



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich



Complete Compiler Code Creation

Master Thesis

Loris Reiff

2. May 2021

Supervisors: N. Badoux, Prof. M. Payer, Prof. Dr. S. Čapkun

Department of Computer Science, ETH Zürich

This thesis was developed with the guidance of Nicolas Badoux and the HexHive research group from EPFL, and under the remote supervision of Prof. Dr. S. Čapkun from ETHZ.



Acknowledgements

I would like to thank Nicolas Badoux for his support, feedback and supervision throughout the thesis. He was always available when I needed guidance. My appreciation also goes to Professor Mathias Payer and Antony Vennard of the HexHive team for their feedback and to Derrick McKee for promptly replying to my questions about *Tinbergen*. Hopefully, you succeed in swaying reviewer two too, eventually.

I would also like to thank my parents for their support, love, and motivational words. Furthermore, I am thankful to my girlfriend for making me smile and for reading my thesis to weed out some grammar mistakes. Reading a thesis of a completely different field is challenging! Last but not least, a shout-out to my best buddy Andrin for his valuable feedback on the thesis and for sending me funny memes and gifs that made me laugh!

Abstract

Modern software is complex. To deal with this complexity, software is translated between different abstraction levels. Various tools exist to ease this process, for instance compilers and binary rewriters. It is clear that the correctness of these tools is essential, as bugs might propagate to the translated software and harm its stability.

Testing these tools is challenging, because of their inherent complexity that stems from Turing-complete in- and output. Therefore, current work focuses on single components. In the case of a compiler this manifests itself by testing only specific optimizations. In this work, we create a program which generates a corpus that aims at testing whole tools. To do so, we leverage fuzzing and coverage indications to create compiler intermediate representation snippets which can be used as a testing input for evaluating the aforementioned type of tools.

Our evaluations show that the generated snippets achieve higher coverage than multiple software projects like *sqlite* and *7zip* combined, while requiring less space. Furthermore, we discovered 4 previously unreported bugs in LLVM and 2 in RetroWrite, a state of the art binary rewriter.

Contents

Contents	iv
1 Introduction	1
2 Background	4
2.1 Compilers	4
2.1.1 LLVM	5
2.1.2 LLVM Compiler Architecture	6
2.1.3 LLVM Intermediate Representation	6
2.2 Fuzzing	7
2.3 Test Suite Reduction	10
2.4 Test Case Reduction	11
2.5 Semantic Comparison	13
3 Design	15
3.1 Goals	15
3.2 Overview	16
3.3 Fuzzing towards an IR Snippet	17
3.3.1 Seed Selection	18
3.3.2 Coverage	18
3.3.3 Mutation	19
3.3.4 Scheduler	21
3.4 Minimization of IR Snippets	23
4 Implementation	25
4.1 Overview	25
4.2 Preprocessing	27
4.2.1 Internalization	27
4.2.2 Distance calculation	28
4.3 Fuzzing	28

4.4	Minimization	29
5	Evaluation	31
5.1	Generated IR Snippets	31
5.2	Case Study: Finding Bugs	35
5.2.1	LLVM	37
5.2.2	RetroWrite	39
5.2.3	Conclusion	39
6	Related Work	41
7	Future Work	44
8	Conclusion	46
A	Appendix	47
A.1	Open Source Contributions	47
A.1.1	AFLGo	47
A.1.2	LLVM	47
A.1.3	RetroWrite	48
	Bibliography	49

Chapter 1

Introduction

Software plays an important role in our society and bugs are often not merely an annoyance, but a costly problem. A study by the U.S Department of Commerce [86] in 2002 estimated the cost of software failures to be between \$22.2 to \$59.5 billion. Since 2002 the costs have only risen with the adoption of more and more software in our daily lives. CISQ estimates the cost caused by bugs, operational mistakes, and vulnerabilities to be around \$1.56 trillion for 2020 in the U.S.A alone [52]. However, CISQ expects the number to be at the low end given the meteoric rise in costly security incidences. In order to have more reliable software with fewer bugs, extensive testing is applied [51]. However, testing tools that translate software between different abstraction layers like compilers and static binary rewriters is challenging [37, 102]. At the same time their correctness is crucial, especially for compilers. Compilers translate source code to optimized machine code. Consequently, virtually every piece of software is processed by a compiler or compiler like tool. Thus, a bug in a compiler can propagate to other software. For instance, a bug in Java 7 caused several Apache projects to crash [10]. Not only do compiler bugs propagate with potentially severe consequences, but also do they hinder debugging. When developers encounter a bug in their software, they, normally rightfully, assume the bug to be in their code, thus they waste a lot of time debugging the wrong parts [36]. Also, the correctness of binary rewriters is essential, they allow the instrumentation of closed-source software and can enhance its security by adding sanity checks, or by enabling fast fuzzing [41, 89].

Compilers are extremely complex pieces of software and have a large surface which is one reason that makes testing tough. A typical compiler performs lexical analysis, translates source code to an intermediate representation (IR), applies optimizations, and generates machine code. What is more, a formal specification is often missing and only the implicit objective of translating source code to efficient machine code is stated. For example, it is

rarely specified when to apply which optimization, which makes verification difficult. Additionally, the input as well as the output is Turing-complete code, which sets it apart from other complex software and makes testing especially challenging. Similar complexities are present in static binary rewriters. Input, and output is Turing-complete code. Additionally, the input executable is to be rewritten as generally as possible. For instance, one should be able to add stack canaries when rewriting a binary, but it should also be possible to add sanitizer instrumentations.

A lot of effort has been put into compiler testing [37, 112, 83, 66, 99, 72]. However, most focus has been spent on the compiler as a whole, from front-end to back-end. This could mask bugs in the optimizer as a subset of optimizations might never trigger for the language used to test [18]. This is a problem, because the front-end is generally exchangeable and a bug in such a component might be triggered with another front-end. This issue is further fueled by design limitations that restrict the test’s capability to certain optimizations only. Additionally, little research has been performed on maximizing code coverage of the compiler due to inherent difficulties when doing it on the complete compiler pipeline [37]. Moreover, the focus was mainly on accelerated compiler testing, i.e, on creating a snippet, testing it and then discarding it. This is in contrast to building a corpus which can be reused for a number of other tools.

Our approach differs from previous research, we generate an IR corpus and target every optimization at the mid-end of compilers. Each snippet targets a particular optimization and is generated with the goal of maximizing the code coverage of the optimization, when compiled. At the same time we try to keep the size of the corpus at a minimal level with heuristic approaches. This should yield a diverse, yet small corpus.

We believe our tool has its *raison d’être*, as it complements existing compiler, and binary analysis testing tools. We focus on the mid-end of compilers and the method easily extends to the back-end. Additionally, our design spends more time generating the snippets with the goal of having a more diverse test set that can be evaluated quickly. This makes it ideal as regression test for compilers. Existing code generation tools test only a limited set of optimizations. Thus, they fail to deliver this property. Moreover, the corpus enables testing a binary rewriter’s or binary analysis tool’s capability with regard to different optimizations. If a binary analysis tool fails when evaluating a snippet, it can be easily linked to a specific optimization. This simplifies debugging of the binary analysis tool. This is in contrast to evaluating binary tools on big projects, where it is often more difficult to find which component caused the tool to misbehave. Moreover, in the case of a binary rewriter, it is difficult to execute all relevant and interesting parts of a big project, which makes it likely that certain edge-cases go untested when

relying on a big project.

In this thesis we introduce a tool to generate an IR snippet corpus using directed mutational grey-box fuzzing, and test case as well as test suite reduction techniques. We aim to maximize the code coverage of the mid-end compiler optimizations, while minimizing the size of the corpus.

We show that compiling the generated snippets by our tool achieve a higher coverage than CTMark, which consists of multiple tools including sqlite and 7zip. At the same time the snippets are five times faster compiled than CTMark. Furthermore, we uncovered 4 previously unreported bugs in LLVM and 2 unreported bugs in RetroWrite a state-of-the-art static binary rewriter.

Our core contributions consists of designing and developing the first tool capable of generating a wide reaching IR corpus that is at the same time minimal in its size. Furthermore, we evaluate the corpus' capabilities on a compiler as well as on a binary rewriter and present promising results.

Background

In this chapter, we introduce techniques and technologies that help in understanding the concepts of our snippet generator. Firstly, we cover the design of compilers, since the core of our idea is to maximize the executed code of the compiler optimizations. Secondly, we discuss fuzzing. Fuzzing is a randomized method to execute deep code paths of a program. This section also touches on how the penetration or rather coverage achieved is measured. Next, we look at test suite and test case reduction. These sections help in understanding on how we tackle the corpus size question. Lastly, we discuss the topic of semantic comparison. Semantic comparison is not relevant for the corpus generation itself, but for some of its use-cases, in particular, we use it in the evaluation.

2.1 Compilers

Compilers are omnipresent in the life of software developers. They translate the source code to an intermediate representation (IR), which is then translated to machine code. These phases help to abstract and facilitate architecture independent optimizations. The compilation can happen ahead of time, which is the case for languages like C and Rust, or during run-time for languages like JavaScript. In the later case one speaks of just-in-time compilation. The compilation process is complex, but all its phases can be assigned into one of three parts, the front-end, middle-end, or back-end. The front-end is responsible for tasks like reading in the source file, verifying the syntactic correctness, and the translation of the code to the IR. The middle-end's job is to perform architecture independent optimizations and code instrumentations. The back-end's tasks include the generation of the machine code and architecture dependent optimizations. In Subsection 2.1.2 we cover how LLVM, a widely used compiler, relates to this model. The complexity of each stage is a surface for bugs and bugs in compilers are

extremely devastating, because developers work with the assumption that compiler produce correct code. To improve the reliability and correctness, there are essentially two approaches, with the first being formal verification of the compiler and the second being extensive testing. Formal verification is extremely time-consuming. Therefore, it did not gain a lot of traction in the compiler community. Nonetheless, formally verified compilers do exist. CompCert [67] is a prominent example. Rigorous testing is widely adapted [16, 9] and of uttermost importance for quality control and error detection [28, 64]. However, testing a compiler well is difficult and has its challenges [37, 102]. The test cases must be extensive in order to cover the complexity of the compiler. Furthermore, compiler inputs are arbitrarily complex programs and there is no oracle that determines if the generated machine code is semantically equivalent to the source input. Therefore, one usually compares the behavior of programs compiled by different compilers. However, this is not a proof for the absence of bugs!

