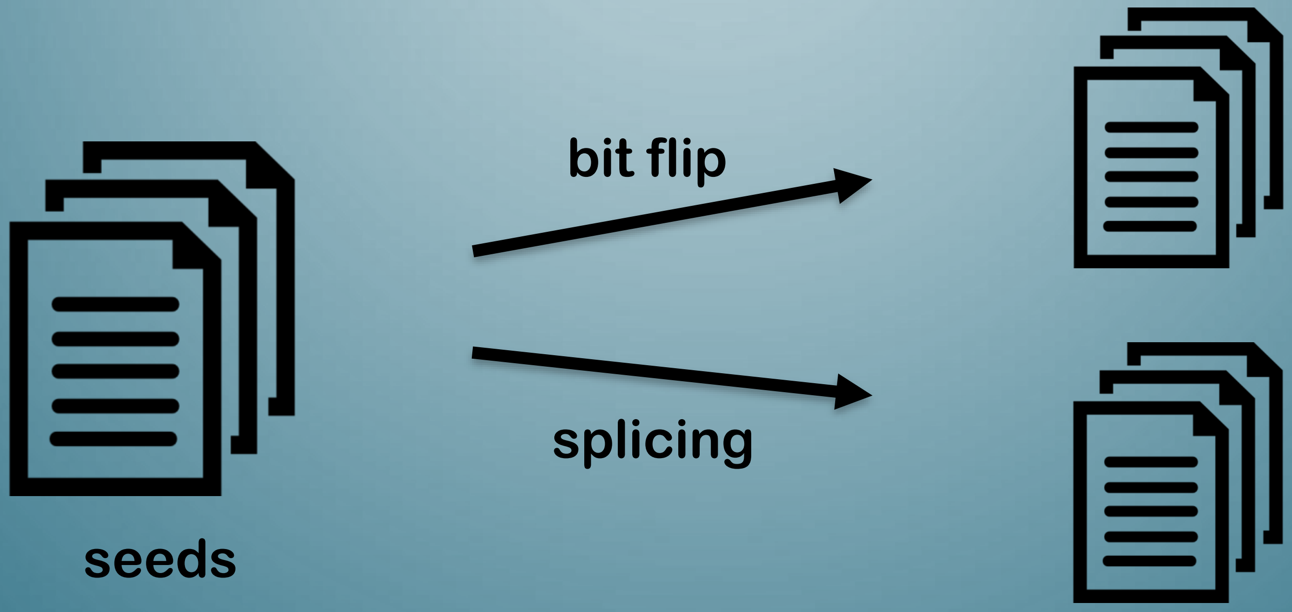
NAUTILUS: FISHING FOR DEEP BUGS WITH GRAMMARS

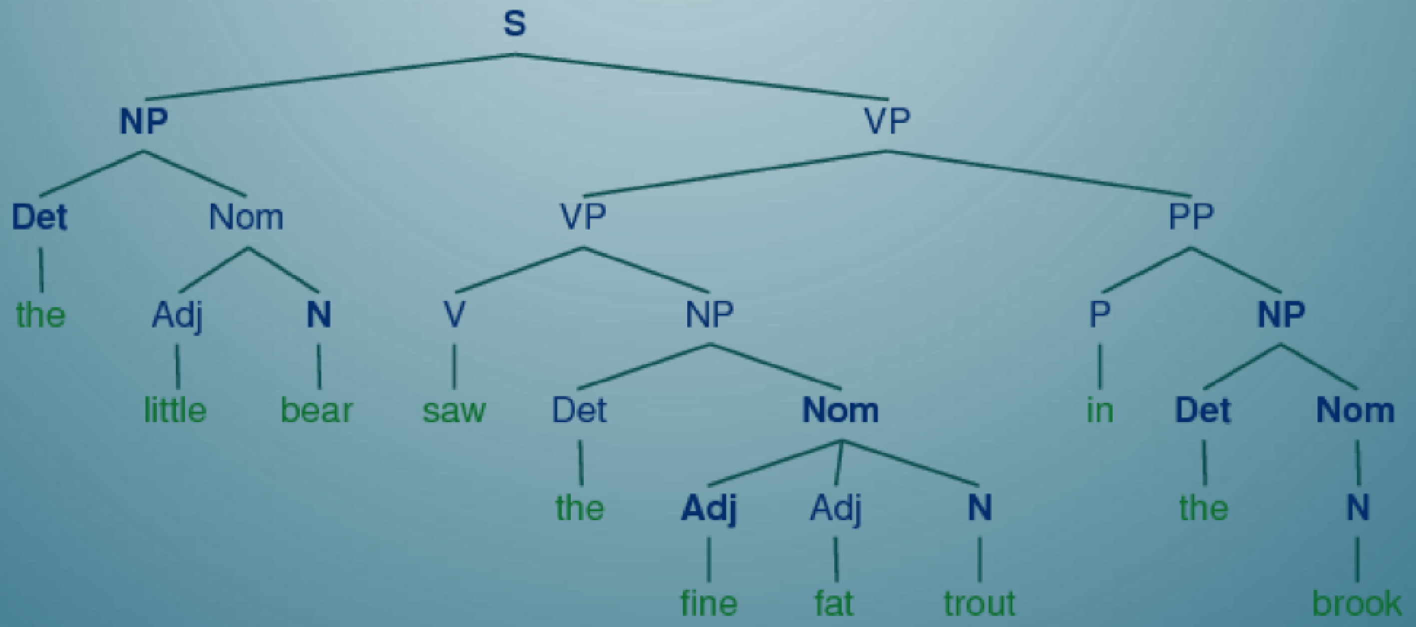
20204222 강우석

FUZZING

MUTATION



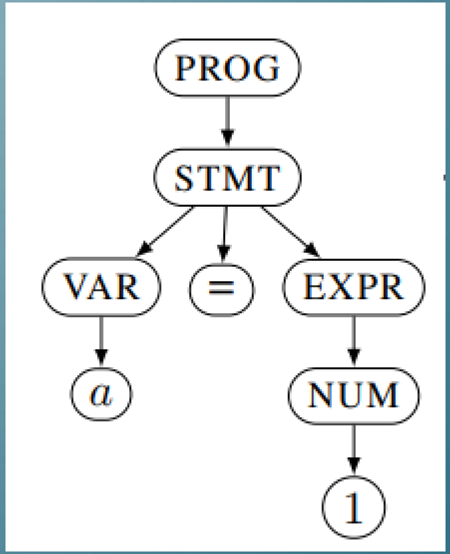
GENERATION


S
NP
VP
Det
Nom
VP
PP
the
Adj
N
V
NP
P
NP
little
bear
saw
Det
Nom
in
Det
Nom
the
Adj
Adj
N
the
N
fine
fat
trout
brook

CONTEXT-FREE GRAMMARS

- (N, T, R, S)

N: {PROG, STMNT, EXPR, VAR, NUMBER}
T : { a , 1 , 2 , = , return 1}
R: {
PROG → STMT (1)
PROG → STMT ; PROG (2)
STMT → return 1 (3)
STMT → VAR = EXPR (4)
VAR → a (5)
EXPR → NUMBER (6)
EXPR → EXPR + EXPR (7)
NUMBER → 1 (8)
NUMBER → 2 (9)
}
S : PROG

