Finding Small Code with Large Binaries - Final Report

VIKTOR GSTEIGER and NIKODEM KERNBACH*, ETH Zürich, Switzerland

We present a parameterized framework that seeks source code examples with a high binary-to-code-size ratio. Evaluating code sample quality using a custom heuristic function, our framework incorporates existing tools to enhance the discovery of code examples with high heuristic values. Through extensive experiments, we collect data for subsequent analysis and present condensed results in plots. Our analysis focuses on the non-equivalence between source code and binary size, identifying common reduction patterns, including compiler-specific and flag-specific ones. We also uncover patterns resulting in large binary sizes. Finally, we provide a critical analysis of the proposed solution.

1 INTRODUCTION

Software development workflows for compiled languages like C and C++ fundamentally depend on the correctness and efficiency of the underlying compiler. However, these attributes are challenging to ensure in practice, given the large-scale and intricate nature of compilers. The task of bug identification and the incorporation of novel optimizations is a collective effort of the programming community, prompting us to question whether such optimizations can be identified automatically.

In this project, we introduce a unique framework designed to aid in discovering potential optimizations and bugs by examining the non-equivalence relation between source code and binary size. It is expected that compilers optimize the compiled binary to produce smaller binaries. However, patterns leading to an unexpected increase in binary size can be valuable targets for optimization and potential bug indicators. Our framework endeavours to pinpoint such instances of source code characterized by a high binary-to-code-size ratio, thereby facilitating the analysis and manual identification of non-equivalence patterns, which can, in turn, be used to optimize or fix compilers.

Our approach's novelty resides in the integration of various pre-existing tools and a custom heuristic function to assess the violation of the binary-to-code-size ratio equivalence relation for a given piece of source code. Furthermore, the high degree of parameterization in our framework enables the investigation of a wide array of research questions.

We then manually extract and investigate common patterns that violate this equivalence relation, with a specific focus on prevalent reduction patterns and large binary size patterns. We pay particular attention to the variances between different compilers and compiler flags during this process. While the compilers we investigated were generally proficient at optimizing away most of the patterns we discovered, our framework proved to be quite effective in finding such patterns more broadly.

We also provide a comparative overview of related work, positioning our results within the broader academic literature. Finally, we critically evaluate the strengths and limitations of our framework, ensuring a thorough assessment of its capabilities.

Authors' address: Viktor Gsteiger, vgsteiger@student.ethz.ch; Nikodem Kernbach, nkernbach@student.ethz.ch, ETH Zürich, Switzerland.

2 APPROACH

In developing our approach, we faced a critical decision regarding the main focus of our investigation. We chose to tackle the issue of non-equivalence relations in compilers by optimizing for source size, while either maintaining or increasing the binary size. This decision necessitated fixing the binary size of the initial source code as a hard threshold that all following reductions must meet. By doing so, we were guiding the reduction program to optimize for large binary size while reducing source code size, which was our explicit aim.

We began our work on the artefact, implemented as a Python framework, by outlining several constraints that the framework had to follow:

(1) The framework must generate semantic-preserving mutations of the original, arbitrary, source code.
(2) The resulting mutant should be smaller in size than the source code.
(3) The binary size to source code ratio of the mutant should be maximized within a reasonable run time compared to the source code.

The workflow of our framework implemented as a Python script and available as an artefact uploaded to *ETH Zürich Moodle* or on *Github* [1], unfolds as follows:

(1) Depending on the input parameters, either a random code sample is initialized with *Csmith*[2] and verified using *diopter*[3], or a source code is read from the input file.
(2) *C-Reduce*[4] is employed to apply code equivalence transformations and concurrently reduce code size, generating multiple candidates for the subsequent round.
(3) The candidates are assessed according to a predefined heuristic.
(4) All potential candidates are fed into a global priority queue, with the heuristic value serving as the key, to identify the next viable candidate.
(5) The current highest-ranking candidate is selected from the priority queue and step 2 is repeated until the number of iterations or the timeout is reached. If the pool of candidates is exhausted before either condition is met, we either halt (if we are solely interested in the result derived from the initial code) or a new source code sample is generated.

Following we will go into more detail on the tools used and our approach of integrating them into our framework. Specifically, we will highlight the main difficulties and novel approaches in using the beforementioned tools.

