IT University of Copenhagen

Exploring grammar-based fuzzing techniques

18th December, 2024

Lasse Faurby Klausen *lakl@itu.dk*
Nicolai Grymer *ngry@itu.dk*

Supervisor Mahsa Varshosaz
Course code KSADSOA1KU

Contents

1 Introduction

Fuzzing plays a critical role in software testing and compiler verification, as it provides an effective way to identify vulnerabilities and edge cases that might not be captured through traditional testing methods. By generating random or semi-random inputs, fuzzing exercises the limits of a program or compiler, exposing issues related to correctness, error handling and security. This testing approach is particularly great at working with complex systems or languages, where exhaustive manual testing is impossible.

In this project, we explore the implementation and testing of various heuristics to improve the performance of our own fuzzer targeting a subset of C[1]. The focus is on gaining a deeper understanding of fuzzing techniques and experimenting with different heuristics—such as max depth constraints, termination through waves, and probabilistic values to examine the fuzzer's efficiency and quality. These heuristics are designed to optimize the balance between code coverage and the time it takes to generate inputs, ensuring that the fuzzer can effectively test a wide range of code paths while remaining computationally efficient.

Through systematic experimentation, we aim to evaluate the performance of these heuristics with respect to their impact on code coverage in the Tiny C Compiler (TCC). By analysing the effects of each heuristic, we seek to identify optimal configurations that improve the fuzzer's effectiveness and practicality.

[1] `https://en.wikipedia.org/wiki/C_(programming_language)`

2 Background

Fuzzing is a software testing technique that generates random or semi-random data into a program to identify vulnerabilities, crashes, or unexpected behaviour. Fuzzing is classified as *negative testing*[3], also known as failure testing, or error path testing. In the domain of fuzzing compilers, it specifically targets the structural and syntactic rules of a program, yielding great results in uncovering subtle flaws in parsers, interpreters, and compilers[6]. Programs may also contain structural or syntactic incorrect parts, which test the compilers ability to diagnose incorrectness.

Fuzzy testing have made its way into various domains, including software development, cybersecurity, and quality assurance, where it is used to uncover vulnerabilities and ensure system robustness. E.g. since its launch in 2016, Google's OSS-Fuzz platform has contributed to the resolution of over 8,800 security vulnerabilities and 28,000 bugs in 850 critical open-source projects [8].

The following describes the state of the art fuzzing techniques that enable projects like Google's OSS-Fuzz platform.

2.1 Grammar-Based Fuzzing

Grammars can serve as a guideline for generating inputs randomly, enabling fuzzers to produce structured data, code or input that adheres to specific formats or protocols. This approach leverages predefined grammar rules to create syntactically valid inputs, making it particularly useful for testing complex systems. Grammar-based fuzzers typically choose a specific rule as their entry point, and generate from there [9]. Great example of grammar-based fuzzers include source code generators for testing compilers [1, 4]. Yang et al. [2011] found more than 325 bugs in C compilers while building CSmith.

2.2 Mutation-Based Fuzzing

Modern fuzzers often use mutation-based techniques, altering existing valid inputs, or generation-based methods that create inputs from scratch based on predefined rules. Effective input generation increases the likelihood of discovering hidden flaws by thoroughly exercising the software under test. A mutation could be e.g. a simple string manipulation, inserting a random character, deleting a charater, flipping a bit, etc. This can ensure that we get a high proportion of valid inputs [9].

2.3 Coverage-Guided Fuzzing

Coverage-guided fuzzers use feedback from code execution to inform their input generation process. By analyzing code coverage metrics, these fuzzers prioritize paths that have not been tested, maximizing exploration and increasing the likelihood of finding edge-case bugs [10].

2.4 White-Box vs. Black-Box Fuzzing

Black-box fuzzing generates inputs randomly without any knowledge of the program's internal structure. It mutates well-formed inputs or uses grammars to produce test cases, making it simple to implement and broadly applicable. However, it often struggles with low code coverage, especially in scenarios requiring specific input values to trigger particular code paths.

White-box fuzzing, in contrast, uses program analysis techniques like dynamic symbolic execution to guide input generation. It gathers constraints from conditional statements during execution and solves them to create new inputs that systematically explore different paths in the code. This approach achieves higher code coverage and is more effective at uncovering bugs deeply nested in software [2].

2.5 Fuzzing heuristics

The different techniques in fuzzing require controllable parameters such that developers or researchers can target specific parts of the targeted domain. This could be controlling the max depth[2], or probability of a certain production rule to be chosen among all rules of a non-terminal. Besides probabilistic choice of production rule, max depth and chance of mutation are examples of heuristics in fuzzing.

2.6 Modern fuzzers

Modern fuzzers improved upon black-box fuzzers by introducing techniques like runtime analysis, taint tracking, and symbolic execution. These methods enable testers to target critical program paths, significantly improving efficiency and effectiveness compared to random input generation. Some fuzzers adjust their inputs based on feedback, and memory-based methods allow them to skip checks to test deeper parts of a system. Hybrid methods, like genetic algorithms, create better inputs, while advanced formats like ASN.1 and XML help test complex systems. Additionally, modern fuzzers integrate into development processes by generating automated reports and providing actionable insights, streamlining bug identification and resolution [5].

Advancements in the field continue to emerge, as seen in the state-of-the-art DOM fuzzer developed by Xu et al. [2020], which demonstrates up to 3.74x higher throughput through browser self-termination, surpassing existing tools. FreeDom's use of context-aware generation helped identify three times more unique crashes in WebKit compared to the leading fuzzer, Domato, and uncovers 24 previously unknown bugs across major browsers [7].

[2]Maximum amount of production rule expansion

3 Methodology

We chose to fuzz the C compiler TinyC (TCC)[3], with a subset of the language C, henceforth refer to as CLN. The subset is based on SmallC[4], which we extended with global variables, more primitive types, arrays, comments, strings, functions, do-while loops and while loops. Initially, we experimented with the TinyC language[5], which proved useful for constructing the core of the fuzzer. However, its limited set of language features constrained our ability to analyze and optimize the fuzzer's heuristics. By switching to CLN and its larger grammar, it enabled the generation of a diverse range of programs, making the process of optimizing our heuristics possible. See the CLN grammar in Appendix A.

The final grammar requires a `main` function as the entry point, which includes variable declarations, a sequence of statements, and a return value. Optional global declarations, such as variables or functions, may precede the `main` function. These global declarations and functions ensure that subsequent code generation can utilize variables and functions without needing to check whether they have been declared earlier. Functions specify parameters via lists and encapsulate logic in compound statements, which may include nested declarations and subordinate statements.

Variable declarations support primitive types like `int`, `float` and `char` with optional modifiers (`const` or signedness). Arrays and strings are included, allowing dimension specifications and optional initialization.

Control flow constructs include `if-else`, `switch`, `while`, `do-while`, and `for` loops. The `switch` construct supports `case` and `default` clauses for complex branching. Expressions allow assignments, conditionals, arithmetic, logical operations (`&&`, `||`), and ternary expressions (`? :`). Statements include function calls, return statements, and compound blocks. Function declarations specify return types and parameter lists. Lastly, both whitespace and multiline comments are supported.

The grammar is limited by the architecture of our grammar-based fuzzer. Each additional syntax constraint further restricts the ability to implement additional language features. For example, pointers were omitted as their inclusion would require extending the variable system to manage both pointers and non-pointers. Additionally, only one-dimensional arrays are supported, and there is no use of external functions such as *printf()* or memory management operations like *malloc()*, *free()*, or *sizeof()*.

Other interesting features that were excluded due to time constraints include:

- **Bitwise Operators**: (&, |, ^, ~)
- **Shift Operators**: («, »)

[3] `https://bellard.org/tcc/`

[4] `https://medium.com/@efutch/a-small-c-language-definition-for-teaching-compiler-design-b70198531a2f`

[5] `http://www.iro.umontreal.ca/~felipe/IFT2030-Automne2002/Complements/tinyc.c`

- **Increment/Decrement Operators**: (++ and –)
- **Recursive Functions**: While we allow functions to call themselves, the system does not implement optimizations typically handled by compilers for recursion. Such optimizations may occur but are not guaranteed.

Lastly, advanced data structures such as structs, custom types, and enums were not implemented. These features would require a more sophisticated variable management system within the fuzzer, which was beyond the scope of this project.

3.1 Heuristics

Two areas sparked our interest when implementing heuristics. First involves tackling the problem of termination and the second involves probabilistic fuzzing. These were chosen given their relevance and impact on the outcome.

3.1.1 Max Depth

This heuristic serves both as a controllable parameter and a technical necessity to prevent the fuzzer from infinitely expanding grammar rules, and thus never finishing.

3.1.2 Wave depth

Merely limiting the depth does not stop the ever expanding nature of the fuzzer as it will merely stay at around the max depth for indefinite time. To accommodate this, the max-depth heuristics would require some form of exit plan. A simple solution is to find the quickest path to termination, which would result in a single `triangle wave`, where the fuzzer immediately seeks max depth, given its expansive nature, and then seek the quickest termination, which is at depth 0. But a more sophisticated solution, would be to introduce multiple waves. This means that the fuzzer would target max-depth, followed by a min-depth, and then a max-depth again. This would repeat as per amount of wave counts. The resulting heuristics for the waves will strongly correlates with the execution time.