2.1.1 LLVM

The LLVM compiler infrastructure project [12] originated in the University of Illinois. The goal was to provide a research tool, which allowed exploring new frontiers in dynamic compiler techniques for both static and dynamic programming languages. Since then, the LLVM project has greatly evolved and turned into an umbrella project for various modular compiler and toolchain technologies. As a result of this development the term LLVM is used to address different parts of the original project. It can refer to the project as a whole, to the LLVM Core, to the LLVM intermediate representation (Subsection 2.1.3) or to an LLVM-based compiler. The LLVM Core is a library that provides a target-independent optimizer which is built around the LLVM intermediate representation. Additionally, the library comes with code generation support for many popular CPU architectures. An LLVM-based compiler uses the LLVM Core as its back-end for compilation. Herein, because we predominantly focus on the LLVM Core, we use the term LLVM representatively for the LLVM Core.

Due to its permissive license and the modularity of the LLVM projects, many of its tools are used in industry as well as in academia [15]. In particular, LLVM Core and Clang – which together act as a C/C++ compiler – have a wide adoption. For instance, Apple uses Clang and LLVM Core as its default compiler and it is an integral part of its development tools. Likewise, Android’s Native Development Kit (NDK) relies on these LLVM products [13]. Furthermore, big projects such as Chrome use LLVM exclusively to compile to all major platforms including Windows [14, 3].

2.1.2 LLVM Compiler Architecture

The LLVM compiler infrastructure was designed with modularity and extensibility in mind. These design decisions date back to the first versions of LLVM [39], where the focus was on back-end optimizations while GCC was used as front-end. The LLVM Core library, which corresponds to the mid-end and back-end of a compiler, can be easily extended with additional architectural independent optimizations or other target architectures. Moreover, the optimizations pipeline is very flexible. For instance, it can be bisected and optimizations can be easily reordered. Figure 2.1 depicts the architecture

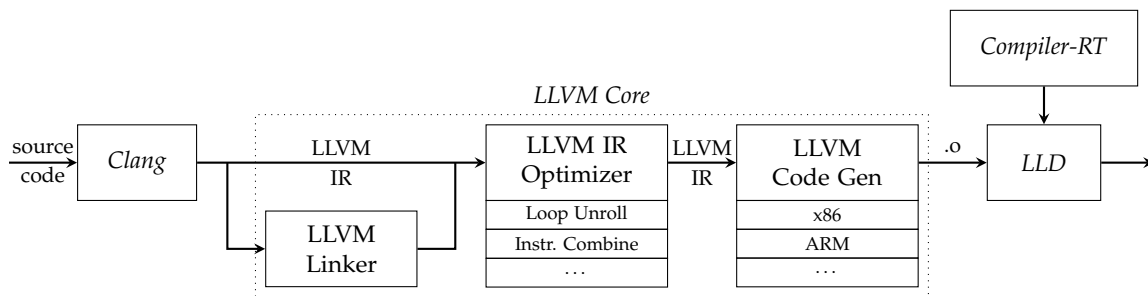


Figure 2.1: LLVM-based C/C++ toolchain. *Italic* components are LLVM projects.

of a typical LLVM compiler toolchain with a focus on LLVM Core. In this case the front-end is *Clang*, but it can be changed to any other front-end that translates source code from a particular language to the LLVM IR. If *link time optimization* (LTO) is enabled, the LLVM IR code files are linked, otherwise the optimization step follows directly and the IR code files are handled separately. The LLVM IR optimizer performs different architecture independent analysis and optimization passes on the LLVM IR code. The optimized IR code is then passed to the LLVM Code Generator, which translates the LLVM IR to the machine code of the targeted architecture. This step is also responsible for target specific optimizations, such as choosing efficient instructions and a good register mapping. The created object files are then linked by a linker of choice, for instance the LLD linker. Given certain compile options, the program is also linked against LLVM's *Compiler-RT* runtime. *Compiler-RT* provides certain built-ins, sanitizer run-times such as AddressSanitizer, and a profiling library.

2.1.3 LLVM Intermediate Representation

The LLVM intermediate representation (LLVM IR) is a type-safe, static single assignment (SSA) representation intending to be low-level and lightweight, yet extensible and expressive. The IR language is extensively documented to facilitate the development and debugging. Also, it has three different,

but semantically equivalent forms, with each having its use-case. There is the human readable assembly representation, which is useful for debugging and reasoning about transformations performed on the IR. Next, there is a bitcode representation, which is the binary equivalent of the human readable form. It is useful for external tools that load the IR code from disk. While the human readable format can also be used for this use-case, the bitcode allows faster parsing. Lastly, the third form, the in-memory compiler IR form, is used inside the compiler suite. Listing 1 shows an annotated example of a “Hello World” program in the LLVM IR.

```

1  ; ModuleID = 'hello.c'
2  source_filename = "hello.c"
3  target triple = "x86_64-unknown-linux-gnu"
4
5  ; Declare a global string constant
6  @.str = private unnamed_addr constant [14 x i8] c"hello, world!\00", align 1
7
8  ; Definition of the main function
9  define dso_local i32 @main(i32 %0, i8** %1) #0 {
10     ; allocate a 32bit integer on the stack
11     %3 = alloca i32, align 4
12     ; write 0 to the stack at location %3
13     store i32 0, i32* %3, align 4
14     ; Call the puts function with the string constant converted to
15     ; an i8* (char pointer)
16     %6 = call i32 @puts(i8* getelementptr inbounds ([14 x i8],
17                                                    [14 x i8]* @.str,
18                                                    i64 0, i64 0))
19     ret i32 0
20 }
21
22 ; Declaration of external puts function
23 declare dso_local i32 @puts(i8*)
24
25 ; attributes of the main function
26 attributes #0 = { noinline nounwind uwtable }

```

Listing 1: Example LLVM IR code in human readable form.

2.2 Fuzzing

Fuzzing, or fuzz testing is roughly speaking the process of repeatedly running a program with random inputs, with the goal of finding an input which results in unintended behavior of the program (e.g., memory corruptions, memory leak, failing code assertions or missing input validation). The idea was first introduced in the early 1990s [78] and has since then gained a wide adoption in vulnerability research, because of its effectiveness [7, 19].

Moreover, big software companies such as Microsoft [29] and Google [4] have adopted fuzzing to harden their software. The repeated running of a program under test (PUT) with random inputs is especially good in the discovery of memory corruptions, which are often exploitable [101] and remain one of the most common bug classes [80]. The reason for its effectiveness is that many memory corruptions terminate the program fatally, which is easy to pick up [75]. Furthermore, fairly lightweight instrumentation exists to detect additional memory corruption bugs [2]. Other bugs such as race conditions require more expensive and more elaborate bug oracles [20].

The effectiveness of a fuzzer greatly depends on the input generation process and the fuzzer's performance, i.e., how many random inputs it can test during the available time. But other aspects such as good error detection are equally essential, as one misses crashes otherwise. To achieve the goal of finding bugs faster, numerous improvements have been suggested for the input generation. This has led to three categories of fuzzers: black-box, white-box, and grey-box fuzzers. A black-box fuzzer does not take the internals of the PUT into account. It solely relies on observations of the input/output behavior of the program. In contrast, white-box fuzzers generate test inputs by analyzing the internals of the PUT beforehand and during execution. White-box fuzzers generally apply static analysis techniques such as symbolic execution. These analyses are expensive, which results in fewer inputs being tested in the available time. Lastly, there are grey-box fuzzers which can be seen as in-between white- and black-box fuzzers. They rely on lightweight instrumentation to gather information during execution, such as code coverage, which is then used to guide the input generation. Here, one tries to strike the right balance between the analysis overhead for the input generation process and the number of inputs that can be tested in the available time.

Given that the goal is to create inputs that trigger bugs, white- and grey-box fuzzers should use their insight to aim for inputs that result in flawed behavior. However, before executing the PUT with the input, it is unclear whether the input causes the program to misbehave. What is more, one does not know how many mutations are required for the discovery of a buggy input. This makes quantifying the quality of inputs difficult. Therefore, white- and grey-box fuzzers generally try to maximize code coverage. Under the assumption that bugs are distributed uniformly in the code base, maximizing code coverage increases the likelihood of finding a bug. Miller's report [79] supports this intuition. It showed that a 1% increase in line coverage – a form of code coverage – results in an increase of detected bugs by 0.92%. Code coverage itself can be measured with different granularity. A straightforward metric is the aforementioned *line coverage*, which assesses the lines of source code covered when executing the PUT with the given input. Other common metrics include *edge coverage* and *block coverage*. Edge coverage measures

the covered edges of the control flow graph (CFG), similarly block coverage considers the blocks of the CFG. Besides different types of code coverage other metrics such as *calling context coverage* or *memory access coverage* have been evaluated as coverage metric alone and in combination with code coverage [105].

When it comes to the creation of the inputs itself, there are two main approaches. Firstly, there is the generation-based way. As the name implies, the inputs are generated from scratch. This usually happens according to some pattern or grammar. Secondly, there is the mutation-based method. With the mutation-based approach, a set of seed inputs is required, on which mutations such as bit flipping, block copying/deletion or arithmetic mutations are performed. In its basic form, the mutations are input agnostic, which can limit the effectiveness, for example, when a file type has a checksum that verifies the integrity of the file. A problem which is not present in a generation-based fuzzer. However, the generation-based method is more difficult to implement in comparison to a mutation-based one. The generation-based method requires more detailed knowledge about the file format to cover many features, whereas in the mutation-based approach one can use a diverse seed corpus. Due to the vast resources of the web, it is relatively easy to assemble such a corpus. By performing grammar-aware mutations the effectiveness is greatly improved. Similarly to the generation-based way, knowledge about the file format is required. However, a limited understanding is often sufficient to pass integrity checks and the like, which makes it arguably still easier to implement. Furthermore, a good seed corpus helps to quickly kick-start the penetration of deep code paths [92].