Csmith. Our strategy employs *Csmith*, a random C program generation tool proposed by Yang et al. [14], to produce random source code samples. *Csmith* generates C source code adhering to both

[1] https://github.com/vGsteiger/ast-project
[2] https://github.com/csmith-project/csmith
[3] https://github.com/DeadCodeProductions/diopter
[4] https://embed.cs.utah.edu/creduce/

static and dynamic aspects of the C99 standard. We utilize a selection of the command-line parameters provided by *Csmith*, including:

(1) The `max-expr-complexity` parameter limits the complexity of expressions to a defined value (default is 10). This parameter modulates the complexity of *Csmith*-generated programs, enabling exploration of binary size in relation to expression complexity.
(2) The `max-block-depth` parameter caps the depth of nested blocks to a defined value (default is 5), thereby regulating the complexity of the generated source code.
(3) The `stop-by-stmt` parameter halts the generation of statements beyond a specified id (default is 100). This is used to control the complexity of the source code regarding statement complexity.
(4) The `seed` parameter ensures the reproducibility of the random source code generation process. Using a seed allows the regeneration of the same sequence of random numbers, hence the identical source code for each run. The default value for this parameter is 0.

Diopter. The sanitiser module from the differential optimization tester *Diopter*[5] is utilized for quality assurance. While *Csmith*-generated programs conform to the C99 standard, they might still elicit compilation warnings or undefined behaviour. To address this, our framework employs *Diopter* for code generation and after each iteration of *C-Reduce*. If issues are detected, a new sample is generated or the existing candidate is bypassed. *Diopter* was added as a submodule to our git repository for easy version switching and updates.

C-Reduce. Our methodology makes use of the test-case reduction tool *C-Reduce*, proposed by Regehr et al. [8]. *C-Reduce* accepts a C program from a source file and a shell script file that provides an *interestingness* test. This test must be satisfied by the original program and any subsequent reductions. Reduction commences by eliminating or simplifying declarations and statements, provided that the generated code continues to pass the test, resulting in a minimal source code.

In our approach, the *interestingness* test is adapted to yield candidates that satisfy our specific criteria (see Listing 1). Candidates of interest maintain or exceed the binary size of the initial code while likely reducing the source code size. Such candidates are duplicated into a subdirectory for the current iteration and inserted into the priority queue, marking them as potential contenders for the subsequent iteration.

```
# If the new binary does not run at all, it is not interesting
if [ $? -ne 0 ]; then
    exit 1
fi

# If the new soure code is smaller than 500 bytes, it is not
    interesting (remove reduction to single main method)
if [ $(wc -c < {local_new_source_code_path}) -lt 500 ]; then
    exit 1
fi

# If the new binary is bigger than the original, it is interesting
if [ $(size tmp.o | awk '{{print $1}}' | tail -n 1) -ge $(size orig
    .o | awk '{{print $1}}' | tail -n 1) ]; then
    random_string=$(mktemp XXXXXXXXXXXXXXXX)
    cp {local_new_source_code_path} {iteration_dir}/interesting_${{
        random_string}}.c
    exit 0
fi
exit 1
```

Listing 1. Interestingness Test: Removed is the compilation of the original code to `orig.o` and the newly generated to `tmp.o` as well as a test if the newly generated code compiles and is larger than 500 to exclude trivial reductions.

We selected *C-Reduce* primarily due to its compatibility with various operating systems and effective algorithmic implementation. However, it is important to note that our current methodology does not incorporate the consideration of all the parameters *C-Reduce* has to offer.

Two parameters that we do consider to be of notable importance are its distinct timeout behaviours, modifiable by command-line arguments. In our framework, we discern between two types of timeouts: (1) timeout on a specific *C-Reduce* activity, which imposes a time limit on each *C-Reduce* reduction attempt, and (2) *C-Reduce* timeout per iteration, where we limit the time a single reduction can take within an iteration.

Empirical observations indicate this differentiation enhances results, primarily because it allows for more precise tuning of *C-Reduce's* behaviour. It was noted that *C-Reduce* frequently spends considerable time on a reduction; thus, allowing more time per iteration can potentially yield better outcomes.