3.1.3 Probabilistic Fuzzing

Instead of selecting rules uniformly at random, the fuzzer uses these probabilities to guide the generation process, prioritizing certain rules based on their likelihood of producing more interesting, diverse, or impactful test cases. This heuristic allows the fuzzer to focus on paths in the input space that are more likely to uncover bugs or edge cases, while still maintaining enough randomness to explore less common scenarios. This is inspired by the section on probabilistic programming written by Zeller et al. [2024].

4 Implementation

In this section, we will elaborate on our fuzzer implementation, cover the core components, heuristic implementations and design choices. The code is publicly available at GitHub [6].

The fuzzer is written in `Go`, which has little overhead meaning we're getting close to system-language performance. Initial implementation attempts using two different types of architectures. One involved using structs to specify properties for each type of grammar rule, whereas the other involved a central looping mechanism, where the grammar rule would be parsed, meaning it would be grammar-agnostic. We continued with the architecture of the second type. It takes a grammar in the form of a string map pointing to an array of grammar-rules. Each grammar-rules has both the rule itself and its probability. The following code example shows how the generator stores the language rules:

```
type languageRules = map[string][]rule

type rule struct {
    ruleString
    ruleProbability
}

type ruleString = string
type ruleProbability = float64
```

Nested grammar rules are processed recursively, with the call stack reflecting the depth of the grammar rules traversed. To accumulate results, we used a string builder, which employs efficient array operations for concatenating strings, rather than constructing a parse tree. While a tree-based approach would have been preferable for enabling advanced post-processing tasks, such as complex mutation-based fuzzing, the string-builder approach kept the solution simple.

Many grammar-based fuzzers allow developers to define both grammatical rules and constraints, enabling not only the specification of a language's syntax but also ensuring correctness in variable declarations, scoping, assignments, and function definitions. Avoiding issues like redeclaration, using only previously declared variables and functions, preserving const immutability, and adhering to scope constraints remains a significant challenge.

Rather than implementing a semantic rules system, we introduced specialized terminal tokens that reference functions, variables, and other constructs. The keywords associated with these tokens are detailed in Table 1. While this approach limits the expression of complex semantic rules, it provided the necessary features to generate valid CLN code effectively.

The fuzzer integrates a variable and function system to prevent invalid operations, such as assigning

[6] `https://github.com/Grymse/go-fuzz-tcc`

Keyword	**Description**
$FUNC_DECL$	Represents the declaration of a function, including its name, parameters, and return type.
ID_DECL	Denotes the declaration of an variable identifier.
ID_DECL_C	Refers to the declaration of a constant identifier.
$ID_DECL_ARR_C$	Represents the declaration of a constant array identifier.
ID_DECL_ARR	Denotes the declaration of an array identifier.
ID	Represents a general identifier, such as a variable name or function name.
ID_AS	Indicates an identifier with an assigned value.
$FUNC$	Represents the invocation or definition of a function.

Table 1: Descriptions of Grammar Keywords

to constant variables or invoking functions with incorrect names or parameter counts. The variable system is designed for extensibility, allowing for future enhancements. In C, the ability to assign any primitive type to another simplifies generating large expressions involving comparisons, arithmetic, and logical operations.

The syntax is defined using a formal grammar similar to Backus-Naur Form (BNF). Non-terminal symbols, such as *"<type_specifier>"* can be expanded into multiple different production rules. In this case it is *"char", "<number_type>" & "const <number_type>"*.

To handle scoping, a system was implemented to track braces (*{* and *}*), adjusting scope and removing out-of-scope variables. However, by only adjusting scope by looking at bracing, constructs like for-loops would not work. For example, in *"for(int i = 0;;) {}"*, the variable *i* is only in scope during the loop, but not after. To manage this, a ^ symbol was introduced at the start of production rules to manually increase the scope, e.g., *"^for (...)"*. This implies that the scope is one higher during the production rule, meaning all variables declared during the line will cease to exist after the production rule.

On top of this, there are two extra variables that increase the possibilities of the grammar language:

- The $*$ operator repeats a rule 1 to 10 times, as in *"<number_type*>"*.
- The ! operator repeats a rule once for each production rule it has, as in *"<statements!>"*.
- The ˆ operator indicates a new scoping level *"^do <statements> while <condition>"*.

As the fuzzer was developed, the CLN grammar evolved to accommodate generation rather than parsing. This led to the restrictions that are not a part of C, but eased the process of generating code. E.g. global variables must be listed first, followed by functions and, finally, the main function. This is suboptimal, as it depends on the specific grammar of CLN, rather than supporting a more general, flexible grammar. This dependency limits the fuzzer's versatility and its ability to adapt to other grammars.

The following is a boiled down version of the generator. It does not include scoping, edge-cases and termination guarantee:

```
func processRule(rules []Rules) {
    unprocessedRule = selectRuleProbabilistic(rules)
    // Replace special rules $INT$, $ID$, $ID_DECL$ etc.
    rule = replaceSpecialTerminals(unprocessedRule)

    for {
        find next '<' and '>'

        if '<' not exist:
            output.append(rule[pointer:])
            break

        if <> pair is nonTerminal:
            processRule(nonTerminal.rules)
            pointer = '>'-index + 1
        else
            output.append(rule[pointer])
            pointer += 1
    }
}
```

4.0.1 Max Depth

To address infinite expansion of grammar rules, we implemented a max-depth constraint. When the max depth is reached, the generator seeks the fastest path to a terminal, completing all open grammar rules with minimal steps. We developed an algorithm based on a modified BFS, where terminals are prioritized at the beginning of the queue. This algorithm calculates the cost of grammar rules dynamically, enabling the generator to select the cheapest option at any point.

4.0.2 Wave depth

To improve code diversity and prevent repeated expansion of specific grammar rules, we introduced a varying depth target. This operates as a triangular wave pattern, alternating between reaching the max depth and targeting a shallower depth (e.g., 4). Each cycle, or wave, is defined by parameters such as wave peak and a ranges of possible wave valleys (`WaveValleyMin`, `WaveValleyMax`). The range makes the fuzzer randomly choose a valley target point between these two. This ensures exploration of different parts of the grammar during each iteration.

4.0.3 Probabilistic Fuzzing

We integrated probabilistic fuzzing by associating production rules with probabilities instead of selecting them uniformly. The fuzzer prioritizes rules dynamically, guiding the generation process toward paths likely to produce diverse or impactful test cases. By adjusting probabilities based on target-specific needs, the implementation balances exploration of error-prone constructs with randomness to maintain variety in the generated code.

We are left with the following heuristics for optimizing the code: `WaveValleyMin`, `WaveValleyMax`, `MaxDepth`, `WaveCount` and probabilities for each language rule.

5 Experiment Design

To evaluate our fuzzer implementation, we designed experiments to assess two key metrics: *TCC compiler coverage* and *time to generate input* for the compiler.

The primary objective of the fuzzer is to maximize code coverage, specifically targeting as much of the source code in the `TCC` compiler as possible. Ideally, a fuzzer should generate diverse inputs that cover a wide range of paths in the code. However, since our fuzzer operates on a subset of the C programming language, achieving complete coverage (i.e., 100%) is not feasible.

In addition to coverage, it is crucial to consider the time it takes to generate input. A heuristic that is overly complex or computationally expensive may render the fuzzer impractical, especially if the input generation time is excessively long. For example, if it takes an hour to generate a single input, it would be impractical for large-scale testing. Therefore, it is important to strike a balance where the heuristic achieves a high coverage percentage while maintaining reasonable input generation times.

5.1 Coverage in `TCC`

One of the primary metrics for evaluating our fuzzer is the extent of code coverage achieved within the `TCC` compiler. Specifically, we aimed to measure how much of the compiler's source code was exercised by the inputs generated by the fuzzer.

To generate coverage reports for the `TCC` compiler, we configured the compilation process of `TCC` to enable code coverage analysis. This was done by including the following flags during the configuration step:

```
./configure --extra-cflags="-fprofile-arcs -ftest-coverage" --extra-ldflags="-
    fprofile-arcs -ftest-coverage"
```

These flags ensure that the `.gcda` and `.gcno` files are generated for all compiled `.c` files. When an input file is executed on the `TCC` compiler, `.gcov` files are produced for the `.c` files that were actually exercised during the run.

To gather sufficient data for analysis, we generated 10 input files and ran them through the `TCC` compiler. For each execution, any new lines, branches, or calls executed were incrementally added to the existing code coverage. This coverage was reset between each test iteration to ensure that the data reflected the current run exclusively.

The results, including the number of lines, branches, and calls executed, were plotted as percentage coverage using Python. These metrics were reported for the relevant files `tccgen.c`, `tccpp.c`, and `x86_64-gen.c`, that had `8727`, `3963` and `2329` lines of code, respectively. These files were identified as the only ones exhibiting significant variations in coverage based on the changes in the applied heuristics.

5.2 Time to generate input

The time required to generate each input was another important metric in evaluating the fuzzer's performance. For each of the 10 input files generated, we measured the time it took to create the input from start to finish. Once the generation times for all 10 inputs were recorded, we calculated the average time across these inputs to obtain a representative measure of the fuzzer's efficiency.

The average time to generate input was then plotted in a graph using Python, providing a clear comparison of how the different heuristics impacted the generation time. This metric is essential for assessing the practical feasibility of the fuzzer, as longer input generation times may render the tool less effective in real-world scenarios, where rapid input generation is often critical.

5.3 Heuristics

The primary focus of the experiments was to modify and refine our heuristics to assess how these changes impacted the performance of our fuzzer. Specifically, we aimed to isolate individual heuristics and systematically adjust their parameters across multiple iterations. By doing so, we could evaluate the performance of the fuzzer under different configurations and identify the specific values that enabled optimal performance. This iterative process allowed us to fine-tune the heuristics, ensuring that the fuzzer achieved the best possible balance between coverage and efficiency.

