REDOCS2020 - Automatic exploit generation

Maxime Bélair [1] Manh-Dung Nguyen [2] Emilien Fournier [3]
Tristan Benoit [4] Gabriel Sauger [4]

Subject by: Jules Villard

[1]Orange Labs / IMT atlantique - maxime.belair@imt-atlantique.fr

[2]CEA LIST & Université Grenoble Alpes - manh-dung.nguyen@cea.fr

[3]ENSTA Bretagne / Lab-STICC - emilien.fournier@ensta-bretagne.org

[4]LORIA - firstName.lastName@loria.fr

Problem Overview

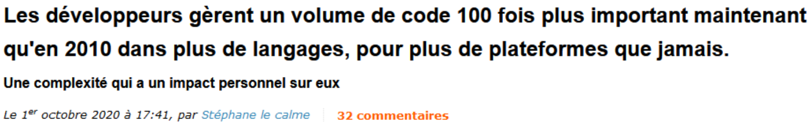
Context

- Codebases are bigger than ever
- More bugs than ever!

Problem Overview

Context

- Codebases are bigger than ever
- More bugs than ever!



Les développeurs gèrent un volume de code 100 fois plus important maintenant qu'en 2010 dans plus de langages, pour plus de plateformes que jamais.

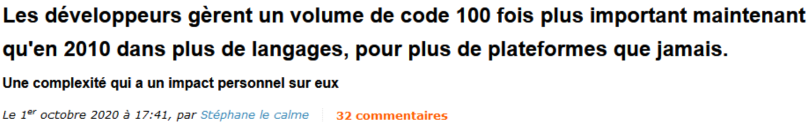
Une complexité qui a un impact personnel sur eux

Le 1^er octobre 2020 à 17:41, par Stéphane le calme | **32 commentaires**

Problem Overview

Context

- Codebases are bigger than ever
- More bugs than ever!



Les développeurs gèrent un volume de code 100 fois plus important maintenant qu'en 2010 dans plus de langages, pour plus de plateformes que jamais.

Une complexité qui a un impact personnel sur eux

Le 1er octobre 2020 à 17:41, par Stéphane le calme **32 commentaires**

Facebook runs on at least 64 millions of line of code. Imagine the debugging.

Background

Static Analysis

- Processes the code of the program
- Don't execute it !

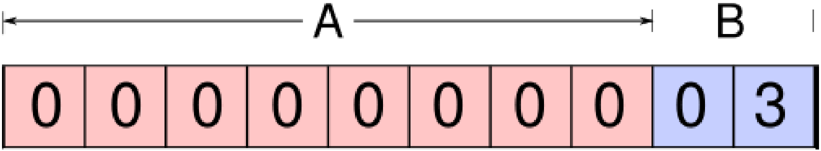
Formal challenge

"Can we automatically turn static analysis reports into executable confirming the vulnerability of a program ?" - Jules Villard, Facebook

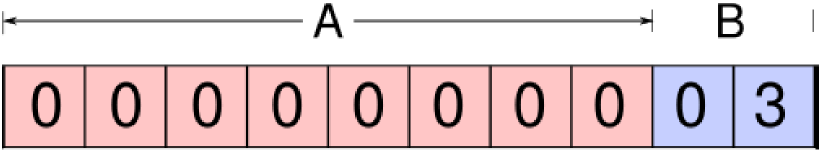
Weakness example: the Buffer Overflow

- Classic attack
- Allow to modify other variables in memory
- A lot of consequences
    - Crash the program (e.g. Segfault)
    - Hijack the execution (e.g. Cracks)
    - Privilege escalation (e.g. Root Shells),
    - ...

