Obfuscation: know your enemy

Ninon EYROLLES
neyrolles@quarkslab.com

Serge GUELTON
sguelton@quarkslab.com

QUARKSLAB
INNOVATIVE SECURITY

Prelude

```
1  print "Charlie says hello"
```

Prelude

```
1  print "Charlie says hello"
```

```
1  a = ''.join([x for x in [chr(0x65 | 0x1), chr(ord('k')-2),
2                           'L'.lower(), "harvest".rstrip('tsve'),
3                           'C']][::-1])
4  b = ''.join(map(chr, [x for x in [0x20, 0x73, 0x61, 0x80, 0x79,
5                                    0x73, 0x20, 0x93, 0x68, 0x65,
6                                    0x6c, 0x94, 0x6c, 0x6f]
7                         if x < 0x80]))
8  print a + b
```

Plan

Code obfuscation

Definition

Obfuscation is used to make code analysis as complex and expensive as possible, while keeping the original behaviour of the program (input/output equivalence).

- Malwares (try to avoid signature detection)
- Protection of sensitive algorithm (DRM, intellectual property...)
- Theoretically: transformation of symetric-key encryption in asymetric-key encryption, homomorphic encryption algorithm...

Don't shoot the messenger

Why this talk ?

→ Obfuscation exists and is widely used.

→ You might be interested in breaking it (to rewrite some code as free software for example).

⇒ If you want to break it, you need to know how it works!

Several obfuscation types

- Control flow obfuscation
- Data-flow obfuscation
- Symbols rewriting: variable names, function names...
- Code encryption, packing...

Several obfuscation types

- **Control flow obfuscation**
- **Data-flow obfuscation**
- Symbols rewriting: variable names, function names...
- Code encryption, packing...

Plan

Control flow

- Illustrates the execution flow of a program: the different paths that are possible during the execution
- Cycles (`for`, `while`...), conditions (`if`), calls to other functions...
- It's represented with a *Control Flow Graph* (CFG): it's formed of *basic blocks* and links between them

Control flow

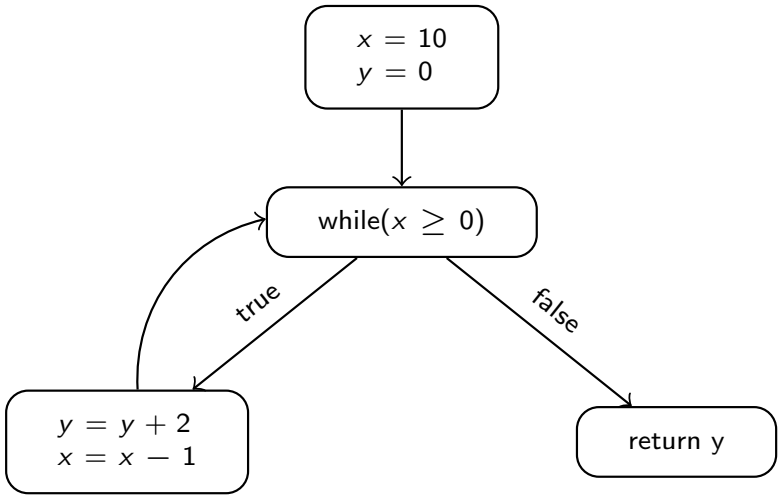


Figure : CFG of pseudo-code

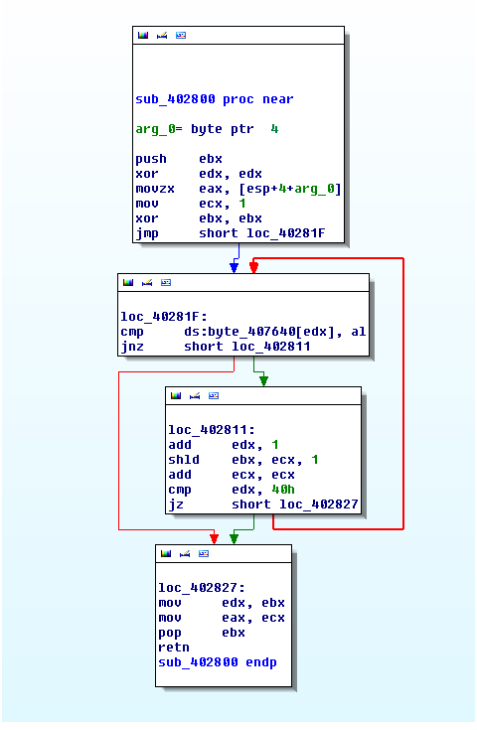


Figure : CFG of assembly code

Various techniques

The goal is to transform the structure of the CFG:

Various techniques

The goal is to transform the structure of the CFG:

- loop unrolling;

Various techniques

The goal is to transform the structure of the CFG:

- loop unrolling; → search for patterns

Various techniques

The goal is to transform the structure of the CFG:

- loop unrolling; → search for patterns
- inlining of function;

Various techniques

The goal is to transform the structure of the CFG:

- loop unrolling; → search for patterns
- inlining of function; → comparison of code

Various techniques

The goal is to transform the structure of the CFG:

- loop unrolling; → search for patterns
- inlining of function; → comparison of code
- junk code insertion;

Various techniques

The goal is to transform the structure of the CFG:

- loop unrolling; → search for patterns
- inlining of function; → comparison of code
- junk code insertion; → liveness analysis

Various techniques

The goal is to transform the structure of the CFG:

- loop unrolling; → search for patterns
- inlining of function; → comparison of code
- junk code insertion; → liveness analysis
- opaque predicates;

Various techniques

The goal is to transform the structure of the CFG:

- loop unrolling; → search for patterns
- inlining of function; → comparison of code
- junk code insertion; → liveness analysis
- opaque predicates; → SMT solver

Various techniques

The goal is to transform the structure of the CFG:

- loop unrolling; → search for patterns
- inlining of function; → comparison of code
- junk code insertion; → liveness analysis
- opaque predicates; → SMT solver
- control flow flattening.

