Offset-Aware Mutation based Fuzzing for Buffer Overflow Vulnerabilities: Few Preliminary Results

Sanjay Rawat, Laurent Mounier

Verimag, University of Grenoble
Grenoble, France

Second International Workshop on Security Testing, Berlin
March 25, 2011

1 Overview

1 Overview

2 Tainted Path Generation and Code Instrumentation

- Tainted Dependency Sequence (TDS)

1. Overview
2. Tainted Path Generation and Code Instrumentation
   - Tainted Dependency Sequence (TDS)
3. Evolutionary Algorithm (EA)
   - Why EA?
   - A Typical EA

1 Overview

2 Tainted Path Generation and Code Instrumentation
- Tainted Dependency Sequence (TDS)

3 Evolutionary Algorithm (EA)
- Why EA?
- A Typical EA

4 String Inputs Generation
- Input Generation
- Crossover and Mutation

1. Overview
2. Tainted Path Generation and Code Instrumentation
   - Tainted Dependency Sequence (TDS)
3. Evolutionary Algorithm (EA)
   - Why EA?
   - A Typical EA
4. String Inputs Generation
   - Input Generation
   - Crossover and Mutation
5. Improved Mutation: Offset-aware Mutation
   - Again the maze analogy
   - Pseudo-code of the Algorithm

1 Overview

2 Tainted Path Generation and Code Instrumentation
- Tainted Dependency Sequence (TDS)

3 Evolutionary Algorithm (EA)
- Why EA?
- A Typical EA

4 String Inputs Generation
- Input Generation
- Crossover and Mutation

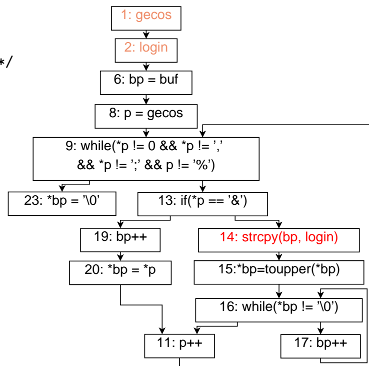
5 Improved Mutation: Offset-aware Mutation
- Again the maze analogy
- Pseudo-code of the Algorithm

6 Experimental Results
- Experimental Results

1 Overview
2 Tainted Path Generation and Code Instrumentation
- Tainted Dependency Sequence (TDS)
3 Evolutionary Algorithm (EA)
- Why EA?
- A Typical EA
4 String Inputs Generation
- Input Generation
- Crossover and Mutation
5 Improved Mutation: Offset-aware Mutation
- Again the maze analogy
- Pseudo-code of the Algorithm
6 Experimental Results
- Experimental Results
7 Conclusions and Future Work
- Conclusions and Future Work

Overview of the Problem

```
void buildfname(char *gecos, char *login, char *buf)
{
/* gecos - tainted; login - tainted; buf - untainted */
  char bp = buf; p = gecos;
  char* p;
    /* fill in buffer */
  while ( *p != '\0' && *p != ',' && *p != ';'
        && *p != '%') {
        if (*p == '&') {
           (void) strcpy(bp, login); /* BAD */
           *bp = toupper(*bp);
           while (*bp != '\0')
                bp++;
           } else {
              *bp++ = *p;
           }
        p++;
  }
  *bp = '\0';
}
```



Overview of the Approach

- It is a light weight *smart* fuzzer with a focus on BO vulnerabilities.

Overview of the Approach

- It is a light weight *smart* fuzzer with a focus on BO vulnerabilities.
- Performs a static analysis of source code (C code) to calculate tainted path (slice: source to sink).

Overview of the Approach

- It is a light weight *smart* fuzzer with a focus on BO vulnerabilities.
- Performs a static analysis of source code (C code) to calculate tainted path (slice: source to sink).
- Makes use of evolutionary strategies to generate inputs (dynamic analysis by code instrumentation).

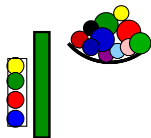
Overview of the Approach

- It is a light weight *smart* fuzzer with a focus on BO vulnerabilities.
- Performs a static analysis of source code (C code) to calculate tainted path (slice: source to sink).
- Makes use of evolutionary strategies to generate inputs (dynamic analysis by code instrumentation).
- Provides information on exploitability of the vulnerability (by inspecting stack).

