Masaryk University
Faculty of Informatics

Adapting SBT-Fizzer for Test-Comp

Bachelor's Thesis

Lukáš Urban

Spring 2023

Declaration

I declare that I have worked on this thesis independently, using only the sources listed in the bibliography.

Lukáš Urban

Supervisor: Mgr. Marek Trtík, Ph.D.

Acknowledgements

I am deeply grateful to my supervisor, Marek Trtík, whose guidance, patience, willingness to help, and never-ending optimism helped me get through the difficulties I encountered working on my thesis. I would also like to thank my consultant, Jan Strejček, for his valuable advice and insight.

Many thanks go to my family, who provided a space to retreat whenever things got too hectic. Lastly, I would like to express my appreciation for my friends, whose company was a welcome and, at times, a much-needed distraction.

Abstract

SBT-Fizzer is a novel grey-box fuzzer being developed by the Statica research group at the Faculty of Informatics, Masaryk University. Initially, the fuzzer was restricted to running only on programs embedded inside the project's source code. This thesis advances the implementation so that SBT-Fizzer can run on benchmarks in the tool competition Test-Comp. The results from the evaluation on the benchmarks show that SBT-Fizzer has the potential to be competitive with the top-ranking tools participating in Test-Comp.

Keywords

SBT-Fizzer, fuzzing, grey-box fuzzing, dynamic program analysis, instrumentation, LLVM, Test-Comp

Contents

Introduction xi

1 Fuzzing 1
1.1 Black, white, and grey-box fuzzers . . . 1
1.1.1 Black-box . . . 1
1.1.2 White-box . . . 2
1.1.3 Grey-box . . . 2
1.2 Mutation and generation-based fuzzers . . . 3
1.2.1 Mutation-based . . . 3
1.2.2 Generation-based . . . 3
1.3 American fuzzy lop (AFL) . . . 3
1.3.1 Coverage measurements . . . 4
1.3.2 Mutation strategies . . . 4
1.3.3 Running the target program . . . 5
1.4 Evaluating fuzzers . . . 5

2 Test-Comp 7
2.1 Tools . . . 7
2.1.1 LibKluzzer . . . 9
2.1.2 VeriFuzz . . . 9
2.1.3 FuSe-BMC . . . 10

3 SBT-Fizzer 11
3.1 Instrumentation . . . 11
3.1.1 LLVM IR . . . 11
3.1.2 SBT-Fizzer pass . . . 13
3.2 Running the target program . . . 16
3.2.1 Communication . . . 16
3.2.2 SBT-Fizzer server . . . 17
3.2.3 SBT-Fizzer client, version 1 . . . 18
3.2.4 SBT-Fizzer client, version 2 . . . 18
3.3 Usage . . . 20

4 Evaluation 23
4.1 Comparison of client version 1 and 2 . . . 24
4.2 Comparison with Test-Comp tools . . . 27

5 Conclusion 33

A Electronic attachments 35

Bibliography 37

Introduction

As software increasingly envelopes our lives, ensuring its reliability and correctness is an essential part of the development process. Finding errors in programs is, however, difficult and time-consuming. Developers use various techniques and tools to help in this regard, from handwritten test cases that verify the program's functionality through dynamic analysis tools that detect memory errors to static analysis and formal methods such as model checking.

Another method widely deployed is *fuzzing*, an automated testing technique that has been highly effective in discovering bugs in real-world programs. During fuzzing, a program called *fuzzer* generates many inputs and feeds them into the program being tested in an attempt to trigger an error.

Coverage-based grey-box fuzzers play a pivotal role in fuzzing. The fuzzers use lightweight instrumentation to collect coverage information from the execution of the target program. The coverage information is used to decide whether a generated input exercised new behavior of the program. If it did, the fuzzer saves the input. The saved inputs are then mutated to generate more inputs.

SBT-Fizzer, a novel grey-box fuzzer, gathers more information from the execution than traditional coverage-based grey-box fuzzers and uses it to guide the input generation better. Since the fuzzer was in early development, it could only fuzz programs embedded inside the project's source code. This thesis advances the implementation so the fuzzer can run on benchmarks in the tool competition Test-Comp, and evaluates its performance.

The first chapter of this thesis presents fuzzing with an overview of the different types of fuzzers, along with examples and brief explanations of the techniques they use. The chapter concludes by commenting on evaluating fuzzers. The second chapter introduces Test-Comp, a tool competition for automatic test generators, Test-Comp's benchmarks for evaluating tools, and the top-ranking tools in the competition. The third chapter details the implementation of the enhancements done to SBT-Fizzer. Finally, the fourth chapter presents the evaluation of SBT-Fizzer on Test-Comp's benchmarks and a comparison with the top-ranking tools.

1 Fuzzing

The first fuzzing tool was proposed by Miller et al. in 1990 [1]. They developed a program called *fuzz* to test the reliability of UNIX utilities. The testing method was simple — the program generated a stream of random characters and fed them into the UNIX utilities to see if they would crash.[1]

While the field has progressed substantially, and the strategies and techniques used in modern fuzzers are more complex and intricate, the method of fuzzing stays true to its roots. Fuzzing (or fuzz testing) is the action of continuously running the target program with generated inputs that it may not be expecting in an attempt to trigger unexpected behavior of the program, such as a crash.

Fuzzing has experienced notable success as it has been effective in discovering software bugs and security vulnerabilities in real software while being automated and often having a low barrier to deployment. Fuzzing has been used to test a diversity of software — applications, compilers, kernels, libraries, protocols, browsers, virtual machines, and more [2].

The following sections will describe possible classifications of fuzzers and closely examine one of the most prominent fuzzers, AFL. The final section addresses how fuzzers are evaluated.

1.1 Black, white, and grey-box fuzzers

The first category is concerned with how much the fuzzer knows about the internal structure of the target program.

1.1.1 Black-box

In the black-box approach, the fuzzer treats the target program as a black box, i.e., does not consider its internal structure. The fuzzer's input generation can be random or directed by the output of the program. Black-box fuzzing is simple, fast, easily parallelizable, and always an option. However, it often fails to expose deeper bugs in the program, especially if the program checks the validity of the input. For instance, in the statement "`if (x == 42) then`" the `then` branch has only a 1 in 2^{32} chance of being executed if `x` is a random 32-bit wide input value. The *fuzz* tool would be considered a classic black-box fuzzer. Nowadays, black-box fuzzers tend to consider information about the structure of the input to generate more relevant

[1] After testing almost 90 different utility programs, they were able to crash more than 24% of them.

semi-valid inputs [3]. For instance, Trinity [4], used for fuzzing Linux system calls, knows the datatypes of arguments, valid flags, and ranges of values the system calls expect.

1.1.2 White-box

On the opposite end of the spectrum are white-box fuzzers with complete knowledge of the fuzz target (often, they use the source code). The most used approach by white-box fuzzers is *dynamic symbolic execution* (also called *concolic execution*). Microsoft pioneered this type of fuzzing with SAGE [5].

Dynamic symbolic execution executes the target program with a concrete input and also symbolically. The symbolic execution follows the same path as the concrete execution but gathers the constraints imposed on the input from the conditional branch statements encountered along the execution path. The constraints are then combined to form a path condition. The fuzzer continues by negating the constraints in the path condition one by one and passing it to a constraint solver. Each solution is a new input that exercises a different execution path. Sage then scores the inputs based on their basic block coverage, and the best one is used as the following concrete input.

White-box fuzzers are thorough and, in theory, able to generate tests to cover all program paths. Even so, the overhead the analysis adds, especially for larger programs where the number of execution paths is numerous (the path explosion problem), along with the limitations of existing constraint solvers, is their main drawback.

1.1.3 Grey-box

Grey-box fuzzers embrace a middle ground and have partial knowledge of the target program. Typically, the technique used in grey-box fuzzing is code instrumentation [6]. Code instrumentation is used to obtain information such as *code coverage*, which is utilized as a feedback mechanism to adjust and direct input generation. These types of fuzzers are also called *coverage-based* or *coverage-guided*. A fuzzer intended for libraries called libFuzzer [7] is an example of a coverage-based grey-box fuzzer.

Another method used by grey-box fuzzers is taint tracing [8]. Taint tracing is used to identify which input bytes influence the program's control flow so that the fuzzer can focus on mutating these specific bytes. To give an example, Angora [9] uses taint tracking to determine which input bytes influence each conditional statement. Furthermore, it also infers the shape (which bytes are used as a single value) and type. Angora uses this information to mutate the input (using gradient descent) to execute the unexplored branch of a conditional statement.

All in all, grey-box techniques have a smaller performance overhead than white-box fuzzing while providing enough information to explore the target program more thoroughly than black-box fuzzing.

1.2 Mutation and generation-based fuzzers

Another way of classifying fuzzers is according to the method of constructing new inputs [10] [3]. These fuzzers are also called model-based and model-less or dumb and intelligent, as they vary in the degree of knowledge they have of the input structure of the target program.