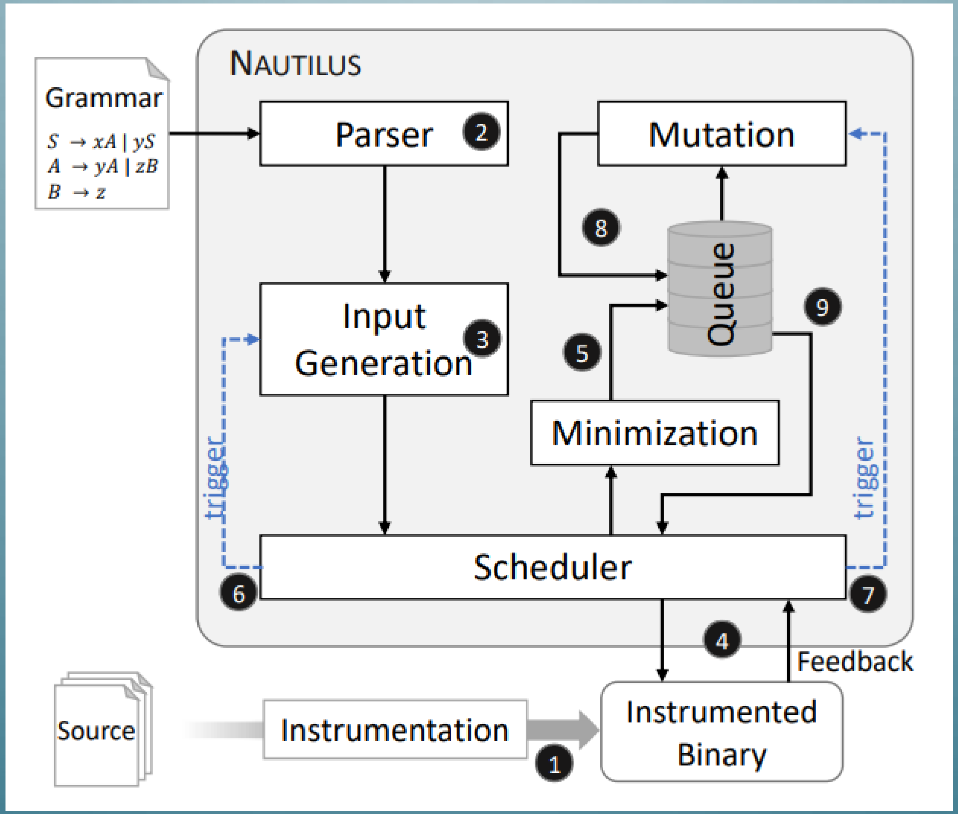


a = 1

CHANLLENGES

- C1 : Generation of syntactically and semantically valid inputs.
- C2 : Independence from corpora.
- C3 : High coverage of target functionality.
- C4 : Good performance.

HIGH-LEVEL VIEW



GENERATION

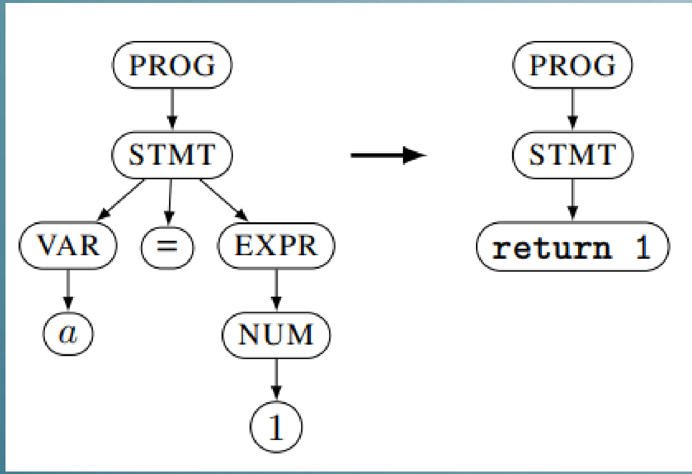
N: {PROG, STMNT, EXPR, VAR, NUMBER}
T : { a , 1 , 2 , = , return 1}
R: {
- PROG → STMT (1)
- PROG → STMT ; PROG (2)
- STMT → return 1 (3)
- STMT → VAR = EXPR (4)
- VAR → a (5)
- EXPR → NUMBER (6)
- EXPR → EXPR + EXPR (7)
- NUMBER → 1 (8)
- NUMBER → 2 (9)