Since fuzzing allows the automatic discovery of inputs that cause the program to misbehave, it is also used for patch testing or crash reproduction. However, in those instances, the assumption of uniformly distributed bugs no longer holds. And even in the case of vulnerability research, targeting specific parts can be fruitful. This has lead to the introduction of *directed fuzzing*. The goal is to generate inputs that reach a certain part of the program. Different approaches have been developed. Some are using white-box fuzzing approaches such as taint analysis [49, 108], which tries to capture the information flow of the input, whereas others are build on mutation-based grey-box techniques [26, 35, 111]. A prominent way in directed grey-box fuzzing is to use a distance metric. This idea was introduced by Böhme et al. [26]. It is based on the intuition that the distance captures how far the trace of a given input is from the target location. If it is close, the input will be assigned a higher *energy*. A higher energy means that more time is spent fuzzing such an input. In other words, more mutants will originate from a seed with high energy (see Figure 2.2). This in turns increases the likelihood of getting closer to the target code location. We cover the distance metric in Subsection 3.3.4, as it is a central part of our design.

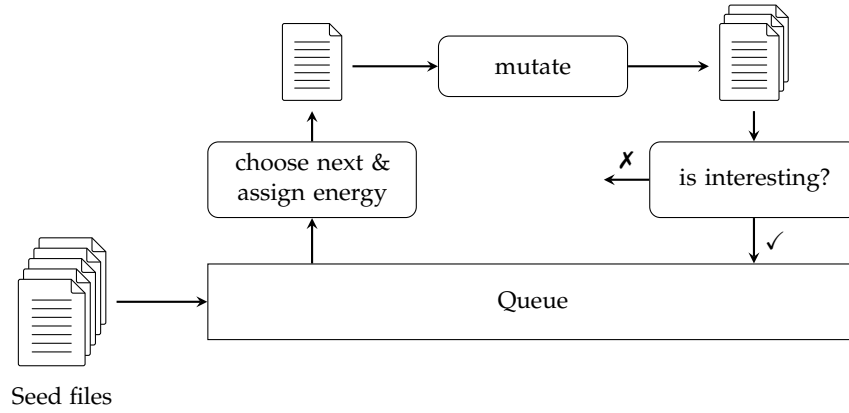


Figure 2.2: General architecture of coverage-based grey-box fuzzing. An input is taken from the queue and an energy is assigned. A higher energy results in more mutant files. A mutant is added to the queue if it is considered interesting. Usually new coverage is regarded as interesting. Adopted from Böhme et al. [27].

2.3 Test Suite Reduction

As software evolves, the test suite grows and regression testing becomes more expensive, especially in terms of time. Indeed, Microsoft reports some regression tests taking several days [33]. The goal of *test suite reduction*, also known as test suite minimization, is to get rid of redundant test cases from a test suite without significantly reducing the fault-detection capabilities. By doing so the regression testing becomes more efficient. Since it is impossible to know which test cases are valuable, some heuristic metric of similarity or quality is necessary. In spite of many different techniques, most research uses data-flow or structural criteria [97] such as edge [94, 55] or block [110, 98] coverage, which we already introduced in Section 2.2. However, other approaches have been proposed as well. For instance methods that leverage the history of the test cases [42, 74] or the execution time [113] as quality metric.

When data-flow or structural criteria are used for test suite reduction, the problem can be restated as finding the minimum subset of test cases such that the same criteria are fulfilled. This is equivalent to calculating the minimum set cover, or if the size of the samples is also taken into account the weighted minimum set cover. However, computing the set cover is known to be NP-complete [60] and finding the minimum one is NP-hard [63]. A number of heuristics have been developed for the finding of an approximated minimum set cover [44, 104, 32]. Naturally, such heuristics have been explored in the setting of test suite reduction [94, 87]. Furthermore, with the improvements made in constraint and satisfiability modulo theories (SMT) solvers, they have been explored as well [31, 46] in the context of test suite reduction.

2.4 Test Case Reduction

A test case normally exhibits some specifically desired behavior. For example, it can trigger a bug or cover some target functions. Any overhead that is unrelated to this desired behavior induce additional costs and is generally unwanted. Hence, it is intriguing to minimize a test case. However, doing so manually is tedious work and time-consuming. In lieu of performing this dull task manually, multiple automated techniques have been proposed that minimize a test case such that it still exhibits the same specific behavior. This process is referred to as both *test case reduction* and *test case minimization* in the literature. Test case reduction is for example especially useful in the compiler setting. Imagine a large project with hundred thousands lines of code that trigger a miscompilation bug. Without synthesizing a smaller code snippet from the project that reproduces the issue, it is extremely difficult to find the transformation that is responsible for the problem. That is why the GCC and LLVM maintainers ask for a reduced test case [8, 1] when reporting bugs. Another prominent use-case for test case reduction is found in fuzzing. Test inputs that are generated by fuzzers can be complicated and execute many bug unrelated paths. Reducing such a test input helps in locating the cause of the bug and hence facilitates exploitation or fixing.

Reducing a bug-causing input is approached in a trial and error fashion. We are not aware of any research that leverages taint analysis or similar to reduce an input. The most straightforward approach would be to simply calculate every subset of a test input. We keep the notion of subset of an input intentionally vague, but this could, for instance, be over the token-position pairs for a source file. In any case, calculating the power set requires exponential runtime! This has lead to first heuristic approaches for test case reduction in the compiler setting in the 1990s [34, 109]. A generic solution was formalized in 2002 by Zeller and Hildebrandt [114], called *delta debugging*. The term delta debugging and the idea was coined by Zeller in a previous paper [115] in a different setting, but is now mostly used in the setting of input reduction. Delta debugging is a divide and conquer method to distill interesting behavior. It starts coarse-grained by trying to remove half of the input as depicted in Figure 2.3. If the input with the removed chunk still exhibits the interesting behavior, the process continues with this part only. If the interesting behavior is not replicated in any of these inputs, more fine-grained chunks of quarter are removed. This process is applied recursively until the chunk size cannot be reduced any further. Then the smallest sample that triggered the interesting behavior is returned.

The original delta debugging implementation by Zeller and Hildebrandt used bytes as finest granularity. McPeak and Wilkerson implemented a variant of delta debugging that is line-based [77], which improves performance for source code files. This was further improved with *hierarchical delta debugging*

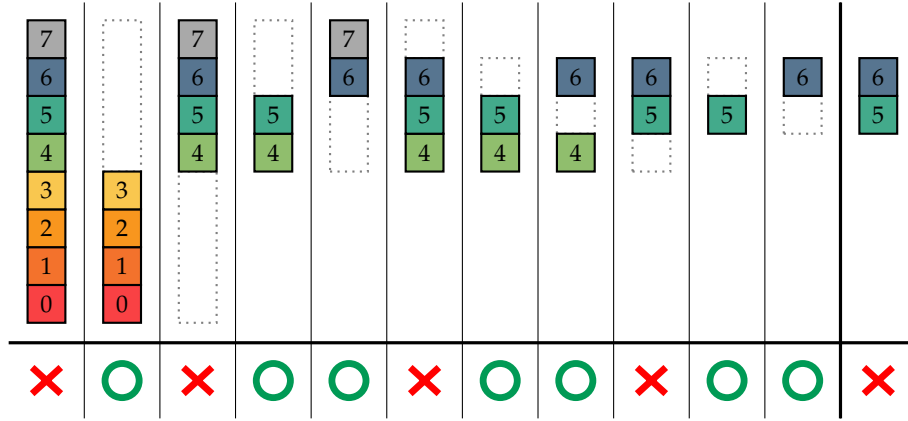


Figure 2.3: Delta debugging example: The stacked blocks correspond to a file. The left most file is the initial input file and the right most the returned reduced input file. The \times symbol indicates that the input exhibits the interesting behavior, e.g., a crash. A colored block is the finest granularity of the reduction. The grayly dotted part corresponds to the removed part in the current iteration.

(HDD) introduced by Mishherghi and Su [81], which leverages the hierarchical structure, in particular the abstract syntax tree (AST), of the input for the chunk selection. Herfert et al. [53] generalized this approach to arbitrary patterns and Sun et al. implemented the *perses* tool [100] which uses grammars provided in Backus–Naur form (BNF) for hierarchical reductions. The use of a grammar enables more advanced transformations and avoids the creation of syntactically invalid programs which can still happen with the AST-based HDD (e.g., by removing the variable name of a declaration). Input-specific approaches can do an even better job in reducing. C-Reduce [93] targets the C language and contains multiple source-to-source transformations next to hierarchical delta debugging support. These transformations help in overcoming local minima in the delta debugging process. This mitigates the risk of getting stuck at a local minimum shortly after start, when the input is still large.

If the test case is a source code file, a further problem has to be taken into account for some programming languages. The reduction of a source code file can result in a syntactically correct code file that exhibits the targeted behavior, but that is not well-defined. Therefore, an additional validation test might be necessary during each reduction step to avoid false positives. This is, for example, true for C programs. The C language is notoriously known for its undefined behavior edge cases, that can even introduce vulnerabilities in some instances. More importantly in this setting, test cases that exhibit undefined behavior are not accepted by compiler maintainers, as it justifies any compilation output. This drastic measure is taken to avoid too many false positive “bugs” being reported [93]. For languages such as C, deciding

whether an input source file has undefined behavior is however a non-trivial task and requires semantic-checking C interpreters such as KCC [43, 11] or Frama-C [40, 6].

2.5 Semantic Comparison

The semantic comparison of code has many applications. It can be used to optimize binaries by determining semantic equivalent code parts and removing redundancies [95]. In malware research, it can help in reverse engineering obfuscated malware [103] or in inferring the lineage of a strain [24, 54]. It can also be applied to detect plagiarism or copyright violations [73]. Furthermore, it can be used for compiler testing [66, 99]. However, determining the true semantic similarity is undecidable in general case [47]. While this is not a problem for aforementioned practical examples, it foreshadows the difficulty of the task.