1.2.1 Mutation-based

Mutation-based fuzzers create new inputs by mutating inputs from a provided set, called a *seed*. Mutation-based fuzzing requires little knowledge of the input structure of the target program. Despite that, they rely on the quality of the provided seed for the fuzzing to be more effective. The typical methods for mutating include bit flipping, replacing bytes with random or predefined values, deleting/inserting blocks of bytes, and others [3]. This class of fuzzers is not as practical for fuzzing programs with complex input requirements, as it leads to lower code coverage than generation-based fuzzers [10].

1.2.2 Generation-based

Generation-based fuzzers generate inputs from scratch by using knowledge of the input specification. Peach [11], a general-purpose black-box fuzzer, allows the user to specify the format of the data that should be generated. Csmith [12], a C compiler fuzzer, generates random C programs according to a grammar corresponding to the C99 standard. The network protocol fuzzer TLS-Attacker [13] is designed with the TLS protocol model and allows the user to create custom message flows and modify message contents.

As opposed to mutation-based fuzzing, this method does not depend on a good seed and is better at passing the initial parsing stage to trigger bugs deeper in the logic of the program. Its downside is the high time cost associated with creating the specification.

1.3 American fuzzy lop (AFL)

American fuzzy lop [14] is a well-known mutation-based coverage-based grey-box fuzzer. AFL has not only had success in finding bugs in various software applications [15], but it has also been used as a baseline tool for many fuzzing-related publications [16]. Its success can be attributed to its speed, ease of use, robustness, and reliability. Although the development of AFL has stopped, a community-driven fork called AFL++ is still being maintained today [17].

AFL's fuzzing process can roughly be summed up in the following steps [18]. The fuzzer starts by filling its input queue with the initial test cases provided by the user. Then, it repeats the following procedure: it takes an input from the input queue and tries to trim it—reduce its size while maintaining the same program behavior.

Afterward, it repeatedly mutates the input and runs the target program with the mutated input. If the mutated input triggers new behavior, the input is saved into the queue.

1.3.1 Coverage measurements

The fuzzer gathers coverage information using lightweight instrumentation at compile-time. The code injected at branch points captures edge coverage, i.e., a tuple of the source block and destination block IDs, along with the hit count for each tuple. These tuples are saved in an array in shared memory. The array acts as a map, where the indices represent the tuples, and a byte at a particular index represents the hit count of the tuple. Note that collisions in the array are possible since it has a fixed size of 64 kB. The size was chosen such that the collisions are sporadic, but the analysis of the map is quick.

In order to detect whether an input exercised new behavior of the target program, the fuzzer keeps a global coverage map of tuples between runs. The global map is similar to the shared map; however, instead of saving the precise hit counts of tuples, it divides them into eight buckets — 1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+. The bucketing allows the fuzzer to distinguish between relevant changes in the control flow and overlook less notable changes (such as executing a loop 42 times versus 43). Whenever an input explores a new bucket for a tuple, it is considered "interesting" and added to the input queue for later mutation.

1.3.2 Mutation strategies

To explore new states of the target program, AFL mutates the inputs from the input queue [19]. The fuzzer starts with deterministic strategies that are applied sequentially:

- bit flips (1-4 bits with stepover one bit)
- byte flips (8, 16, or 32-bit wide bitflips with stepover one byte)
- addition or subtraction of small integers (to 8, 16, or 32-bit values of the input with stepover one byte)
- substitutions (of 8, 16, 32-wide values) with "interesting" values (0, -1, 1024, INT_MAX, etc., using both endians; with stepover one byte).

It then continues in a non-deterministic manner, where the mutations mentioned above are randomly stacked with additional operations:

- random single-byte sets
- block deletion, duplication, and memset (of randomly chosen block sizes)

Finally, AFL will try test case splicing. There, the current input is spliced with another input from the queue (that is different enough) and mutated using the randomized mutations.

1.3.3 Running the target program

A notable performance feature of AFL is the *fork server* [20]. Initially, AFL used to spawn processes from scratch to fuzz the target program. This way, the fuzzer is isolated from the running program. However, this requires calling `execve` for each execution of the target, which adds undesired overhead. AFL minimizes the execution time problem by injecting code of the fork server into the target, which stops the execution of the program early (usually just before `main`). The fork server then waits for a message from the fuzzer. When the fuzzer requires a new target to fuzz, it sends a message to the fork server through a pipe. At this point, the fork server calls `fork`, and the newly cloned process continues with the execution of the target program. As a result, the fork server makes the target go through `execve`, the linker, and libc initialization only once while providing an adequate level of isolation. It is also possible to stop the execution later than `main` and skip over more initialization code.

AFL also has *persistent* mode, where multiple inputs are executed in a single process. This eliminates the `fork` calls that are, especially for fast target programs, a slowdown of the fuzzing performance. Of course, for the persistent mode to work correctly, the target program needs to cause minimal state changes between executions.

1.4 Evaluating fuzzers

The effectiveness of fuzzers is evaluated experimentally. However, evaluating fuzz testing well is difficult. As a developer, you have to demonstrate that your fuzzer finds more bugs in "real" software than the current state-of-the-art.

A key choice of the evaluation is the performance metric. Ideally, if the bugs for the benchmarks are known, the fuzzer is judged by the number of bugs it can find. If the bugs in the target programs are not known, performance is often measured with heuristics that identify "unique" crashes. Sadly, these tend to overcount the number of actual bugs [16]. Code coverage is an objective metric that is often used. However, there are questions about whether it is an efficacious measure [21], as the correlation with finding more bugs may be weak [16].

Finally, choosing a diverse set of target programs to fuzz is essential. Unfortunately, there is no standard benchmark suite, and there is often little overlap in the benchmarks used between different papers, which makes the results hard to compare across papers [16]. Popular benchmark suites include LAVA-M [22], a benchmark suite consisting of coreutils programs injected with bugs, as well as binaries from CGC (Cyber Grand Challenge) [23]. Another option is a tool competition, such as Test-Comp.

2 Test-Comp

The Competition on Software Testing (Test-Comp)[1] is a competition for automatic test generators. The goal of Test-Comp is to "establish a set of benchmarks for software testing in the community" and "provide an overview of available tools for test-case generation and a snapshot of the state-of-the-art in software testing to the community" [24]. As a result, it is easy to compare the performance of software testers and see whether, for example, a new tool has potential.

Test-Comp currently has two categories, *Branch-Coverage* and *Error-Coverage*. In the Error-Coverage category, the goal is to generate a test case that explores the bug (meaning a specified function is called during execution). The Branch-Coverage category is scored by the coverage of branches executed (value between 0 and 1) by the test suite generated by the tool. The test suite validator TestCov [25] is used to validate both error and branch coverage.

The benchmark programs used in Test-Comp are C programs taken from an open-source repository of benchmarks[2] used by SV-Comp[3]. The benchmarks consist of multiple sub-categories focused on recursive programs, programs working with arrays, floats, and bit-operations, as well as programs from more complex software such as BusyBox, SQLite, or SystemC. As the repository is open for submissions from the community, the whole process can be independently scrutinized, and tools are less likely to overfit to the benchmarks.

The benchmarks utilize functions with the prefix `__VERIFIER_nondet_` to feed inputs from the tool into the program as demonstrated in figure 2.1.

2.1 Tools

Though Test-Comp focuses on software testing in general and not solely on fuzzing, it is currently dominated[4] by tools that, albeit in combination with different techniques, utilize fuzzing. Hence, the rest of the chapter will be devoted to the inner workings of the top three performing tools.

[1] https://test-comp.sosy-lab.org
[2] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks
[3] https://sv-comp.sosy-lab.org/
[4] https://test-comp.sosy-lab.org/2022/results/results-verified/

```c
#include <stddef.h>
#include <stdlib.h>
#include <alloca.h>

extern int __VERIFIER_nondet_int(void);
extern char __VERIFIER_nondet_char(void);

int atoi(const char* s) {
    long int v = 0;
    int sign = 1;
    while (*s == ' ') s++;
    switch (*s) {
        case '-': sign = -1;
        case '+': ++s;
    }
    while ((*s - '0') > 0 && (*s - '0') < 10) {
        v = v * 10 + *s - '0'; ++s;
    }
    return sign == -1 ? -v : v;
}

int main() {
    int in_len = __VERIFIER_nondet_int();
    if (in_len < 1) { return 1; }
    char* in = alloca(in_len);
    for (int i = 0; i < in_len - 1; i++) {
        in[i] = __VERIFIER_nondet_char();
    }
    in[in_len - 1] = 0;

    return atoi(in);
}
```

Figure 2.1: Example C benchmark `atoi.c`, which implements the `atoi` function. Taken from the repository of benchmarks used for Test-Comp and slightly reformatted for better readability.

2.1.1 LibKluzzer

LibKluzzer [26] is a hybrid fuzzer, as it combines the strengths of white-box and coverage-guided fuzzing. The tool uses libFuzzer as the coverage-guided fuzzer, and KLUZZER [27], a white-box fuzzer built on top of KLEE [28], a symbolic execution tool.