Definition

Control flow flattening

- Transforms the structure of the program to make CFG reconstruction difficult
- Encodes the control flow information and hide the result in the data flow

Principle

Implementation

- Basic blocks are numbered
- A *dispatcher* handles the execution
- A variable contains the value of the next block to be executed
- At the end of every block, this variable is updated, and the execution flow goes back to the dispatcher which then jumps to the next block

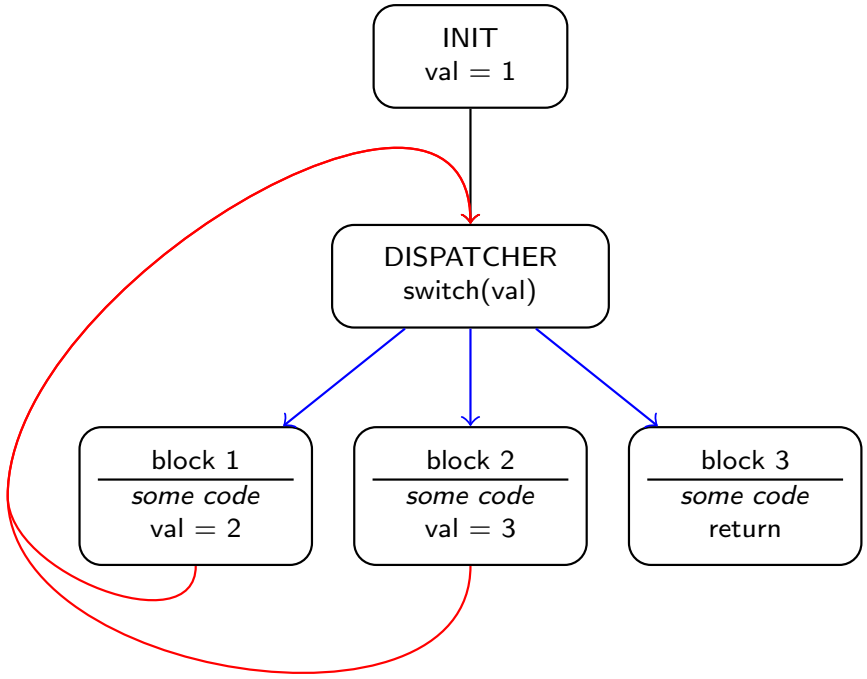


Figure : Principle of control flow flattening

Example

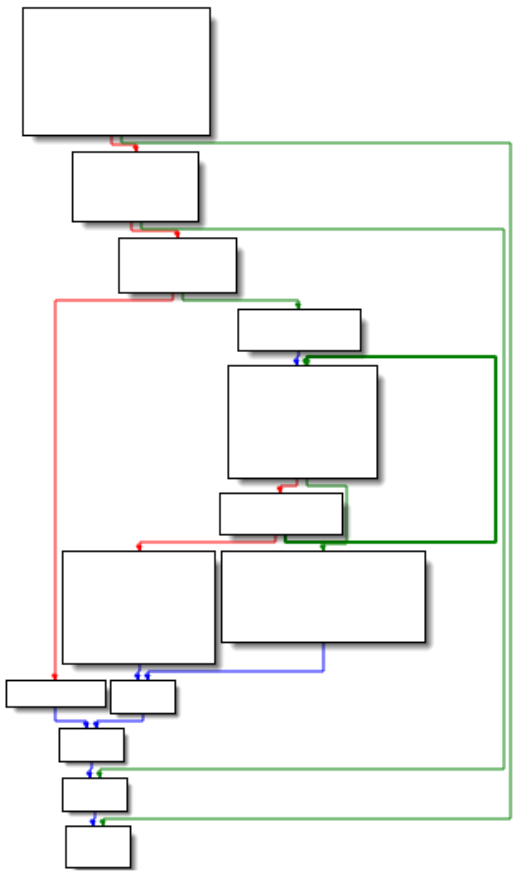


Figure : original CFG

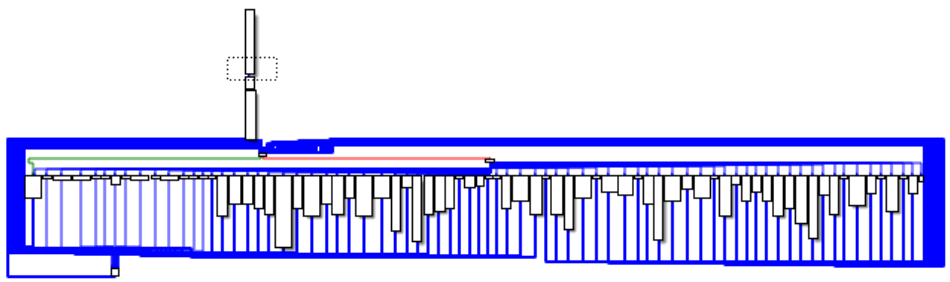
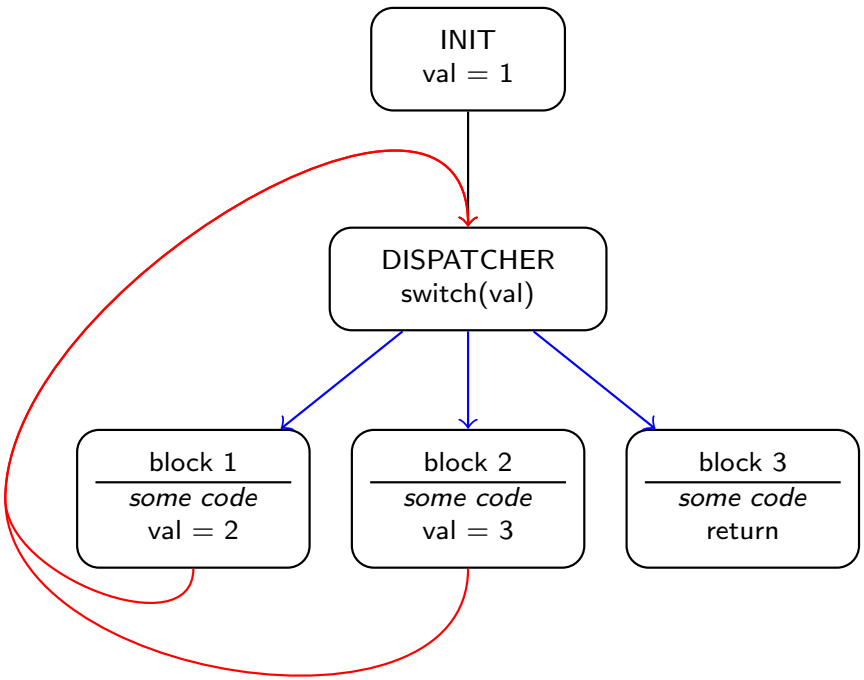