Even on the source code level it is hard to synthesize the semantics of a function and compare it to another one, as the same behavior can be implemented with very different looking code. Despite this, for some applications, e.g., plagiarism detection, it can be sufficient to use syntactic similarity to approximate semantic similarity. The syntactic similarity is generally calculated over syntax trees to be resistant to minor code changes [59, 58]. Other approaches [71, 62] leverage the *program dependence graph* (PDG) [45]. The PDG represents functions in which the nodes are statements and control flow predicates. The edges represent the data and control dependencies. Subgraph isomorphism is then used to spot equivalences. In other settings, e.g., compiler verification, such approaches based on the syntax are not enough, since the assumption that syntactic similarity approximates semantic similarity does not hold. Le et al. proposed *Equivalence Modulo Inputs* (EMI) [66], which is a relaxed definition for semantic equivalence that does not rely on the syntax. Two programs are defined to be equivalent modulo inputs with respect to a finite input set, when the output of the first program equals the output of the second for every input of the input set. Jiang and Su also used a similar method in prior work on functions with random input generation to detect equivalent code based on I/O behavior [57].

Inferring the semantics from binaries poses additional challenges, as the compilation process results in a loss of information (e.g., typing information). Like in the source code case, many solutions rely on the assumption that code similarity approximates function semantic similarity. However, as binary code differs from high-level programming languages other metrics such as control flow [91], instruction type and order [107], or memory accesses [70] are used. This information must be recovered through static binary analysis techniques first, which introduces an additional source of inaccuracy. Since

different compilers or optimizations flags can already result in vastly different assembly, the assumption is even more often violated than in the source code case. In order not to rely on the syntactic equality, solutions that use symbolic execution have been introduced [50, 65]. The expressions computed by the symbolic execution engine are compared with a constraint solver on a basic block level. As this technique still depends on the structure of the basic blocks, improvements enhancing the quality using taint-analysis were proposed [73]. McKee et al. proposed *software ethology* [76] which, instead of using code properties of functions, abstracts inputs and corresponding program changes. The tool they implemented leverages mutational, coverage-guided fuzzing to discover a subset of inputs for each function. It then stores these inputs together with the measurable state changes as a fingerprint for a function. Each fingerprint corresponds to a semantic behavior and allows for an easy comparison between function.

Chapter 3

Design

3.1 Goals

The goal of this thesis is to create a tool which generates a minimal corpus of LLVM IR code snippets, that when compiled explore the maximal diversity of the source code of each optimization performed. Big projects in the size of Firefox or Chrome will trigger most optimizations, but the sheer complexity of the projects make it difficult to pinpoint which part of the source is transformed by which optimization. In contrast, a snippet, as proposed by us, mainly focuses on a particular optimization. This property is critical for some potential applications of the snippets. Imagine that one wants to evaluate an executable lifter, in such a case one is interested with which optimizations the executable lifter has problems. Simply knowing that the lifter fails is not enough. Another down-side of big projects is the compilation time, which is related to their size. Mozilla’s documentation states that it takes 120 min of CPU core time to compile files on a modern machine¹. Therefore, we aim to create snippets that are small (around 5 KiB) and fast to compile (0.5–1 s). The small size is a benefit in the previous example of a executable lifter as well. It helps in debugging and finding the cause of a potential failure. Another advantage of the IR snippets is their independence of the source language and their focus on the compiler back-end. Front-ends that translate source code to IR create “language-biased” IR, which can prevent optimizations from being triggered [18].

We focus on LLVM IR, but the concept extends to any IR. Furthermore, we limit our investigations to the architecture independent optimization of LLVM, as they are easier to reason about and analyze. Architecture specific optimizations require an additional intermediate representation. Furthermore, the optimizations are implemented in a domain specific language (DSL) unique to LLVM, which is “very hard for newcomers” according to

¹<https://firefox-source-docs.mozilla.org/build/buildsystem/slow.html>

the documentation². However, our approach does easily translate to architecture specific optimizations. Our implementation even supports defining architecture optimizations as targets. LTO is also not in our scope. LTO is expensive and not enabled by default, which makes it not a priority.

3.2 Overview

Figure 3.1 depicts a high-level overview of the snippet generation process. As an input of the snippet generator, one can specify the maximal number of snippets per optimization that should be generated and which optimizations should be targeted. The snippets are created with a directed mutational fuzzing based approach that uses compiler provided unit and regression tests as seeds. Note, for each optimization an individual fuzz campaign is launched, since we want snippets that mainly target a specific optimization as stated in the goals. If we ran one campaign which maximizes the coverage over all optimizations, the snippets would cover several optimizations and a distinction would be difficult. To assess the quality of a snippet, we have to know which optimizations are performed when the snippet is compiled. We use the edge coverage to track which parts of the optimizations are executed. The motivations for using edge coverage and directed fuzzing are covered in Section 3.3. The random mutations can add many fragments of code to a snippet that is not relevant for the targeted optimization. That is why we want to minimize each snippet afterwards. Each snippet is minimized individually using delta debugging on the LLVM IR structure. During delta debugging a minimization step is only performed when the edge coverage does not decrease. Lastly, we minimize over all generated snippets by calculating the

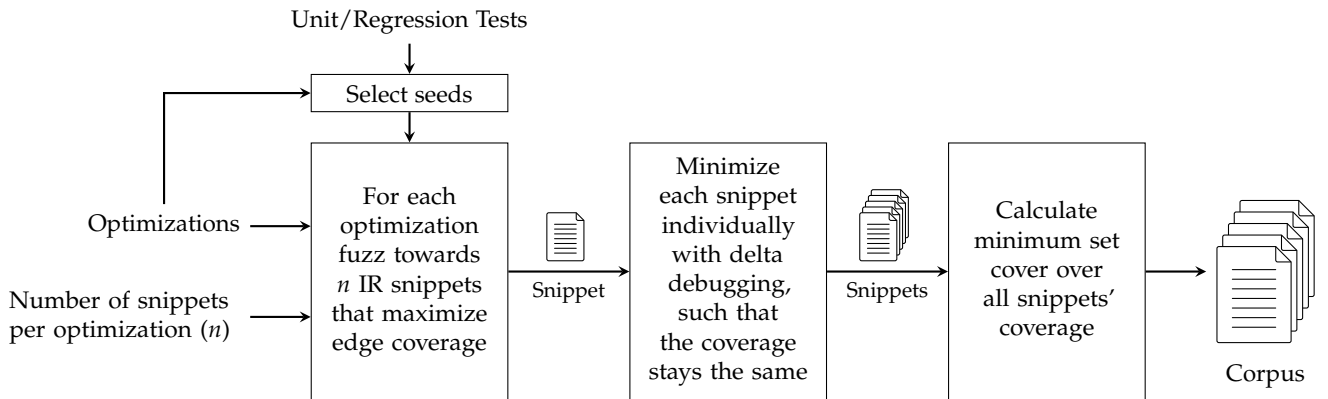


Figure 3.1: Design Overview

²<https://llvm.org/docs/TableGen/#tablegen-deficiencies>

minimum set cover, such that the union over the coverage stays the same. The design decisions of the minimization process are discussed in Section 3.4.

3.3 Fuzzing towards an IR Snippet

We aim to find IR snippets that trigger specific parts in a compiler. Ideally, one could perform some sort of static analysis like symbolic execution to determine an input file that leads to the execution of an optimization. However, this is not possible for large, real world software like a compiler. The constraints are too complex to solve and the state space is too big to be explored [22] which prevents a purely static solution. Fuzzing, on the other hand, has been shown to be effective in discovering inputs for real world software that lead to the execution of well hidden paths [61, 75]. We aim to exploit this property in the snippet generation process. We opt for grey-box fuzzing, as it provides the best trade-off between analysis overhead and valuable insights for the input generation. Since we operate on structured data we must use a generation-based or grammar-aware mutation-based fuzzing strategy. A uniformly random bit sequence is extremely unlikely to be well-formed IR code, similarly mutations like bit-flips or byte deletions are unlikely to create interesting IR snippets for the compiler. Encouraged by the many positive results of mutation-based fuzzing [17] we decided to take this route. Additionally, promising recent results [26, 35, 111, 88] in directed grey-box fuzzing use mutation-based approaches, which is another motivation. It is self-evident that we are not interested in maximizing code coverage over the whole compiler, as we want to trigger specific optimizations. With directed fuzzing we can steer the input generation towards IR snippets that execute the targeted optimization code in the compiler. The scheduler is the responsible part for the directed fuzzing, we explain its mechanics in Subsection 3.3.4.

During the fuzzing process we keep track of the n heuristically best inputs, as well as the interesting input files, which will be mutated. We limit the number in order to keep the corpus size small. Let M denote the set of the n best input files ($|M| = n$) and $cov : S \rightarrow \mathcal{C}$ the mapping from an input file to its coverage, when compiled. We replace a file $m' \in M$, if we have $|cov(f) \setminus \bigcup_{m \in M} cov(m)| \geq |cov(m') \setminus cov(f)|$ for a new input file f , i.e., a new file is added to M if its unique coverage is greater or equal than the maximal coverage that could be lost by removing m' . If this holds for multiple files $m' \in M$, we take the one such that $|cov(f) \setminus \bigcup_{m \in M} cov(m)| - |cov(m') \setminus cov(f)|$ is maximized. This heuristic guarantees that the total coverage ($\bigcup_{m \in M} cov(m)$) never decreases. Ideally, one would check for $|cov(f) \setminus \bigcup_{m \in M} cov(m)| \geq |cov(m') \setminus (cov(f) \cup \bigcup_{m \in M \setminus m'} cov(m))|$, however this is computationally more expensive. The previous test can

be added with $\mathcal{O}(1)$ complexity, the latter has $\mathcal{O}(\sum_{m \in M} |\text{cov}(m)|)$. Each file which is put into the n heuristically best is also considered interesting and will be mutated. Mutating those snippets may further increase their coverage. Additionally, all files which lead to unique coverage in the targeted optimization are considered interesting and further mutated. At this moment they may not have enough coverage to be chosen as one of the n best, but a following mutation has good chances of increasing the coverage of the n best snippets.

3.3.1 Seed Selection

In mutational fuzzing, the seeds are a crucial part of the fuzzer's success [68]. The seeds provide a base coverage which the mutations expand. Therefore, a seed corpus with good coverage saves time, as the fuzzer does not have to discover inputs with the behavior of the corpus first. While a larger corpus is more likely to provide good coverage, one must make sure that the corpus has only little redundant coverage among the different files. If there is too much redundant coverage with a large seed corpus, mutations are applied on similar inputs and are in turn less likely to create new, interesting coverage. In other words, time is wasted. Many compilers come with unit and regression tests. We deem them to be perfect for our application, as they are generally small, numerous, and neatly organized by compiler feature and optimization. In the case of LLVM there are 23481 LLVM IR files in total, 5639 of which cover optimization transformations. To avoid redundancies, we only select the test files that are specifically written for the targeted optimization.