6 Results

Our implementation primarily focused on two heuristics: *probabilistic fuzzing* and our *wave heuristics*. These heuristics were essential to enhance input diversity, improve code coverage, and ensure the fuzzer avoided non-terminating states.

6.1 Probabilistic fuzzing

In Figure 1, we present the results of varying the probabilities assigned to grammar rules. In `Test 01`, all probabilities were set equally, ensuring uniform selection of production rules. In `Test 02`, `Test 03`, and `Test 04`, we incrementally increased the probabilities of higher-cost terms by 1, 10, and 100, respectively. Notably, the results for `Test 01` through `Test 03` were consistent, achieving approximately 49% code coverage in `tccgen.c`. However, in `Test 04`, where probabilities were disproportionately skewed by increasing the weight to 100, we observed a clear decline in code coverage.

In `Test 05`, we retained uniform probabilities for all grammar rules but assigned a slightly higher weight to statement-producing rules. This configuration produced results comparable to those of `Test 01` through `Test 03`.

From these observations, we conclude that evenly distributed probabilities across all grammar rules yield the most reliable results. Minor improvements can be achieved by giving a slight priority to statement-producing rules, but excessively skewed probability distributions negatively impact code coverage.

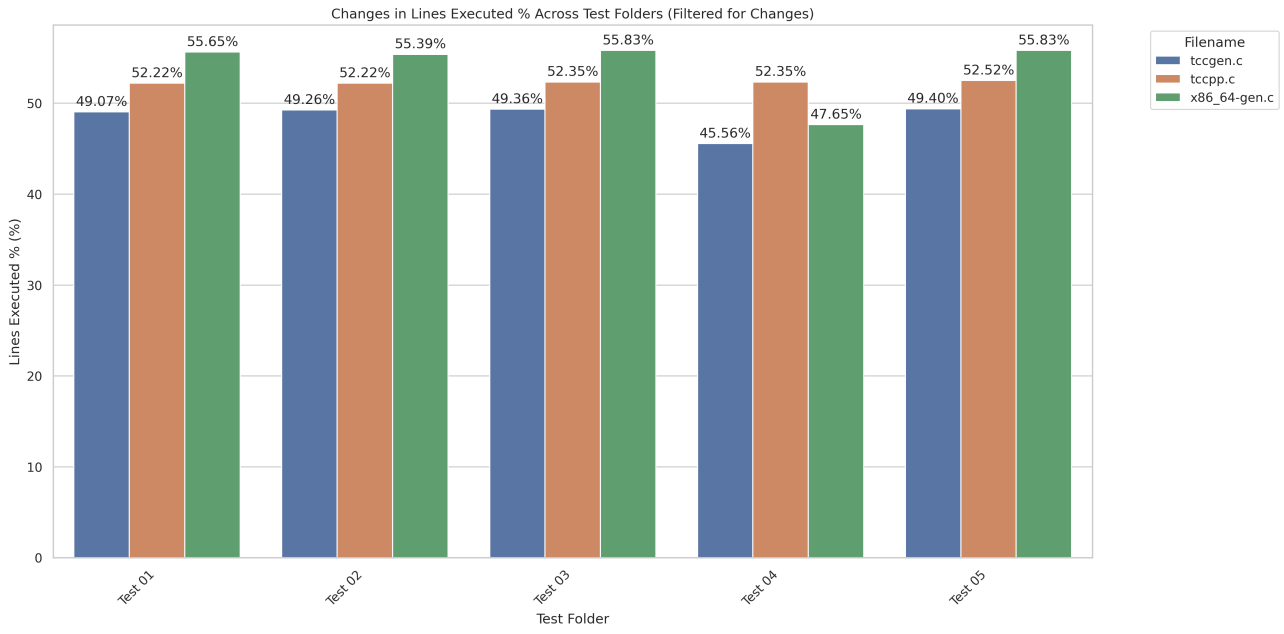

Figure 1: Lines executed changing the probabilistic heuristic.

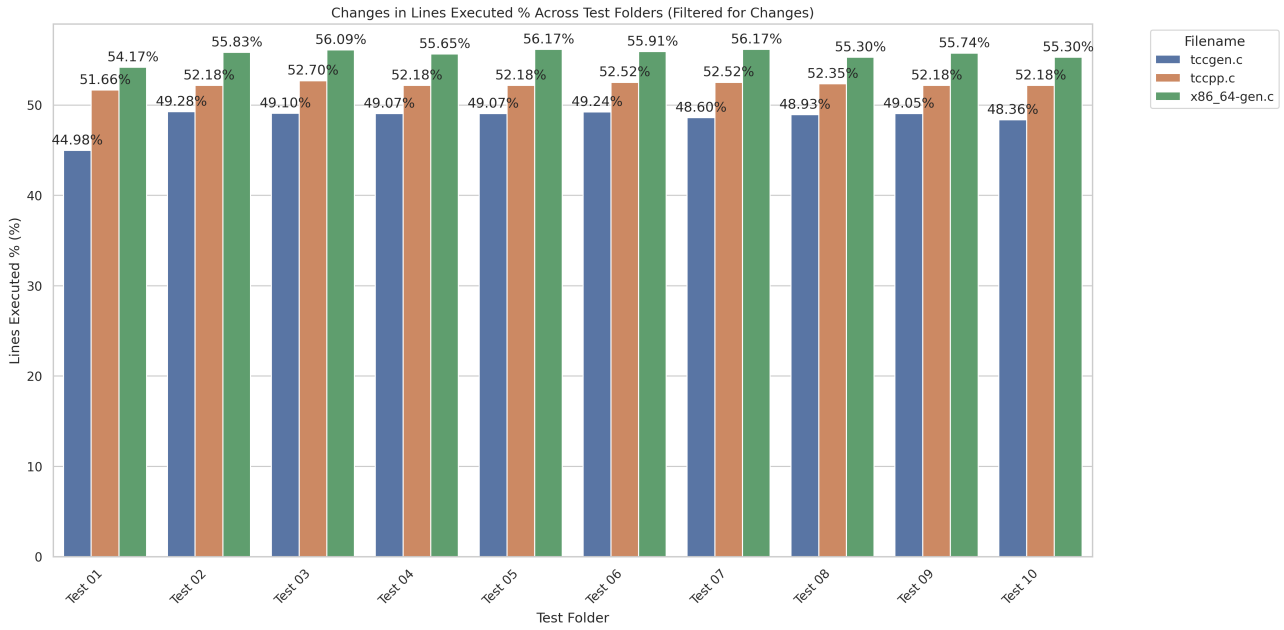

Figure 2: Lines executed by changing the max waves heuristic.

6.2 Wave heuristics

As discussed earlier, the wave heuristics incorporate several heuristics that can be adjusted. In this section, we analyse these heuristics to identify their optimal values.

6.2.1 Max Waves

We evaluated the impact of the *max waves* heuristic, which determines the number of waves before attempting to reach a terminal node in the grammar. The tested values were:

$$\{1, 2, 3, 5, 10, 20, 50, 100, 200, 500\}$$

As shown in Figure 2, in `Test 01`, only a single wave was used and achieved 44.98% code coverage in `tccgen.c`. For all subsequent tests with two or more waves, the coverage stabilized at approximately 49%, indicating no significant improvement when increasing the number of waves beyond a minimal threshold (e.g., 2 waves).

To mitigate the risk of variability due to chance (e.g., particularly "lucky" input generation in the case of 2 waves), it is crucial to use a slightly higher wave count to ensure more diverse input generation. Given that the input generation time increased by only around 20 milliseconds (see Appendix B.1), increasing the wave count to a value like 10 represents a practical choice that balances input diversity and computational efficiency.

6.2.2 Wave Peak

We examined the impact of the *wave peak* heuristic, which defines the maximum depth reached during each wave before attempting to terminate. The tested values were:

$$\{10, 50, 100, 150, 200, 250, 300, 350, 400, 450, 500\}$$

As shown in Figure 3, the most significant increase in the number of lines executed in the TCC compiler occurred between `Test 01` (with a wave peak of 10) and `Test 02` (with a wave peak of 50). Beyond this initial jump, increasing the wave peak from 50 to 100 did not result in a substantial improvement. However, both 50 and 100 as wave peak values still outperformed a wave peak of 10, demonstrating that a moderate increase in the wave peak provides better coverage without significant diminishing returns.

