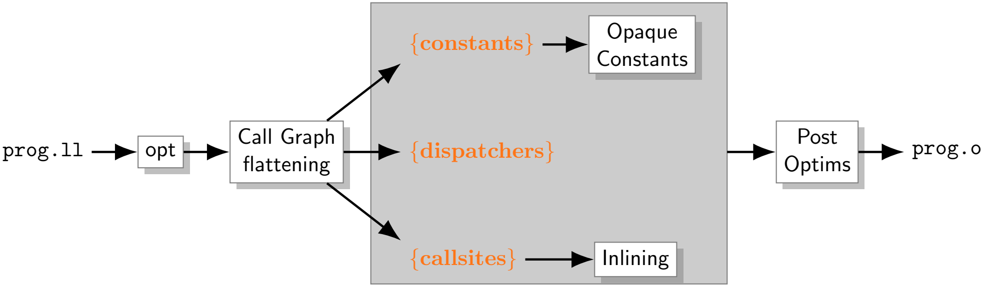
**Value Views** are:

- type safe llvm value containers: `ValueView<BasicBlock>`, `ValueView<Function>`, ...
- produced and consumed by **Value Views aware passes**
- consistent over pass chaining
  - automatic updates through **Value Handlers**
  - special case for operands of instructions (no *Use Handler*)

# Value Views: consistently conveying information between passes

**Value Views** are:

- type safe llvm value containers: `ValueView<BasicBlock>`, `ValueView<Function>`, ...
- produced and consumed by **Value Views aware passes**
- consistent over pass chaining
  - automatic updates through **Value Handlers**
  - special case for operands of instructions (no *Use Handler*)



# Related work

- **Attributes**
- **CHiLL** *(M. Hall et al.)*
  - loops transformations scheduling
  - Decision algorithm based on loop transformation descriptions
- **Obfuscation Executive** *(K. Heffner & C. Collberg)*
  - Dynamic pass manager
  - Goal: reach a terminating condition
  - Obfuscations are associated to a *cost*, a *potency*, pre- and post requirements, and pre-and-post suggestions
  - The impact of randomness is not considered.
- **PyPS** *(S. Guelton)*
  - Dynamic python pass manager API for the PIPS Fortran/C parallelizing compiler
  - Various levels of granularity (passes, modules, functions, loops, ...)
  - Control through python control flow constructs allowed

# Gaining fine-grain control over pass-management?

**Achievements**

- **Optsh**: control over function and option selection
- **Properties**: control over pass inputs filtering
- **Value views**: control over pass input/output values and constants