LibFuzzer is an in-process (meaning all of the executions of the target program occur in one process) fuzzer that uses SanitizerCoverage [29] to track function, basic block, or edge level coverage (the default is edge level). Its mutation strategies include randomly erasing, inserting, shuffling, and changing bytes, randomly changing bits and integers, and others.

KLUZZER uses concolic execution and works similarly to SAGE. The modifications done to KLEE allow the symbolic execution to follow the path of the concrete input and delay the feasibility checks of the other directions of branches. The inputs generated by the SMT solver are then scored not only by the uncovered basic blocks but also by the number of new edges and their hit counts.

The integration of these two techniques is straightforward. Both libFuzzer and KLUZZER are run in parallel, individually generating inputs to cover new code. Though, the tools share their progress by writing to and reading from a common corpus of inputs. LibKluzzer is also responsible for performing transformations of the benchmark to allow for in-process fuzzing, defining the fuzzing target, and modeling the `__VERIFIER_nondet_` functions.

2.1.2 VeriFuzz

VeriFuzz [30] [31] employs a fuzzing engine built from AFL, which is aided by lightweight static analysis and a bounded model checker (BMC) to generate the initial set of inputs.

The static analysis determines an over-approximation of the range of input values that could reach a particular point in the program. The fuzzing engine is fine-tuned, so it considers these ranges while mutating the input.

As AFL (and other grey-box fuzzers) struggle with generating inputs that move beyond complex checks, VeriFuzz deploys CBMC as the bounded model checker to generate such inputs for the seed. The target program is unwound to a "short" depth chosen heuristically, and CBMC is used to produce inputs that cover the branches of the unwound program. These inputs form valid prefixes of the complete inputs for the original program. The inputs are completed using random inputs respecting the ranges discovered from the static analysis and used as the seed for the fuzzing engine. The short unwinding allows the BMC to perform better on programs with loops with large bounds.

2.1.3 FuSe-BMC

The competition was won by FuSe-BMC [32] [33], which utilizes ESBMC (a bounded model checker) and a fuzzer based on AFL.

First, FuSe-BMC uses Clang to produce an abstract syntax tree (AST) from the target program. Next, every conditional statement, loop, and function in the AST is injected with a goal label so FuSeBMC can measure coverage. The goal labels are used to produce a reachability graph, and each goal is ranked depending on the chosen strategy. For Test-Comp, the goals are ranked by their depth and coverage metrics with different weights (for example, conditional coverage is preferred over loop coverage).

FuSe-BMC also has a seed generation phase, where the BMC and the fuzzer run with a small timeout. The fuzzer is aided by static analysis to establish bounds on the input values it should generate. The highest-rated inputs generated by the fuzzer and ESBMC, according to the number of unique labels and the depth reached, are used as seeds for the next phase.

In the primary phase, the ESBMC and fuzzer are run in parallel with longer timeouts. The fuzzer takes advantage of the seeds from the previous phase. FuSe-BMC also maintains two shared memory segments — one for an array of goals that have not been covered and a set of inputs that are the most beneficial to the fuzzer. While the fuzzer is running, ESBMC selects an uncovered goal from the array, tries to generate an input to cover it, and, if it succeeds, writes it into the set of inputs.

3 SBT-Fizzer

SBT-Fizzer[1] is a prototype fuzzer developed by the Statica research group at the Faculty of Informatics, Masaryk University. It is a mutation-based grey-box fuzzer, as it leverages coverage information to create inputs to maximize code coverage.

The fuzzer was developed and could only be launched on a set of benchmarks embedded inside the fuzzer's source code. The benchmarks used redefined `if`, `while`, and `for` (using macros) to collect the required coverage information.

In this chapter, I detail the modifications made to SBT-Fizzer to allow it to run on Test-Comp's benchmarks. I describe the instrumentation process for gathering coverage information, the execution of the target program by the fuzzer, and the method of passing coverage information to the fuzzer.

3.1 Instrumentation

SBT-Fizzer gleans information about the target program by executing instructions instrumented (i.e., inserted) into it. The instrumentation is done by compiling the target program from C to LLVM assembly language (called LLVM IR) and instrumenting the fuzzer's monitoring code into it. The following sections will explain the basics of the LLVM assembly language and describe the information the fuzzer gets from the execution of the target program.

3.1.1 LLVM IR

LLVM[2] IR is an intermediate representation that acts as a high-level assembly language [34]. The language is strongly typed and based on Single Static Assignment[3] (SSA). The representation aims to be universal, enabling high-level language constructs to be mapped into it. LLVM IR can be analyzed and transformed using LLVM passes, which traverse parts of the program, modifying the code or collecting information. The instrumentation is implemented as a pass that runs after the program is compiled into LLVM IR but before any optimizations.

[1] SBT stands for Symbiotic (https://github.com/staticafi/symbiotic), a tool for verifying programs developed by Statica, and Fizzer is a blend of fuzzer and the Faculty of Informatics "FI".

[2] The LLVM project is a collection of compiler and toolchain technologies, providing an optimizer along with code generation support (LLVM Core), a compiler (Clang), tools for static analysis, implementation of the C++ Standard Library (libc++), runtime libraries for dynamic testing (various sanitizers), and more.

[3] Static single assignment form requires each variable (or in the case of LLVM, each register) to be defined before it is used and assigned at exactly one place.

Structure

LLVM programs are composed of modules, which are translation units of the input programs. A module consists of global variables, symbol table entries, and functions. Functions are comprised of basic blocks, which define the control flow graph of the function. A basic block is formed of a list of instructions that execute sequentially. Each basic block contains a single terminating instruction at the end, which terminates the block and transfers control flow to another basic block.

```
int main() {
    unsigned int x = __VERIFIER_nondet_uint();
    int y = __VERIFIER_nondet_int();
    if (x <= 4 && y > 2) {
        *((int*) 0xdeadbeef) = 42;
    }
    return 0;
}

define i32 @main() {
entry:
  %x = call i32 (...) @__VERIFIER_nondet_uint()
  %y = call i32 (...) @__VERIFIER_nondet_int()
  %lhs = icmp ule i32 %x, 4
  br i1 %lhs, label %and_lhs_true, label %end

and_lhs_true:
  %rhs = icmp sgt i32 %y, 2
  br i1 %rhs, label %then, label %end

then:
  store i32 42, i32* inttoptr (i64 3735928559 to i32*), align 4
  br label %end

end:
  ret i32 0
}
```

Figure 3.1: Example C program with its LLVM IR equivalent. Likely to cause a segfault if the `if` statement is evaluated to true.

Figure 3.1 shows a simple program in LLVM IR with four basic blocks. The program reads an unsigned integer and an integer by calling the `__VERIFIER_nondet_` functions. It then uses a `icmp` instruction, which takes a predicate — in this case, `ule` (unsigned less than or equal to), and two arguments to perform the comparison on. The result of the `icmp` instruction is used as an argument for the conditional branching instruction `br`. If `x` is less than or equal to 4, the `br` instruction transfers control flow to the basic block `and_lhs_true`. This basic block checks whether `y` is

greater than 2 (`sgt`, signed greater than), and if so, control flows to `then`, where the program writes 42 to a constant randomly chosen address. Otherwise, the program just ends by returning 0 using the `ret` instruction.

Note that the integer and unsigned integer are represented using the same type `i32`. LLVM IR does not distinguish between signed and unsigned types.[4] However, when needed, the operations on the integers are translated according to the original type, hence the signs associated with the predicates `ule` and `sgt`.

3.1.2 SBT-Fizzer pass

SBT-Fizzer collects information about the conditional `br` instructions and conditions (values of type `i1`) executed during a run of the target program.

First, the pass inserts a call to the function `__sbt_fizzer_process_br_instr` before every conditional `br` instruction. This function collects *br instruction coverage information*, which is composed of:

- the identifier of the `br` instruction
- the branch of the `br` instruction that was covered (true or false).

The pass numbers the basic blocks of the LLVM module sequentially starting from 1, so the identifier of the `br` instruction is simply the number of the basic block it is in. The covered branch is the value of the condition of the `br` instruction. The execution of the program produces a sequence of *br instruction coverage information* called a *br instruction trace*.

To generate new inputs that increase code coverage, SBT-Fizzer tries to flip the conditions it encountered during a run of the target program. Therefore, the pass adds a call to the function `__sbt_fizzer_process_condition` after values of type `i1` that are the results of comparisons (`icmp`/`fcmp`), truncations[5] (`trunc`), or calls (`call`) to a function. The function saves the following data, together called *branching coverage information*:

- the identifier of the condition
- the value of the condition (true or false)
- the branching function value, which expresses how far the fuzzer is from flipping the condition
- the index to the br instruction trace.

A sequence of *branching coverage information* collected during a run is called a *trace*. The conditions are again numbered sequentially from 1, and this number is used as the identifier. The branching function value is the primary value the

[4] Not distinguishing signs simplifies the type system.