3.3.2 Coverage

As discussed in Section 2.2 there are multiple ways to measure the coverage of an executed program given an input. Since we want to target optimizations, using a form of code coverage is the most practical. Thereby we can steer the directed fuzzing towards covering the code that implements the targeted optimization. A metric such as memory-access coverage is not useful, because it is difficult to predict how the compiler accesses memory for a specific optimization. For the same reason, it is unlikely that combining code coverage with memory-access coverage yields benefits in our case. Basic block, edge, and path coverage are common code coverage metrics [68] with inherent trade-offs between accuracy and overhead. Basic block coverage is the least accurate with a small overhead and path coverage the most accurate with a lot of overhead. We decided to use edge coverage, as it is more accurate than basic block coverage and captures the “diversity” of an optimization better, while having moderate overhead. Consider Figure 3.2 which shows an excerpt from the *instruction combination* optimization and the corresponding control flow graph when compiled. This piece of code is part of a 67 lines

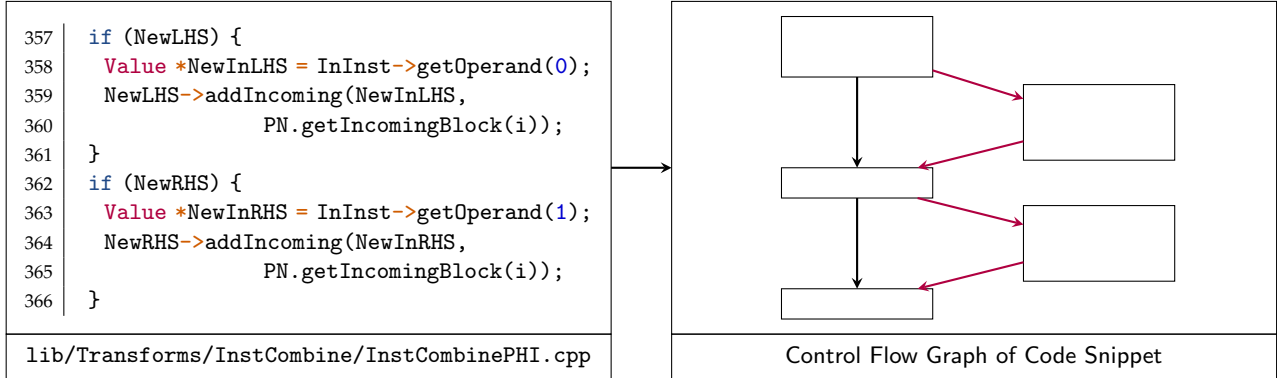


Figure 3.2: Edge coverage is more expressive than block coverage. 100% block coverage of this snippet does not cover each outcome of this part of the optimization.

function, and depending on whether `NewLHS` and `NewRHS` both are set, or only one of the two is set, a different optimization step is performed. As one can see in the control flow graph, it is possible to reach 100% block coverage if one follows the red path, i.e., when `NewLHS` and `NewRHS` are set. So good code coverage appears possible, even though not every optimization step of this particular optimization was performed. In the case of edge coverage this is less of a problem, as the red path does not cover all edges, which would imply that we have to improve our input such that the other edges are also covered. However, we lose information with edge coverage too. If this piece of code is executed once with `NewLHS` and `NewRHS` both set, and once with both not being set, we will get 100% edge coverage. The cases where only one of the two variables is set are not accounted for. To have an even more accurate view, one could use path coverage. With path coverage the complete execution trace is tracked. The memory overhead is, however, too big [48] to keep track of the exact paths. Wang et al. [105] introduced *n-gram coverage* which approximates path coverage by maintaining a history of the last n edges. This comes with some cost and we believe edge coverage to be a sufficient approximation for our use-case, after some unrepresentative manual analysis of parts of the LLVM compiler code. Moreover, it is also widely used in other applications with great success. The code snippet in Figure 3.2 is, for instance, guarded by a check that makes sure that at least one of the two variables is set, which minimizes the aforementioned issue. Whether a path coverage approximation is more expressive in our setting is left to further research.

3.3.3 Mutation

As described in the introduction of this section, we use grammar-aware mutations. Arbitrary mutations on LLVM IR would not result in valid IR code in most cases and thus not help in increasing the code coverage. We

use three mutations which we explain in the following paragraphs. Each mutation has a weight assigned and the fuzzer takes a weighted random sample to decide which mutation to perform next.

Instruction insertion The first mutation inserts a new instruction into an IR snippet. The extension of an IR snippet should allow us to discover new instruction combinations that trigger other optimization steps. We cannot simply insert an instruction without using previous values or without using the result of the instruction, because it would be trivially removed. In other words, only the dead code elimination optimization would be triggered. Therefore, an instruction must use previously created variables and if the instruction returns a result it must be used somewhere afterwards. At the EuroLLVM conference in 2017, Bogner presented an instruction insertion mutation technique for LLVM IR [25]. To add an instruction into a random

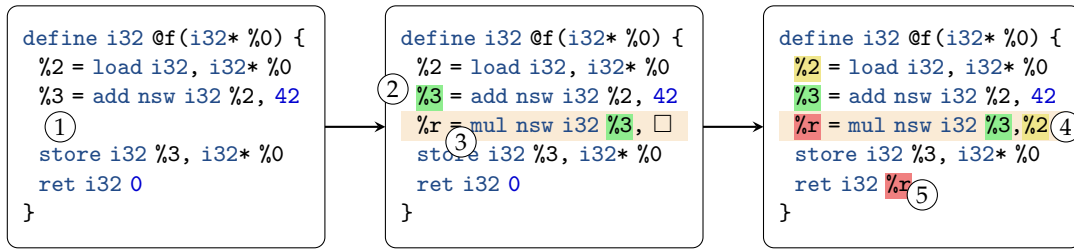


Figure 3.3: The instruction insertion mutation process: ① Divide a basic block at random location. ② Choose a source register randomly ③ Choose an instruction randomly based on the previous register type ④ Fill in the remaining arguments of the operation ⑤ Take an argument of a later instruction as sink and clean up if necessary.

basic block, the block is split into two parts. The registers before are potential sources and the ones after are potential sinks. Note that is possible, because LLVM IR is in SSA form, i.e., each variable has a single definition. In a next step, a random source register is selected which acts as an instruction constrain. After that, a random instruction, which has at least one argument of the same type as the previously chosen register, is selected. Then, the other arguments are filled in with randomly chosen sources. With this, any later argument is taken as a sink. Eventually, dead code resulting from the transformation is cleaned up. In Figure 3.3 one can see an example. The weight of this mutation equals the number of supported instructions.

Instruction deletion To prevent that the snippets get too big while fuzzing, we have an instruction deletion mutation. In order to avoid removing instructions that contribute to the coverage of the targeted optimization, this mutation can only be selected by the fuzzer, if the input file is only 1 KiB away from reaching the maximum size. While removing instructions can

still lead to worse coverage, it is more likely that we choose an unnecessary instruction. If the input is 100 B away from the maximum size we always perform this mutation, to avoid having larger snippets than desired. Otherwise, the weight with which this mutation is selected is linearly interpolated from zero to two times the weight of the other mutations. That is to regulate the aggressiveness of the instruction deletion and increase its likelihood when we approach the size limit. The deletion process is straightforward. An instruction is selected randomly and then removed. If the instruction has a return value, all usages of the value are replaced by another of the same type. If no other variable of the same type is available, a constant is used instead.

Function extraction Since we want to create only a few snippets per optimization, we must also mingle different input files. Some snippet might, for instance, have an interesting sequence of instructions that triggers a specific part of the optimization, but that is the only interesting part about the snippet and besides that it has bad coverage. Another snippet with already quite good coverage could profit from this instruction sequence. That is the motivation for this mutation. The mutation selects a random function from another snippet and inserts it into the snippet that is being mutated. The advantage of taking a whole function is that it is relatively cheap and no cycles on the analysis of the types have to be spent. The definition of the functions called in the cloned function are not copied, only the declaration thereof. While this can result in the loss of some *interprocedural optimizations* (IPOs) like inlining, we believe this to be a sensible limitation. Firstly, not every IPO is affected (e.g., the merging of constants or deducing function attributes) and secondly, copying functions recursively might increase the mutated input significantly. We acknowledge that there is room for improvement in this regard, by applying a more flexible solution that takes the different variables such as the target optimization and the size into account. The weight of this mutation is 1, i.e., it is chosen with the same probability as a specific instruction type is inserted.