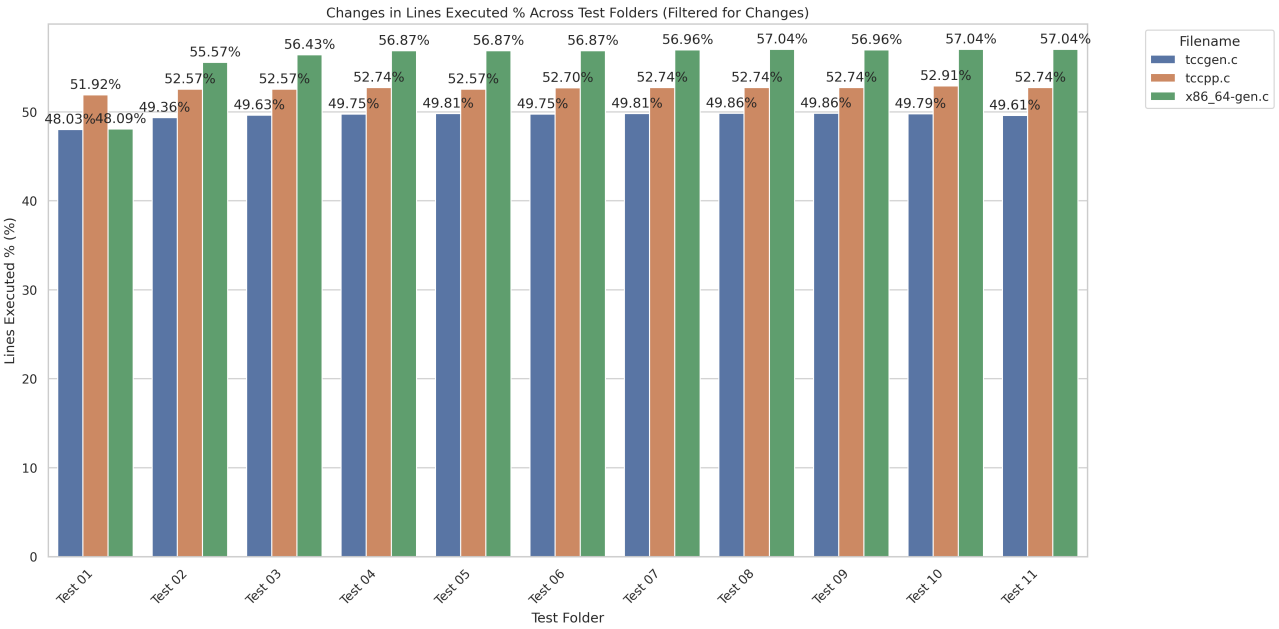

Figure 3: Lines executed by changing the wave peak heuristic.

6.2.3 Wave Valleys

We tested the number of wave valleys the algorithm reaches before attempting to terminate. The tested values were:

$$\{(1,2),\ (1,3),\ (1,6),\ (1,12),\ (5,10),\ (10,20),\ (2,3),\ (1,50),\ (50,100),\ (100,200)\}$$

Where the first part of the tuple is `waveValleyMin` and the second part is `waveValleyMax`.

As shown in Figure 4, there is a significant difference between $(1,2)$ in `Test 01` and $(1,3)$ in `Test 02`. From `Test01` to `Test 04`, we only increased the `waveValleyMax` value, and no significant improvement was observed when using 3 or 12 as the value.

We also tested other pairs, such as $(5, 10)$ in `Test 05`, $(10, 20)$ in `Test 06`, and $(2, 3)$ in `Test 07`. These configurations resulted in similar code coverage, suggesting that using $(2, 3)$ from `Test 07` to $(1, 50)$ from `Test 08` would not yield significant differences. However, we observed a notable drawback when using high values for `waveValleyMin` and `waveValleyMax`. For example, `Test 09` $(50, 100)$ and `Test 10` $(100, 200)$ resulted in only around 45% code coverage.

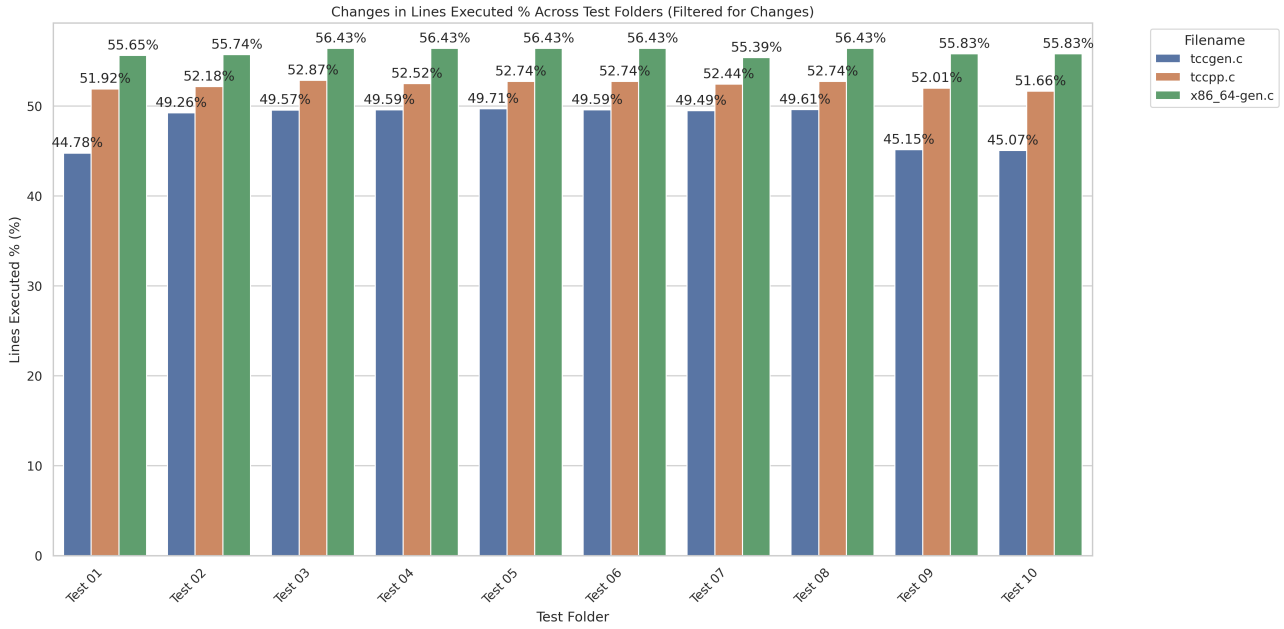

Figure 4: Lines executed of changing wave peak heuristic.

6.3 Summary of Results

In our evaluation, we primarily focused on *line execution* as the main metric for assessing code coverage across the different heuristics, as it provides the most direct measure of the fuzzer's ability to explore different parts of the code. The results from call execution and branch execution closely mirrored the line execution data, reinforcing our findings.

With *input generation time*, we observed that while some heuristics led to slightly longer generation times, the differences were generally minimal (on the order of 20 milliseconds). Given that these small variations did not significantly impact overall performance, we did not consider the time to generate inputs a critical factor in evaluating the heuristics' effectiveness.

7 Threats To Validity

7.1 No baseline to compare against

We attempted to find an external baseline to validate the effectiveness of our fuzzer by utilizing CSmith and Gramminator. However, CSmith only allowed for grammar specification via flag parameters and was not able to produce our CLN grammar. With Gramminator, which requires an ANTLR grammar, we were unable to develop a correct set of language constraints that would generate valid code for evaluation within the projects timeline. As a result, we cannot determine whether the final version of our fuzzer misses significant portions of the TCC compiler within the specified grammar.

7.2 Small subset of the C language

The CLN grammar itself represents a relatively small subset of the C language, as evidenced by the minimal impact our probabilistic approach had on the results. We did not assess the extent to which CLN covers the full C grammar, making it difficult to confirm whether the limited results are attributable to this subset's constraints.

8 Conclusion

In this project, we explored the implementation and evaluation of various heuristics to enhance the performance of a grammar-based fuzzer targeting our subset of the C programming language. Our primary goal was to understand different techniques effect on code coverage of the TCC compiler while maintaining computational efficiency. Through systematic experimentation, we focused on heuristics such as max depth constraints, termination through wave patterns, and probabilistic fuzzing to optimize input generation.

We presented a specific approach to obtaining a max depth constraints, which prevent infinite expansions of grammar rules to ensure computational feasibility. The wave termination heuristic allowed for greater code diversity by introducing multiple depth targets, and proved useful even using moderate wave counts and wave peaks. Lastly, probabilistic fuzzing yielded consistent code coverage when probabilities were uniformly distributed, with slight benefits observed when increasing probability of specific statement production rules. The probability fuzzing results are likely a product of the small subset of C, rather than applicable to ones general understanding of probability generation.

8.1 Future Work

Future work could focus on expanding the fuzzer's capabilities and improving its evaluation. First, the grammar could be expanded to include a larger subset of C or the full C language, incorporating features like pointers, bitwise operators, and multi-dimensional arrays for broader testing. Second, once a larger set of language features is introduced, we could increase coverage by introducing mutation-based fuzzing to enhance input diversity. Third, alternative performance metrics, such as program complexity, grammar coverage, input uniqueness or introduce symbolic execution to aim for edge cases, could provide a more comprehensive evaluation of the fuzzer. Fourth, implementing our grammar in Grammarinator would establish a baseline for comparison, helping to validate the effectiveness of our approach. Lastly, to support other grammars, the generator should be refactored to remove its inherent dependency on the C language for generation. This would enable alternative EBNF notations, such as those used by ANTLR, thereby enhancing its flexibility and broadening its range of applications.

References

[1] Kenneth V. Hanford. “Automatic generation of test cases”. In: *IBM Systems Journal* 9.4 (1970), pp. 242–257.

[2] Patrice Godefroid. “Random testing for security: blackbox vs. whitebox fuzzing”. In: *Proceedings of the 2nd International Workshop on Random Testing: Co-Located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*. RT ’07. Atlanta, Georgia: Association for Computing Machinery, 2007, p. 1. ISBN: 9781595938817. DOI: 10.1145/1292414.1292416. URL: https://doi.org/10.1145/1292414.1292416.

[3] Ari Takanen, Jared DeMott, and Charlie Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2008.

[4] Xuejun Yang et al. “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.

[5] Hongliang Liang et al. “Fuzzing: State of the art”. In: *IEEE Transactions on Reliability* 67.3 (2018), pp. 1199–1218.

[6] Rahul Gopinath and Andreas Zeller. “Building fast fuzzers”. In: *arXiv preprint arXiv:1911.07707* (2019).