[5] The `trunc` instruction is generated for example when the `_Bool` type is used in C.

fuzzer uses to generate new inputs. For a condition that is a comparison, i.e., `icmp/fcmp <pred> <type> <a> <b>`, the value is calculated as `a – b` (and cast to `double`). For example, the fuzzer uses this value to detect which bits of the input influence the condition it is trying to flip. If flipping a bit changes the value, the bit is considered to be sensitive. After determining sensitive bits, the fuzzer tries changing them to minimize the branching function's absolute value (in the minimization analysis phase), which increases the chance of flipping the condition.[6]

If the condition is a `trunc` or a `call`, the branching value is assumed to be 1. Note that the condition can also be a comparison of two pointers. However, the notion loses meaning when comparing pointers pointing to completely different addresses, so the branching value is again assumed to be 1.

While the fuzzer's strategy uses the branching function value of conditions, the br instruction coverage information is collected so that the fuzzer can use the help of JetKlee[7] when it is unable to flip a condition. Since the instrumentation does not have a direct correspondence between the `br` instructions and the conditions they use, it saves the index to the br instruction trace with every condition. The index points to the first `br` instruction in the br instruction trace that comes after the condition in the LLVM IR code (as that is most likely a `br` instruction that uses the condition).

The pass runs before optimizations so that more complex conditions consisting of `&&` or `||` are unfolded. For example, the condition

```
if (a && b) { t } else { f }
```

will correspond to

```
if (a) { if (b) { t } else { f } } else { f }
```

and have 2 `br` instructions in LLVM IR.[8] This way, both `br` instructions are instrumented before optimizations simplify them into one. For instance, the program in figure 3.1 would have one `br` instruction, taking the result of an `and` instruction of the two conditions.

The pass transforms any `switch` instruction into a sequence of `br` instructions by invoking the built-in LLVM pass `LowerSwitch`, thereby instrumenting the branches of the switch. Finally, the pass renames the `main` function to `__sbt_fizzer_method_under_test`, so the program can later be linked with the fuzzer's libraries that define a `main`. The program in figure 3.1 instrumented with the SBT-Fizzer pass can be seen in figure 3.2.

[6] Of course, this is a shortened and simplified version of what happens, but it shows the incentive for capturing the value.

[7] JetKlee (https://github.com/staticafi/JetKlee) is a clone of Klee developed by Statica. The integration is not present in this thesis.

[8] Illustrated in figure 3.1.

```
define i32 @__sbt_fizzer_method_under_test() {
bb1:
  %x = call i32 (...) @__VERIFIER_nondet_uint()
  %y = call i32 (...) @__VERIFIER_nondet_int()
  %lhs = icmp ule i32 %x, 4
  %0 = zext i32 %x to i64
  %1 = sub i64 %0, 4
  %2 = uitofp i64 %1 to double
  call void @__sbt_fizzer_process_condition(
    i32 1, i1 %lhs, double %2
  )
  call void @__sbt_fizzer_process_br_instr(
    i32 1, i1 %lhs
  )
  br i1 %lhs, label %bb2, label %bb4

bb2:
  %rhs = icmp sgt i32 %y, 2
  %3 = sext i32 %y to i64
  %4 = sub i64 %3, 2
  %5 = sitofp i64 %4 to double
  call void @__sbt_fizzer_process_condition(
    i32 2, i1 %rhs, double %5
  )
  call void @__sbt_fizzer_process_br_instr(
    i32 2, i1 %rhs
  )
  br i1 %rhs, label %bb3, label %bb4

bb3:
  store i32 42, i32* inttoptr (i64 3735928559 to i32*), align 4
  br label %bb4

bb4:
  ret i32 0
}
```

Figure 3.2: The program in figure 3.1 instrumented using the SBT-Fizzer pass. The calculation of the branching function value is also added during instrumentation — the comparison arguments are extended to a higher type if an overflow is possible, the subtraction is performed, and the result is cast to a double and passed to the instrumented function (the operations extension and casting to a double are chosen according to the signedness of the comparison).

3.2 Running the target program

The fuzzer needs to execute the target program numerous times. The simplest way to accomplish this is to link the target program to the fuzzer and execute the program's `main` function (called in-process fuzzing). While this approach is fast, it also has its drawbacks. For example:

Crashes An unexpected crash of the target program (e.g., a segmentation fault, such as in the program in figure 3.1) would lead to a crash of the fuzzer.

Memory leaks If the target program contains memory leaks, executing it many times can result in running out of memory, at which point the fuzzer has to be stopped.

State Remnants of the previous execution of the program could affect the subsequent execution. As an example, static variables would not get reset, possibly leading to unexpected behavior.

As discussed briefly in sections 1.3.3 and 2.1.1, different fuzzers deal with these issues by using varying levels of isolation. While libFuzzer uses in-process fuzzing, AFL executes a new process every time (albeit using a workaround with the fork server), and Nyx [35] uses a snapshot of a virtual machine (VM) and resets it after each run.

Since one of the design goals of SBT-Fizzer is to prioritize correctness, the target program should be executed in its process to avoid the issues mentioned above. After the target program is compiled into LLVM IR and instrumented, it is linked with libraries that provide the definitions of the instrumented functions, a means of communication with the fuzzer, definitions of `__VERIFIER_nondet_` functions, and a `main` function to create an executable. Whenever the fuzzer needs to execute the target program, it runs the executable, supplies it with an input (just a sequence of bytes), and retrieves the result — the trace, the br instruction trace, and the bytes read from stdin (i.e., from the `__VERIFIER_nondet_` functions) — called *execution information*.

3.2.1 Communication

Since the target program is now isolated from the fuzzer, it must use some inter-process communication mechanism. The method chosen was communication via a socket using TCP. One of the reasons for choosing this method was to determine its viability (in terms of speed). The other reasons were:

Speed of deployment Communication via a socket was one of the more straightforward methods to implement, mainly because of the structure of the SBT-Fizzer code base. Moreover, the project used the Boost libraries[9], which contain the Boost.Asio library for network programming.

[9] https://www.boost.org/

Cross-platform Of course, sockets are cross-platform, as well as the Boost.Asio library.

Increasing isolation The method would still be applicable if there were a need to increase the isolation of the target program (such as by using a VM).

Remote communication Furthermore, the method would be suitable for fuzzing programs running on a different computer than the fuzzer.

Inspired by the communication method, the fuzzer is called *server*, and the executable responsible for running the target program is the *client*.

3.2.2 SBT-Fizzer server

To allow for the new communication method, I extended the existing code of the fuzzer to also work as a server. The server handles fuzzing logic, accepting connections, communicating with clients, and executing clients. The server uses three threads to handle these tasks.

One thread is responsible for accepting incoming connections. When a connection is accepted, it is stored in a thread-safe queue (as the other threads also access the queue).

The second thread manages the execution of new clients. The thread executes a new client whenever the size of the connections queue falls below a threshold[10]. Separating the execution of clients from the fuzzing loop speeds up the fuzzing process, as the time during which the fuzzer is busy (when communicating with the client or analyzing the trace) is also spent launching new client processes.

Finally, the third thread runs the fuzzer's algorithm. The algorithm is essentially the following loop[11]:

1. generate an input for the client
2. pop client connection from the queue
3. send the input to the client
4. wait for a message from the client containing execution information
5. parse the message and analyze the trace
6. save the input if it uncovered a new branch of a condition.

The loop runs until the given timeout or maximum executions is reached, the fuzzer runs out of strategies, or all reachable branchings are covered. The fuzzer then outputs the saved inputs in the test format used in Test-Comp[12].

[10] Currently five connections.
[11] The actual fuzzing logic is omitted.
[12] https://gitlab.com/sosy-lab/test-comp/test-format

3.2.3 SBT-Fizzer client, version 1

The instrumented target program in LLVM IR is compiled with the SBT-Fizzer client libraries to produce the client executable. The build process is illustrated in figure 3.3.

The client connects to the server and waits until it receives input for the target program. When it does, it executes `__sbt_fizzer_method_under_test` (the `main` function of the target program).

Whenever a `__VERIFIER_nondet_X` function is called, the client replays the received input sequence of bytes to produce a value of the specified type `X`. If the client depletes the input bytes it was given, it repeats a byte specified in the options[13].

The implementation of the instrumented `__sbt_fizzer_process_condition` and `__sbt_fizzer_process_br_instr` functions is simple — they save the branching coverage information and br instruction coverage information in vectors.

After `__sbt_fizzer_method_under_test` ends, the client serializes the trace, br instruction trace, and bytes read using `__VERIFIER_nondet_` functions into a message (which just acts as a byte buffer) and sends it to the server.

Because the client requires `__sbt_fizzer_method_under_test` to return so it can send the execution information back to the server, the instrumentation also replaces functions `exit` and `reach_error`[14] with variants that do not immediately terminate the process.