3.3.4 Scheduler

The scheduler manages the energy of an input. A higher energy means the input is modified more often, i.e., more mutants originate from this input. Our main scheduler is adopted from Böhme et al.'s AFLGo [26], who introduced directed grey-box fuzzing. Directed fuzzing is applicable to us, because we want to steer our fuzzing effort towards specific code locations in the compiler, namely the pieces of code that implement the optimizations. As briefly mentioned in Section 2.2, with directed grey-box fuzzing, inputs get an energy assigned according to their *input distance*. A lower distance implies that the trace of the input is closer to the target location. Inputs with closer traces need fewer mutations to generate an input whose trace covers the

target code location. We briefly recap the distance metric definitions from the paper by Böhme et al. here, for the motivation of this metric we refer to the source [26]. Let T_f denote the target functions and T_b the target basic blocks. In order to compute the *input distance* (called *seed distance* in Böhme et al.'s paper), a *function-level target distance* is assigned to each node in the call graph (CG). Similarly, each node in the intra-procedural control-flow graphs (CFGs) has a given *basic-block-level target distance*. The *function-level target distance* between a function f and the target functions T_f is defined as the harmonic mean of the function distances between f and any reachable target function $t_f \in T_f$. The function distance from f to f' refers to the shortest path from f to f' in the CG:

$$d_f(f, T_f) = \begin{cases} \text{undefined} & \text{if no target function is reachable from } f \\ \left[\sum_{t_f} d_f(f, t_f)^{-1} \right]^{-1} & \text{otherwise} \end{cases}$$

The *basic-block-level target distance* is the harmonic averaged distance to any other basic block that calls a function in the call chain towards the target location or zero if the block itself is a target. More formally, let $d_b(b, b')$ be the shortest path from basic block b to b' if they are in the same CFG of some function f . We define $N(b)$ to be the set of functions called from basic block b such that every element has a path to a target function. Furthermore, let T be the set of basic blocks that have at least one such function call, i.e., $\forall b \in T. N(b) \neq \emptyset$. The basic-block-level target distance between a basic block b and the target basic block T_b is then defined by:

$$d_b(b, T_b) = \begin{cases} 0 & \text{if } b \in T_b \\ 10 \cdot \min_{f \in N(b)} \{d_f(f, T_f)\} & \text{if } b \in T \\ \left[\sum_{t \in T} (d_b(b, t) + d_b(t, T_b))^{-1} \right]^{-1} & \text{otherwise} \end{cases}$$

Now, we have the necessary definitions for the input distance. Given the execution trace $\xi(s)$ of an input s , the input distance $d(s, T_b)$ is defined to be:

$$d(s, T_b) = \frac{\sum_{b \in \xi(s)} d_b(b, T_b)}{|\xi(s)|}$$

In the fuzzing campaign a normalized input distance with regard to the maximum and minimum distance inputs over all previous inputs S is used:

$$\tilde{d}(s, T_b) = \frac{d(s, T_b) - \min_{s' \in S} \{d(s', T_b)\}}{\max_{s' \in S} \{d(s', T_b)\} - \min_{s' \in S} \{d(s', T_b)\}}$$

At this point, we finally have all the ingredients for the description of the actual scheduler. Here, we slightly digress from the paper, which is due to implementation differences (see Section 4.3). AFLGo differentiates between an “exploration” and “exploitation” phase. Initially, its scheduler does not give any weight to the distance metric, but as the time processes, the distance metric is gradually weighted in more for the energy assignment. We, on the other hand, use the distance metric right from the start as follows due to the aforementioned implementation differences:

$$p_{libfuzzer}(s, T_b) = 2^{2.0 \cdot (0.5 - \tilde{d}(s, T_b))}$$

This corresponds roughly to AFLGo’s exploitation phase, where the closer inputs are assigned more energy.

We have two additional power schedulers. In those two cases, we only instrument³ the targeted optimization. In other words, we only get coverage information of the optimization. One is the default scheduler of libfuzzer, which assigns more energy to interesting newer inputs. The other one assigns energy proportional to the coverage of a given input. This can be seen as a poor man’s directed fuzzing as it favors inputs that cover more of the optimization already, which might help in finding additional coverage.

3.4 Minimization of IR Snippets

As mentioned in the design overview we perform two types of minimizations. First, we perform a test case reduction, followed by a test suite reduction over all snippets. We do not perform any reductions during the fuzzing process as it might hinder the discovery of new interesting coverage. Additionally, reducing an input is time intensive, so it would be wasteful to spend time minimizing inputs that will not land in the final corpus. The motivation for the test suite reduction step is that we might end up with snippets that also trigger a good part of another optimization, even to an extent that some specific snippets created for that optimization are redundant. With the reduction step over all snippets we can eliminate those redundancies.

The individual snippet minimization is done using a hierarchical delta debugging approach on the LLVM IR. In each reduction step we want to keep the same coverage, so we consider a reduction to be interesting if the edge coverage does not change. LLVM ships with a customizable reduction tool for IR, which performs delta debugging with the following different granularities:

³Technically this is not correct. We rather ignore the not targeted instrumented parts to avoid recompilation.

- | | |
|-----------------|-----------------------|
| 1. Functions | 5. Function arguments |
| 2. Basic blocks | 6. Instructions |
| 3. Globals | 7. Operand bundles |
| 4. Metadata | 8. Attributes |

As in the delta debugging process itself, one wants to start with coarse-grained reductions and then move to finer representations such as instructions. Consequently, big unnecessary fragments are removed right away and less time is spent with the finer granularities.

For the minimization over all snippets, we first compute the union of the coverage gained by executing the snippets. Afterwards we calculate the minimum set cover, which is feasible in our setting thanks to SMT solvers. More formally, let S be the corpus of snippets and $cov : S \rightarrow \mathcal{C}$ the function that maps a snippet to the set of covered edges of the CFG when the snippet is compiled. We want to find $S' \subseteq S$ with a minimal number of elements such that $\bigcup_{s' \in S'} cov(s') = \bigcup_{s \in S} cov(s)$.

Implementation

4.1 Overview

In this chapter, we discuss our implementation of the proposed design. The project totals to 2178 lines of code, 1269 of which are in C++, the rest – mostly glue – consists of Python. Figure 4.1 depicts an overview of the different parts. We have a script that creates so called *target descriptions* from the optimization names. A target description can contain the following information

- seed files, e.g., `seeds:/path/to/seeds/*.ll`
- source files, e.g., `src:/path/to/InstCombinePhi.cpp`
- functions, e.g., `fun:*FoldIntegerTypedPHI*`

We create these target descriptions before the snippet generation process, as it allows users to hand-tune the targets. To be easily modifiable, the target descriptions support wildcards. The target descriptions generated by the script only specify which source files belong to an optimization and which seeds should be used. For our general purpose this is enough precise. But some optimizations have functions which are only for debugging purposes, by hand-tuning the target description one can select the exact functions of targets. Furthermore, this step allows users to write target descriptions for architecture dependent optimizations, which are not supported by the target description generation process.

The target description generator selects the seeds from LLVM’s regression tests, which can be altered in the target description if wished. The regression tests of LLVM are small pieces of code that test a specific feature of LLVM or have triggered a bug in LLVM previously. Thus, they should provide good base coverage. The tests are also neatly organized by which part of compiler they test. We only select the tests specifically written for the targeted optimization. This allows us to avoid most coverage redundancies without spending a lot of CPU cycles.

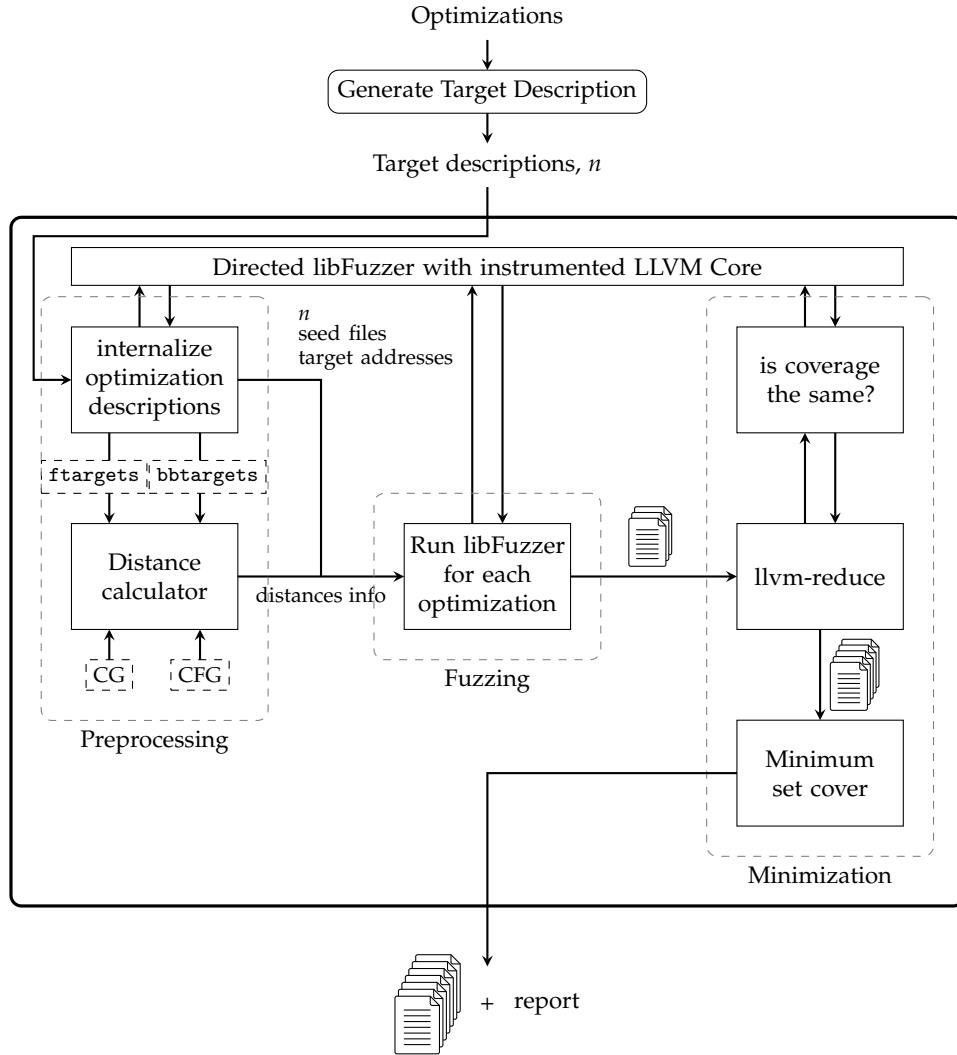


Figure 4.1: Implementation subdivided into the different parts.

In a next step, the target descriptions are passed to the *snippet generator*. The snippet generation process is subdivided into three parts. The preprocessing, which takes care of cacheable computations, the fuzzing itself, and the minimization of the snippets. All three parts have a component that communicates with an instrumented LLVM Core version 11 which is linked with a fuzzer in order to get coverage or symbol information, or to start the fuzzing process. We instrumented all optimizations passes and the code generation part. By not instrumenting components like IR parsing we gain efficiency as well as accuracy. Note, we instrument all optimizations and not simply the targeted one, as this would require recompilations. For the fuzzer, we considered AFLGo and libFuzzer. AFLGo has the directed fuzzing

implemented and builds on top of AFL, a widely adopted fuzzer. LibFuzzer is a LLVM project that has been integrated with LLVM Core in previous work and is also considered state-of-the-art. We opted for libFuzzer, as the changes necessary to integrated LLVM Core and support directed fuzzing seemed smaller. In particular, AFLGo requires that the PUT is compiled twice per directed fuzzing target and one of the compilation has to be done with LTO. This is very time-consuming and would result in a compiler per optimization. This could be changed with engineering effort, but it is not straightforward. The design of libFuzzer was well-suited for modifications to support directed fuzzing, while not requiring recompilations.