[7] Wen Xu, Soyeon Park, and Taesoo Kim. “FREEDOM: Engineering a State-of-the-Art DOM Fuzzer”. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 971–986. ISBN: 9781450370899. DOI: 10.1145/3372297.3423340. URL: https://doi.org/10.1145/3372297.3423340.

[8] Oliver Chang. *Taking the next step: OSS-Fuzz in 2023*. https://security.googleblog.com/2023/02/taking-next-step-oss-fuzz-in-2023.html. Posted on the Google Security Blog by the OSS-Fuzz team. Feb. 2023.

[9] Andreas Zeller et al. “Mutation-Based Fuzzing”. In: *The Fuzzing Book*. Retrieved 2024-11-09 17:25:56+01:00. CISPA Helmholtz Center for Information Security, 2024. URL: https://www.fuzzingbook.org/html/MutationFuzzer.html.

[10] GitLab. *Coverage-guided fuzz testing*. Tier: Ultimate, Offering: GitLab.com, Self-managed, GitLab Dedicated. n.d. URL: https://docs.gitlab.com/ee/user/application_security/coverage_fuzzing/ (visited on 12/16/2024).

9 Appendix

A Grammar

```
program:
        global_decl? 'int' 'main' '(' ')' '{' variable_decl_as statement* '

global_decl: variable_decl_as | func_decl*;

func: FUNC '(' (expr (',' expr)*)? ')';

func_decl:
        type_specifier FUNC_DECL '(' (param_list)? ')' compound_statement;

param_list: param (',' param)*;

param: type_specifier ID;

compound_statement: '{' var_decl* statement* '}' | '{}';

var_decl: type_specifier var_decl_list ';';

type_specifier: number_types | 'char';

number_types:
        'int'
        | 'signed int'
        | 'unsigned int'
        | 'short'
        | 'signed short'
        | 'unsigned short'
        | 'long'
        | 'signed long'
        | 'unsigned long'
        | 'float'
        | 'double';

arr_decl:
```

```
        'const' number_types ID_DECL_ARR_C arr_decl_array
        | number_types ID_DECL_ARR arr_decl_array;

arr_decl_array:
        '[' INT ']'
        | '[' INT ']' '=' comma_separated_value
        | '[' INT ']' '=' '{}'
        | '[]'
        | '[]' '=' comma_separated_value
        | '[]' '=' '{}';

string_decl:
        'const' 'char' ID_DECL_ARR_C '[]' '=' LOREM
        | 'char' ID_DECL_ARR '[]' '=' LOREM;

comma_separated_value: value ',' comma_separated_value | value;

var_decl_list:
        variable_id_as
        | variable_id_as ',' var_decl_list;

variable_id_as: ID_DECL | ID_DECL '=' expr;

variable_decl_as:
        type_specifier ID_DECL '=' value ';'
        | type_specifier ID_DECL ';'
        | 'const' type_specifier ID_DECL_C '=' value ';'
        | string_decl ';'
        | arr_decl ';';

statement:
        switch_statement
        | do_while_statement
        | for_statement
        | compound_statement
        | cond_statement
        | while_statement
        | '//' LOREM statement
        | func ';'
```

```
        | 'return' ID ';';

switch_statement:
        'switch' '(' expr ')' '{' case_statement* '}'
        | 'switch' '(' expr ')' '{' case_statement* 'default' ':' statement

case_statement:
        'case' INT ':' statement 'break' ';'
        | 'case' CHAR ':' statement 'break' ';'
        | 'case' INT ':' statement
        | 'case' CHAR ':' statement;

do_while_statement:
        'do' loop_statement 'while' '(' expr ')' ';';

while_statement: 'while' '(' expr ')' loop_statement;

for_statement:
        'for' '(' 'int' ID_DECL '=' '0' ';' condition ';' ID_AS '=' expr ')
        | 'for' '(' ';' condition ';' ')' loop_statement;

loop_statement:
        '{' loop_inner_statement* '}'
        | '{}'
        | loop_inner_statement;

loop_inner_statement: statement | 'break' ';' | 'continue' ';';

cond_statement:
        'if' '(' expr ')' statement
        | 'if' '(' expr ')' statement 'else' statement;

expr: ID_AS '=' expr | condition;

condition: disjunction | disjunction '?' expr ':' condition;

disjunction: conjunction | disjunction '||' conjunction;

conjunction: comparison | conjunction '&&' comparison;
```

```
comparison:
        relation
        | relation '==' relation
        | relation '!=' relation;

relation:
        sum_
        | sum_ '<=' sum_
        | sum_ '<' sum_
        | sum_ '>=' sum_
        | sum_ '>' sum_;

sum_: term | sum_ '+' term | sum_ '-' term;

term: factor | term '*' factor | term '/' factor;

factor: '!' factor | '-' factor | primary;

primary:
        value
        | ID
        | func
        | '(' expr ')';

value: INT | INT '.' INT 'f' | CHAR;

INT: [0-9]+;

CHAR: '\'' [a-zA-Z0-9] '\'';

ID: [a-zA-Z_][a-zA-Z0-9_]*;

ID_DECL: ID;

ID_AS: ID;

ID_DECL_ARR: ID;
```

```
ID_DECL_ARR_C: ID;

ID_DECL_C: ID;

FUNC_DECL: ID;

FUNC: ID;

LOREM: '"' .*? '"';

WS: [ \r\n\t]+ -> skip;

COMMENT:
        '/*' LOREM '*/'
        | '/*' '\n' '*' LOREM '\n' '*/';
```

B Result graphs

This section presents the result graphs associated with various heuristics tested in the experiments. These graphs provide visual insights into the time to generate input, and the coverage of branches, calls, and lines executed under different configurations of the fuzzing algorithm.

B.1 Max Waves

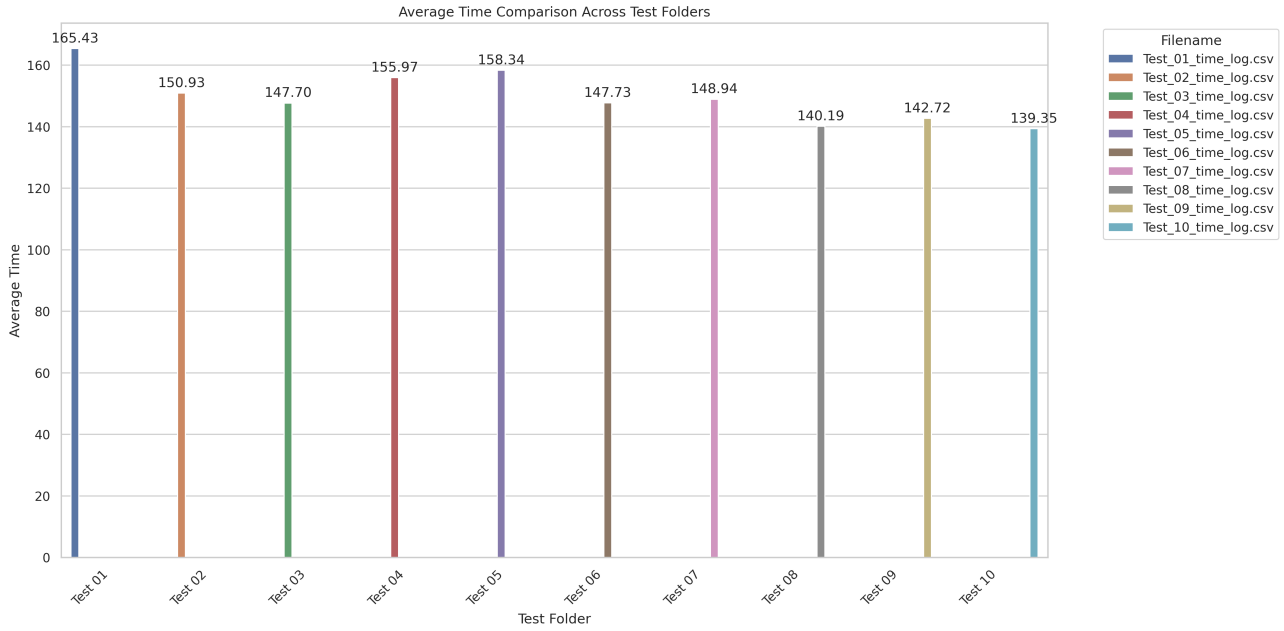

Figure B.1: Time to generate input by varying max waves.