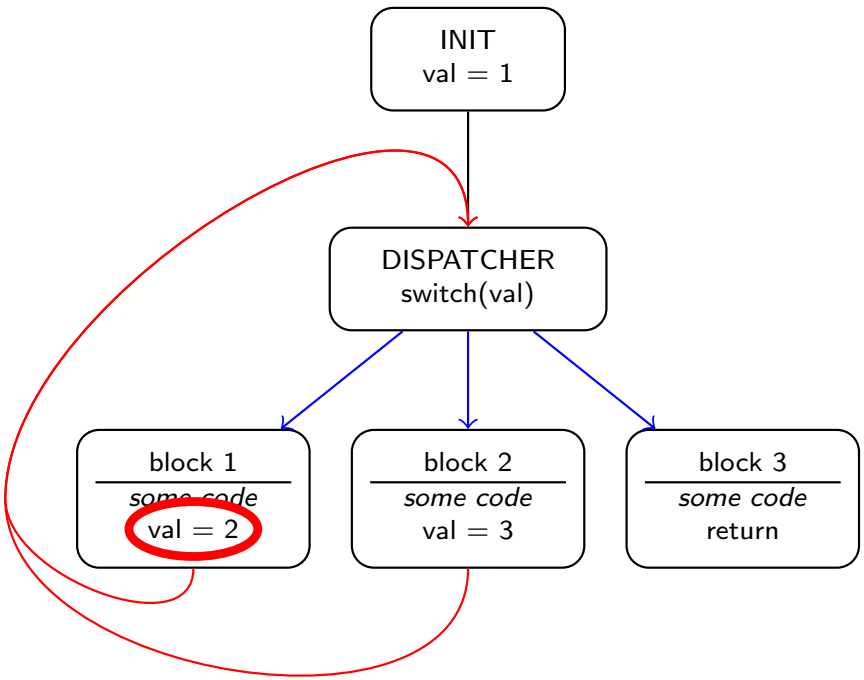
Figure : CFG after the control flow flattening

Weakness



What is the weakness of the control flow flattening ?

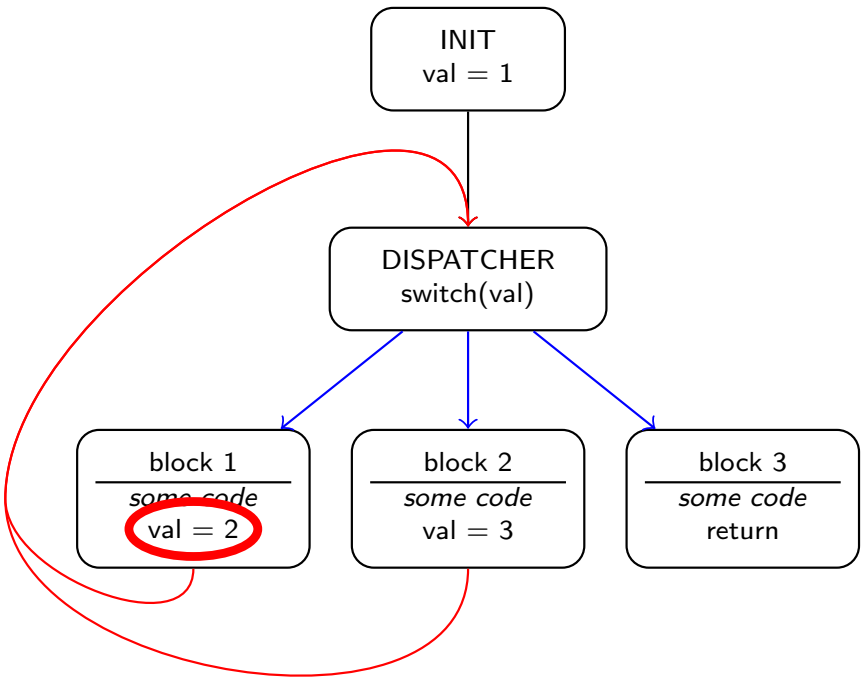
Weakness



What is the weakness of the control flow flattening ?

$\Rightarrow$ variable containing the execution flow

Weakness



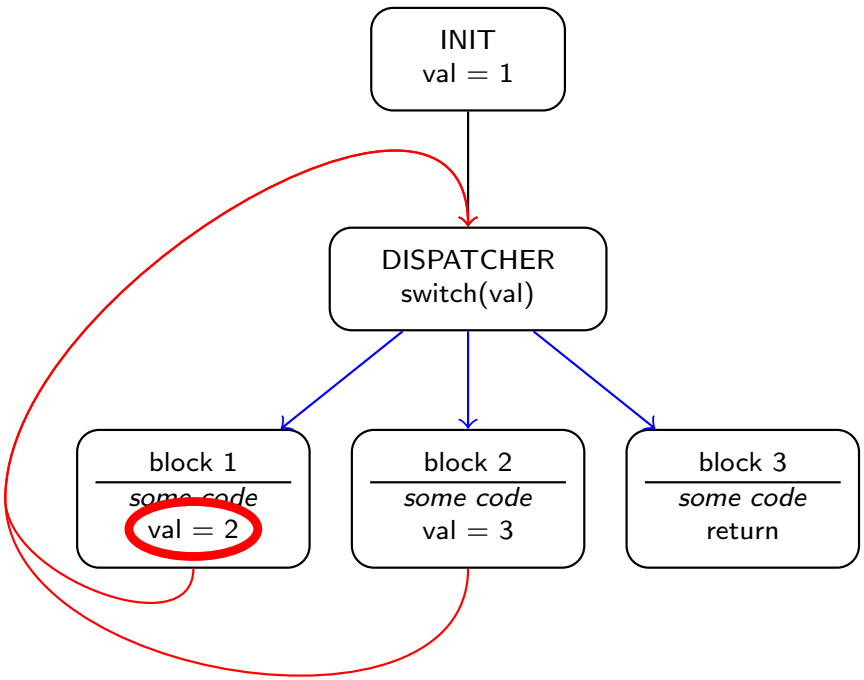
What is the weakness of the control flow flattening ?

