Fizzer: New Gray-Box Fuzzer*
(Competition Contribution)

Martin Jonáš, Jan Strejček, Marek Trtík✉, and Lukáš Urban

Masaryk University, Brno, Czech Republic

`trtikm@mail.muni.cz`

Abstract. Fizzer is a new gray-box fuzzer. In contrast to common gray-box fuzzers that aim to cover both `true` and `false` branches of branching instructions, Fizzer primarily aims to cover both possible values `true` and `false` of Boolean expressions in the program. When a generated test evaluates a so-called *atomic* Boolean expression to one of these values, our fuzzer computes the distance to the other value, detects bytes that influence this distance, and applies gradient descent on these bytes to flip the value. In Test-Comp 2024, Fizzer placed third in the category *Cover-Branches* after FuSeBMC and FuSeBMC-AI.

Keywords: gray-box fuzzing · dynamic analysis · gradient descent

1 Test-Generation Approach

Fuzzing [5] is an automatic technique that generates test inputs for a given program. *Gray-box fuzzers* first instrument the given program with a code that tracks selected information about a program execution. The instrumented program is then repeatedly executed on various inputs and the tracked information is used to generate new inputs that should execute parts of the program not executed in previous runs.

Successful gray-box fuzzers like AFL [6] collect only very limited information about each program execution and try to quickly perform as many executions as possible. In Fizzer, we use an approach that gathers slightly more information about program executions and uses it to select uncovered parts of the code and make more targeted attempts to cover it.

While typical gray-box fuzzers track only the information about the basic blocks visited during a program execution, our approach tracks also evaluation of each *atomic Boolean expression* (ABE). A Boolean expression is atomic if it is not a variable, not a call of a function whose definition is a part of the program, and not a result of applying a logical operator. Many LLVM instructions yielding `i1` type (i.e., Boolean) from other types are ABEs. An important example is the `icmp` instruction used in translations of `C` expressions like `(x > 42)` or `(string[i] == 'A')`. Each time an ABE is evaluated to `true` or `false`, the instrumented

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

code saves the *calling context* (i.e., the sequence of currently evaluated function calls, which loosely corresponds to the call stack), the value of the ABE, and the *distance* to the opposite value. For example, if ABE `(x > 42)` is evaluated to `true`, the distance to `false` is computed as `x - 42`.

Our fuzzer 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 to `true` and we want to evaluate it to `false` in the same calling context. We first repeatedly execute the program on various mutations of the input to detect the bytes of this input that have some influence on the distance of the ABE evaluation. This process is called a *sensitivity analysis* and the detected bytes are called *sensitive*. Then we apply the following two analyses that use the sensitive bytes. One analysis performs a *gradient descent* on the sensitive bytes with the aim to minimize the absolute value of the distance and to evaluate the ABE in the considered calling context to `false`. Alternatively, if we already know another input evaluating the ABE to `false` in a different calling context, we can try to use the value of its sensitive bytes instead of the sensitive bytes of the current input. This analysis is called *byteshare analysis*.

The fuzzer maintains the information about ABEs evaluated in all program executions, their calling contexts, values, and distances in a binary tree called *atomic Boolean execution tree*. The tree is used to select the ABE and its value to be covered.

For a more detailed and formal description of our approach, we refer to the corresponding research paper [4].

2 Software Architecture

FIZZER is implemented in C++, consists of around 11,000 lines of code in 125 files and uses the LLVM infrastructure. The compiled tool is dependent only on the CLANG compiler. FIZZER consists of two 64-bit executables, namely SERVER and INSTRUMENTER, and a collection of static LIBRARIES provided in both 32-bit and 64-bit versions. Finally, there is a Python script offering a user friendly interface to the tool.

The input program is first translated to LLVM by CLANG. The INSTRUMENTER then instruments the LLVM program with the code for tracking and collecting data during program execution, as explained in the previous section. The inserted code calls functions from the static LIBRARIES. The instrumented program linked with the corresponding static LIBRARIES is called TARGET.

The SERVER controls the actual test generation process. In particular, SERVER generates inputs using the sensitivity analysis, gradient descent, and byteshare analysis mentioned above and runs the TARGET on these inputs. It also receives and processes the information tracked by the TARGET during its executions and builds the atomic Boolean execution tree. The tree is used to select an ABE value to be covered.

The SERVER is one process and each execution of TARGET runs in another process. The exchange of information between the SERVER's process and the

TARGET's process is done via *shared memory*. This ensures that the SERVER can receive the information about TARGET's execution even if the execution crashes.