4.2 Preprocessing

Certain tasks have to be done only once per target description. For instance computing the distance information of a target. Therefore, we perform a preprocessing step to avoid such recomputations. Consequently, when one wants to rerun the generation process for an optimization, e.g., to spend more time on fuzzing, one will not have to perform those time-consuming steps again. The preprocessing step is subdivided into two components. The distance computation, and the internalization of the target descriptions, which is essentially the translation of the target descriptions into another format.

4.2.1 Internalization

The target descriptions contain filenames and function names. In order to be compatible with AFLGo's distance calculator we want to have basic block labels and the function names in mangled form (`bbtargets` and `ftargets` in Figure 4.1). Basic block labels are strings with the filename and code line to the first line to which the basic block corresponds in code (e.g., `InstCombinePHI.cpp:358`). So we translate the lines of the target description that specify the filenames and function names into this format in this step. Additionally, we also translate the targeted functions to the corresponding program counter addresses in the instrumented LLVM Core. The reason thereof is that we have to know which instrumented points belong to the targeted optimization while fuzzing to keep track of the best n snippets. By knowing the address we can check if the program counter is set to such a target address in a computationally cheap way. The alternative of symbolizing the program pointer at each step to get a filename and function name string, and comparing it to the values from the target description is much more expensive. This would slow down fuzzing. AFLGo inserts the distance right at the instrumentation, while this is even cheaper this is not an option in our case, as it would require a recompilation for each optimization.

4.2.2 Distance calculation

The distance calculation has to be performed once per target description. The target information, the CG, and the CFG are required to calculate it. Strictly speaking we generate the CG and CFG in the first run of the snippet generation process from a LLVM IR file of the compiler. This is a left-over from the experimenting phase and it rather belongs to the build process, as the CFG and CG will not change for a particular LLVM version. This interface to the distance calculator is the same as in AFLGo. AFLGo's implementation was, however, too slow for a project of the size of LLVM. We rewrote this part in 267 lines of C++ and made algorithmic improvements. Thanks to the same interface we could contribute our improvements to the AFLGo project (see Appendix A).

4.3 Fuzzing

As explained before we build upon libFuzzer for the fuzzing process as previous work has integrated the LLVM Core with libFuzzer already. These integrations only focused on specific subparts of the LLVM Core, like for instance the instruction selection process of the code generation. With relatively little effort, we adapted it to the complete compilation process from LLVM IR to machine code. We fuzz the compilation from IR to machine code, because we wanted our implementation to also work with architecture dependent optimizations, but more importantly we wanted to reject untranslatable LLVM IR code. LLVM IR code can be architecture dependent, e.g., when it contains inline assembly. If the inline assembly is, for instance, for ARM we cannot compile it to x86 machine code, it is untranslatable. However, these cases are easily excludable and would not require a compilation to machine code. The problem arises when generic LLVM IR fails to compile. Theoretically this should not happen and it is considered a bug by LLVM, but these kinds of bugs have a low priority¹. That is why we want to reject such cases during fuzzing to avoid ending up with uncompileable IR snippets. To perform the directed fuzzing we modified libFuzzer such that it parses a distance file and a target file as generated in the internalization step. As a reminder, this file contains addresses of the code locations that belong to the targeted optimization. Furthermore, there is a flag for the number of snippets that should be generated in this session. This information is used in the execution phase of the fuzzing, i.e., when we compile the input, to decide whether the input is interesting and if yes, what its distance is. The input distance is used to calculate the energy as explained in Section 3.3. LibFuzzer implements the scheduling differently to AFL. A snippet that is considered

¹e.g., https://bugs.llvm.org/show_bug.cgi?id=40451, https://bugs.llvm.org/show_bug.cgi?id=45649

interesting is not added to a queue, but to a corpus set from which the next seed to mutate is chosen randomly according to some weight distribution. LibFuzzer’s default weight distribution is that the i -th value has weight i (one indexed). In our proportional scheduler the i -th element has weight $i \cdot |cov(f_i)|$, with the intuition that we want to mutate snippets with good coverage more. Lastly, with our directed fuzzing scheduler we set the weight as described in Subsection 3.3.4. Since even seeds with a longer distances stay in the corpus, they can be selected with a low probability, which is why we do not distinguish between an exploration and exploitation phase from an implementation perspective. Recall, the exploration phase in AFLGo essentially corresponds to not using the input distance. AFLGo gradually increases the weight of the input distance until a point where it is dominant which is called the exploitation phase.

LibFuzzer has a simple interface that lets one set a custom mutator. The existing LLVM libFuzzer bindings already ship with the insertion and deletion mutations proposed at LLVMEuro 2017 [25], which we used. However, the instruction deletion mutation was buggy, which we fixed. We submitted the patch upstream to the LLVM project². We added the function insertion mutation, which takes a random function from an uniformly random IR file that is in the “interesting IR corpus”. When copying a function from one IR to another, one has to be extra careful with references to metadata and attributes to create a true deep copy. Missing some data can result in invalid IR or lead to a crash.

4.4 Minimization

In order to reduce a snippet individually we leverage *llvm-reduce*, which is part of LLVM Core. It performs hierarchical delta debugging on the LLVM IR and is customizable which makes it perfect for our use-case. We can setup *llvm-reduce* such that it queries an oracle which says whether the reduction resulted in a loss of coverage or not. The oracle is trivially implementable. We compile the snippet with the instrumented LLVM Core and compare the coverage to the reference. If the coverage of the reduced input is a superset of the reference coverage, we accept the reduction. We consider the superset, because a reduction might even increase coverage. We do not simply look at the number of coverage points as the coverage points could change to other parts of the optimization which are covered by another snippet. To speed up the process we perform multiple reductions on different files in parallel.

This minimization over all snippets is done using Z3 [82]. Z3 is a satisfiability modulo theories (SMT) solver from Microsoft licenced under the MIT licence.

²<https://reviews.llvm.org/D96207>

It received the 2018 Test of Time Award³ and is considered the state-of-the-art in SMT solvers. Therefore, we chose it as a basis to calculate the minimum set cover. Z3 allows users to formulate a minimization problem. We introduce a boolean symbolic variable for each snippet. The boolean is the indicator variable if a snippet is part of the cover. We constrained the solution set such that the union of the coverage of all snippets that have the indicator set equals the union of every file's coverage by formulating it as a boolean satisfiability problem (SAT). We, then, minimize over the sum of indicator variables. More formally, let S denote the set of snippets, and let $I_s \in \{0, 1\}$ be an indicator variable for each snippet $s \in S$. Furthermore, we define $cov : S \rightarrow \mathcal{C}$ to be the mapping from a snippet to its covered edges of the CFG when compiled. We instruct Z3 to solve the following minimization problem:


$$\begin{aligned}
& \text{minimize} && \sum_{s \in S} I_s \\
& \text{subject to} && \bigwedge_{c \in cov(S)} \bigvee_{\substack{s \in S \\ c \in cov(s)}} I_s \text{ is satisfied} \\
& && I_s \in \{0, 1\} \quad \forall s \in S
\end{aligned}$$

³<https://etaps.org/2018/test-of-time-award>

Evaluation

In this chapter we evaluate the generated corpus. Firstly, we look at the properties of the snippets like the achieved edge coverage, size, and compilation duration. This gives some idea about the quality of the snippets, and in what context they might be useful. Secondly, we assess how the snippets performs in finding bugs in LLVM, and RetroWrite [41] a state-of-the-art binary rewriter. Our evaluation shows that the generated snippet corpus is fast to compile, has good coverage, and is able to uncover previously unreported bugs.


5.1 Generated IR Snippets

In this section we evaluate the achieved coverage, the size, and compilation time of the snippets. All tests were performed on a Dell OptiPlex 7080 running Ubuntu 18.04 with an i7-10700 CPU clocked at 2.90 GHz and with 32 GB of RAM. We restricted the evaluation to targeting level two optimizations (-O2), because the -O2 flag is predominantly used by big software projects (e.g., Firefox, Chrome, Linux). Due to time limitations we present some preliminary results that were run with turbo boost enabled and on multiple cores. We clearly indicate for which results this is the case with the following symbol: . If the symbol is not present, the measurements were performed on a single core with turbo boost disabled. The former type of results should be taken with a grain of salt when comparing, since the scheduling over multiple cores and Intel's turbo boost feature can cause some unwanted variance.

In a first step we ran the generation process on a subset of optimizations in order to decide on a number of snippets per optimization. The *SLP-Vectorizer* and *InstCombineCompares* optimization were selected based on the high number of instrumented code points and their coverage achieved with one snippet, as previous investigations hinted that optimizations with

5.1. Generated IR Snippets

n	SLPVectorizer	MergeICmps	Reassociate	VectorCombine	InstCombineCompares
1	20.02%	37.17%	36.42%	39.76%	11.22%
5	40.48%	37.55%	37.03%	36.59%	13.93%
25	44.46%	41.26%	38.99%	36.34%	24.17%
50	46.49%	41.82%	40.55%	39.02%	28.37%
75	46.60%	42.94%	40.51%	46.59%	31.34%
80	46.52%	42.01%	41.22%	47.07%	31.64%
100	47.02%	41.82%	41.29%	47.32%	31.25%

Table 5.1: Coverage achieved by optimizations with different number of snippets (n), when fuzzing for 1 h in parallel. 

many instrumented points resulted in worse coverage. The others were chosen randomly to inspect how increasing the number of snippets act on other optimizations. The results are listed in Table 5.1. Each result is the measurement of one run. While the random effect of fuzzing can skew such results, it is acceptable in this instance, because we are interested in the trend and we have 5 different optimizations. As one can gather from the table, more snippets increase the coverage (with some small exceptions caused by the aforementioned randomness). Hence, the function extraction mutation is not enough to increase the diversity of one snippet, at least not when each snippet size is limited. It appears that with around 50 snippets per optimization we reach a coverage wall. Since we only looked at a subset of optimizations and in order to be conservative given that we ran the measurements once, we decided to continue with $n = 80$ for further evaluations. That way we should be sure that we saturate the coverage and do not lose valuable snippets for other optimizations. Note, reaching 100% code coverage is extremely unlikely. This can be due to several reasons. For instance, the program can contain dead code or parts that are only executed with a more aggressive optimization level. Moreover, in the case of LLVM there are two different pass managers which schedule the optimizations. Both require boilerplate code for each optimization, but only one of the two can be chosen by the user at a time.