3 IMPLEMENTATION AND RESULTS

In this subsection, we will delve into our framework implementation in more granular detail, highlighting both its strengths and limitations.

3.1 Framework Implementation

As previously outlined, our framework exploits the capabilities of three open-source tools: *Csmith*, *Diopter*, and *C-Reduce*, to discover small source code samples that yield large binaries. We have also outlined the approach with regard to the steps our implementation takes to achieve the goal of finding small code with large binaries. In the following, we will go over the implementation in more detail and highlight the weaknesses and strengths of our approach.

Framework Parameters. Our framework offers extensive configurability, allowing users to tweak parameters influencing both *Csmith*'s random source code generation and *C-Reduce*'s reduction speed and behavior. Additionally, users can specify a compiler (e.g., `clang`), an optimization level (e.g., `-O3`), and the paths to the executables of the aforementioned tools, making our framework a highly parametrized tool. Lastly, we also let the user define an input file or random generation as well as an output directory location.

Source Code Generation and Sanitation. To explore diverse patterns of compact code producing large binaries, our framework commences with random code generation. This is facilitated by the `gen_and_save_src_code` function, which leverages *Csmith* as discussed previously. Generated source code samples are subsequently validated with *Diopter* for compatibility and sanity. This generation-validation loop continues until a sample passes the sanity check.

[5] https://github.com/DeadCodeProductions/diopter

Reduction Iterations. Our methodology utilizes an iterative heuristic optimization approach to realize the primary objective of the project. Initially, candidate source code is generated using the `gen_and_save_src_code` function and assigned a heuristic value via the `calculate_heuristic_value` function and tested against *Diopter*. If a candidate does not pass the *Diopter* test, it is dismissed. The candidates that pass the beforementioned selection process are then added to a priority queue, `candidates_pq`, for subsequent reduction.

During each iteration of the reduction process, the candidate with the highest heuristic value is extracted from the queue. The `generate_reduced_source_code_candidate` function then generates a set of reduced source code alternatives. Candidates that successfully pass the sanitizer and compilation steps are reintroduced into the priority queue according to their heuristic value.

The best candidate for the current iteration is evaluated against the global best, updating the latter if superior. This process continues until the maximum number of iterations is reached, or the allocated time expires. The framework concludes by logging the globally optimal source code, its heuristic value, and the initial source from which it was derived. All of these are copied to a user-defined output directory for further investigation.

Heuristic Function. Our approach employs the `calculate_heuristic_value` function as the core heuristic evaluator. This function employs the `calculate_size_difference` function to compute the differences in source code and binary sizes between the original and the reduced candidate. The binary file size is calculated as the size of the `.text`-section in the binary file.

It is crucial to note that if the reduced source code size is larger than the original, or if the reduced binary size is smaller, the heuristic value defaults to zero, thereby discarding the candidate. Additionally, source codes of size 500 bytes (corresponding to a trivial single function declaration) are also disregarded. This 500-byte threshold was empirically determined to prevent trivial code scenarios, which offer little to no insights for analysis.

If a candidate is not discarded, its details are logged, and the heuristic value is computed as the ratio of the reduced binary size to the reduced source code size. This ratio is reflective of our core objective: small source code generating large binaries.

3.2 Results

In this section, we'll present our research findings using the previously detailed framework. The research delves into the various aspects and parameters of our implementation, and the selected tools. We have scrutinized our results in the context of existing secondary literature to enrich our analysis. The experimental data was gathered on the hardware system discussed in the succeeding paragraph. Additionally, we manually examined multiple generated source code samples to identify common reduction patterns that yield compact code and sizable binaries, as well as patterns resulting in extensive binary sizes.

System. For all experiments a machine with the following specifications was used:

- **CPU:** AMD Ryzen™ 5 2500U @ 2.00GHz
- **OS:** Ubuntu 22.04.2 LTS
- **Compiler:** gcc (Ubuntu 11.3.0)
- **Csmith version:** 2.3.0
- **Diopter version:** 0.0.21
- **C-Reduce version:** 2.11.0

If not stated differently, we used the following parameters to run the framework:

- **Compiler flag:** O3
- **Timeout:** 500s
- **Interestingness test timeout:** 10s
- **Iteration timeout:** 50s
- **Csmith max. no. of statements:** 100
- **Csmith max. expr. complexity:** 10
- **Csmith max block depth:** 5

The plots in this section illustrate the mean values as points, while the whiskers represent the standard deviation.