The Buffer Overflow – Technically

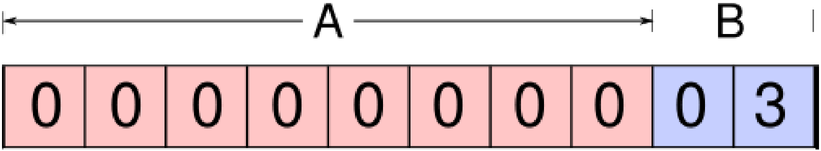


The Buffer Overflow – Technically

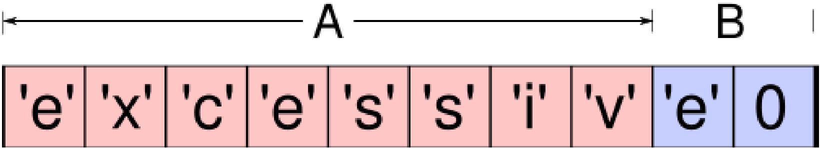


```
strcpy(A, "excessive")
```

The Buffer Overflow – Technically



`strcpy(A, "excessive")`



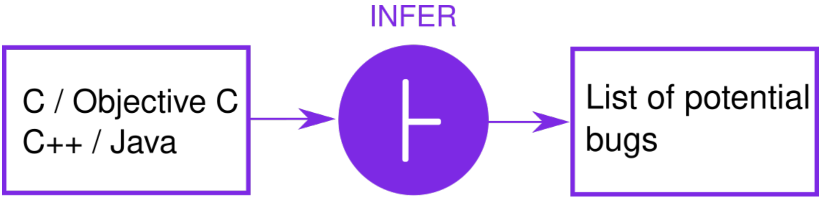
Weakness example: the Buffer Overflow

```c
int main() {
    short valid = 0;
    char buf[8];
    strcpy(buf, "excessive");
    printf("%d", valid); // Overflow !!!
}
```

Program bug example

- Example of vulnerability – 101 Buffer overflow
- Exploit it manually
- Automatize it through Infer later!

Infer tool



INFER

C / Objective C
C++ / Java

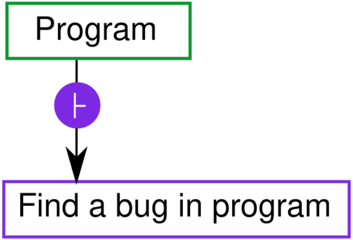
List of potential bugs

- Static analysis tool from Facebook
- **Capture** phase, then **Analysis** phase

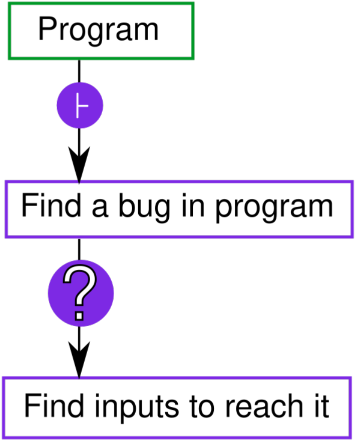
Practical approach

Program

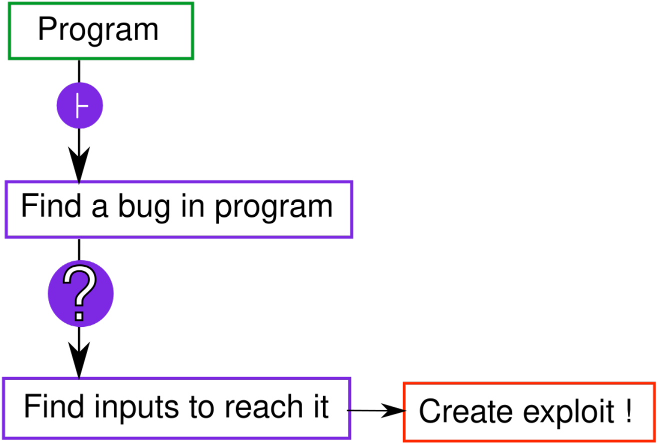
Practical approach



Practical approach



Practical approach



Program

Find a bug in program

Find inputs to reach it

Create exploit !

Practical approach

This week's challenge

Given the Infer information about bugs of a program A, find a **set of inputs** that **crashes** A

Table of content

Proposed approaches

Model checking

Overview

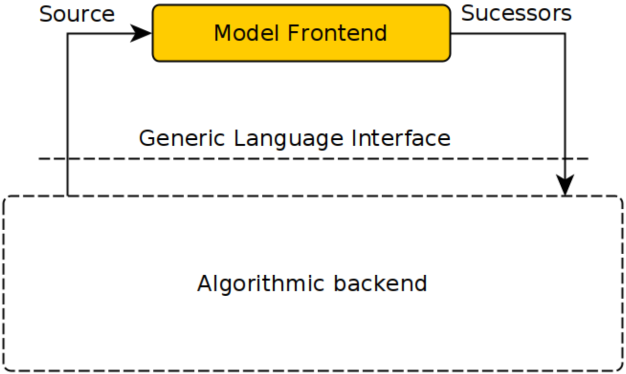
- ■ Formal
- ■ Intuitive
- ■ Automated
- ■ Provides counter-examples
- × State-space explosion

