Fizzer with Local Space Fuzzing*
(Competition Contribution)

Martin Jonáš, Jan Strejček, and Marek Trtík(✉)

Masaryk University, Brno, Czech Republic
trtikm@mail.muni.cz

Abstract. Fizzer is a gray-box fuzzer introduced at Test-Comp 2024. This paper summarizes the lessons learned with the original version and describes the major changes including new analyses implemented in the current version of Fizzer. In particular, Fizzer now uses dynamic taint-flow analysis and local space fuzzing. We also provide experimental results showing the progress between the two versions.

Keywords: gray-box fuzzing · dynamic analysis · taint analysis

1 Test-Generation Approach

Fuzzers [8] are tools that generate test inputs for a given program with the use of dynamic analysis. Gray-box fuzzers first instrument the given program to get some information about each program execution (e.g., which basic blocks were visited during the run). The instrumented program is then executed on some input and the obtained information about the execution is used to prepare the input for the next execution with the aim to cover some previously uncovered code. This process is repeated until some goal or limit is reached. Fizzer focuses solely on achieving high branch coverage and applies the same process also in the *Cover-Error* category. Hence, only branch coverage is discussed in this paper.

While standard gray-box fuzzers prefer fast executions and thus gather only a little information, Fizzer collects more information and aims to create more targeted inputs. More precisely, Fizzer tracks the evaluation of *atomic Boolean expressions* (abe) in the given program, which are the Boolean expressions built from expressions of other types, e.g., `(x > 21)` or `(string[i] == 'B')`. Fizzer instruments the program such that each time an abe is evaluated, the program stores the current *calling context* (i.e., the sequence of function calls that are on the call stack), the value `true` or `false` of the abe, and the *distance* to the opposite value. For example, if the abe `(x > 21)` is evaluated to `true`, the distance to `false` is computed as `x - 21`. Fizzer aims to generate tests that evaluate each abe in each reached calling context to both `true` and `false`.

Assume that some *input* leads to the evaluation of an abe in some calling context to `true`. The original version of Fizzer [6] applies the following steps

* This work has been supported by the Czech Science Foundation grant GA23-06506S. M. Trtík—Jury member.

A. Boronat and G. Fraser (Eds.): FASE 2025, LNCS 15693, pp.275–280, 2025.
https://doi.org/10.1007/978-3-031-90900-9_14

to evaluate it to `false`. First, it runs the *sensitivity analysis* to detect the *input* bytes that affect the distance (and thus probably also the value) of the considered ABE in the considered calling context. For each bit of *input*, sensitivity analysis executes the program on *input* with the bit flipped. If the distance changes, the whole byte containing the bit is marked as *sensitive*. As the second step, FIZZER runs the *byteshare analysis* if it has seen some $input'$ that evaluates the same ABE to `false` in a different calling context. The analysis replaces sensitive bytes in *input* by the corresponding sensitive bytes of $input'$ and executes the program. If this still does not evaluate the ABE in the considered calling context to `false`, FIZZER applies the last step. It performs a *gradient descent* on the sensitive bytes with the aim to minimize the absolute value of the distance and thus evaluate the ABE to `false`. We refer to the full paper [7] for more details.

The original version of FIZZER received the bronze medal in *Cover-Branches* category of Test-Comp 2024. Still, we have identified some drawbacks. One of them is that the sensitivity analysis is very slow on programs with a large input, because it may require a program execution for each bit of the input. Moreover, it does not detect bytes that affect the value of ABE if flipping more than one bit is needed to change the distance. In the new version of FIZZER, sensitive bytes are computed by a *dynamic taint-flow analysis*. The program is executed on *input* and bytes returned from each call to `__VERIFIER_nondet_*()` are tainted by a fresh taint. All the taints are propagated through instructions, from their input to output arguments. Input bytes whose taint reaches the expression of the ABE are marked as sensitive. While the original sensitivity analysis under-approximated the sensitive bytes, the new one over-approximates them.