C-Reduce Timeouts versus Size Difference. *C-Reduce* strives to minimize code samples, iterating over possible mutations until there are no more improvements in code size. Each generated sample has to undergo an *interestingness* test, which could be time-consuming. Consequently, as discussed in Section 2, we implement two timeouts to constrain the duration of this process: one limiting the maximum time spent on calculating the *interestingness* test value, and another curtailing the overall time of a single *C-Reduce* run (which we refer to as an iteration).

To investigate the correlation between the quality of produced samples and these timeouts, we executed the framework ten times for five different timeout combinations. For each run, the best sample was selected based on its heuristic value. The results are illustrated in Figure 1. The iteration timeout was consistently set to be exactly five times the depicted *interestingness* test timeout.

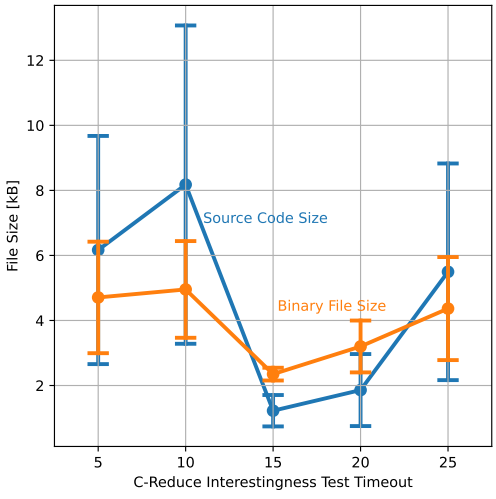

Fig. 1. Average file sizes under different C-Reduce timeouts.

The plot reveals no significant correlation between *C-Reduce* timeouts and the binary-to-code-size ratio. However, a general trend

is noticeable in that the range of samples tends to broaden with lower timeouts. This can be attributed to the fact that the premature termination of *C-Reduce* might not result in considerable reductions in either source code or binary size. As timeouts increase, both source code and binary file size demonstrate a downward trend.

Complexity of Source Program versus Size Difference. As discussed in Section 2, *Csmith* affords us the ability to manipulate the complexity of the resulting code samples. To explore any possible correlation between program complexity and the maximum achievable binary-to-code-size ratio of the reduced sample, we conducted experiments, collecting ten data points across three levels of complexity, as delineated in Table 1.

Level	expr-complexity	block-depth	stop-by-stmt
Low	5	2	50
Medium	10	5	100
High	15	8	150

Table 1. Parameters chosen for each complexity level.

The resulting average file sizes and standard deviation can be seen in Figure 2.

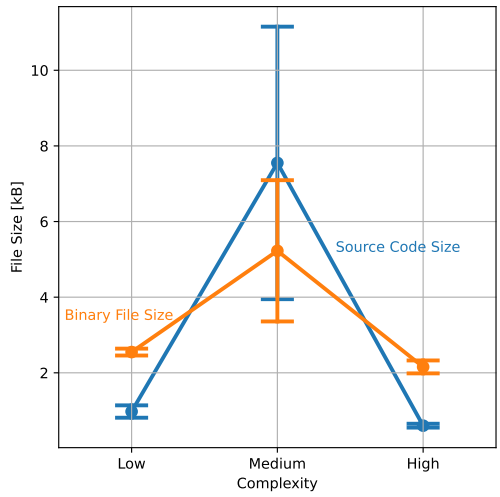

Fig. 2. Average file sizes under different initial code complexities.

Similar to the previous experiment, we couldn't identify a correlation between the complexity of the initial code and the resultant maximum binary-to-code-size ratio. This finding is unexpected as one would anticipate that programs of higher complexity present more challenges to optimization and reduction. One potential explanation could be that adjusting *Csmith*'s parameters do not necessarily result in more complex programs. Simply increasing the number of statements or depth of nested blocks doesn't necessarily make it more challenging for state-of-the-art compilers like *gcc* and *clang* to optimize the program.