}
S : PROG

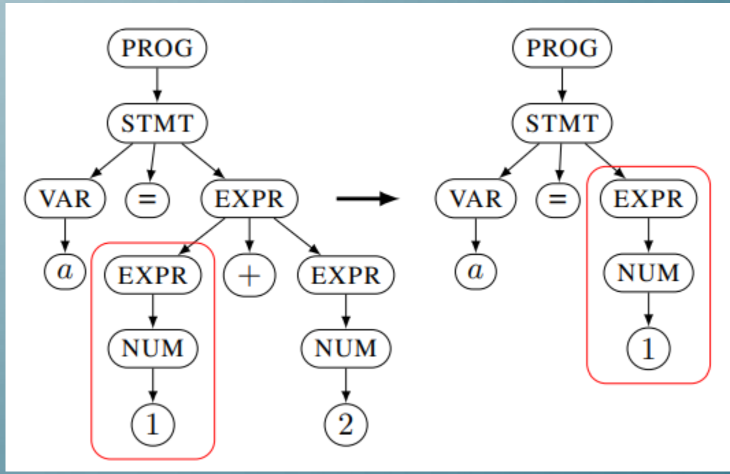
- Naïve generation
  : randomly

- Uniform generation
  : uniformly

MINIMIZATION



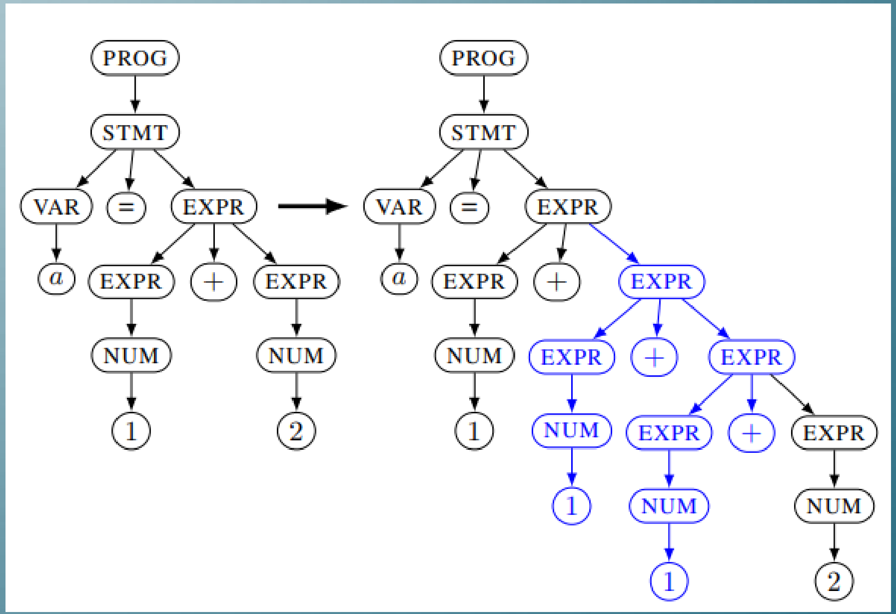
subtree
minimization



recursive
minimization

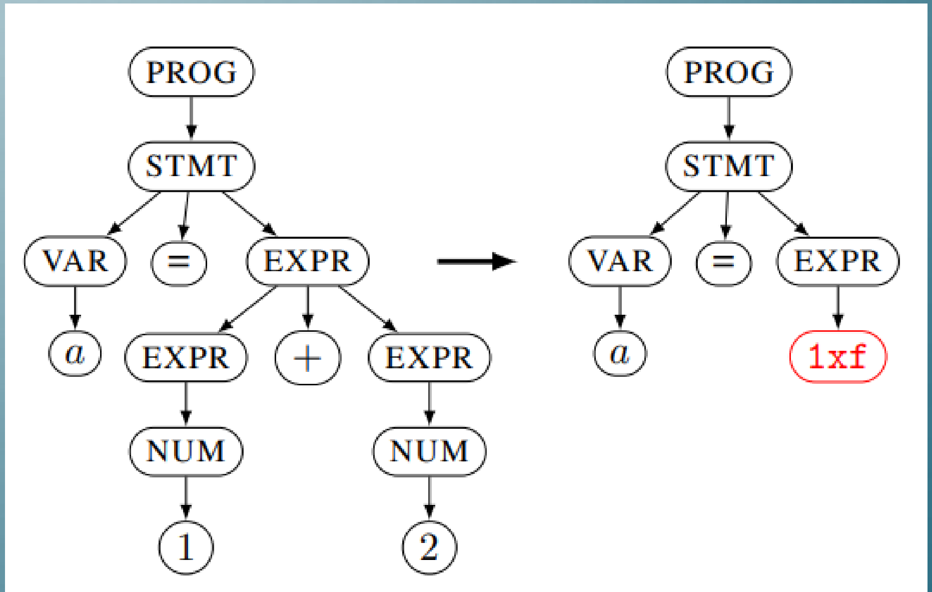
MUTATION

- Random Mutation
- Rules Mutation
- Random Recursive Mutation



MUTATION

- Splicing Mutation
- AFL Mutation



IMPLEMENTATION

1. Target Application Instrumentation
2. ANTLR Parser
3. Preparation Phase
4. Fuzzing Phase

EVALUATION

- Research Question

1. Identify new bugs in real-life applications
2. More efficient than other state-of-art fuzzers
3. Improve the fuzzing efficiency for target applications with highly structed inputs

EVALUATION

4. How much does the use of feedback increase the fuzzing performance?
5. Does our complex generation method actually increase fuzzing performance?
6. How much does each of the mutation methods used contribute to find new paths?

TARGET & ENVIRONMENT

- Ruby (mruby), Lua, PHP, JavaScript (ChakraCore)

CPU : Intel Core i5-650 CPU clocked at 3.2 GHz
RAM : 4GB
OS : Ubuntu 16.04.4 LTS

VULNERABILITIES

Target	Type	CVE
mruby	Use after free caused by integer overflow	CVE-2018-10191
	Use after free in `initialize_copy`	CVE-2018-10199
	Use of uninitialized pointer in `hash.c`	CVE-2018-11743
	Segmentation fault in `mrb_class_real`	CVE-2018-12247
	Segmentation fault in `cfree`	CVE-2018-12248
	Heap buffer overflow caused by `Fiber::transfer`	CVE-2018-12249
	Stack overflow (not fixed yet)	none yet