Obelix : Modular Model-Checker

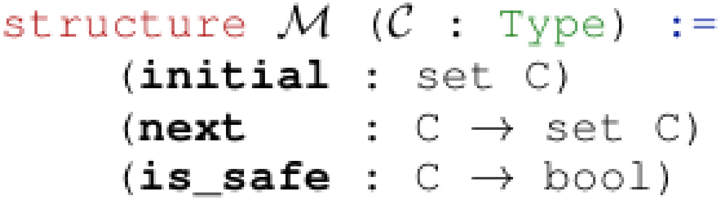
Highlights

- Modular model-checker
- 3 languages : DVE, EMI-UML, C/C++
- 12+ algorithms
- Bitstate hashing
- Swarm model-checking

Obelix : Generic Language Interface

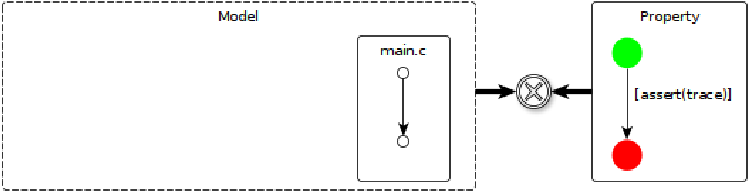


Obelix : Generic Language Interface

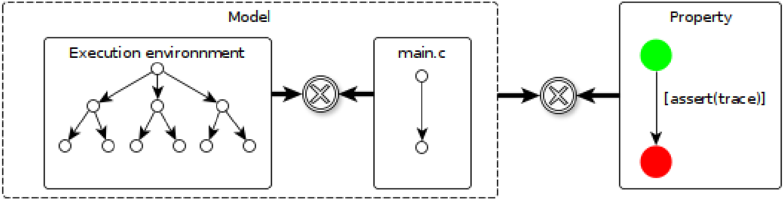


```
structure ℳ (𝒞 : Type) :=
     (initial : set C)
     (next    : C → set C)
     (is_safe : C → bool)
```

Application : How ?



Application : How ?