Compiler Optimization Flags vs. Size Difference. The next experiment examined the influence of compiler optimization flags on the source code and binary file sizes. The results can be seen in figure 3.

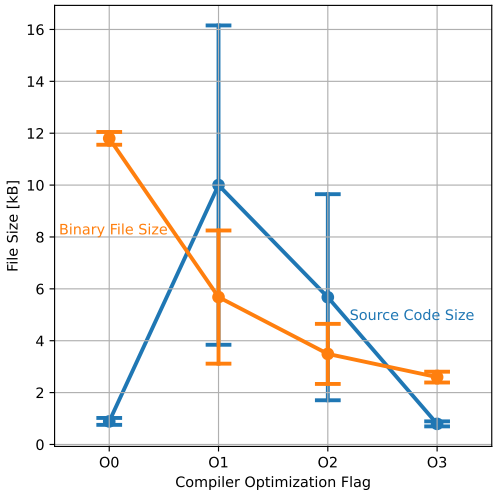

Fig. 3. Average file sizes under different compiler flags.

As evident from the plot, the compilers lived up to the expectation of enhanced code optimization with higher optimization flags, evidenced by the strictly monotonic decreasing curve of the binary file size. Intriguingly, aside from the `O0` data point, the source code size also diminishes. We attribute this to the possibility that code that can be readily reduced also tends to be easily optimized during compilation into a binary file.

Best Result According to Heuristic. In the course of executing the previous experiments, the highest heuristic value was observed during the testing of compiler optimization flags. The source code initially sized 2,394 bytes, while the binary was a size of 11,421 bytes. After execution, the outcome was a reduced source code size of 501 bytes with an unchanged binary size, corresponding to a heuristic value of 22.796. The distribution across iterations is illustrated in Figure 4.

Interestingly, the best heuristic value was achieved right after the first iteration. This pattern recurred frequently in our experiments. Once a local minimum is attained and further exploration of paths is hindered due to the limit on *C-Reduce* execution time, subsequent iterations become redundant. To manage this recurring situation, we implemented a local minima detection mechanism, which generates a new initial code using *Csmith*, thereby preventing the wastage of resources. This option can be activated in our framework by using the `regenerate` flag.

Common Reduction Patterns. In the course of our manual inspection of various reductions produced by our framework, we noticed certain common patterns. We classified these patterns according to the compiler used and the optimization flag applied, while also noting the role played by the initial source code size in the types of observable reduction patterns.

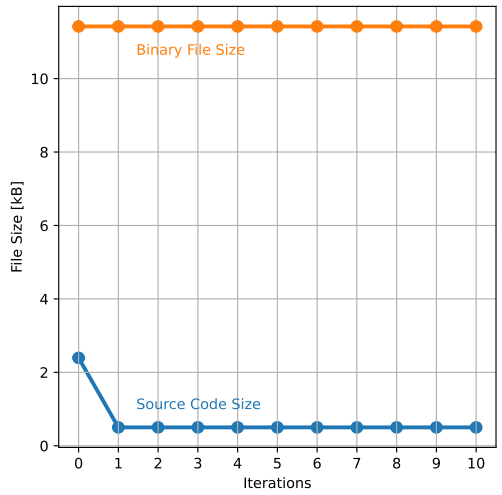

Fig. 4. This code sample, compiled using the `O0` flag had the highest encountered heuristic value of 22.796407.

Some reductions, like the removal of comments generated by *Csmith*, or blank lines, were frequent but not particularly relevant to our investigation. Likewise, reducing calls to the frequently used *Csmith* function `transparent_crc`, which computes the checksum of global variables, was a common pattern. This indicates that the compiler is likely already optimizing these calls for symbolic variables, replacing computations with symbolic equivalents. Similarly, the removal of syntactic sugar, trivial conditional branches, and other language elements that do not significantly contribute to the binary file was also commonly seen.

One very common reduction after all the calls to *Csmith* functions were removed is to also remove the include of the *Csmith* library. It is very clear that removing a library results in a reduced source code size, however, it also reduces the binary size as the external library is included in the binary. Thus removing the include may result in a smaller binary which is discouraged by the current framework implementation.