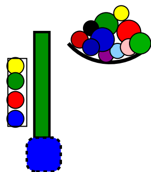
Overview of the Approach

- It is a light weight *smart* fuzzer with a focus on BO vulnerabilities.
- Performs a static analysis of source code (C code) to calculate tainted path (slice: source to sink).
- Makes use of evolutionary strategies to generate inputs (dynamic analysis by code instrumentation).
- Provides information on exploitability of the vulnerability (by inspecting stack).
- Consider an analogy...

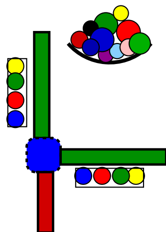
Maze Analogy



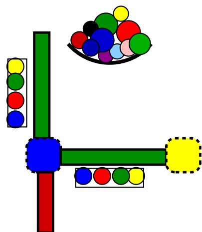
Maze Analogy



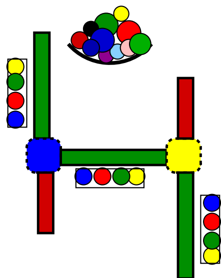
Maze Analogy



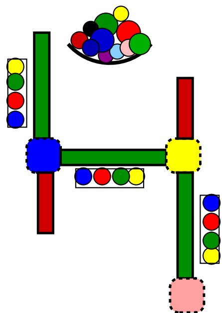
Maze Analogy