A few results

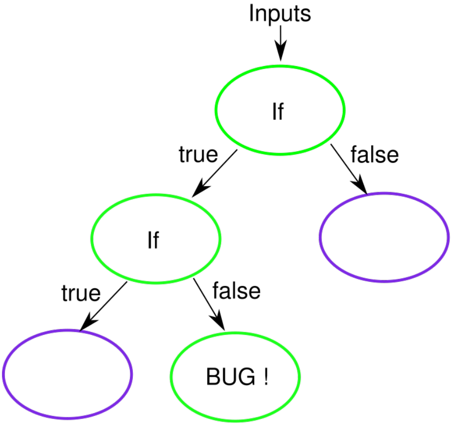
Results

- Custom model-checker
- 16 reachability algorithms supported

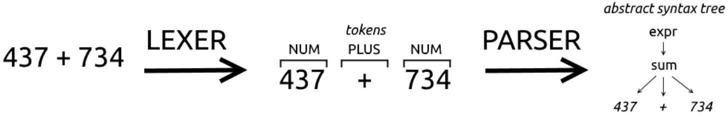
Symbolic approach : SMT

Find output using a logic formula

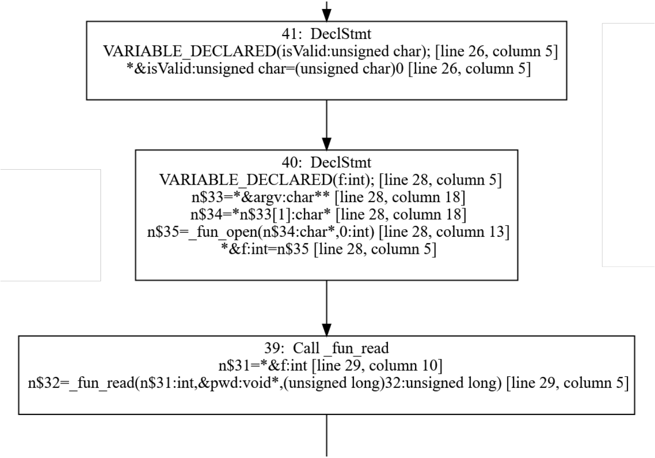
CFG and BFS



Compiler


437 + 734
LEXER
tokens
NUM
437
PLUS
+
NUM
734
PARSER
abstract syntax tree
expr
sum
437
+
734

Smallfoot Intermediate Language

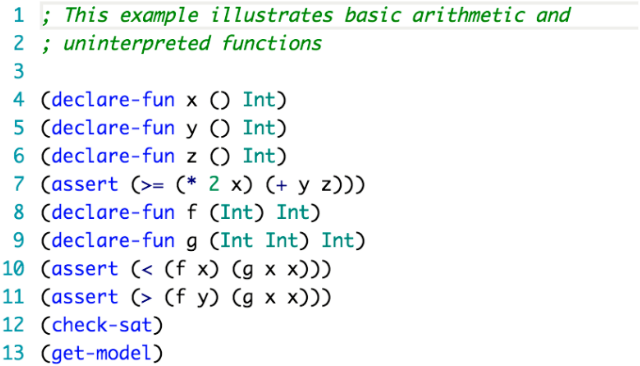


```
41: DeclStmt
VARIABLE_DECLARED(isValid:unsigned char); [line 26, column 5]
*&isValid:unsigned char=(unsigned char)0 [line 26, column 5]
```

```
40: DeclStmt
VARIABLE_DECLARED(f:int); [line 28, column 5]
n$33=*&argv:char** [line 28, column 18]
n$34=*n$33[1]:char* [line 28, column 18]
n$35=_fun_open(n$34:char*,0:int) [line 28, column 13]
*&f:int=n$35 [line 28, column 5]
```

```
39: Call _fun_read
n$31=*&f:int [line 29, column 10]
n$32=_fun_read(n$31:int,&pwd:void*,(unsigned long)32:unsigned long) [line 29, column 5]
```

Formula



```
; This example illustrates basic arithmetic and
; uninterpreted functions

(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)
(assert (>= (* 2 x) (+ y z)))
(declare-fun f (Int) Int)
(declare-fun g (Int Int) Int)
(assert (< (f x) (g x x)))
(assert (> (f y) (g x x)))
(check-sat)
(get-model)
```

Input Language

Restriction :

- No pointer
- Only int
- No exterior function
- Only use argv as a vector of int

Buffer Overflow example

Does not fit our example :

- Array
- char* (String)
- Standard functions : memcpy

Compiler

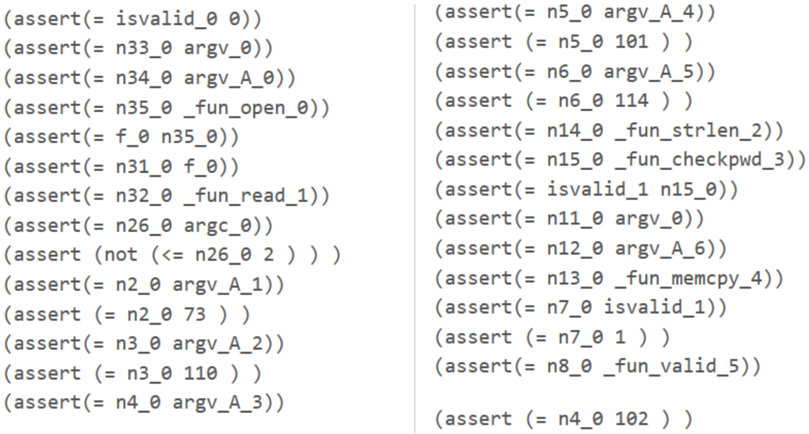
Does not fit our example :

- Array
- char* (String)
- Standard functions : memcpy

Principle

What we won't understand won't hurt us.