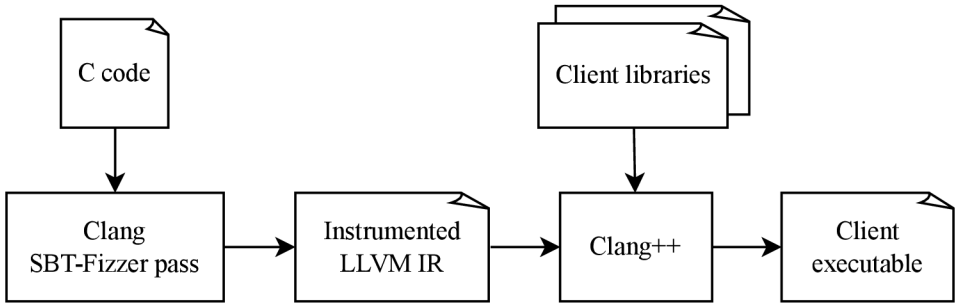

Figure 3.3: Build process of client version 1. The SBT-Fizzer pass is loaded using Clang to compose the compilation and instrumentation steps into one.

3.2.4 SBT-Fizzer client, version 2

Although SBT-Fizzer client version 1 sufficed for performing the evaluation, it became apparent that it had a significant limitation. More than 200 benchmarks (out of 3460) crashed during the execution, which means the client did not have the chance to send back results from the execution. While the fuzzer could technically

[13] At the moment, `00000000` or `01010101` (85).

[14] Test-Comp specific function used in the benchmarks.

work without feedback, it should always receive it, even when the target program behaves unpredictably.[15]

There were two potential solutions to this problem. The first solution involves sending the information in chunks whenever one of the instrumented functions is called, rather than collecting all the information and sending it at once at the end. However, the messages would be small (roughly 20 bytes), meaning the packet would consist mostly of data added by the TCP protocol. Due to this considerable overhead, the second solution was chosen.

The second solution entails splitting the client into two binaries executed in separate processes. The first binary, SBT-Fizzer client, is the one that connects to the server and communicates with it. It also creates a shared memory segment for communicating with the second process. The rationale for choosing shared memory is that it is the fastest method of interprocess communication. The second binary, SBT-Fizzer target, is the one that executes the `main` function of the target program and saves execution information into the shared memory segment (note that the shared memory segment simply acts as an array of bytes).

The client's function is now to:

1. connect to the server
2. wait until it receives input from the server
3. create the shared memory segment
4. start the SBT-Fizzer target executable
5. wait until it finishes
6. copy the results from the shared memory into a message
7. send the message to the server.

Since the target program is now completely separate from the client, it is built only once during the build stage of the project (as opposed to SBT-Fizzer client version 1).

SBT-Fizzer target

SBT-Fizzer target's build process is analogous to SBT-Fizzer client version 1. Despite that, the two executables work differently; SBT-Fizzer target only opens the shared memory segment and executes `__sbt_fizzer_method_under_test`.

The `__VERIFIER_nondet_` functions operate similarly to client version 1. The instrumented functions behave differently; because SBT-Fizzer target can terminate at any time, they have to write the information into shared memory immediately

[15] Even if the problem did not appear on the benchmarks in Test-Comp, it would have to be solved eventually when SBT-Fizzer gets deployed on real software.

when called. Since the shared memory segment acts as an array of bytes, any data written into it is first prefixed with a byte that identifies the data's type. When the fuzzer parses the message (that contains the bytes from the shared memory), it uses the identifiers to interpret the bytes correctly.

The new approach has further implications. The instrumentation does not have to replace functions such as `exit`. Furthermore, the fuzzer can terminate the target if it never finishes[16].

Direct mode

There is no dispute that SBT-Fizzer client version 2 has a substantial amount of overhead — the fuzzer launches two processes, and there is communication via TCP and shared memory for each execution of the target program.

To mitigate the issue, I added another mode of target execution and communication to the fuzzer. In this mode, the server runs SBT-Fizzer target and communicates with it directly using the shared memory segment, thus skipping the execution of the client and communication through TCP. As a result, the mode is even faster than using SBT-Fizzer client version 1 while still being able to handle programs that crash during their execution.

The user can choose between the two modes. Currently, the direct mode is strictly better than using the client. The client will become relevant in the future when the fuzzer is used for fuzzing a program running on a different computer or the program needs to be isolated even further.

3.3 Usage

SBT-Fizzer is available on GitHub[17] under the zlib license. The main branch currently works with client version 1, while the shared-memory branch contains client version 2. The shared-memory branch will be merged into the main branch to replace client version 1.

The fuzzer requires Boost (minimum version 1.69), as well as LLVM and Clang (minimum version tested is 10.0) to compile. The project is built using CMake. After building and installing, the executables can be found in the `./dist/tools` directory.

Because the fuzzing process is now cumbersome and contains many steps, I also wrote three Python scripts to simplify the procedure (installed in the `./dist/tools` directory). The script `sbt-fizzer` covers all the steps necessary to fuzz a program. It instruments the target program, builds the SBT-Fizzer target (or client version 1) with it, and runs the server with the built target (client). The scripts `sbt-fizzer_instrument` and `sbt-fizzer_build_target (client)` can be used to carry out the intermediary steps as indicated by their name. Note that the scripts

[16] Such as in two benchmarks in Test-Comp that contain an infinite loop.
[17] https://github.com/staticafi/sbt-fizzer

use Clang and LLVM's opt tool[18] from the `PATH` environment variable. To fuzz the program `program.c`, run:

```
./dist/tools/sbt-fizzer program.c
```

[18] Used for running passes on LLVM source files (https://llvm.org/docs/CommandGuide/opt.html).

4 Evaluation

The performance of SBT-Fizzer was evaluated on the Branch-Coverage category of Test-Comp 2022. The category Error-Coverage was omitted as the fuzzer does not utilize any specific techniques for reaching a particular point in the program. The top three tools LibKluzzer, VeriFuzz, and FuSe-BMC, were also evaluated to compare their performance with SBT-Fizzer.

The experiments were performed on machines with a 6-core Intel Core i7-8700 CPU with a frequency of 3.20 GHz and 33 GB of RAM. As in Test-Comp, I used the tool BenchExec[1] to run the experiments to ensure reliable measurements, as well as FormelaCloud[2] to distribute the test-case generation and validation.

LibKluzzer, VeriFuzz, FuSe-BMC, and TestCov were obtained from the Test-Comp 2022 tester archives[3], and their system requirements were installed on the machines. The tools were limited with `--limitCores 2 --memorylimit 7GB --timelimit 300` (the timelimit limits the CPU time).[4] TestCov used the default options as per its definition[5].

SBT-Fizzer ran with `--limitCores 1 --memorylimit 7GB --timelimit 300`, as it uses wall-time to measure its execution time to stop itself and output the test suite (limiting to one core ensures it does not get killed by BenchExec due to hitting the CPU time limit before stopping itself and outputting the test suite).

The options SBT-Fizzer ran with were:

`--max_trace_length 10000` (limit the trace)
`--max_br_instr_trace_length 30000` (limit the br instruction trace)
`--timeout 500` (kill the target if it does not finish within 500ms)
`--stdin_model stdin_replay_bytes_then_repeat_zero` (repeat 0 byte)
`--max_stdin_bytes 65536` (limit the number of bytes read)
`--test_type testcomp` (produce the test suite for Test-Comp)
`--max_seconds 295` (ensure the fuzzer has time to output the test suite)

The values for these options are initial educated guesses and were not optimized in any way.

[1] https://github.com/sosy-lab/benchexec

[2] Test-Comp uses VerifierCloud (https://vcloud.sosy-lab.org/cpachecker/webclient/help/) to distribute the verification and validation process over a cluster of 167 machines. FormelaCloud is an instance of VerifierCloud used by Formela, installed and running on 14 machines in the laboratory.

[3] https://gitlab.com/sosy-lab/test-comp/archives-2022

[4] These options were used during the pre-runs of Test-Comp.

[5] https://gitlab.com/sosy-lab/test-comp/bench-defs/-/tree/testcomp22/benchmark-defs

4.1 Comparison of client version 1 and 2

The first comparison is between SBT-Fizzer's three modes of communication — client version 1 (v1), client version 2 (v2), and direct mode. Client v1 used the same options as written above, except without the timeout and limit for the br instruction trace.

Table 4.1 shows the coverage the three modes reached on all subcategories. Client v2 achieved better total coverage than client v1 but a lower score. The differences between client v1 and client v2 can also be seen in figure 4.1. Both client v2 and direct mode gained coverage thanks to being able to deal with crashing programs, which is apparent in the subcategories MainHeap, Array, and DeviceDrivers. While the performance of client v2 is similar to v1, the overhead of spawning another process and communicating through shared memory is especially noticeable in the subcategory Combinations. On the other hand, as expected, the direct mode surpassed the two versions both in total coverage and score. Unexpectedly, client v2 had slightly better coverage on the subcategory MainHeap. The difference appears to be the result of undefined behavior in the benchmarks.