PHP	Division by Zero triggered by range() caused by a type conversion.	-
	Segmentation fault in `zend_mm_alloc_small`	-
	Stack overflow caused by using too many parameters in a function call.	-
ChakraCore	Wrong number of arguments emitted in JIT-compiled code	-
	Segmentation fault in out-of-memory conditions	-
Lua	Type confusion	-

TABLE I: Vulnerabilities found by NAUTILUS in our targets

AGAINST OTHER FUZZERS

Target	Baseline Coverage	Fuzzer	Mean	Median	**Median New Coverage Found**	Std Deviation	Skewness	Kurtosis
ChakraCore	14.7%	NAUTILUS	34.0%	34.1%	**19.4 pp**	0.60 pp	−0.29	−0.44
		NAUTILUS - No Feedback	18.6%	18.5%	**3.8 pp**	0.24 pp	1.42	0.53
		AFL	15.8%	15.8%	**1.1 pp**	0.27 pp	0.10	−0.58
		IFuzzer	15.9%	16.0%	**1.3 pp**	0.20 pp	−1.08	0.35
mruby	25.7%	NAUTILUS	53.7%	53.8%	**28.1 pp**	1.60 pp	−0.16	−0.38
		NAUTILUS - No Feedback	37.7%	37.8%	**12.1 pp**	0.34 pp	−0.81	−1.01
		AFL	28.0%	27.6%	**1.9 pp**	1.28 pp	2.36	4.20
PHP	2.2%	NAUTILUS	11.7%	12.3%	**10.0 pp**	2.17 pp	−0.65	−0.43
		NAUTILUS - No Feedback	6.1%	6.1%	**3.9 pp**	0.09 pp	0.08	−1.69
		AFL	2.2%	2.2%	**0.0 pp**	0.00 pp	−1.40	0.61
Lua	39.4%	NAUTILUS	66.7%	66.6%	**27.2 pp**	1.33 pp	−0.11	−0.72
		NAUTILUS - No Feedback	47.9%	47.8%	**8.4 pp**	1.02 pp	0.11	−1.80
		AFL	54.4%	54.6%	**15.2 pp**	0.54 pp	−1.80	2.42

TABLE II: Statistics about branch coverage. The new coverage found is the additional coverage that was found by the fuzzer w.r.t. the initial corpus. “pp” stands for “percentage points”. Note: AFL was able to find some coverage on PHP, but the results round to zero.