The original approach also struggles with divergencies: a small modification of the input changes the execution path such that the desired ABE in the desired calling context is missed. On the positive side, these divergences can cover some program parts not covered so far. On the negative side, the divergencies disrupt the original sensitivity analysis and gradient descent. To prevent them, we developed the *local search analysis* that internally applies several strategies for input generation including gradient descent. An important feature of the local search analysis is that it runs all its strategies in a *local space* of the target ABE. We sketch this idea using an execution path with only two ABEs `x = y` and `x = 1`, where `x`, `y` are 32-bit signed integers. Assume that the path was explored using the input where $\mathtt{x} = \mathtt{y} = 0$ and that we want to evaluate the second ABE to `true`. The distance functions of the ABEs are $f(\mathtt{x}, \mathtt{y}) = \mathtt{x} - \mathtt{y}$ and $g(\mathtt{x}) = \mathtt{x} - 1$, respectively. Observe that the second distance depends only on `x`. Mutating `x` alone would produce an input for which the execution diverges from the original path on the first ABE. In order to prevent this, we build a stack of local spaces, one for each ABE along the path. Intuitively, the local space captures all values of sensitive bytes that keep the distance of the corresponding ABE unchanged. For example, the distance for the first ABE `x = y` given $\mathtt{x} = \mathtt{y} = 0$ is $f(0, 0) = 0$ and thus the local space is given by equality $f(\mathtt{x}, \mathtt{y}) = f(0, 0)$, i.e., $\mathtt{x} - \mathtt{y} = 0$. Now assume that gradient descent applied on the distance of the second ABE wants to execute the program with $\mathtt{x} = 1$. The original FIZZER would directly run the

program on `x` = 1 and `y` = 0 and the execution would diverge on the first ABE. The local search analysis uses the local space of the first ABE to figure out that in order not to diverge from the path, `y` should be set to 1 and runs the program on `x` = `y` = 1 and the execution will not diverge. Due to space limitations we provide more details about the approach in [5].

2 Software Architecture

Fizzer is implemented in C++, significantly depends on the LLVM infrastructure, and is divided into two executable parts: Instrumenter and Server. The task of Instrumenter is to instrument a given program with the code tracking and reporting the information about program executions. Server schedules and runs the analyses described in the previous section, in particular for generating new inputs and starting executions of the instrumented program. The instrumented program runs in a separate process and communicates with the Server using shared memory, so if the program crashes, Server can still get the information about the run and continue with test generation.

The current version of Fizzer additionally compiles the given program to our new custom *Simple Assembly LAnguage* (SALA), which is basically a simplified version of LLVM, but SALA instructions do not contain type information, there are no LLVM registers nor intrinsics, and functions are not required to be in SSA form. Server contains a SALA interpreter that performs the taint-flow analysis described in the previous section.

3 Strengths and Weaknesses

The current Fizzer has mostly the same strengths and weaknesses as the original version [6]. Fizzer is still a relatively simple and very compact tool with minimal external dependencies. It can be applied to programs of arbitrary size and programs that use external functions available only in compiled form.

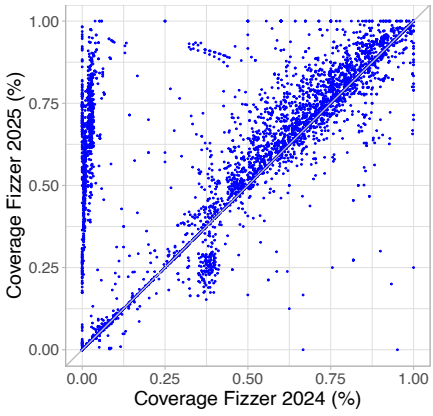

Fig. 1: Comparison of coverages achieved by the original [6] and new Fizzer on *Cover-Branches* benchmarks of Test-Comp 2024.

The main weaknesses of Fizzer stem from the fact that it is a fuzzer that significantly relies on gradient descent. First, being a fuzzer, it generates tests by executing the program. These executions must have some resource limits specified (e.g., number of evaluated ABEs, input size, size of calling context, time limit). Fizzer thus explores only *prefixes* of program paths and consequently tends to focus on parts of the program close to the entry point. This weakness is partially

mitigated by running an *optimizer* after the fuzzing finishes. Optimizer extends the limits and re-runs the program executions that exceeded the standard limits. Second, being reliant on gradient descent, it tends to perform poorly if the branching conditions are non-linear as the standard gradient descent only computes information about linear approximations of the objective function.