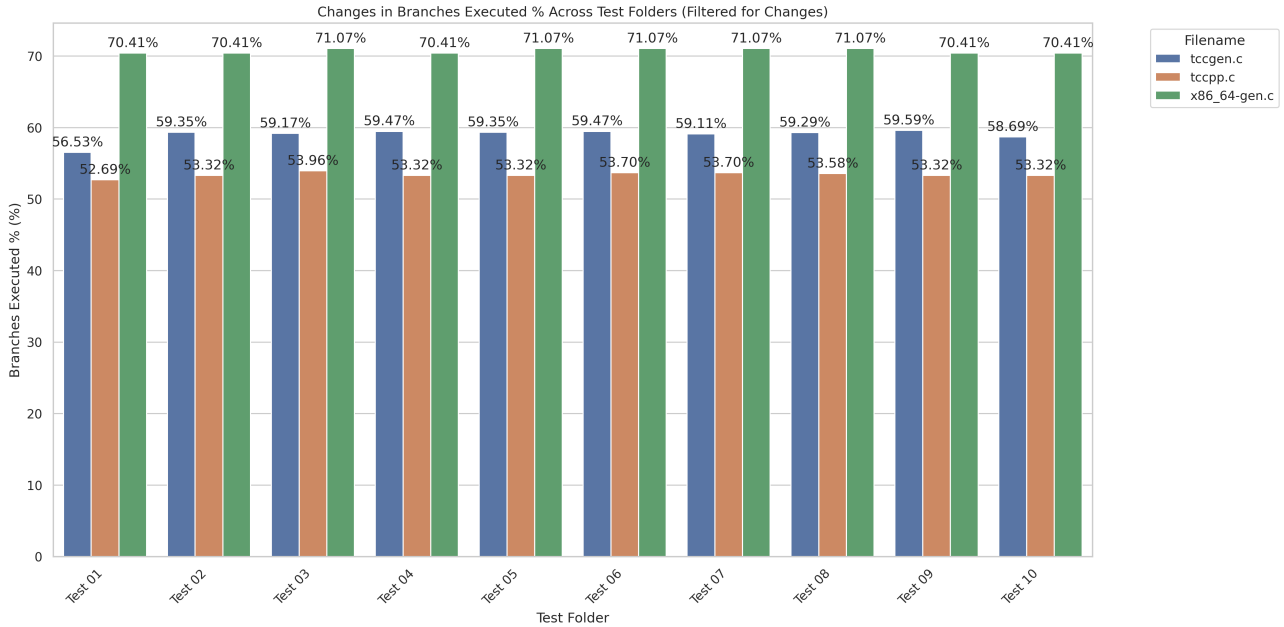

Figure B.2: Branches executed by changing the max waves heuristic.

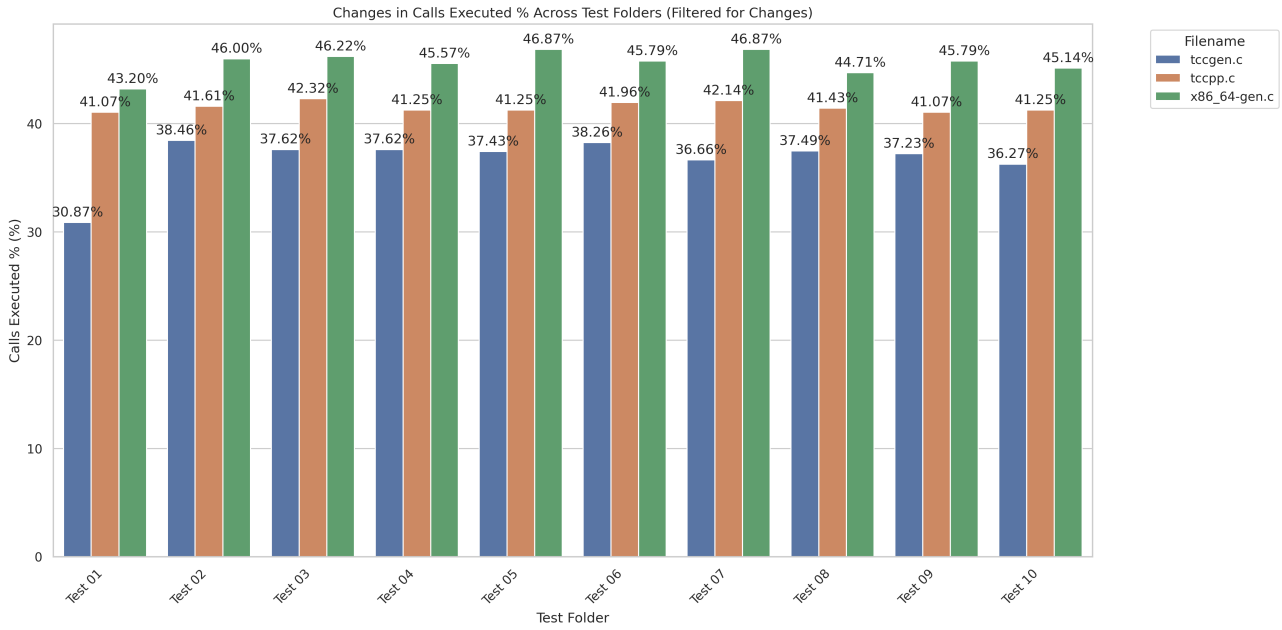

Figure B.3: Calls executed by changing the max waves heuristic.

B.2 Probability Heuristic

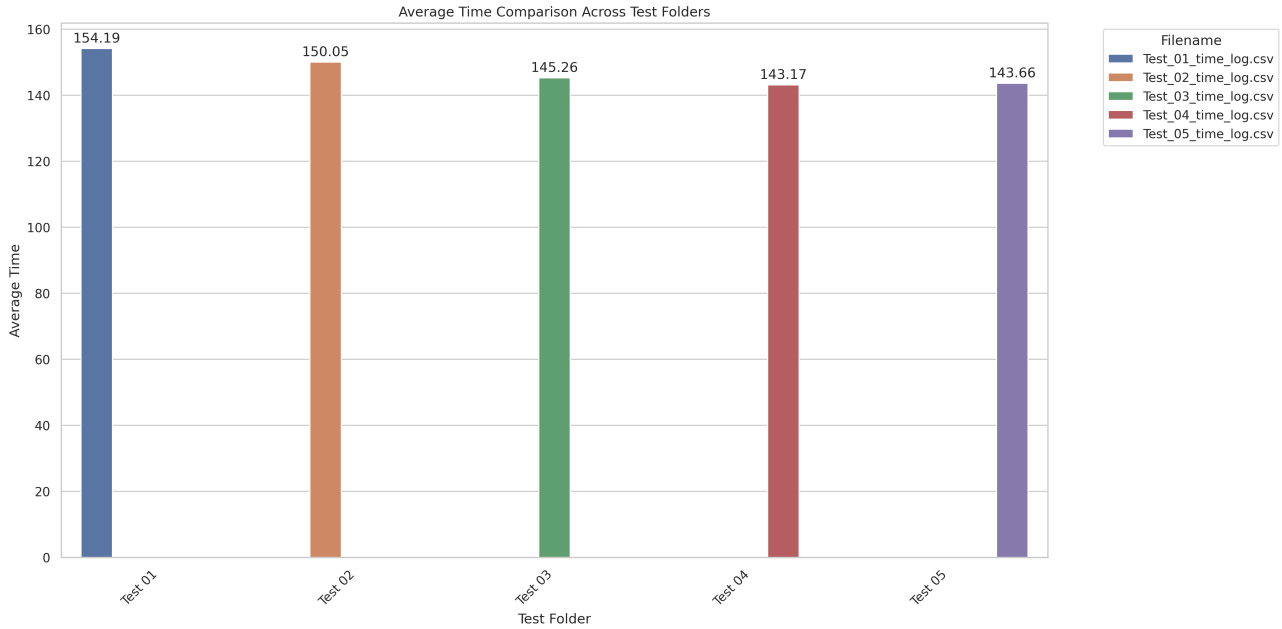

Figure B.4: Time to generate input by varying probabilities in the probabilistic fuzzing heuristic.

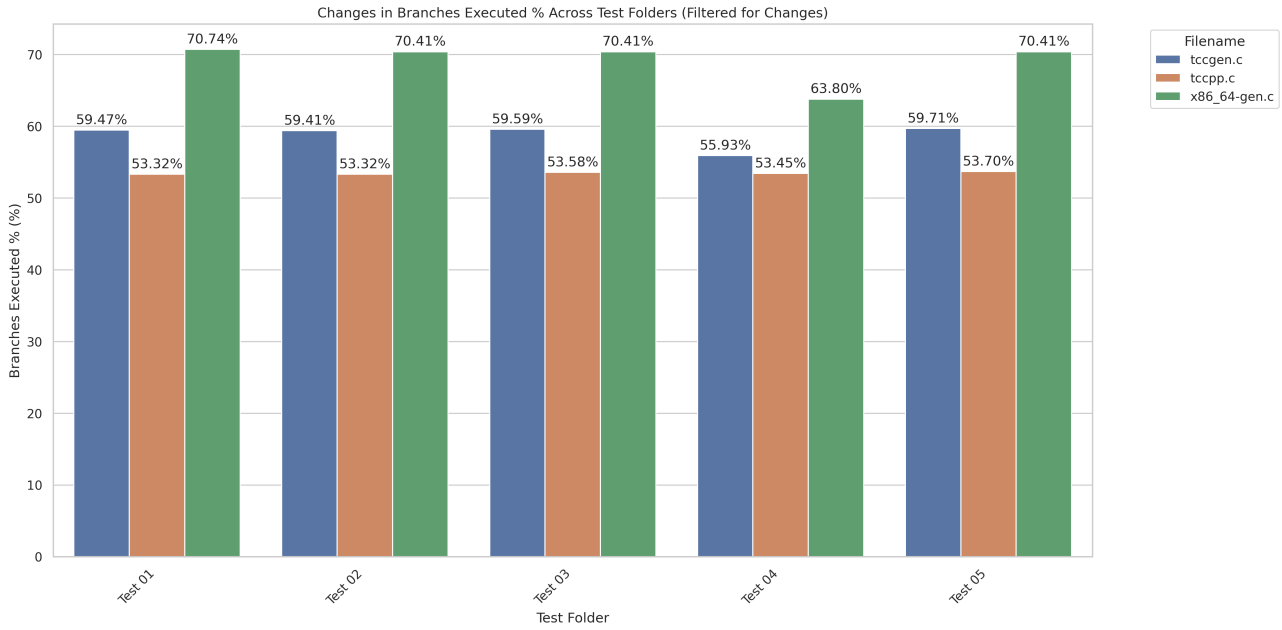

Figure B.5: Branches executed by varying probabilities in the probabilistic fuzzing heuristic.