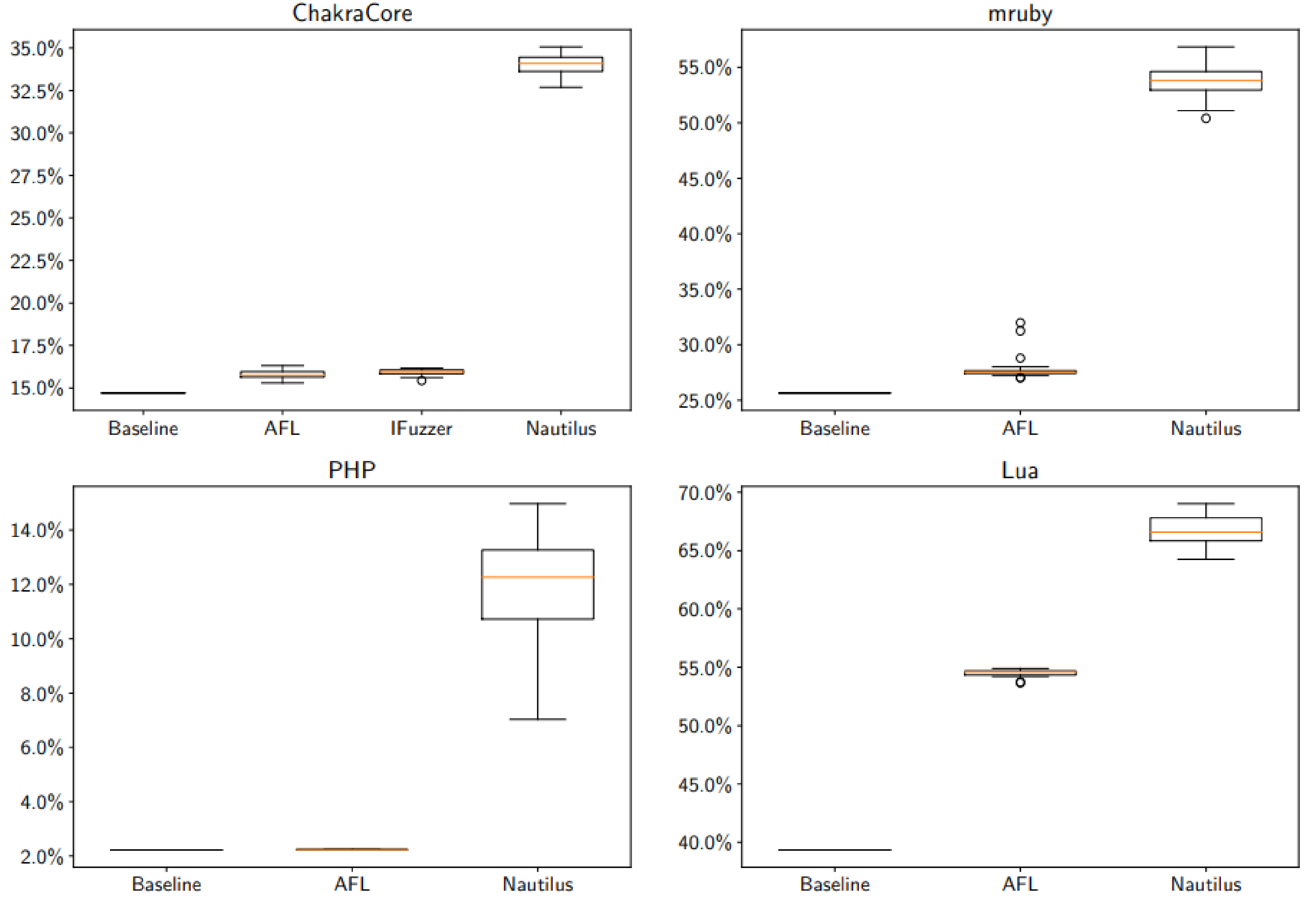


Fig. 3: Branch coverage generated by our corpus of 1000 generated inputs *(Baseline)* and by the different fuzzers over 20 runs of 24 hours each.