Client v2 did, however, introduce more server crashes. After inspecting some of the crashing benchmarks, the server errors can be classified into two categories: actual bugs in the fuzzing algorithm and errors during parsing of the message containing bytes from the shared memory segment. The parsing errors stem from SBT-Fizzer target crashing while opening the shared memory segment. Some of the benchmarks in Test-Comp define functions with names that collide with functions used by SBT-Fizzer target's libraries. For instance, the benchmark `fo_test.i` defines the function `open`, which collides with C'`open` used to open the shared memory segment.

Surprising is the relatively poor performance of client v2 and direct mode on the subcategory ControlFlow (which accounts for the lower score of client v2), even without considering server crashes. The difference is caused by `openssl` benchmarks, all of which are of very similar structure (hundreds of lines with many branches testing for equality with specific values). I examined the behavior of the fuzzer (with both client versions) on two benchmarks under Valgrind. The output of the two versions diverges after the benchmark reads the value of an uninitialized variable. Thus, I speculate that the disparity arises from undefined behavior, but further examination is needed.

Table 4.1: Coverage (%) achieved on the subcategories of Test-Comp's Branch-Coverage category by SBT-Fizzer using client version 1, client version 2, or direct mode. The coverage was rounded except for the SQLite subcategory. The total is the sum of the unrounded coverages, rounded at the end. The score for each subcategory is the coverage divided by the number of benchmarks in the subcategory. The total score is the sum of the scores of the subcategories multiplied by the average number of benchmarks.

Subcategory	Programs	Client v1	Client v2	Direct
Arrays	400	25772	26418	26418
BitVectors	62	4488	4392	4477
Combinations	671	32364	31818	32927
ControlFlow	67	3649	2810	2941
ECA	29	844	832	858
Floats	226	7944	7944	7944
Heap	143	10073	10180	10180
Loops	727	57121	57173	57186
ProductLines	263	7552	7541	7571
Recursive	53	4420	4482	4482
Sequentialized	103	6741	6691	6750
XCSP	119	10162	10162	10162
BusyBox	75	2207	2251	2269
DeviceDrivers	290	5470	5851	5846
SQLite	1	3.58	3.64	3.67
MainHeap	231	16913	18664	18640
Total	3460	195722	197212	198652
Score	—	181823	181249	182698

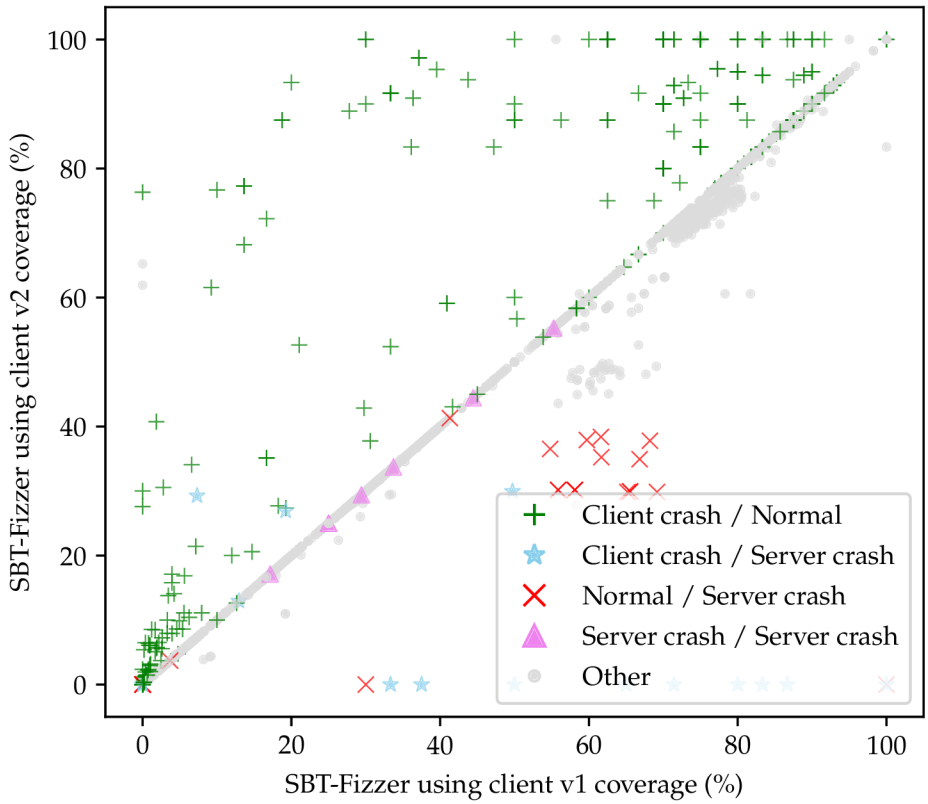

Figure 4.1: Scatter plot of the coverage of SBT-Fizzer using client version 1 and client version 2. The points are divided based on the end status of SBT-Fizzer with client version 1 / version 2, as depicted in the legend. The end status client crash means the target program crashed the client version 1 process. As a result, the fuzzer did not get any feedback and could not continue.

4.2 Comparison with Test-Comp tools

This section compares SBT-Fizzer (using client v2) with LibKluzzer, VeriFuzz, and FuSe-BMC. Table 4.2 lists the tools' coverage on all subcategories, along with their total coverage and score. The ranking of the tools by total coverage aligns with Test-Comp. However, ranking by score places FuSe-BMC third. According to [33], FuSe-BMC is optimized for the 900-second timelimit. That is particularly apparent in the SQLite subcategory, where it did not generate any tests in time (which would likely place it higher). Unfortunately, SBT-Fizzer does fall behind the other tools on total coverage, score, and coverage in most subcategories. Nonetheless, the result is understandable, as LibKluzzer, VeriFuzz, and FuSe-BMC employ state-of-the-art grey-box fuzzers in combination with other techniques — bounded model checking, whitebox fuzzing, and static analysis.

Figures 4.2, 4.3, and 4.4 show SBT-Fizzer's coverage compared with the other tools, as well as SBT-Fizzer's end status. There are a few observations to be made here. First, SBT-Fizzer frequently achieves worse coverage than the other tool, even when it exhausts the allocated time. For example, the other tools have a distinct advantage in the subcategories ECA and ControlFlow, which contain long sequences of `if` conditions that check for equality with specific values. In these cases, bounded model checking and concolic execution thrive at finding inputs satisfying these conditions using an SMT solver.

The performance of SBT-Fizzer on programs that produce a long trace and br instruction trace is primarily limited[6] by the speed of the exchange of information between the fuzzer and the program. For contrast, when a branch is executed multiple times, AFL (used by VeriFuzz and FuSe-BMC) buckets the count as described in subsection 1.3.1, while SBT-Fizzer saves information about each execution of the branch. The benchmarks in the subcategory Arrays, where SBT-Fizzer performed worse, often contain loops that iterate numerous times. In such instances, the fuzzer cannot fuzz most of the program because the trace reaches the enforced size limit. However, the fuzzer's performance would significantly deteriorate if the limit was not imposed.

Furthermore, SBT-Fizzer performs suboptimally in the subcategory Floats. The fuzzer often stops early on these benchmarks because it depletes its fuzzing strategies. The benchmarks consist of floating-point arithmetic and branches that check if the input falls within a specific range or is equal to a specific value. The fuzzer's current strategies are inadequate in these cases.

The fuzzer obtains the best coverage from the tools on the subcategories Recursive and DeviceDrivers. Additionally, SBT-Fizzer beats LibKluzzer and VeriFuzz on some benchmarks where the two tools are terminated due to hitting the memory limit.

[6] Measured by profiling with Callgrind.

Table 4.2: Coverage (%) achieved on the subcategories of Test-Comp's Branch-Coverage category by SBT-Fizzer, LibKluzzer, VeriFuzz, and FuSe-BMC. The total and the score are calculated as in table 4.1.