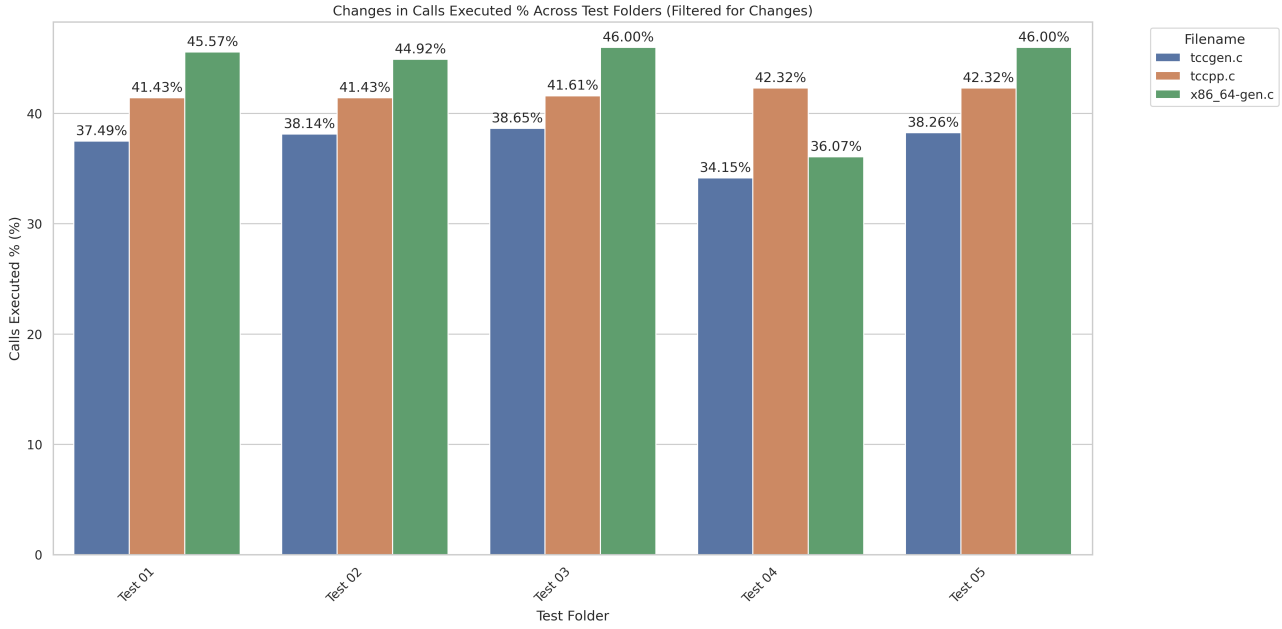

Figure B.6: Calls executed by varying probabilities in the probabilistic fuzzing heuristic.

B.3 Wave Peak Heuristic

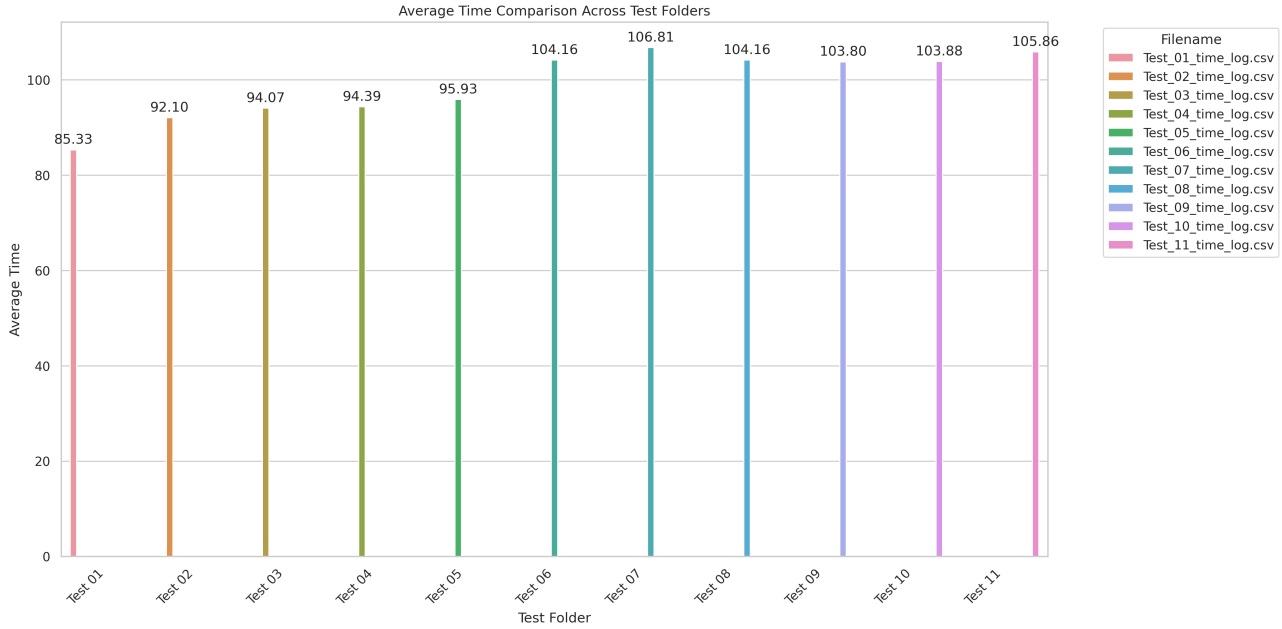

Figure B.7: Time to generate input by varying wave peaks.

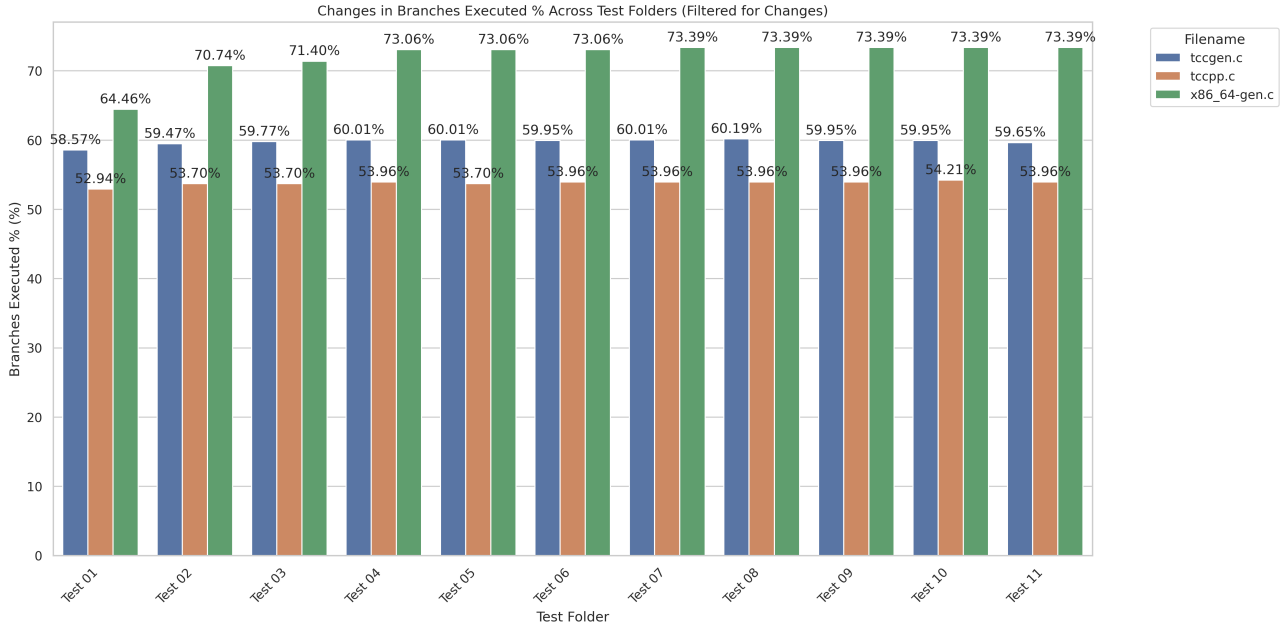

Figure B.8: Branches executed by changing the wave peak heuristic.