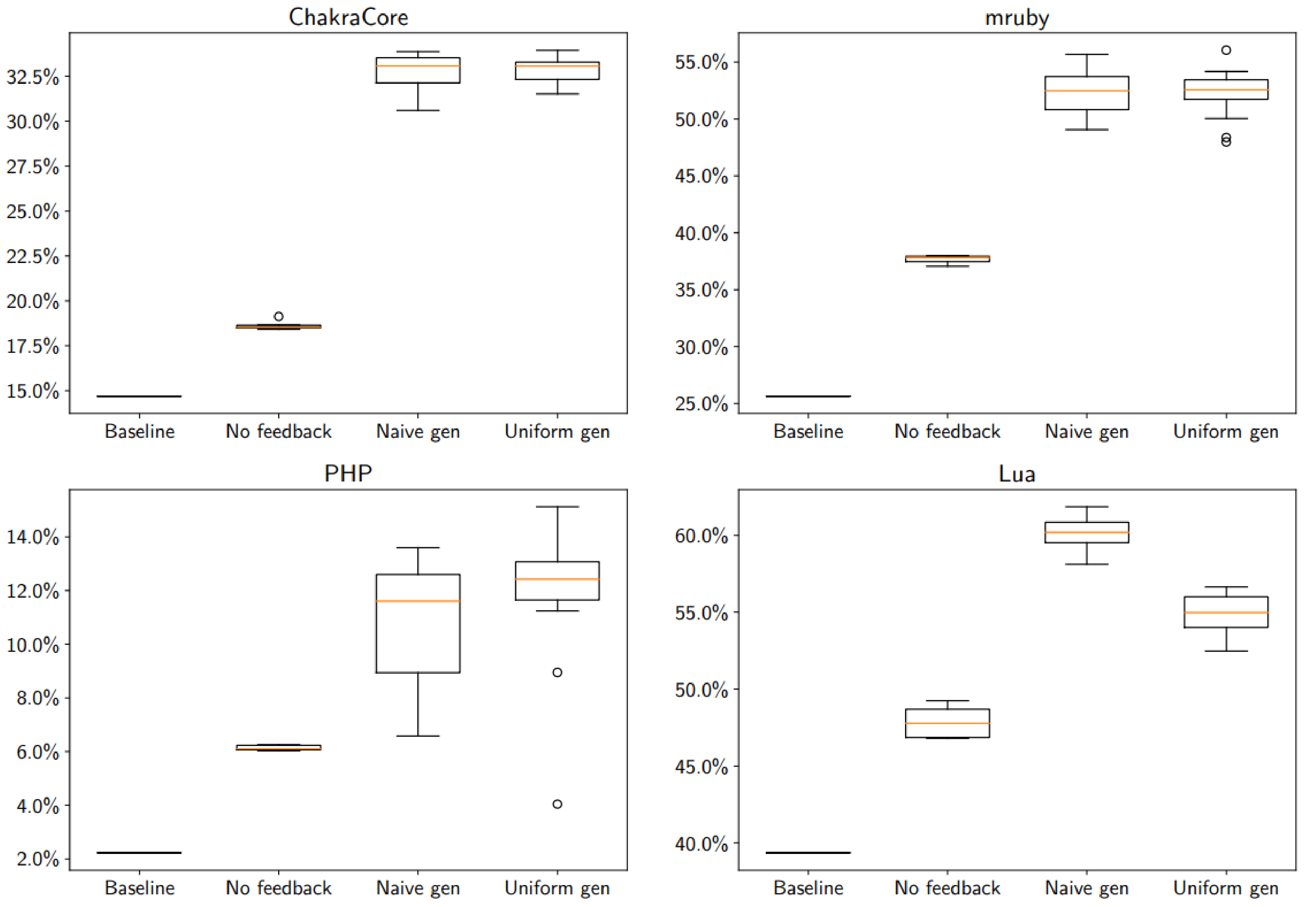


Fig. 4: Branch coverage generated by our corpus *(Baseline)* and by different configurations of NAUTILUS.