$\Rightarrow$ variable containing the execution flow

Obfuscation techniques:

- multiple (context) variables
- opaque predicates
- hash

Weakness



What is the weakness of the control flow flattening ?

⇒ variable containing the execution flow

Obfuscation techniques:

- multiple (context) variables
- opaque predicates
- hash

⇒ dynamic analysis (tracing) can also be used

Plan

Data Flow analysis

Several ways to do it

- Information provided by the program's data: strings, numbers, structures...
- Relations between the data or between the input and output (of a program, a function, a basic block)
- Interactions between the program and the data: reading, writing, location in memory...
- Formal notions: live variable, data flow equations, backward and forward analysis...

Examples

To make data analysis more complex:

Examples

To make data analysis more complex:

- encode constants (strings for example);

Examples

To make data analysis more complex:

- encode constants (strings for example);
  $\rightarrow$ look for decoding routine

Examples

To make data analysis more complex:

- encode constants (strings for example);
  $\rightarrow$ look for decoding routine
- insert useless data (close to junk code);

Examples

To make data analysis more complex:

- encode constants (strings for example);
  → look for decoding routine
- insert useless data (close to junk code);
  → use symbolic execution, data tainting / slicing

Examples

To make data analysis more complex:

- encode constants (strings for example);
  $\rightarrow$ look for decoding routine
- insert useless data (close to junk code);
  $\rightarrow$ use symbolic execution, data tainting / slicing
- complexify arithmetic operations on data;

$$x + y \quad \Leftrightarrow \quad (x \oplus y) + 2 * (x \wedge y)$$

Examples

To make data analysis more complex:

- encode constants (strings for example);
  $\rightarrow$ look for decoding routine
- insert useless data (close to junk code);
  $\rightarrow$ use symbolic execution, data tainting / slicing
- complexify arithmetic operations on data;

$$x + y \quad \Leftrightarrow \quad (x \oplus y) + 2 * (x \wedge y)$$

  $\rightarrow$ use bruteforce and build heuristic

Examples

- modify the way data are stored / manipulated: split tables, change the calling convention of functions, etc;

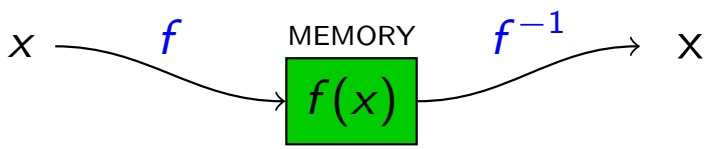
Examples

- modify the way data are stored / manipulated: split tables, change the calling convention of functions, etc;
  → spot similar elements (probably processed by the same instructions)
  → dynamic analysis to get arguments of a function

Examples

- modify the way data are stored / manipulated: split tables, change the calling convention of functions, etc;
  $\rightarrow$ spot similar elements (probably processed by the same instructions)
  $\rightarrow$ dynamic analysis to get arguments of a function

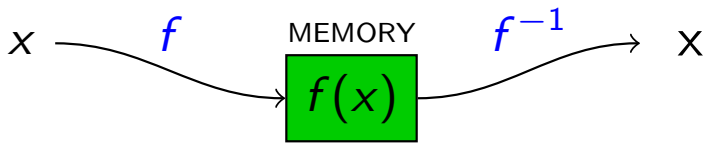
- encode data while reading and writing.



Examples

- modify the way data are stored / manipulated: split tables, change the calling convention of functions, etc;
  → spot similar elements (probably processed by the same instructions)
  → dynamic analysis to get arguments of a function

- encode data while reading and writing.



  → find the relevant variables, and look for the corresponding encoding

Plan

1. Introduction
2. Control flow obfuscation
3. Data flow obfuscation
4. Python obfuscation
   - Modified Interpreter
   - Source-to-source obfuscation
   - A few examples

Context

- Applications are developed in Python (DropBox for example): a modified interpreter is delivered with the binary
- Creation of "packers" to make access to the code difficult
- Few traditional obfuscations here!
- Three ways to obfuscate:
  - modified interpreter so that access to compiled files is difficult;
  - measures to make decompilation harder;
  - source to source obfuscation in case of decompilation success.

State of the art

Based on the work of Kholia and Wegrzyn[1]:

Change the magic number

Number specific to each version of CPython, prevent decompilation
$\rightarrow$ bruteforce ($\sim$ 50 possibilities)

[1] *Looking Inside the (Drop) Box*, by D. Kholia and P. Wegrzyn

State of the art

Based on the work of Kholia and Wegrzyn[1]:

Change the magic number

Number specific to each version of CPython, prevent decompilation
$\rightarrow$ bruteforce ($\sim$ 50 possibilities)

Suppress some features

Remove some functions like `PyRun_File()`, or attributes like `co_code`

[1] *Looking Inside the (Drop) Box*, by D. Kholia and P. Wegrzyn

State of the art

Based on the work of Kholia and Wegrzyn[1]:

Change the magic number

Number specific to each version of CPython, prevent decompilation
$\rightarrow$ bruteforce ($\sim$ 50 possibilities)

Suppress some features

Remove some functions like `PyRun_File()`, or attributes like `co_code`

Opcode encryption

Encrypt compiled files $\rightarrow$ find decryption routine

[1] *Looking Inside the (Drop) Box*, by D. Kholia and P. Wegrzyn

State of the art

Opcode remapping

Applies a permutation on the opcodes of the instruction set.