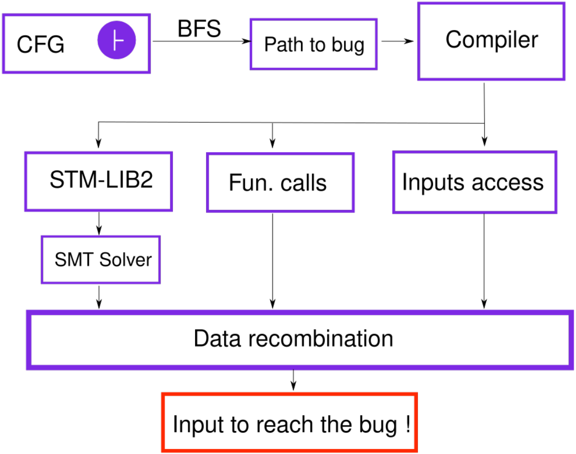
SMT Output



```
(assert(= isvalid_0 0))
(assert(= n33_0 argv_0))
(assert(= n34_0 argv_A_0))
(assert(= n35_0 _fun_open_0))
(assert(= f_0 n35_0))
(assert(= n31_0 f_0))
(assert(= n32_0 _fun_read_1))
(assert(= n26_0 argc_0))
(assert (not (<= n26_0 2 ) ) )
(assert(= n2_0 argv_A_1))
(assert (= n2_0 73 ) )
(assert(= n3_0 argv_A_2))
(assert (= n3_0 110 ) )
(assert(= n4_0 argv_A_3))
```

```
(assert(= n5_0 argv_A_4))
(assert (= n5_0 101 ) )
(assert(= n6_0 argv_A_5))
(assert (= n6_0 114 ) )
(assert(= n14_0 _fun_strlen_2))
(assert(= n15_0 _fun_checkpwd_3))
(assert(= isvalid_1 n15_0))
(assert(= n11_0 argv_0))
(assert(= n12_0 argv_A_6))
(assert(= n13_0 _fun_memcpy_4))
(assert(= n7_0 isvalid_1))
(assert (= n7_0 1 ) )
(assert(= n8_0 _fun_valid_5))

(assert (= n4_0 102 ) )
```

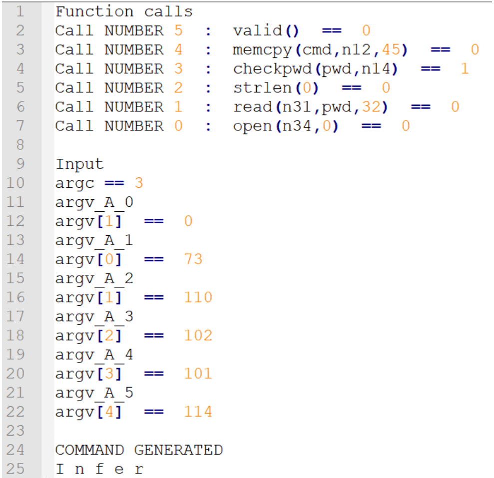


CFG
⊦
BFS
Path to bug
Compiler
STM-LIB2
Fun. calls
Inputs access
SMT Solver
Data recombination
Input to reach the bug !

SMT Solver Response

```
{
  'argc_0': 3, '_fun_valid_5': 0, 'n8_0': 0, 'n7_0': 1,
  'isvalid_1': 1, '_fun_memcpy_4': 0, 'n13_0': 0,
  'argv_A_6': 0, 'n12_0': 0, 'argv_0': 0, 'n11_0': 0,
  'n15_0': 1, '_fun_checkpwd_3': 1, '_fun_strlen_2': 0,
  'n14_0': 0, 'n6_0': 114, 'argv_A_5': 114, 'n5_0': 101,
  'argv_A_4': 101, 'n4_0': 102, 'argv_A_3': 102,
  'n3_0': 110, 'argv_A_2': 110, 'n2_0': 73,
  'argv_A_1': 73, 'n26_0': 3,'_fun_read_1': 0,
  'n32_0': 0, '_fun_open_0': 0, 'n31_0': 0, 'f_0': 0,
  'n35_0': 0, 'argv_A_0': 0, 'n34_0': 0,
  'n33_0': 0, 'isvalid_0': 0, 'pwd_0': 0
}
```