In the context of the *clang* compiler, the following patterns were commonly observed:

- In some extreme instances, the source code of 193,201 bytes, producing a binary of 16,384 bytes, was reduced to a source code of merely 501 bytes, while the binary size remained at 16,384 bytes. This is due to *clang*'s characteristic of producing binaries of at least 16,384 bytes, enabling large source code reductions without affecting the binary size.
- As depicted in Figure 3, there are differences in reduction patterns with varying compiler flags. Higher optimization flags, as anticipated, are adept at removing unnecessary code. For example, `O0` resulted in a reduction of source code size from 152,328 to 48,885, while preserving the binary size at 49,152. Higher optimization flags resulted in a more drastic source code reduction, from 152,328 to 503, while maintaining the binary size at 16,384.
- Using the `O0` optimization flag, we observed patterns like merging multiple statements into a single line, and removal of unused functions and declarations, among other compiler optimizations.

In the case of the `gcc` compiler, the following patterns were commonly observed:

- The disparity in source code reductions between various optimization flags wasn't as significant as in *clang*. For instance, `O3` reduced source code from 152,328 to 97,815 bytes, while preserving binary size at 16,384. Conversely, `O0` resulted in a smaller source code reduction, from 152,328 to 136,790 bytes, while maintaining binary size at 98,304.
- A noticeable pattern emerged where the reducer could remove a large function in `O3` while retaining the same function in `O0`. This is inferred from the minimal binary size of 16,384 in `O3`, where the reducer was then limited in further reductions.
- Another commonly observed reduction pattern under the `O3` flag involved the removal of all unused variable declarations, the removal of constant declarations with trivial uses, and other apparent optimizations most likely also performed by the compiler.

In the context of the *clang* compiler, the following patterns were commonly observed:

- In some extreme instances, the source code of 193,201 bytes, producing a binary of 16,384 bytes, was reduced to a source code of merely 501 bytes, while the binary size remained at 16,384 bytes. This is due to *clang*'s characteristic of producing binaries of at least 16,384 bytes, enabling large source code reductions without affecting the binary size.
- As depicted in Figure 3, there are differences in reduction patterns with varying compiler flags. Higher optimization flags, as anticipated, are adept at removing unnecessary code. For example, `O0` resulted in a reduction of source code size from 152,328 to 48,885, while preserving the binary size at 49,152. Higher optimization flags resulted in a more drastic source code reduction, from 152,328 to 503, while maintaining the binary size at 16,384.
- Using the `O0` optimization flag, we observed patterns like merging multiple statements into a single line, and removal of unused functions and declarations, among other compiler optimizations.

In the case of the `gcc` compiler, the following patterns were commonly observed:

- The disparity in source code reductions between various optimization flags wasn't as significant as in *clang*. For instance, `O3` reduced source code from 152,328 to 97,815 bytes, while preserving binary size at 16,384. Conversely, `O0` resulted in a smaller source code reduction, from 152,328 to 136,790 bytes, while maintaining binary size at 98,304.
- A noticeable pattern emerged where the reducer could remove a large function in `O3` while retaining the same function in `O0`. This is inferred from the minimal binary size of

16,384 in O3, where the reducer was then limited in further reductions.
- Another commonly observed reduction pattern under the O3 flag involved the removal of all unused variable declarations, the removal of constant declarations with trivial uses, and other apparent optimizations most likely also performed by the compiler.

These reduction patterns offer valuable insights for our research. They identify which parts of the code do not significantly impact the binary size, indicating that our framework effectively reduces source code to its minimum while maintaining a certain binary size. This understanding aids in comprehending compiler optimizations and variations in compiler behaviour, such as the impact of different compiler flags and compilers on the minimal binary size.

The patterns also underscore the non-equivalence between source code and binary size, revealing substantial code patterns that do not directly contribute to the binary size. This shows which parts of the code can be safely removed with minimal or no impact on the size.

Upon relaxing the bound for binary size reductions, we discovered the framework could eliminate even more code patterns, such as:

- **Declarations of Local Variables:** Local variables used sparingly have little impact on the binary size and are therefore removed when the bound is relaxed.
- **Excessive Use of Intermediate Variables:** If a calculation involves several steps with intermediate variables storing each step's results, these can often be consolidated into one expression. This reduces the source code size without significantly affecting the binary size.
- **Control Flow Statements:** Control flow statements can be eliminated without significantly impacting the binary size.