34	LOAD_GLOBAL
35	CALL_FUNCTION
36	POP_TOP

$$\begin{array}{l} 34 \rightarrow 75 \\ 35 \rightarrow 23 \\ 36 \rightarrow 12 \end{array} \Rightarrow$$

75	LOAD_FAST
23	LOAD_CONST
12	ROT_TWO

State of the art

Opcode remapping

Applies a permutation on the opcodes of the instruction set.

34	LOAD_GLOBAL
35	CALL_FUNCTION
36	POP_TOP

$$\begin{array}{l} 34 \rightarrow 75 \\ 35 \rightarrow 23 \\ 36 \rightarrow 12 \end{array} \Rightarrow$$

75	LOAD_FAST
23	LOAD_CONST
12	ROT_TWO

→ Compare permuted bytecode with standard bytecode for standard Python module

State of the art

Opcode remapping

Applies a permutation on the opcodes of the instruction set.

34	LOAD_GLOBAL
35	CALL_FUNCTION
36	POP_TOP

$$\begin{array}{l} 34 \rightarrow 75 \\ 35 \rightarrow 23 \\ 36 \rightarrow 12 \end{array} \Rightarrow$$

75	LOAD_FAST
23	LOAD_CONST
12	ROT_TWO

→ Compare permuted bytecode with standard bytecode for standard Python module

→ Get into the application runtime and execute arbitrary code

New techniques

Addition of new opcodes

Substitution of series of opcodes with a new opcode

LOAD_GLOBAL
CALL_FUNCTION
POP_TOP

$\Rightarrow$

LOAD_GLOBAL
CALL_AND_POP

$\rightarrow$ Analyse the interpreter!

Insertion of junk opcode

New techniques

Addition of new opcodes

Insertion of junk opcode

- Use opcodes for stack manipulation: `ROT_TWO`, `ROT_THREE` or `POP_TOP`
- Combine it to modify bytecode without changing computed values

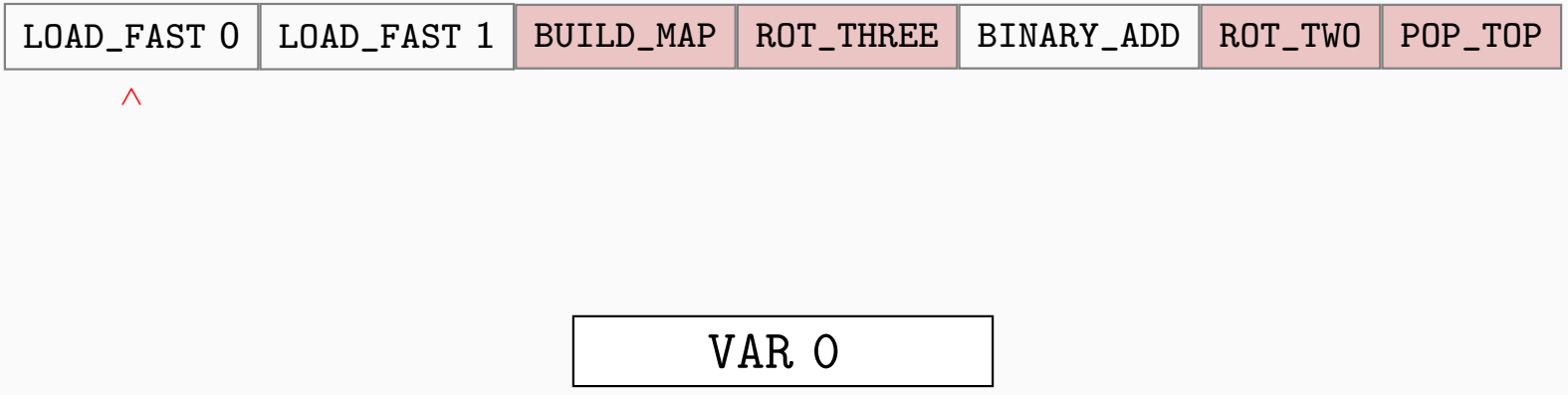
`LOAD_FAST 0`	`LOAD_FAST 1`	`BUILD_MAP`	`ROT_THREE`	`BINARY_ADD`	`ROT_TWO`	`POP_TOP`

New techniques

Addition of new opcodes

Insertion of junk opcode

- Use opcodes for stack manipulation: `ROT_TWO`, `ROT_THREE` or `POP_TOP`
- Combine it to modify bytecode without changing computed values

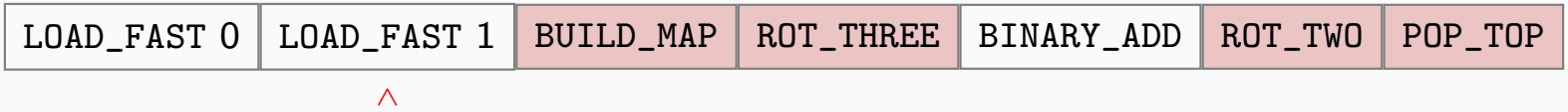


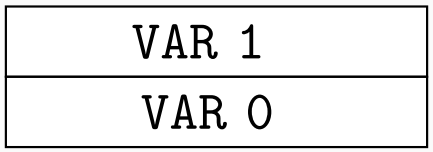
New techniques

Addition of new opcodes

Insertion of junk opcode

- Use opcodes for stack manipulation: `ROT_TWO`, `ROT_THREE` or `POP_TOP`
- Combine it to modify bytecode without changing computed values