Evaluation



```
Function calls
Call NUMBER 5  :  valid()  ==  0
Call NUMBER 4  :  memcpy(cmd,n12,45)  ==  0
Call NUMBER 3  :  checkpwd(pwd,n14)  ==  1
Call NUMBER 2  :  strlen(0)  ==  0
Call NUMBER 1  :  read(n31,pwd,32)  ==  0
Call NUMBER 0  :  open(n34,0)  ==  0

Input
argc == 3
argv_A_0
argv[1]  ==  0
argv_A_1
argv[0]  ==  73
argv_A_2
argv[1]  ==  110
argv_A_3
argv[2]  ==  102
argv_A_4
argv[3]  ==  101
argv_A_5
argv[4]  ==  114

COMMAND GENERATED
I n f e r
```

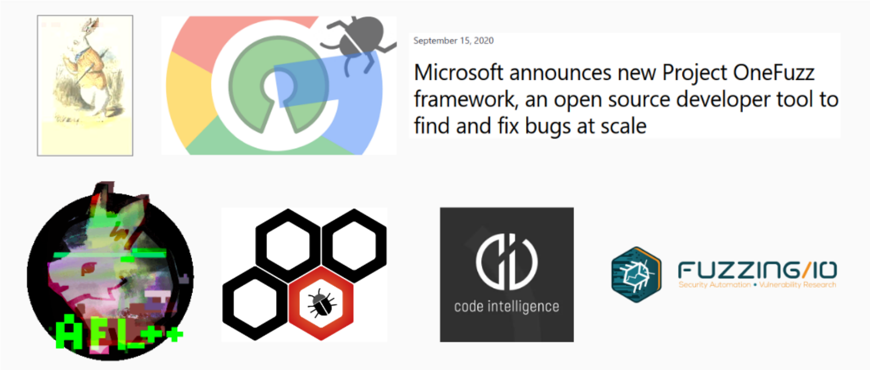
- ≈ 0.30 seconds

Fuzzing Technique

Third approach: Directed Fuzzing

Fuzzing technique

Idea: Generate random inputs in the hope of crashing programs

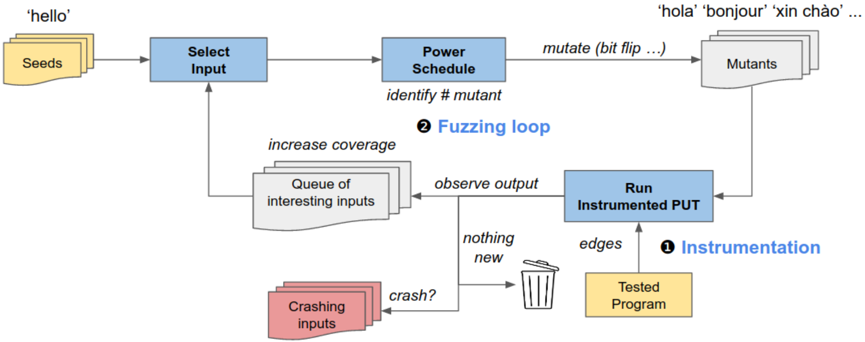


September 15, 2020

Microsoft announces new Project OneFuzz framework, an open source developer tool to find and fix bugs at scale

> **100 fuzzers** (Google, Facebook, Microsoft) in recent years

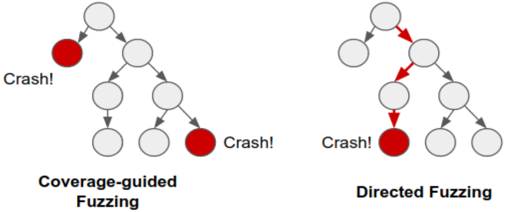
Coverage-guided greybox fuzzing [AFL, libFuzzer]