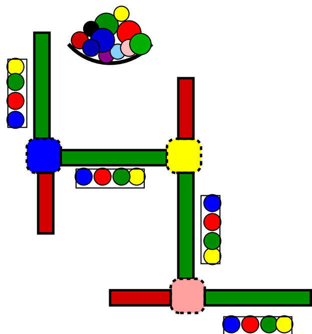
Maze Analogy



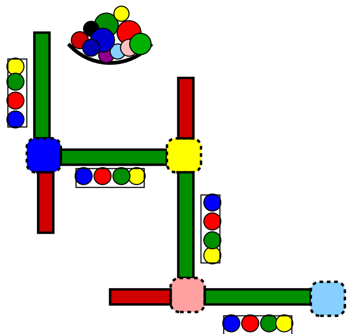
Maze Analogy



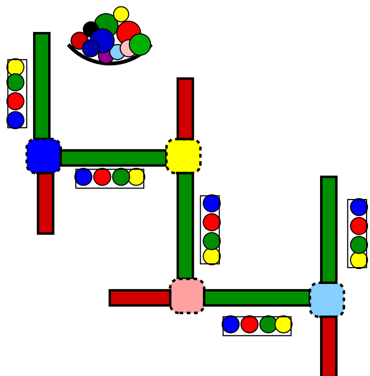
Maze Analogy



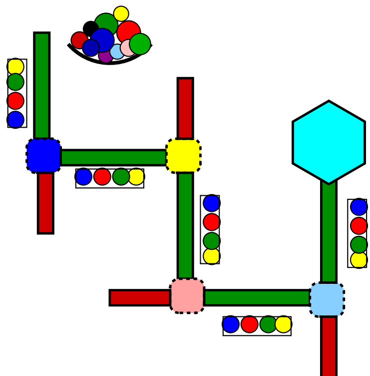
Maze Analogy



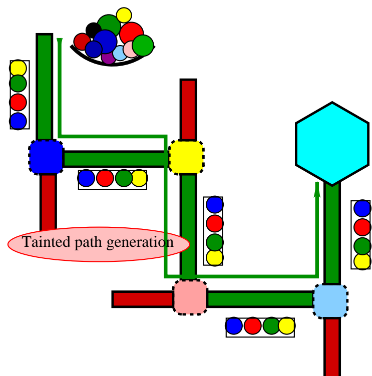
Maze Analogy



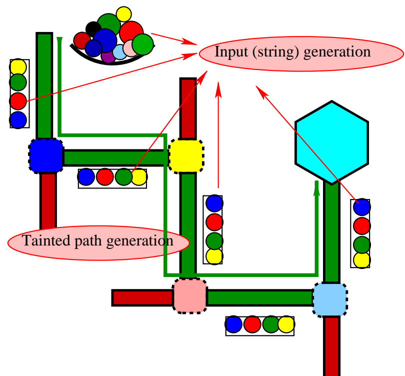
Maze Analogy



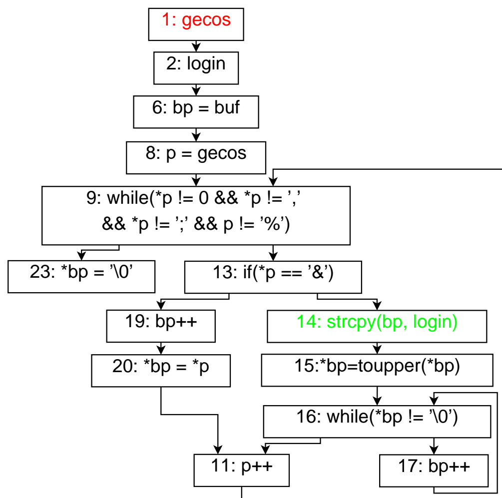
Maze Analogy



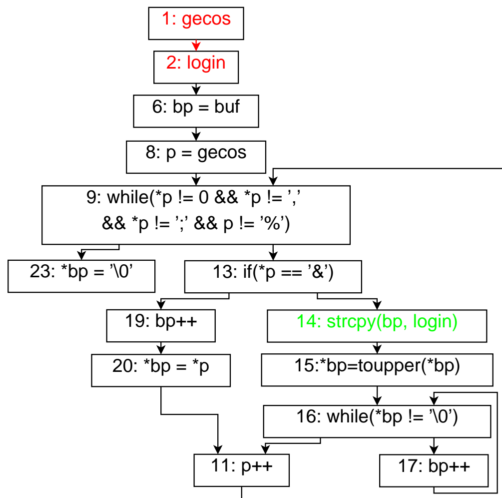
Maze Analogy



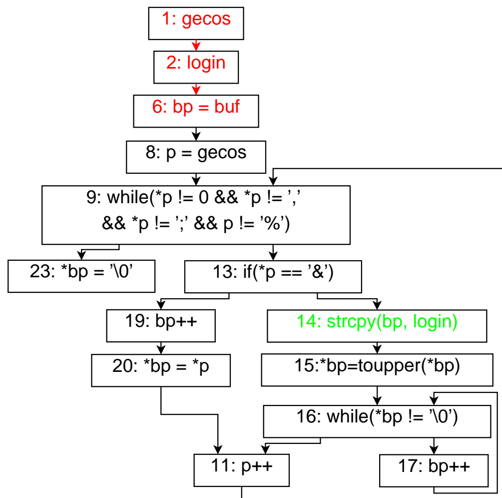
What is a tainted path (slice)?



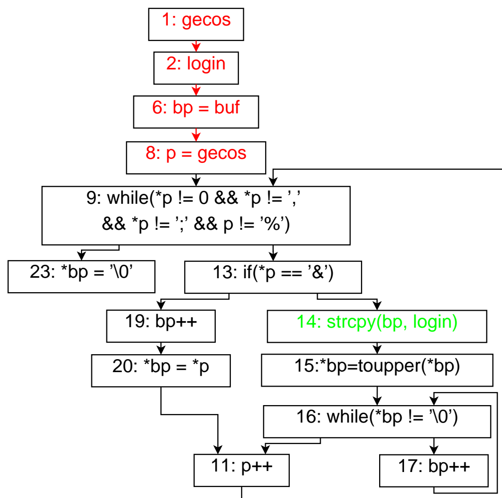
What is a tainted path (slice)?



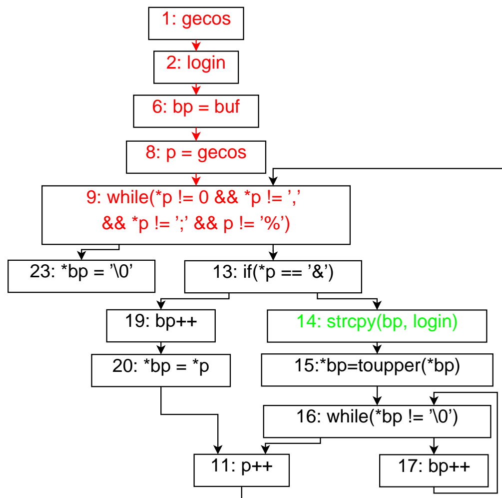
What is a tainted path (slice)?