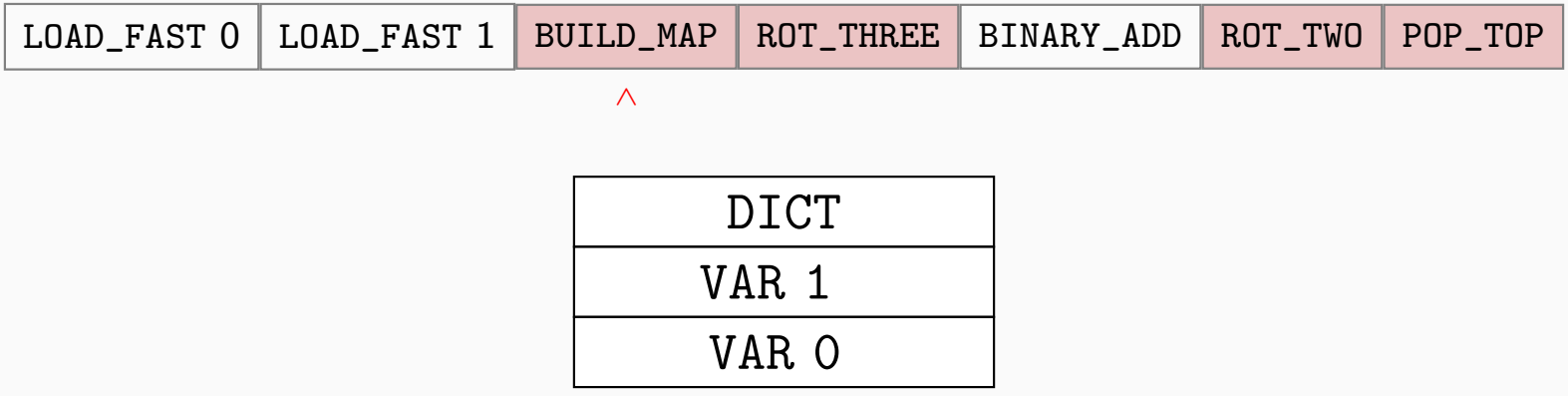


New techniques

Addition of new opcodes

Insertion of junk opcode

- Use opcodes for stack manipulation: `ROT_TWO`, `ROT_THREE` or `POP_TOP`
- Combine it to modify bytecode without changing computed values

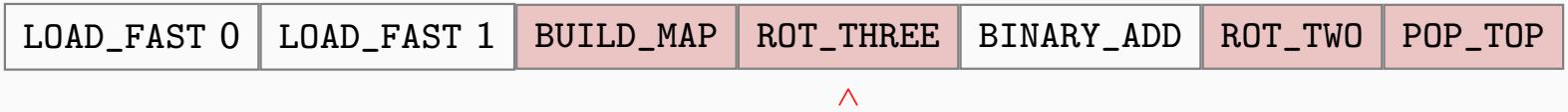


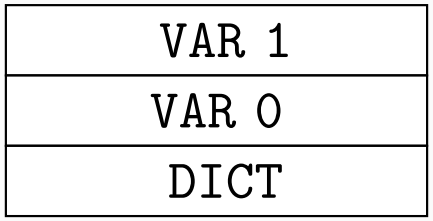
New techniques

Addition of new opcodes

Insertion of junk opcode

- Use opcodes for stack manipulation: `ROT_TWO`, `ROT_THREE` or `POP_TOP`
- Combine it to modify bytecode without changing computed values



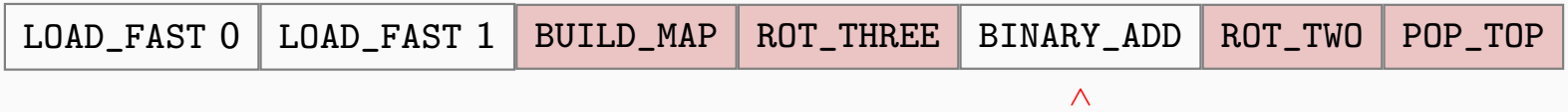


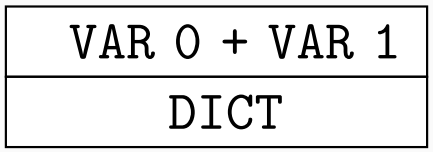
New techniques

Addition of new opcodes

Insertion of junk opcode

- Use opcodes for stack manipulation: `ROT_TWO`, `ROT_THREE` or `POP_TOP`
- Combine it to modify bytecode without changing computed values



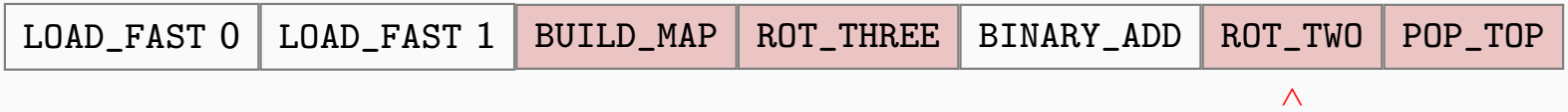


New techniques

Addition of new opcodes

Insertion of junk opcode

- Use opcodes for stack manipulation: `ROT_TWO`, `ROT_THREE` or `POP_TOP`
- Combine it to modify bytecode without changing computed values



DICT
VAR 0 + VAR 1

New techniques

Addition of new opcodes

Insertion of junk opcode

- Use opcodes for stack manipulation: `ROT_TWO`, `ROT_THREE` or `POP_TOP`
- Combine it to modify bytecode without changing computed values

LOAD_FAST 0	LOAD_FAST 1	BUILD_MAP	ROT_THREE	BINARY_ADD	ROT_TWO	POP_TOP

^

```
VAR 0 + VAR 1
```

New techniques

Addition of new opcodes

Insertion of junk opcode

- Use opcodes for stack manipulation: `ROT_TWO`, `ROT_THREE` or `POP_TOP`
- Combine it to modify bytecode without changing computed values

`LOAD_FAST 0`	`LOAD_FAST 1`	`BUILD_MAP`	`ROT_THREE`	`BINARY_ADD`	`ROT_TWO`	`POP_TOP`

→ Prevent decompilation with `uncompyle`, but `pycdc` still works.