Some of the weaknesses of the original Fizzer mentioned in Section 1, i.e., expensive sensitivity analysis and divergences caused by small modifications of inputs, are partially mitigated by the new taint-flow analysis and search in local spaces. This is supported by our experimental evaluation on all *Cover-Branches* benchmarks from Test-Comp 2024 [1] (our experiments use less resources than the Test-Comp 2024 setting, in particular the time limit was set to 300 s per benchmark). The results presented by the scatter plot in Figure 1 show that the new version of Fizzer generally achieves better branch coverage, sometimes even by an order of magnitude. On average, the new version of Fizzer achieved the coverage 66.5% while the original version achieved 59.8%. In Test-Comp 2025 [3], the new version of Fizzer finished in the 4th place in *Cover-Branches* category and won a bronze medal in *Overall* [2].

4 Tool Setup and Configuration

Fizzer can be downloaded either as a binary or as a source code (links are in Section 6). For the source code of the version used in the competition, check out the tag `TESTCOMP25`. The `README.md` file in the root of the repository contains instructions for building the tool. The tool is used via `sbt-fizzer.py` script:

```
sbt-fizzer.py [options] --input_file <c-program>
                        --output_dir <output-dir>
```

All results including the generated tests will be stored under the directory `<output-dir>`. The list of all available options can be obtained by the command `sbt-fizzer.py --help`. Options used in the competition are:

- `max_seconds 865` The timeout for the entire fuzzing process.
- `optimizer_max_seconds 30` The timeout for the optimizer.
- `max_exec_milliseconds 500` The timeout for each program execution.
- `max_exec_megabytes 13312` The memory limit for each program execution.
- `max_stdin_bytes 65536` The upper bound for the number of input bytes.
- `stdin_model stdin_replay_bytes_then_repeat_zero` An input model: given input bytes followed by bytes of the value 0.
- `test_type testcomp` The format for the generated tests.

5 Software Project and Contributors

Fizzer has been developed at the Faculty of Informatics of Masaryk University by Marek Trtík and Lukáš Urban (contributed to the original version). Martin Jonáš and Jan Strejček participated in discussions and contributed to the project by some ideas. The tool is open-source and available under the zlib license.

6 Data-Availability statement

Fizzer is available in a binary form at Zenodo [4] and the source code is available at GitHub:

https://github.com/staticafi/fizzer

References

1. Test-Comp 2024 benchmarks repository (checkout testcomp24-final), https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/
2. Test-Comp 2025, table with results, https://test-comp.sosy-lab.org/2025/results/results-verified/
3. Beyer, D.: Advances in automatic software testing: Test-Comp 2025. In: Proc. FASE. Springer (2025)
4. Jonáš, M., Strejček, J., Trtík, M.: Fizzer: binary (Dec 2024). https://doi.org/10.5281/zenodo.14246517
5. Jonáš, M., Strejček, J., Trtík, M.: Gray-box fuzzing in local space (2025), https://arxiv.org/abs/2501.18046
6. Jonáš, M., Strejček, J., Trtík, M., Urban, L.: Fizzer: New gray-box fuzzer (competition contribution). In: Beyer, D., Cavalcanti, A. (eds.) Fundamental Approaches to Software Engineering - 27th International Conference, FASE 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings. Lecture Notes in Computer Science, vol. 14573, pp. 309–313. Springer (2024), https://doi.org/10.1007/978-3-031-57259-3_17
7. Jonáš, M., Strejček, J., Trtík, M., Urban, L.: Gray-box fuzzing via gradient descent and Boolean expression coverage. In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part III. Lecture Notes in Computer Science, vol. 14572, pp. 90–109. Springer (2024), https://doi.org/10.1007/978-3-031-57256-2_5
8. Liang, H., Pei, X., Jia, X., Shen, W., Zhang, J.: Fuzzing: State of the art. IEEE Transactions on Reliability **67**(3), 1199–1218 (2018). https://doi.org/10.1109/TR.2018.2834476