MUTATION METHODS

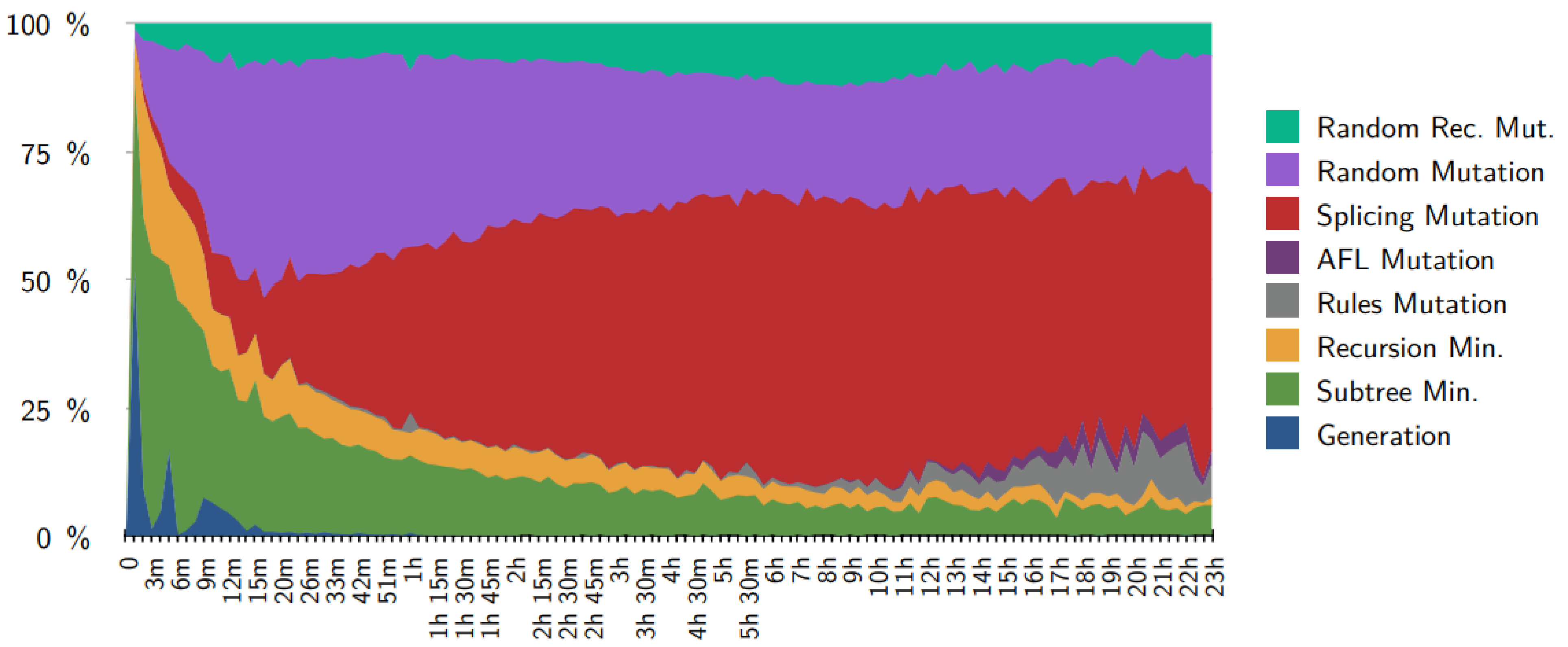


Fig. 5: Percentage of identified new paths for each mutation method, over 20 runs on each target.

LIMITATIONS

- Needs source level access
- Needs a grammar, and a list of important symbols

CONCLUSION

- Grammar + Feedback
- Splicing mutation is effective methods
- Found thirteen new bugs and received 2600 USD in bug bounties

Thanks!