Self-modifying code

```
1   def foo(b):
2       b += 1
3       ...
```

```
1   def foo(b):
2       modify_bytecode()
3       b -= 1
4       ...
```

During execution:

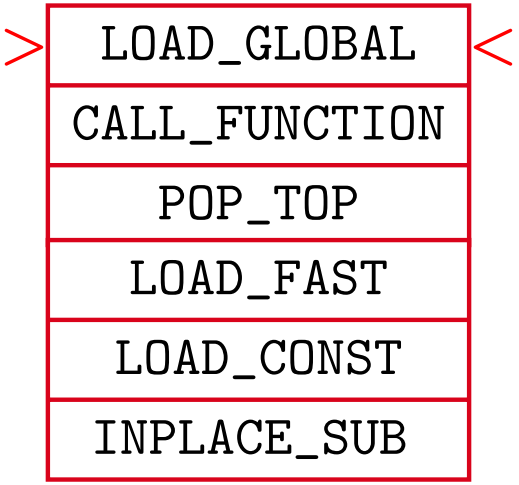
LOAD_GLOBAL
CALL_FUNCTION
POP_TOP
LOAD_FAST
LOAD_CONST
INPLACE_SUB

Self-modifying code

```
1   def foo(b):
2       b += 1
3       ...
```

```
1   def foo(b):
2       modify_bytecode()
3       b -= 1
4       ...
```

During execution:

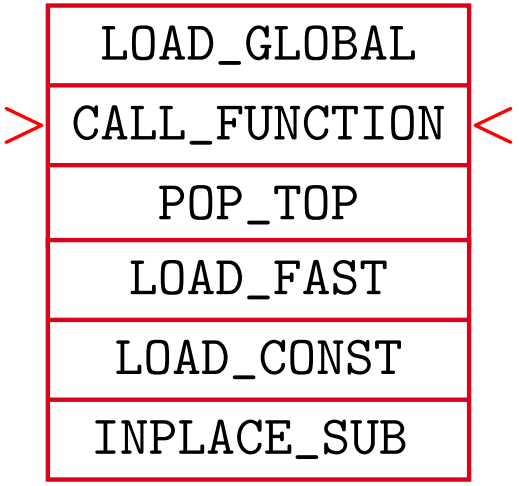


Self-modifying code

```
1   def foo(b):
2       b += 1
3       ...
```

```
1   def foo(b):
2       modify_bytecode()
3       b -= 1
4       ...
```

During execution:

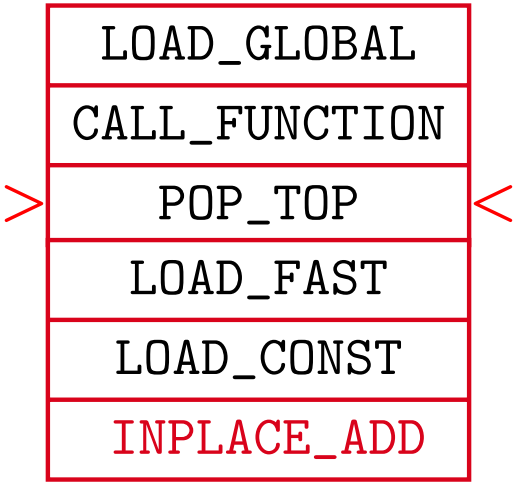


Self-modifying code

```
1   def foo(b):
2       b += 1
3       ...
```

```
1   def foo(b):
2       modify_bytecode()
3       b -= 1
4       ...
```

During execution:

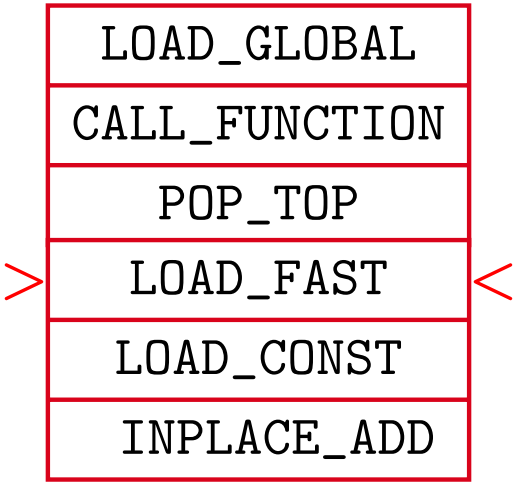


Self-modifying code

```
1   def foo(b):
2       b += 1
3       ...
```

```
1   def foo(b):
2       modify_bytecode()
3       b -= 1
4       ...
```

During execution:

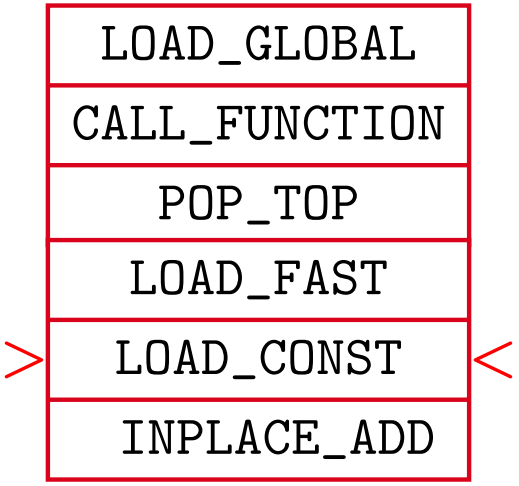


Self-modifying code

```
1   def foo(b):
2       b += 1
3       ...
```

```
1   def foo(b):
2       modify_bytecode()
3       b -= 1
4       ...
```

During execution:

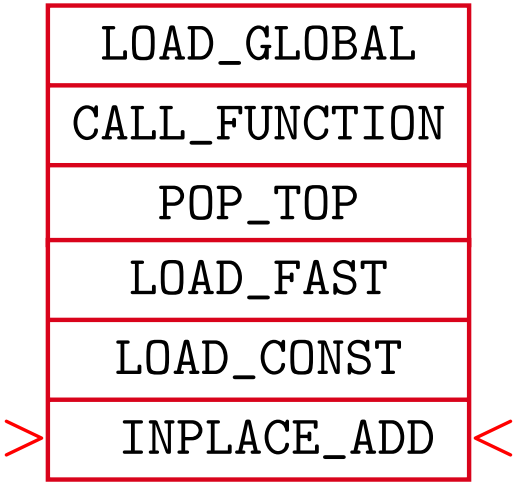


Self-modifying code

```
1   def foo(b):
2       b += 1
3       ...
```

```
1   def foo(b):
2       modify_bytecode()
3       b -= 1
4       ...
```

During execution:



Abstract Syntax Tree

Abstract Syntax Tree (AST): tree representation of the abstract structure of source code.