Subcategory	Programs	SBT-Fizzer	LibKluzzer	VeriFuzz	FuSe-BMC
Arrays	400	26418	32265	28231	32402
BitVectors	62	4392	4820	4839	4706
Combinations	671	31818	31747	35597	37432
ControlFlow	67	2810	3956	4356	4109
ECA	29	832	1075	1163	945
Floats	226	7944	10331	11657	11162
Heap	143	10180	10388	10137	9918
Loops	727	57173	57561	57826	57765
ProductLines	263	7541	7672	7670	5701
Recursive	53	4482	4261	3536	4230
Sequentialized	103	6691	7463	9003	8833
XCSP	119	10162	11691	9753	9850
BusyBox	75	2251	2077	2680	2392
DeviceDrivers	290	5851	5716	5645	5740
SQLite	1	3.64	3.67	4.15	0
MainHeap	231	18664	20365	19982	20845
Total	3460	197212	211390	212079	216030
Score	—	181249	198704	198926	198250

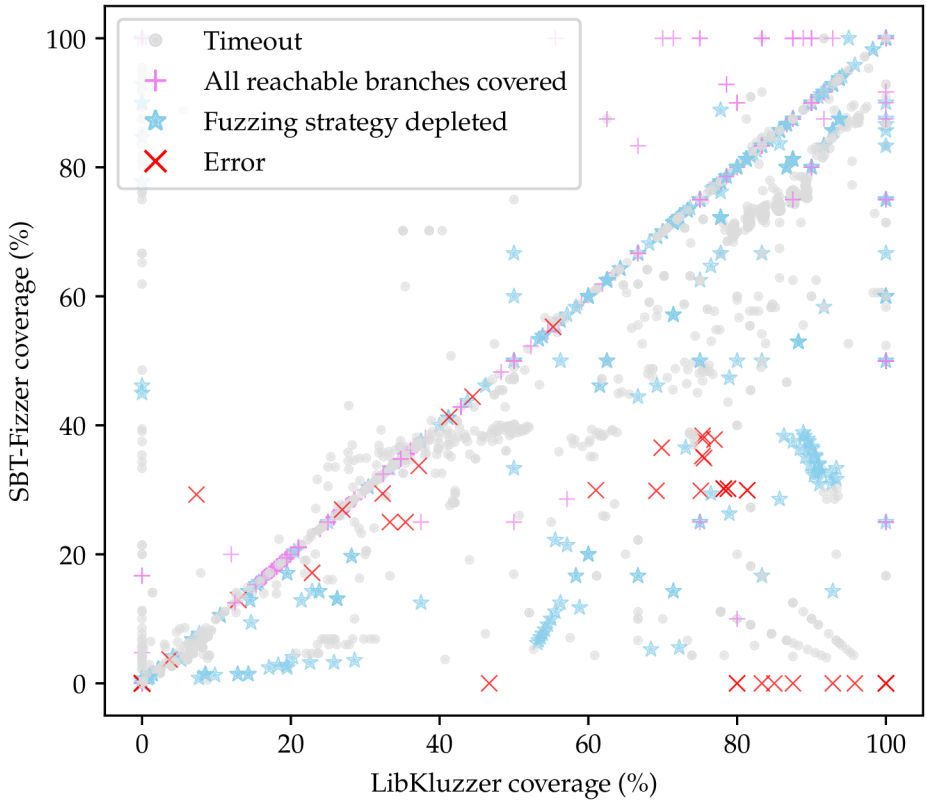

Figure 4.2: Scatter plot of the coverage of SBT-Fizzer and LibKluzzer. The points are divided based on SBT-Fizzer's end status, as depicted in the legend.

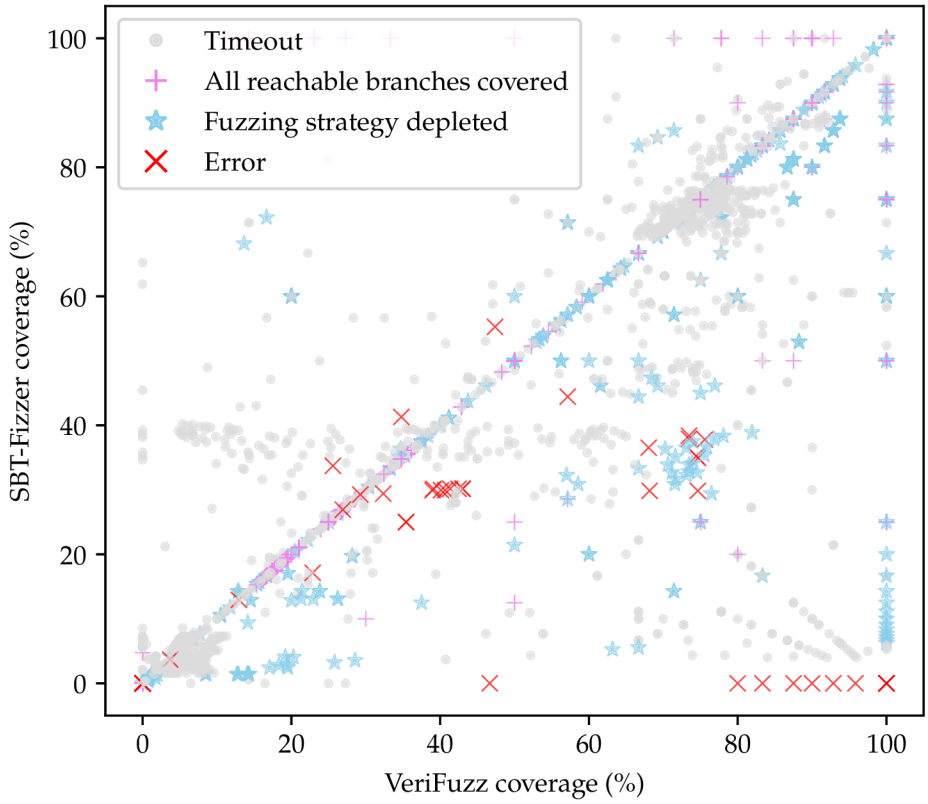

Figure 4.3: Scatter plot of the coverage of SBT-Fizzer and VeriFuzz. The points are divided based on SBT-Fizzer's end status, as depicted in the legend.

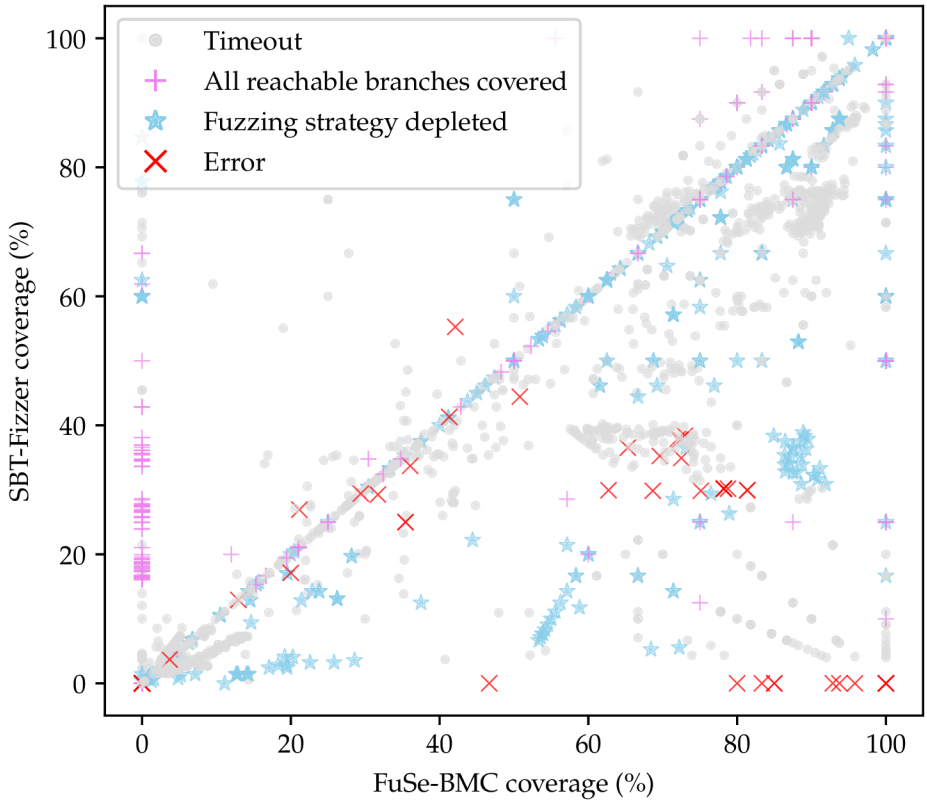

Figure 4.4: Scatter plot of the coverage of SBT-Fizzer and FuSe-BMC. The points are divided based on SBT-Fizzer's end status, as depicted in the legend.

5 Conclusion

The main contribution of this thesis, described in the third chapter, is the improvement of SBT-Fizzer, which allows it to fuzz benchmarks in the tool competition Test-Comp. Fuzzing a C program now involves compiling it into LLVM IR, instrumenting it with functions that collect coverage information, and linking it with the fuzzer's libraries to create an executable. The fuzzer then executes the executable to fuzz the target program. The fuzzer and the executable exchange data using TCP or shared memory.

SBT-Fizzer was then evaluated on the benchmarks and compared to the tools LibKluzzer, VeriFuzz, and FuSe-BMC participating in Test-Comp. Although the fuzzer did not outperform any of the other tools, the results are promising, as it uses purely grey-box fuzzing, whereas the other tools combine it with different techniques. The evaluation highlights areas where the fuzzer has room for improvement. It is worth investigating combining the fuzzer with other program analysis techniques and reducing the amount of information passed from the program to the fuzzer. The performance can be further increased by adding another execution mode of the target program, drawing inspiration from AFL's fork server.