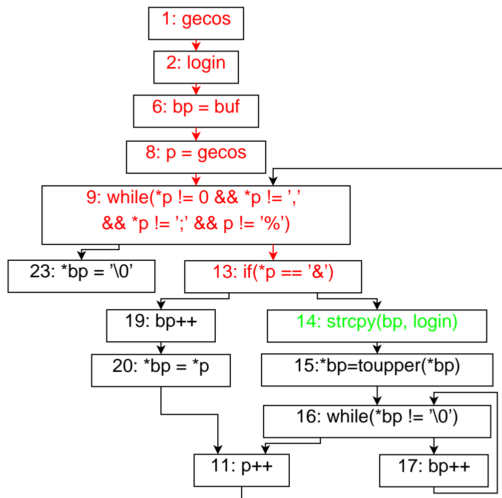
What is a tainted path (slice)?



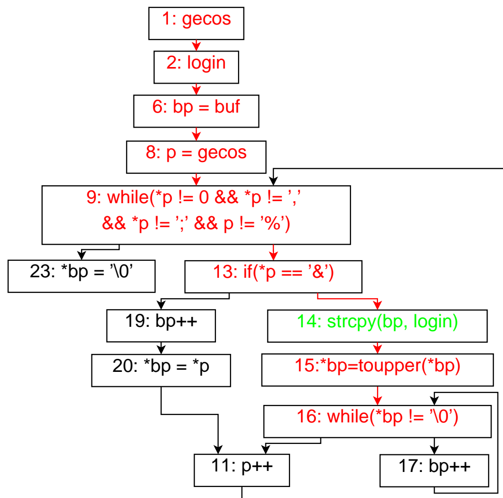
What is a tainted path (slice)?



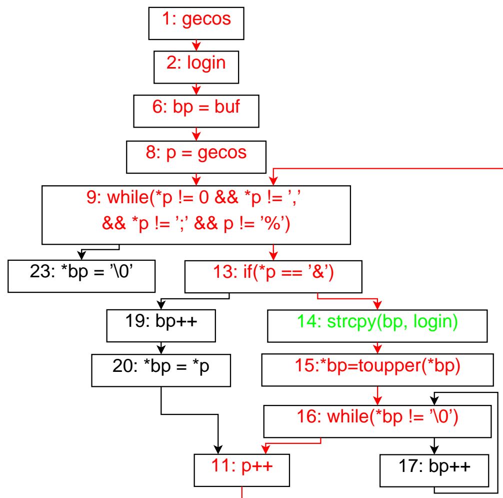
What is a tainted path (slice)?



What is a tainted path (slice)?



What is a tainted path (slice)?



Tainted Path Generation and Code Instrumentation

- Makes use of a previous study on *Taint Dependency Sequences* (TDS) [4] to choose the paths to be exercised by the fuzzer.
- TDSs identify a sequence of instructions whose execution can be influenced by user inputs.

Tainted Path Generation and Code Instrumentation

- Makes use of a previous study on *Taint Dependency Sequences* (TDS) [4] to choose the paths to be exercised by the fuzzer.
- TDSs identify a sequence of instructions whose execution can be influenced by user inputs.
- Each TDS $t = < l_1, l_2, \ldots, l_n >$ associated to a variable v is a sequence of program locations l_i that a program execution path should traverse in order to reach l_n with an input-dependent value assigned to v.

Tainted Path Generation and Code Instrumentation

- Makes use of a previous study on *Taint Dependency Sequences* (TDS) [4] to choose the paths to be exercised by the fuzzer.
- TDSs identify a sequence of instructions whose execution can be influenced by user inputs.
- Each TDS $t = < l_1, l_2, \ldots, l_n >$ associated to a variable v is a sequence of program locations l_i that a program execution path should traverse in order to reach l_n with an input-dependent value assigned to v.
- Based on the labels $< l_1, l_2, \ldots, l_n >$, the source code is instrumented at l_ith line and compiled binary is used for further analysis.

Evolutionary Algorithm (EA)

Objective: Generate input that produces an execution sequence corresponding to the chosen path (TDS).

- *Smart* fuzzing can be considered as a *search problem*.
- We make use of *evolutionary strategies* (ES) to generate candidate inputs.