- Nodes are operators
- Leaves are operands

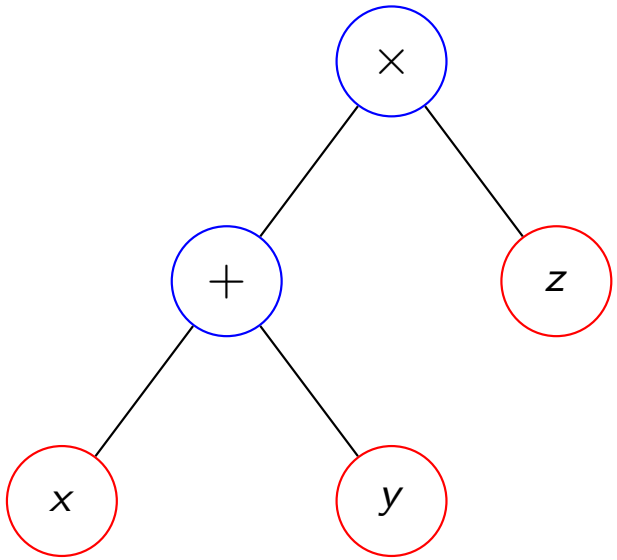


Figure : AST representation of $(x + y) \times z$

Python source-to-source

Principle

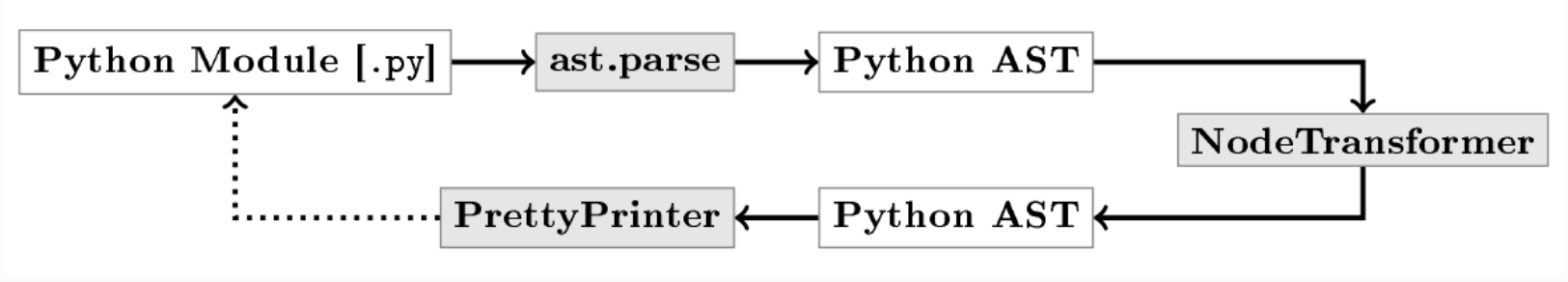


Figure : Compilation flow for Python source-to-source

Obfuscation Examples

Python source-to-source

Principle

Obfuscation Examples

- **Control flow transformations**: loop unrolling, mixing `if` and `while` with opaque predicates, transformation in functional style
- **Data flow transformations**: string encoding, use of mixed boolean-arithmetic expressions.
- **Symbols obfuscation**: replacing names of functions and variables with random string

Loop unrolling

```
1   for i in range(3):
2       if b & 1 == 1:
3           p ^= a
4       hiBitSet = a & 0x80
5       a <<= 1
```

```
1   i = 0
2   if ((b & 1) == 1):
3       p ^= a
4   hiBitSet = (a & 128)
5   a <<= 1
6   i = 1
7   if ((b & 1) == 1):
8       p ^= a
9   hiBitSet = (a & 128)
10  a <<= 1
11  i = 2
12  if ((b & 1) == 1):
13      p ^= a
14  hiBitSet = (a & 128)
15  a <<= 1
```

→ Look for patterns (instructions, variables)

Mixing if and while

```
1   # original code
2   if cond1:
3       work()
```

```
1   # obfuscated if
2   opaque_pred = 1
3   while opaque_pred & cond1:
4       work()
5       opaque_pred = 0
```

→ Holds only if the predicates are difficult to evaluate statically and not obvious for a human

Opaque predicates

```
1  x = ((((2 * ((-816744550) | 816744552)) -
2          ((-816744550) ^ 816744552)) *
3         (((3783141896 ^ 3921565134) -
4           (2 * ((~3783141896) & 3921565134))) |
5          ((4009184523 & (~3870761249)) -
6           ((~4009184523) & 3870761249)))) -
7        (((2105675179 & (~2244098417)) -
8          ((~2105675179) & 2244098417)) ^
9         ((3657555079 + (~3519131805)) + 1)))
```

→ Use constant folding: `x = 36`

```
1  x = (80*b**2 + 160*b*(~ b) + 36821*b +
2       80*(~ b)**2 + 36821*(~ b) + 4236969) % 256
```

→ Bruteforce, heuristics...

Transformation in functional style

```
1  def fibo(n):
2    return n if n < 2 else (fibo(n - 1) + fibo(n - 2))
```

```
1  fibo = (lambda n: (lambda _: (_.__setitem__('$', ((_['n'] if ('
       n' in _) else n) if ((_['n'] if ('n' in _) else n) < 2)
       else ((_['fibo'] if ('fibo' in _) else fibo)(((_['n'] if (
       'n' in _) else n) - 1)) + (_['fibo'] if ('fibo' in _) else
        fibo)(((_['n'] if ('n' in _) else n) - 2))))), _)[(-1)])
       ({'n': n, '$': None})['$'])
```

→ Either you're comfortable with functional style, or you use input/output analysis or symbols information.

Conclusion

- There's a lot of obfuscation techniques
- Understanding obfuscation can be useful (interoperability)
- Keep focused on the context and what you want to know
- Every obfuscation can be broken with time and resources

Questions?

www.quarkslab.com

contact@quarkslab.com | @quarkslab.com

Table of contents