The fuzzer is limited to fuzzing programs using `__VERIFIER_nondet_` functions to read input. Fuzzing programs consisting of multiple source files is also quite tedious. For SBT-Fizzer to be easily deployable on real software, further development is needed. For example, a Clang wrapper, which would instrument the code with SBT-Fizzer pass and compile it to make the SBT-Fizzer target, would allow the user to compile a larger project and fuzz it readily. Another key feature is supporting other ways the target program can read input, such as from stdin or from a file.

A Electronic attachments

The archive submitted with this thesis contains:

- Directory `sbt-fizzer-client-v1` with the source code of SBT-Fizzer with client version 1.
- Directory `sbt-fizzer-client-v2` with the source code of SBT-Fizzer with client version 2.
- File `README.md` with instructions for building and running the fuzzer.

Bibliography

1. MILLER, B. P.; FREDRIKSEN, L., and SO, B. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM*. 1990, vol. 33, no. 12, pp. 32–44. ISSN 0001-0782. Available from DOI: 10.1145/96267.96279.
2. CHEN, C.; CUI, B.; MA, J.; WU, R.; GUO, J., and LIU, W. A systematic review of fuzzing techniques. *Computers & Security*. 2018, vol. 75, pp. 118–137.
3. MANES, V. M.; HAN, H.; HAN, C.; CHA, S.; EGELE, M.; SCHWARTZ, E. J., and WOO, M. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering*. 2021, vol. 47, no. 11, pp. 2312–2331. ISSN 1939-3520. Available from DOI: 10.1109/TSE.2019.2946563.
4. JONES, D. *Trinity: Linux system call fuzzer* [online]. [visited on 2023-05-03]. Available from: https://github.com/kernelslacker/trinity.
5. GODEFROID, P.; LEVIN, M. Y.; MOLNAR, D. A., et al. Automated whitebox fuzz testing. In: *NDSS*. 2008, vol. 8, pp. 151–166.
6. LIANG, H.; PEI, X.; JIA, X.; SHEN, W., and ZHANG, J. Fuzzing: State of the Art. *IEEE Transactions on Reliability*. 2018, vol. 67, no. 3, pp. 1199–1218. Available from DOI: 10.1109/TR.2018.2834476.
7. LLVM PROJECT. *libFuzzer - a library for coverage-guided fuzz testing.* [online]. [visited on 2023-01-29]. Available from: https://llvm.org/docs/LibFuzzer.html.
8. GODEFROID, P. Fuzzing: Hack, art, and science. *Communications of the ACM*. 2020, vol. 63, no. 2, pp. 70–76.
9. CHEN, P. and CHEN, H. Angora: Efficient Fuzzing by Principled Search. In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 711–725. Available from DOI: 10.1109/SP.2018.00046.
10. MILLER, C.; PETERSON, Z. N., et al. Analysis of mutation and generation-based fuzzing. *Independent Security Evaluators, Tech. Rep*. 2007, vol. 4.
11. *Peach Fuzzer Community Edition v3* [online]. [visited on 2023-05-06]. Available from: https://gitlab.com/peachtech/peach-fuzzer-community.
12. YANG, X.; CHEN, Y.; EIDE, E., and REGEHR, J. Finding and understanding bugs in C compilers. In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 2011, pp. 283–294.

13. SOMOROVSKY, J. Systematic Fuzzing and Testing of TLS Libraries. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. Vienna, Austria: Association for Computing Machinery, 2016, pp. 1492–1504. CCS '16. ISBN 9781450341394. Available from DOI: 10.1145/2976749.2978411.

14. ZALEWSKI, M. *Technical "whitepaper" for afl-fuzz* [online]. [visited on 2023-01-29]. Available from: https://lcamtuf.coredump.cx/afl/technical_details.txt.

15. ZALEWSKI, M. *american fuzzy lop* [online]. [visited on 2023-05-03]. Available from: https://lcamtuf.coredump.cx/afl/.

16. KLEES, G.; RUEF, A.; COOPER, B.; WEI, S., and HICKS, M. Evaluating Fuzz Testing. 2018. Available from DOI: 10.48550/ARXIV.1808.09700.

17. FIORALDI, A.; MAIER, D.; EISSFELDT, H., and HEUSE, M. AFL++: Combining Incremental Steps of Fuzzing Research. In: *Proceedings of the 14th USENIX Conference on Offensive Technologies*. USA: USENIX Association, 2020. WOOT'20.

18. *Motivation behind AFL* [online]. [visited on 2023-05-03]. Available from: https://afl-1.readthedocs.io/en/latest/motivation.html.

19. ZALEWSKI, M. *Binary fuzzing strategies: what works, what doesn't* [online]. 2014. [visited on 2023-01-29]. Available from: https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html.

20. ZALEWSKI, M. *Fuzzing random programs without execve()* [online]. 2014. [visited on 2023-01-29]. Available from: https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html.

21. BOEHME, M.; CADAR, C., and ROYCHOUDHURY, A. Fuzzing: Challenges and Reflections. *IEEE Software*. 2021, vol. 38, no. 3, pp. 79–86. Available from DOI: 10.1109/MS.2020.3016773.

22. DOLAN-GAVITT, B.; HULIN, P.; KIRDA, E.; LEEK, T.; MAMBRETTI, A.; ROBERTSON, W.; ULRICH, F., and WHELAN, R. Lava: Large-scale automated vulnerability addition. In: *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 110–121.

23. *DARPA Cyber Grand Challenge* [online]. [visited on 2023-05-06]. Available from: https://github.com/CyberGrandChallenge/.

24. BEYER, D. Advances in Automatic Software Testing: Test-Comp 2022. In: JOHNSEN, E. B. and WIMMER, M. (eds.). *Fundamental Approaches to Software Engineering*. Cham: Springer International Publishing, 2022, pp. 321–335. ISBN 978-3-030-99429-7.

25. BEYER, D. and LEMBERGER, T. TestCov: Robust Test-Suite Execution and Coverage Measurement. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019, pp. 1074–1077. Available from DOI: 10.1109/ASE.2019.00105.

26. LE, H. M. LLVM-based Hybrid Fuzzing with LibKluzzer (Competition Contribution). In: WEHRHEIM, H. and CABOT, J. (eds.). *Fundamental Approaches to Software Engineering*. Cham: Springer International Publishing, 2020, pp. 535–539. ISBN 978-3-030-45234-6.
27. LE, H. M. KLUZZER: Whitebox Fuzzing on Top of LLVM. In: CHEN, Y.-F.; CHENG, C.-H., and ESPARZA, J. (eds.). *Automated Technology for Verification and Analysis*. Cham: Springer International Publishing, 2019, pp. 246–252. ISBN 978-3-030-31784-3.
28. CADAR, C.; DUNBAR, D.; ENGLER, D. R., et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: *OSDI*. 2008, vol. 8, pp. 209–224.
29. THE CLANG TEAM. *SanitizerCoverage* [online]. [visited on 2023-05-06]. Available from: https://clang.llvm.org/docs/SanitizerCoverage.html.
30. BASAK CHOWDHURY, A.; MEDICHERLA, R. K., and R, V. VeriFuzz: Program Aware Fuzzing. In: BEYER, D.; HUISMAN, M.; KORDON, F., and STEFFEN, B. (eds.). *Tools and Algorithms for the Construction and Analysis of Systems*. Cham: Springer International Publishing, 2019, pp. 244–249. ISBN 978-3-030-17502-3.
31. METTA, R.; MEDICHERLA, R. K., and KARMARKAR, H. VeriFuzz: Good Seeds for Fuzzing (Competition Contribution). In: JOHNSEN, E. B. and WIMMER, M. (eds.). *Fundamental Approaches to Software Engineering*. Cham: Springer International Publishing, 2022, pp. 341–346. ISBN 978-3-030-99429-7.
32. ALSHMRANY, K. M.; MENEZES, R. S.; GADELHA, M. R., and CORDEIRO, L. C. FuSeBMC: A White-Box Fuzzer for Finding Security Vulnerabilities in C Programs (Competition Contribution). In: GUERRA, E. and STOELINGA, M. (eds.). *Fundamental Approaches to Software Engineering*. Cham: Springer International Publishing, 2021, pp. 363–367. ISBN 978-3-030-71500-7.
33. ALSHMRANY, K. M.; ALDUGHAIM, M.; BHAYAT, A., and CORDEIRO, L. C. FuSeBMC v4: Smart Seed Generation for Hybrid Fuzzing. In: JOHNSEN, E. B. and WIMMER, M. (eds.). *Fundamental Approaches to Software Engineering*. Cham: Springer International Publishing, 2022, pp. 336–340. ISBN 978-3-030-99429-7.
34. LLVM PROJECT. *LLVM Language Reference Manual* [online]. [visited on 2023-01-29]. Available from: https://llvm.org/docs/LangRef.html.
35. SCHUMILO, S.; ASCHERMANN, C.; ABBASI, A.; WÖRNER, S., and HOLZ, T. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types. In: *USENIX Security Symposium*. 2021, pp. 2597–2614.