A Typical EA

Algorithm 1 Pseudo-code of a typical evolutionary algorithm

INITIALIZE population with *random* candidates
repeat
    SELECT parents
    RECOMBINE parents to generate children (crossover)
    MUTATE offsprings
    EVALUATE new candidates with fitness function
    SELECT fittest candidates for the next population
until TERMINATION CONDITION is met
return BEST Solution, if found

ES differs from GA mainly because of the introduction of *strategies parameters*, i.e. each individual x is mutated to x' by performing following operation:

$$x' = x + N(0, \sigma)$$

Initial Population

- Considers string inputs.
- Regular expression is used to generate initial set of inputs.
- We start with a set of inputs $I := \langle i_1, i_2, ..., i_m \rangle$.

We construct a frequency matrix $freq = \begin{pmatrix} \cdots \\ f_{ij} \\ \cdots \end{pmatrix}$, where f_{ij} is the frequency of l_j for the input $i \in I$. This matrix is used later while evaluating fitness.

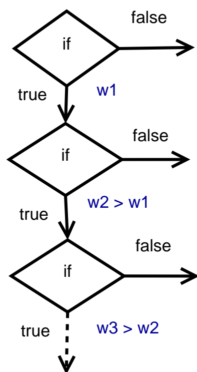
Fitness Function:

- For a given TDS $t =< l_1, l_2, \ldots, l_n >$, fitness value of each $i \in I$ (set of inputs), we define fitness function F_i as follows:

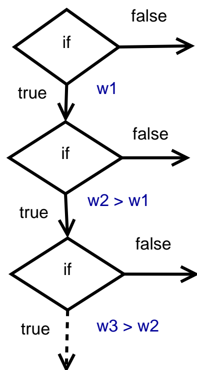
$$F_i = \sum_{j=1}^{n} w_j \times f_{ij} \qquad (1)$$

  where $w_j \in W$ (set of weights) corresponds to weight associated with $l_j \in t$ and f_{ij} is the frequency of l_j for $i \in I$.
- F_i is affected by the structure of the program i.e. rare statements, nested conditional statements etc.

Fitness Function: Effect of nested statements



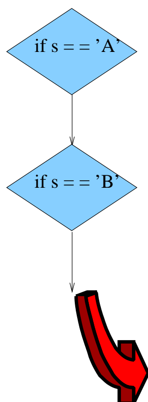
Fitness Function: Effect of nested statements



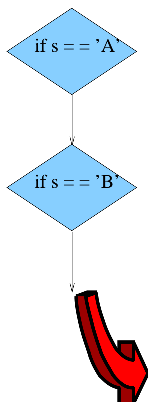
Frequency Spectrum Analysis

A careful analysis reveals a lot about execution trace of the program by means of frequencies corresponding to t_js (Thomas Ball *et. al.* [2][3][6]).

String generation: an example

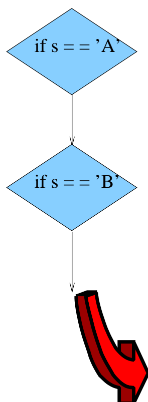


String generation: an example



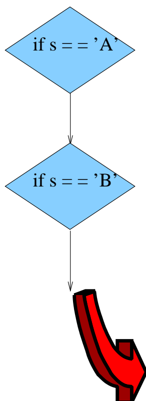
- s1 = abcAxy

String generation: an example



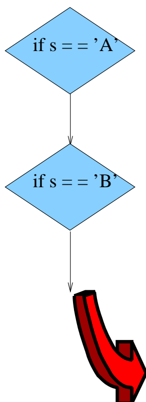
- s1 = abc A xy
- s2 = xy B jka

String generation: an example



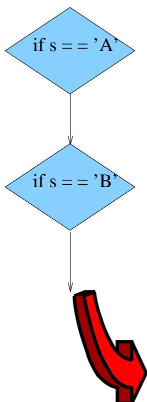
- s1 = abc A xy
- s2 = xy B jka
- Desired string = $f(s1 + s2) \supseteq \{A, B\}$

String generation: an example



- s1 = abc A xy
- s2 = xy B jka
- Desired string = $f(s1 + s2) \supseteq \{A, B\}$
- Set-theory is used for such a purpose, especially *Symmetric Difference*.

String generation: an example



- s1 = abc A xy
- s2 = xy B jka
- Desired string = $f(s1 + s2) \supseteq \{A, B\}$
- Set-theory is used for such a purpose, especially *Symmetric Difference*.
- For characters (constraints) learning, we maintain a set M which is used to generate new strings.

Crossover and Mutation

Crossover

Crossover is performed by interchanging substrings of parent string (using n-point crossover).