3 Strengths and Weaknesses

On the positive side, FIZZER is a relatively simple and very compact tool with minimal external dependencies. As it is a pure fuzzer, it can be applied to programs of an arbitrary size and it can also handle programs that use external functions available only in compiled form. And covering (in)equality constraints, which is often difficult for fuzzers, is boosted by the gradient descent.

Fuzzers in general limit each execution of the program as they need to perform many of these executions. FIZZER sets upper bounds (passed to the tool via command line options) on the number of evaluated ABEs, the size of the input bytes read, the size of the calling context, and other properties. If an execution of the TARGET exceeds some of the bounds, it is terminated. FIZZER thus obtains information about prefixes of real executions and thus it can effectively generate tests only for parts of the program close to the program entry point. This weakness correlates with the well known practical experience with fuzzers in general: they are effective in covering code close to the entry point, but have troubles to get deeper. In FIZZER, we do not attempt to properly deal with this phenomenon. We only use so-called *optimizer* after fuzzing stops (usually due to reaching its timeout). The optimizer simply sets up the upper bounds to large numbers and executes the program on those generated inputs that exceeded some upper bound during fuzzing.

Some weaknesses of FIZZER also come from the fact that it is only a prototype implementation taking advantage of some specific features of the Test-Comp benchmarks. In particular, the only way of reading an input currently supported by FIZZER are the functions `__VERIFIER_nondet_*()`.

Another weakness is related to the use of gradient descent as one of the main techniques to cover a selected ABE. The technique is efficient when flipping Boolean values depending on functions with only few extremes (e.g., quadratic functions), but it can struggle on functions with a complex behavior (e.g., functions used for hashing). To mitigate this issue, we implemented a second version of the gradient descent adjusted for functions with many local extremes and we apply it e.g. on function XOR.

In Test-Comp 2024, FIZZER won the bronze medal in the category *Cover-Branches* where 18 tools were competing. Moreover, it obtained the highest score in 3 out of 23 sub-categories of *Cover-Branches*, namely in *ReachSafety-Floats*, *SoftwareSystems-AWS-C-Common-ReachSafety*, and *SoftwareSystems-BusyBox-MemSafety*. FIZZER also participated in the *Cover-Error* category. It is important to stress that FIZZER cannot currently be instructed to focus on covering one particular location, like the target `reach_error()` of this category. FIZZER thus attempted to cover all ABEs in the program, just like in the other category. Despite of that FIZZER placed seventh out of 19 participants in this category. More details can be found on competition's website [1] and report [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, checkout the commit tagged `TESTCOMP24` in order to build the version participating in the competition. The `README.md` file in the root of the repository contains detailed instructions for building the tool. Once the tool is built, all binaries are under `./dist` directory. The content of the directory can be copied "as-is" to a target computer, i.e., no installation is necessary. The tool should be used via `sbt-fizzer.py` script:

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

All results for the given `C` program `<my-c-program>` will be stored under the directory `<my-output-dir>` (including generated tests). The list of all available options can be obtained by command `sbt-fizzer.py --help`. Here are the options we used in the competition:

- `max_seconds 865` The timeout for the fuzzing.
- `optimizer_max_seconds 30` The timeout for the optimizer.
- `max_exec_milliseconds 500` The timeout for each Target's 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: Read generated input bytes and then read zeros.
- `test_type testcomp` The format for the generated tests.

Please note that Fizzer currently does *not* execute the given program in an isolated environment. It is thus *not* advised to run Fizzer directly (outside a container) on any `C` program accessing disk or other external resources.

5 Software Project and Contributors

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

6 Data-Availability statement

Fizzer is available in a binary form at Zenodo [3] and the source code is available at GitHub:

`https://github.com/staticafi/sbt-fizzer`

References

1. Test-Comp 2024, table with results, https://test-comp.sosy-lab.org/2024/results/results-verified/
2. Beyer, D.: Automatic Testing of C Programs: Test-Comp 2024. Springer (2024)
3. Jonáš, M., Strejček, J., Trtík, M., Urban, L.: Fizzer: binary (Nov 2023). https://doi.org/10.5281/zenodo.10183158
4. Jonáš, M., Strejček, J., Trtík, M., Urban, L.: Gray-box fuzzing via gradient descent and Boolean expression coverage. In: Finkbeiner, B., Kovács, L. (eds.) TACAS 2024. LNCS, vol. 14572, pp. 90–109 (2024). https://doi.org/10.1007/978-3-031-57256-2_5
5. 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
6. Zalewski, M.: American fuzzy lop (2013), http://lcamtuf.coredump.cx/afl/.