Large Binary Size Patterns. A number of C language constructs result in relatively large binary files. The most noticeable examples include:

- **Static Global Variables:** Static global variables used in functions that cannot be optimized away are essential to maintain the fixed binary size. Code featuring many such variables results in larger binaries that cannot be reduced.
- **Function Calls to External Libraries:** These code patterns generate large binary files. Many of these calls are not reduced when a hard binary size boundary is in place. This is because the logic in these external functions cannot be reduced. When potentially in-lined by the compiler, they make a disproportional contribution to the binary while adding minimally to the source code.
- **Loops and Multidimensional Arrays:** For or while loops, especially those involving large multidimensional array instantiation, are difficult to optimize and thus are rarely reduced. In some cases, these constructs and operations on large multidimensional arrays were the only parts of the code that remained unreduced.
- **Function Declarations without a Body:** This pattern has appeared on multiple occasions. Many binary instructions are required to make a function executable, while an empty body results in a small source code size.

Thus, if the goal is to generate synthetic source code that results in the largest possible binary, we would integrate many global static variables, numerous empty function declarations, several function calls to external libraries, and for loops with computations on multidimensional arrays. This understanding offers new avenues for further research, especially considering the established literature on optimization of multi-dimensional array operations in software [5; 10].

4 RELATED WORK

Recent surveys into the characteristics of compiler bugs by Sun et al. [13] found that programs that trigger bugs are usually small but complex. This supports our endeavour into small source code that generates large binaries by reducing the source code while optimizing for a large source code to binary ratio.

Our research into the non-equivalence relation and code patterns was influenced by differential testing between different source code reductions. Randomized differential testing of compilers has had great success in finding compiler crashes and silent compilations that result in nonoptimal binaries. Our approach of enabling source code reductions that focus on errors in compilations of source code concerning small source code and large binaries extends the existing research. Research into missed compiler optimizations by Barany [2] shows a more holistic approach where they use a custom binary analysis tool to compare the output programs. A similar visual analytics system Compilation Navigator was proposed by Devkota et al. [3] which could be additionally used to analyse the large binary patterns described earlier. And Ren et al. [9] proposed to systematically study the effectiveness of compiler optimization on binary code differences which ties into our manual analysis of code patterns that result in large binary code. A tool such as *Cornucopia* as proposed by Singhal et al. [11] could prove valuable to generate a large number of binary files for such a systematic extended analysis of source code that generates large binaries.

Additionally, our approach to increasing the complexity of the source codes to be reduced is supported and advanced by Nagai et al. [7] which presents an enhanced method of testing the validity of arithmetic optimization of C compilers using randomly generated programs by scaling up the size and numbers of expressions.

One concern we have not touched upon was that *Csmith* may produce a lot of dead code that is optimized away by the compiler and that can be removed without an influence on the binary size. Research by Barany [1] introduces a liveness-driven random program generation. Additionally, Le et al.[4] proposed *Athena*, which introduces a guided mutation strategy of compiler input programs that maximise the distance between the mutant and the original file. This can also be extended by tools proposed by Sun et al. [4; 12] that delete code or mutate code that gets executed to ensure the liveness of code is reduced.

An additional topic which has created a lot of buzz recently was the usage of machine learning for all sorts of tasks. Compiler testing and optimization have not been spared by this craze as shown by research into binary comprehension by Zhang [15] and similar works.

Additionally, tools for source-to-binary similarity, such as BinPro proposed by Miyani et al. [6], is an additional approach to analysing source code that resulted in large binaries.

5 CONCLUSION

As discussed in the previous sections, we implemented a framework using external tools to reduce source code with a fixed binary size goal, aiming to produce source code resulting in large binaries. The motivation behind this study was to delve deeper into the non-equivalence relation between source code size and binary size, serving as an additional tool for compiler debugging. Our research has unveiled patterns of compiler optimizations that are readily reduced while simultaneously hinting at optimization problems warranting further research.

Although our attempts to identify a specific pattern demonstrating a clear non-equivalence were unsuccessful, we managed to develop a tool that can pave the way for additional research. Moreover, the results from our study pointed towards potential optimization issues in modern compilers, which may significantly impact today's most utilized applications.