Crossover and Mutation

Crossover

Crossover is performed by interchanging substrings of parent string (using n-point crossover).

Mutation

- Mutation is based on *set-theoretic* definitions. We try to learn from bad inputs too!!!
- This is the main step which is responsible for learning (light) constraints automatically.

Crossover and Mutation

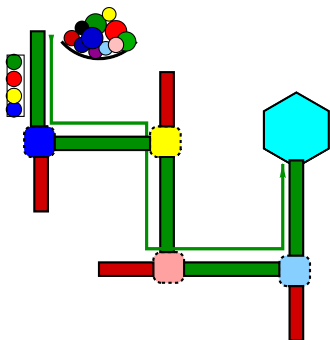
Crossover

Crossover is performed by interchanging substrings of parent string (using n-point crossover).

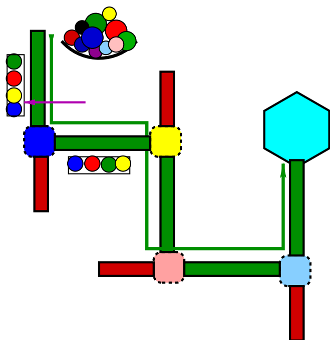
Mutation

- Mutation is based on *set-theoretic* definitions. We try to learn from bad inputs too!!!
- This is the main step which is responsible for learning (light) constraints automatically.
- Under mutation, each input is manipulated either by adding or deleting characters from a set M (at pseudo-random positions!).

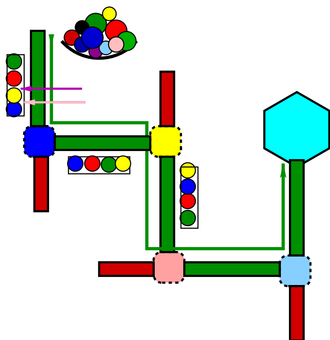
Maze example, again!



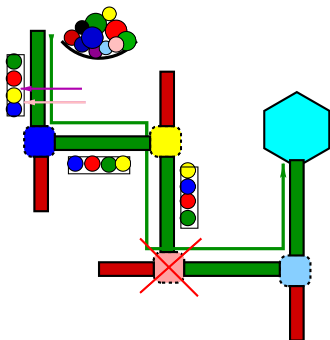
Maze example, again!



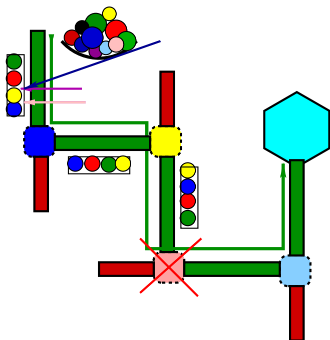
Maze example, again!



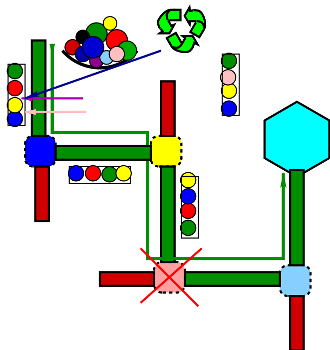
Maze example, again!



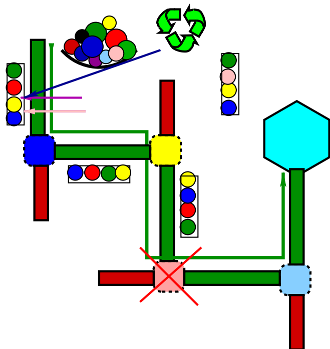
Maze example, again!



Maze example, again!

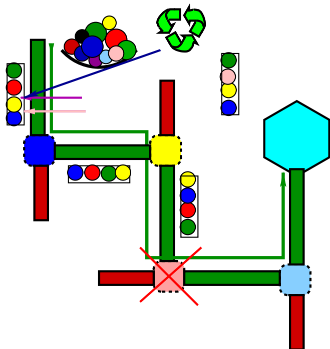


Maze example, again!



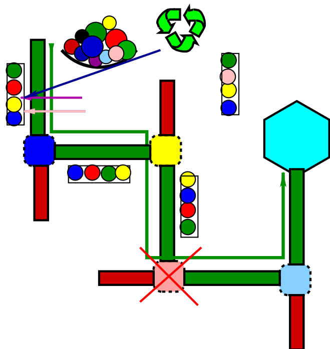
- s1 = abc A xy; s2 = xy B jka

Maze example, again!