Goal: Cover as many paths as possible

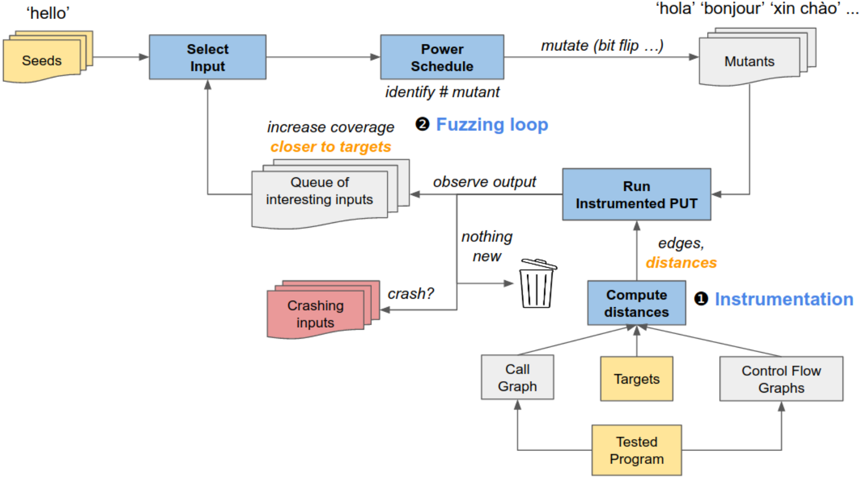
- Feedback: code coverage (e.g., lines, branches)
- Mutation operators: bitflip, insert/delete/overwrite bytes ...

Intuitions of directed approach



- **Goal**: Reach predefined targets (e.g., recent code changes, vulnerable functions, patches, static reports)
- New distance-based input metric
- Favor inputs that are "closer" to targets
  - Select them more often
  - Produce more mutants from those inputs

Directed greybox fuzzing [AFLGo]

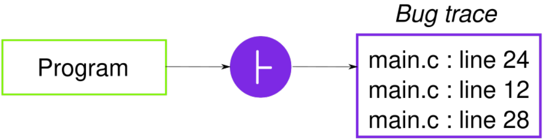


"Directed Greybox Fuzzing", M. Böhme et al, CCS'17

Putting it all together

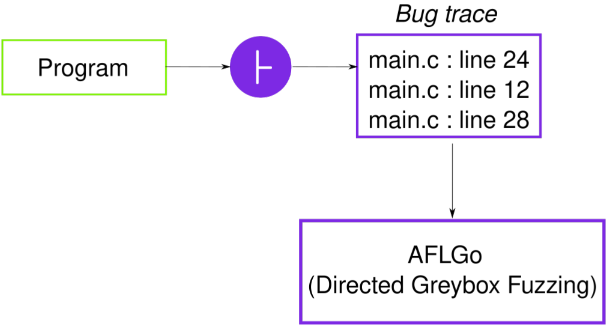
Program

Putting it all together



main.c : line 24
main.c : line 12
main.c : line 28

Putting it all together



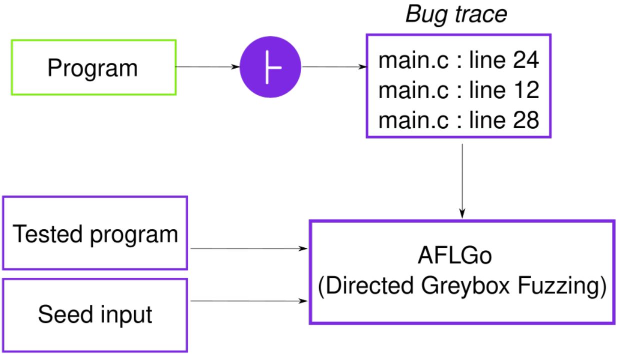
Program

Bug trace

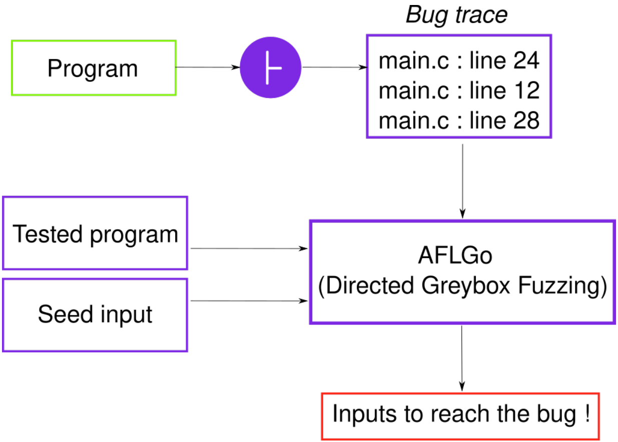
main.c : line 24
main.c : line 12
main.c : line 28