5.1 Limitations

As we hinted in the results section, our current approach excels at eliminating code patterns that are optimized by compilers in any case. Therefore, identifying the code patterns not optimized well by compilers in the current setup becomes tedious. One must understand both the initial and the reduced source code, as well as the compiler used well enough to discern which code patterns are reduced due to their negligible effect on binary size owing to optimizations. Conversely, they may contribute to a large binary and the non-equivalence relation between source code and binary size.

Moreover, our current approach does not test the generated code for liveness. Meaning that there could be significant parts of the code that are trivially reduced without impacting the binary size, thereby obscuring the patterns that generate large binaries.

Lastly, we focused on two of the most used C-compilers during our work, which inherently reduced the chances of identifying significant issues. While this does not imply that the compilers in question are devoid of flaws, it certainly makes it harder as these compilers are thoroughly tested and trusted.

5.2 Future Work

As hinted in the previous subsection, for future iterations, we would consider liveness as a factor in random code generation. This would ensure the elimination of large trivial reductions that currently cloud the research output of our framework.

Furthermore, as we have observed, many patterns that were reduced away did not contribute significantly to our research. In a future iteration of this work, we propose a two-stage reduction process. First, a rapid reduction towards the mostly obtained optimal, leaving only parts that had no impact on the binary size due to compiler optimizations. Following this, a more meticulous approach with multiple steps where we relax the hard boundary to obtain a minimal set of language constructs contributing most to binary size. In our research, we manually executed this two-step solution. For future iterations, we would incorporate this functionality into the research framework. This functionality was not added to the framework in the current version due to time constraints.

Lastly, the code our framework produces is challenging to read and interpret. For further manual investigations, it would be prudent to include some sort of auto-formatting. Although syntactic sugar like new lines and spaces might add to the source code, it's only a concern during intermediate reduction steps. At the point of manual inspection, we aren't particularly concerned about the exact number but rather about the constructs that contribute to the source code resulting in relatively large binaries.

REFERENCES

[1] Gergö Barany. Liveness-driven random program generation. *CoRR*, abs/1709.04421, 2017.

[2] Gergö Barany. Finding missed compiler optimizations by differential testing. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, page 82–92, New York, NY, USA, 2018. Association for Computing Machinery.

[3] Sabin Devkota, Pascal Aschwanden, Adam Kunen, Matthew P. LeGendre, and Katherine E. Isaacs. Ccnav: Understanding compiler optimizations in binary code. *CoRR*, abs/2009.00956, 2020.

[4] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. *ACM SIGPLAN Notices*, 50(10):386–399, 2015.

[5] Chun-Yuan Lin, Jen-Shiuh Liu, and Yeh-Ching Chung. Efficient representation scheme for multidimensional array operations. *IEEE Transactions on Computers*, 51(3):327–345, 2002.

[6] Dhaval Miyani, Zhen Huang, and David Lie. Binpro: A tool for binary source code provenance, 2017.

[7] Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. Reinforcing random testing of arithmetic optimization of c compilers by scaling up size and number of expressions. *IPSJ Transactions on System and LSI Design Methodology*, 7:91–100, 2014.

[8] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for c compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 335–346, 2012.

[9] Xiaolei Ren, Michael Ho, Jiang Ming, Yu Lei, and Li Li. Unleashing the hidden power of compiler optimization on binary code difference: An empirical study, 2021.

[10] Oliver Serang and Florian Heyl. TRIOT: Faster tensor manipulation in c++11. *The Art, Science, and Engineering of Programming*, 1(2), apr 2017.

[11] Vidush Singhal, Akul Abhilash Pillai, Charitha Saumya, Milind Kulkarni, and Aravind Machiry. Cornucopia : A framework for feedback guided generation of binaries. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ACM, oct 2022.

[12] Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications*, pages 849–863, 2016.

[13] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. Toward understanding compiler bugs in gcc and llvm. In *Proceedings of the 25th international symposium on software testing and analysis*, pages 294–305, 2016.

[14] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294, 2011.

[15] Yifan Zhang. Leveraging artificial intelligence on binary code comprehension. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ACM, oct 2022.