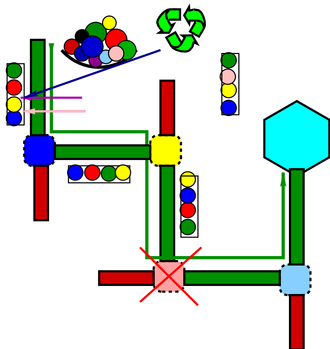
- s1 = abc A xy; s2 = xy B jka
- $M = s1 \bigtriangleup s1$ = bcABjk

Maze example, again!



- s1 = abc A xy; s2 = xy B jka
- $M = s1 \triangle s1 =$ bcABjk
- offset-mutation(s1,M,4)

Maze example, again!



- s1 = abc A xy; s2 = xy B jka
- $M = s1 \triangle s1 =$ bcABjk
- offset-mutation(s1,M,4)
- abcAbc...; abcAk..; abcA B ...

Algorithm 2 Pseudo-code of the proposed EA based approach

```
INITIALIZE population I with λ random candidates
TERMINATION CONDITION: program crashed or 1000 iterations are over
M ← {}
repeat
    EXECUTE program with I, RECORD offset
    CALCULATE fitness
    CALCULATE diff1, diff2 and notImp (till 2 generations)
    UPDATE M as M ← M ∪ (diff1 △ diff2)
    if N(0, σ) > large_number then
        M ← M \ notImp
    end if
    if N(0, σ) > large_number then
        M ← M ∪ new_chars
    end if
    SELECT μ fittest candidates for the next population
    for each of μ candidates do
        if probability_mutation then
            for i = 1 to λ/μ do
                MUTATE candidate using M and offset
            end for
        else
            for i = 1 to λ/μ do
                SELECT two parents randomly
                RECOMBINE parents to generate child
            end for
        end if
    end for
    UPDATE I with newly generated candidates
until TERMINATION CONDITION is met
```

High Level Flow Diagram

Experimental Results

- We experiment on Verisec dataset [1]. We choose two (representative!)programs which do string manipulations on inputs.
- We compare results of our algorithm with blind fuzzing approach and a GA based approach without offset based mutation.

Table: Experimental Results. Legend: A1 (resp. A2)- evolutionary strategy with (resp. without) offset-aware mutation, A3- random fuzzing

S.No.	Application	Name	Constraints	A3	A2	A1
1	sendmail	mime_fromqp	'=n'	370	154	27
2	edbrowse	ftpls	'-- '	*	*	290

Conclusions

This work:

- presents preliminary results on our ongoing work [5].
- approximates constraints (characters) on inputs based on execution trace.
- proposes a new mutation technique for generating strings based inputs, called *offset-aware mutation*.
- obtains experimental results which are significantly better than the previous results.

Future work

A lot!!

- Static analysis on executables to generate tainted path.
- During static analysis, learn dependency on inputs and *critical nodes* in tainted path to enhance mutation.
- Do a rigorous analysis on the crash to infer exploitability of the vulnerability.

Verisec.
http://se.cs.toronto.edu/index.php/Verisec_Suite.

Thomas Ball.
What's in a region? or computing control dependence regions in near-linear time for reducible control flow.
ACM Letters on Programming Languages and Systems, 2(1-4):1–16, 1993.

Thomas Ball.
The concept of dynamic analysis.
In *Proc. of 7th European Software Engineering Conference*, volume 1687 of *LNCS*, pages 216–234. Springer, 1999.

Dumitru Ceara, Laurent Mounier, and Marie-Laure Potet.
Taint dependency sequences: A characterization of insecure execution paths based on input-sensitive cause sequences.
In *Proc. of the IEEE Int. workshop MDV'10*. IEEE Computer Society, 2010.

Sanjay Rawat and Laurent Mounier.
An evolutionary computing approach for hunting buffer overflow vulnerabilities: A case of aiming in dim light.
In *Proc of sixth EC2ND 2010*. IEEE Computer Society, 2010.

Thomas W. Reps.
The use of program profiling for software testing.
In *Informatik 97, GI Jahrestagung*, pages 4–16, 1997.

Thank You!!
&

Sanjay.Rawat@imag.fr
Laurent.Mounier@imag.fr