AFLGo
(Directed Greybox Fuzzing)

Putting it all together



Putting it all together



Program

Bug trace

main.c : line 24
main.c : line 12
main.c : line 28

Tested program

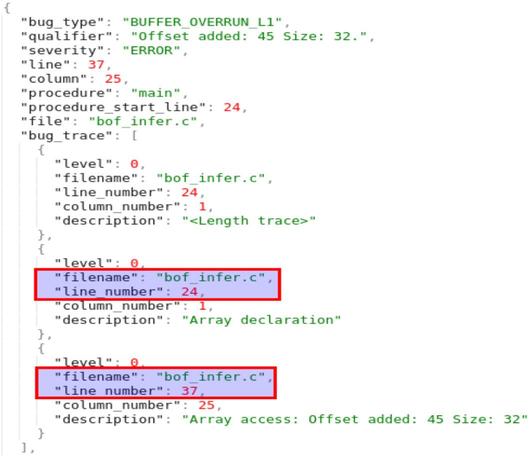
Seed input

AFLGo
(Directed Greybox Fuzzing)

Inputs to reach the bug !

Implementation

- Python script to parse Infer's output “report.json"
  - Targets: ['bof_infer.c:24', 'bof_infer.c:37']
- Generate a script to run AFLGo `https://github.com/aflgo/aflgo`



```
{
  "bug_type": "BUFFER_OVERRUN_L1",
  "qualifier": "Offset added: 45 Size: 32.",
  "severity": "ERROR",
  "line": 37,
  "column": 25,
  "procedure": "main",
  "procedure_start_line": 24,
  "file": "bof_infer.c",
  "bug_trace": [
    {
      "level": 0,
      "filename": "bof_infer.c",
      "line_number": 24,
      "column_number": 1,
      "description": "<Length trace>"
    },
    {
      "level": 0,
      "filename": "bof_infer.c",
      "line_number": 24,
      "column_number": 1,
      "description": "Array declaration"
    },
    {
      "level": 0,
      "filename": "bof_infer.c",
      "line_number": 37,
      "column_number": 25,
      "description": "Array access: Offset added: 45 Size: 32"
    }
  ],
```

Evaluation

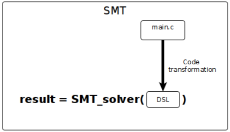
Run the fuzzer 5 times against the buffer overflow example (due to randomness)

Run	**Time-to-Exposure**
1	4m 44s
2	18m 4s
3	17m 11s
4	14m 50s
5	18m 13s
Average	14 m 36 s

Conclusions and perspectives

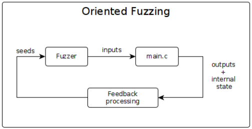
Results comparison

SMT



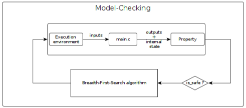
- + Quick
- + Function complexity
- – Semantic limitations

Fuzzing



- + Can be efficent
- – Does not always find the solution

Model Checking



- + Respect the semantic
- – Combinatorial explosion

Other Approaches – Hybrid methods

- Hybrid methods
  - Prune interesting branches with Fuzzing + Heuristics
  - SMT on subproblems

Future Work – Automatic exploit generation

- We can trigger bugs
- How to automatize their exploits?
- Create an attack model
- Find all possible exploits
- → FULLY AUTOMATED EXPLOITS!

Convert vulnerabilities to bug

```
void onBufferOverflow(env, bufsz, entrypoint)
  if(canOverWriteRip())
    addExploit("Can rewrite on rip!");
  for(var=0 to env.nb_vars)
    if( var.rewrittable && var.used)
      addExploit("Can Hijack the program!");
  /* [...]  */
```

Next steps to improve the framework

- General
  - Test on code with several path to the same bug
- Fuzzing
  - Resolve magic-bytes comparisons, (e.g. `strcmp`)
  - Distance computation is long with large programs
- SMT
  - Add more types
  - Support arbitrary memory accesses
  - Improve function with hooks
- Model Checker
  - Partial order reduction
  - Hardware model-checking

It works on my machine

Thank you !