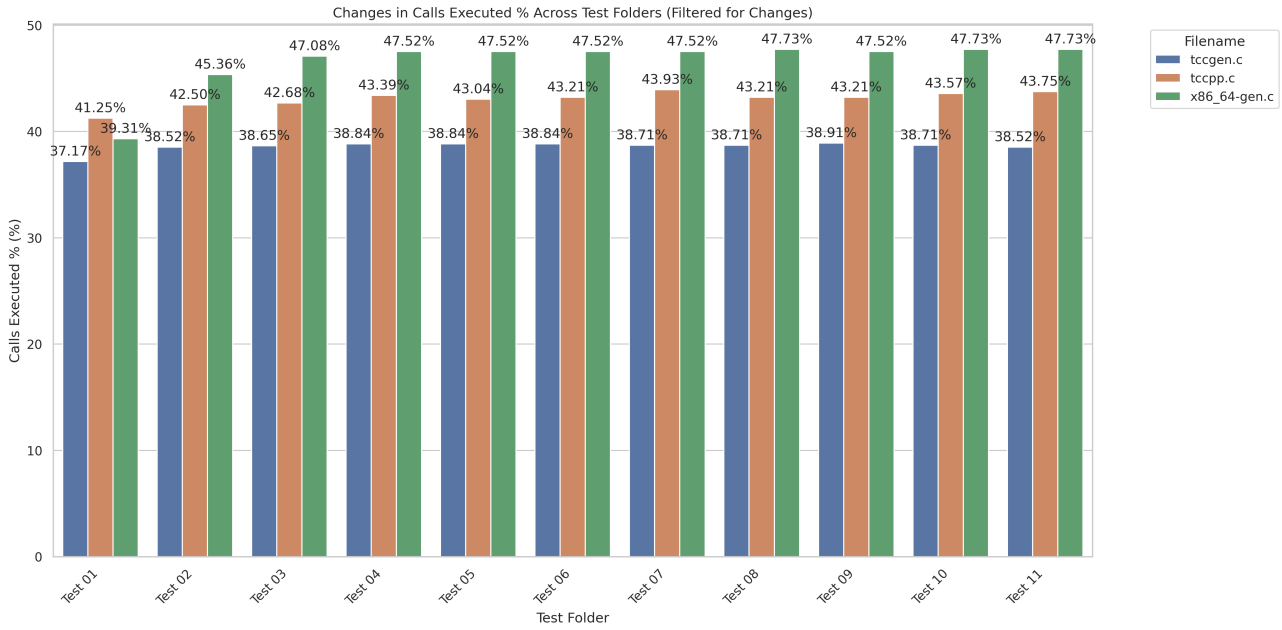

Figure B.9: Calls executed by changing the wave peak heuristic.

B.4 Wave Valleys Heuristic

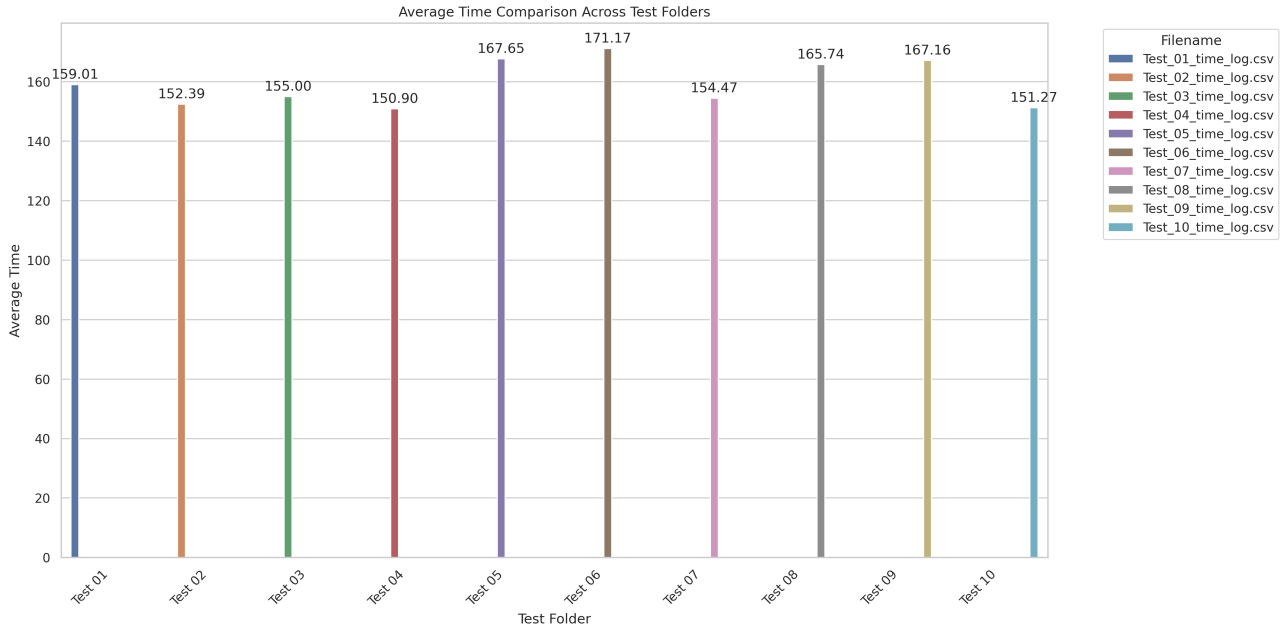

Figure B.10: Time to generate input by varying wave valleys.

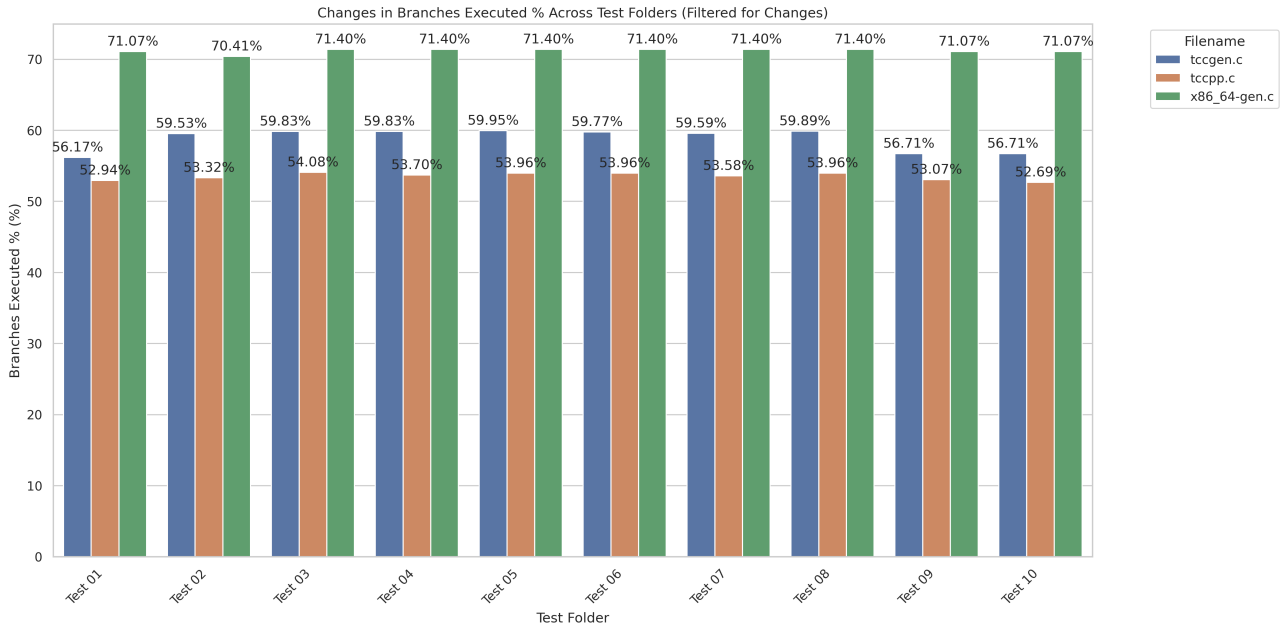

Figure B.11: Branches executed by changing the wave valleys heuristic.

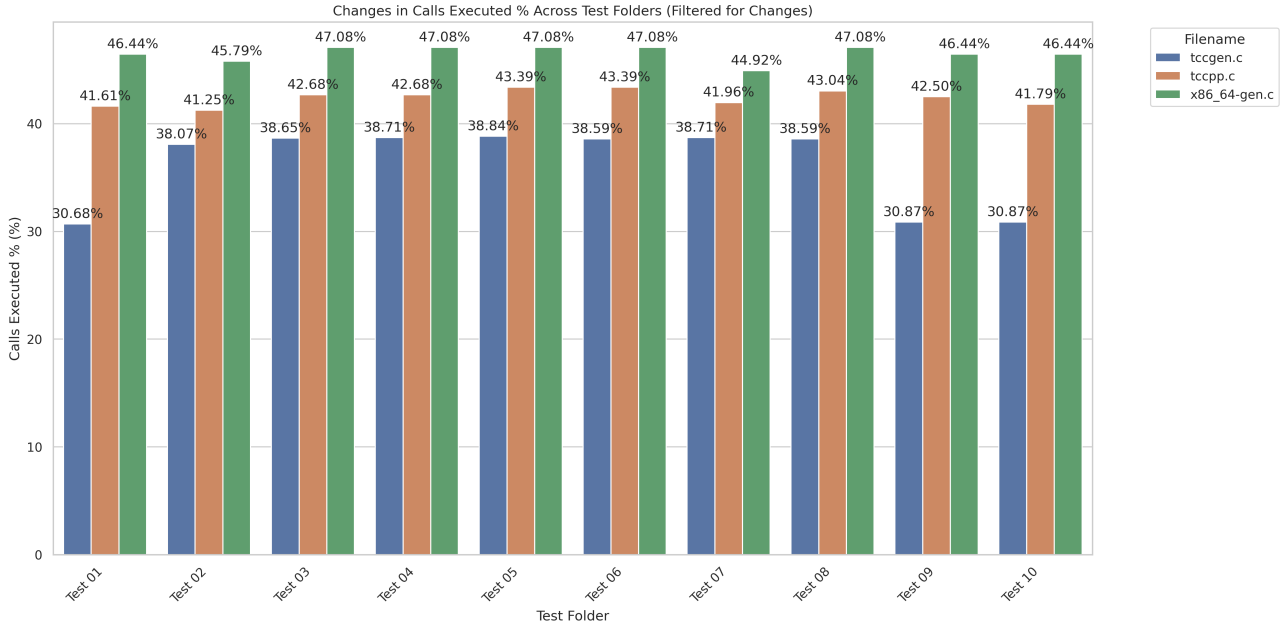

Figure B.12: Calls executed by changing the wave valleys heuristic.