Since generating snippets for all optimization takes rather long (the fuzzing alone takes almost 2.5 days) and due to time limitations, we ended up generating the whole corpus for $n = 80$ only twice, once with libFuzzer’s default scheduler and once with the directed fuzzing scheduler. We omitted the proportional scheduler for time reasons. Since fuzzing relies on randomness, running it once is not sufficient, even though previous papers assumed that running a fuzzer for a long time *evens out* the result [61]. Arcuri and Briand [21] suggest using a Mann-Whitney-U-Test to compare results. Therefore, we evaluated a subset of optimizations with five runs and did a Mann-Whitney-U-Test. Additionally, we report the coverage discovery of the

median result in a plot. These evaluations were performed without turbo boost and on a single core to guarantee comparability. We still included the data of the *one-run-sample* snippets, as it gives some interesting insight about certain optimizations and snippet sizes. Furthermore, we use those snippets in Section 5.2.

Table 5.2 contains the coverage information of the *one-run-sample* snippets, once generated with the directed fuzzing scheduler, once with the default scheduler. The table additionally includes the coverage of CTMark [5], a compiler benchmark. CTMark is a collection of *7zip*, *Bullet*, *CalmAV*, *consumer-typeset*, *kimwitu++*, *lencod*, *mafft*, *SPASS*, *sqlite3*, *tramp3d-v4* and should therefore have a fairly good code coverage. As shown in Table 5.2, the *LoopSink*, *ISCCP*, *LoopRotation* and *ForceFunctionsAttrs* optimization have the same coverage for all three types of inputs. This could have several reasons. One possibility is that these optimizations are fully triggered and the other edges are pessimistic error handling paths. Or maybe it has some rare edge cases that are extremely difficult to trigger. Unfortunately, we did not have time to investigate this further. Furthermore, the *InstSimplifyPass* results indicate that our mutations are not capable of triggering this optimization. CTMark has a way higher coverage while the default and directed fuzzing scheduler have the same coverage. This is also the case for the following optimizations: *PartiallyInlineLibCalls*, *ElimAvailExtern*, and *InferFunctionAttrs*. The *InstSimplifyPass* optimization performs transformations such as `add i32 1, 0 → i32 1` or `and i32 %x, %x → i32 %x`, but it also includes more complex ones. As of now we have not pinpointed the issue that leads to the poor performance of our approach. In the case of *PartiallyInlineLibCalls* it is clear why we do not increase the coverage. The reason is that our mutations do not create any library calls. Similarly, for *InferFunctionAttrs* which infers the attributes from the libc library functions. Lastly, *ElimAvailExtern* eliminates available external global definitions, this is also not handled by our mutations. This implies improvement potential in the mutations. The directed fuzzing created snippets that had strictly greater coverage than CTMark in 35 of 59 cases. In comparison to the default scheduler it reached higher coverage in 43 cases.

The total size of the directed fuzzed corpus in the bitcode format before the minimization step was 18 810 200 B. The minimization process resulted in a reduction of slightly more than a third, i.e., the corpus size was 12 409 252 B after the minimization. While we only casually measured the time Z3 had to calculate the minimum set cover, it might interest the reader. In this instance it took around 13 min, which is quite impressive! Since there is no particular reason to have multiple files per optimization, we also combined snippets of the same optimization into one file, if possible. This reduced the size further to 8 342 356 B, as we do not have to store some format details of the IR multiple times. Compiling the snippets in this form takes 44.99 s, in comparison to CTMark which takes 253.00 s this is more than five times

5.1. Generated IR Snippets

	ADCE	Alignment From Assumptions	BDCE	Called Value Propagation	Canonicalize FreezelnLoops	Constant Hoisting	Constant Merge	Correlated Value Propagation	Dead Argument Elimination	Dead Store Elimination	DivRemPairs	EarlyCSE
DF	45.53%	48.72%	50.69%	43.26%	45.80%	30.46%	41.98%	42.86%	51.08%	22.15%	43.71%	43.07%
Default	43.66%	47.44%	48.85%	43.46%	43.51%	29.24%	41.98%	40.51%	50.41%	21.38%	42.38%	42.13%
CTMark	45.82%	1.28%	49.31%	51.51%	32.82%	35.99%	35.85%	43.58%	44.32%	22.32%	38.41%	43.88%
	ElimAvail Extern	Float2Int	ForceFunction Attrs	Function Attrs	GVN	Global DCE	Global Opt	ISCCP	IndVar Simplify	InferFunction Attrs	InstCombine AddSub	InstCombine AndOrXor
DF	25.53%	49.91%	4.42%	45.29%	36.07%	46.25%	42.76%	21.74%	46.62%	21.21%	39.34%	37.81%
Default	25.53%	49.40%	4.42%	44.39%	32.17%	44.73%	34.36%	21.74%	42.25%	21.21%	38.27%	32.92%
CTMark	29.79%	48.36%	4.42%	44.54%	41.95%	33.80%	19.40%	21.74%	48.41%	24.24%	25.39%	31.38%
	InstCombine AtomicRMW	InstCombine Calls	InstCombine Casts	InstCombine Compares	InstCombine LoadStore Alloca	InstCombine MulDivRem	InstCombine Negator	InstCombine PHI	InstCombine Select	InstCombine Shifts	InstCombine Simplify Demanded	InstCombine VectorOps
DF	57.48%	18.82%	41.33%	31.64%	47.63%	43.40%	51.54%	41.48%	46.27%	51.89%	46.95%	46.92%
Default	57.48%	14.89%	41.67%	28.68%	40.73%	39.14%	49.49%	37.33%	42.00%	50.38%	44.54%	45.64%
CTMark	0.00%	6.28%	37.42%	30.59%	32.77%	25.82%	42.66%	42.29%	33.74%	41.31%	31.49%	20.67%
	InstSimplify Pass	Instruction Combining	Jump Threading	Loop Deletion	Loop Distribute	LoopIdiom Recognize	LoopLoad Elimination	Loop Rotation	Loop Simplify	Loop Sink	LoopUnroll Pass	Loop Vectorize
DF	13.14%	37.79%	40.96%	20.30%	44.52%	36.68%	43.54%	17.17%	51.10%	2.08%	39.82%	40.68%
Default	13.14%	39.24%	38.22%	19.19%	44.38%	34.97%	42.53%	17.17%	48.76%	2.08%	40.94%	37.10%
CTMark	42.34%	40.72%	44.74%	27.31%	4.52%	31.39%	42.28%	17.17%	52.56%	2.08%	46.76%	40.30%
	LowerConstant Intrinsic	LowerExpect Intrinsic	Mem2Reg	MemCpy Optimizer	Merge ICmps	Partially InlineLibCalls	Reassociate	SCCP	SLP Vectorizer	TailRecursion Elimination	Vector Combine	
DF	53.45%	39.69%	42.62%	43.33%	42.01%	19.10%	41.22%	51.82%	46.52%	45.65%	47.07%	
Default	53.45%	36.19%	42.62%	41.00%	26.21%	19.10%	39.36%	49.44%	23.67%	43.93%	36.10%	
CTMark	41.38%	15.95%	16.39%	31.00%	11.15%	24.72%	30.10%	45.67%	48.39%	48.42%	34.39%	

Table 5.2: Comparing coverage of the directed fuzzing approach (DF), libFuzzer default scheduler (Default), and CTMark. The number of snippets per optimization is 80 and we fuzzed 1 h/opt. DF has in 35 of 59 cases strictly higher coverage than CTMark and in 4 the same 🎯.

faster (median out of 31 runs in both cases).

Optimization	Default (median)	Directed Fuzzing (median)	p -Value
SLPVectorizer	39.57%	41.15%	0.0297
Reassociate	29.15%	32.06%	0.1481
VectorCombine	27.32%	24.39%	0.3766
InstCombineCompares	18.84%	19.13%	0.3381
MergeICmps	34.20%	38.10%	0.1043

Table 5.3: Coverage median out of 5 runs ($n = 80$) and p -value for Mann-Whitney-U-Test with alternative hypothesis: the directed fuzzing approach is better.

Table 5.3 compares the default and the directed scheduler on the same subset of optimizations on which we evaluated the different choices of n . This test is run without turbo boost and on a single core. When comparing these runs with the ones from Table 5.2 where turbo boost was enabled and 16 cores were used, it is immediately clear that the additional computing power helped in discovering more interesting inputs. This is true for both schedulers. With exception to the *SLPVectorizer* result of the default scheduler the multi-core results were better. Even though, the directed scheduler outperformed the default one in the *one-run-sample* evaluation in most cases, we could only show for the *SLPVectorizer* optimization that the directed scheduler is significantly better than the default. It is likely that the non-significance is caused by the small sample size, given that the median coverage of the directed fuzzing is better in 4 out of these 5 optimizations and given the results from Table 5.2. Due to time limitations we, unfortunately, did not have time to perform more repetitions, thus this is left for future work.

Figures 5.1, 5.2, 5.3, 5.4, and 5.5 depict the coverage discovery of the two different schedulers of the results in Table 5.3. The most edges are discovered in the first 15 min for both schedulers. Afterwards only smaller coverage gains happen. It appears that those gains are a bit more frequent for the directed fuzzing approach, but conclusive interpretations are difficult. More data, especially over a longer fuzzing campaign, is needed to interpret the different coverage discovery pattern.

5.2 Case Study: Finding Bugs

We evaluate the effectiveness of the snippets in two use-cases. Firstly on LLVM itself and secondly on RetroWrite [41] a state-of-the-art binary rewriting tool. Since the snippets contain arbitrary code, we need a method to semantically compare two differently built versions. While multiple papers have been published covering different approaches for semantical comparison

5.2. Case Study: Finding Bugs

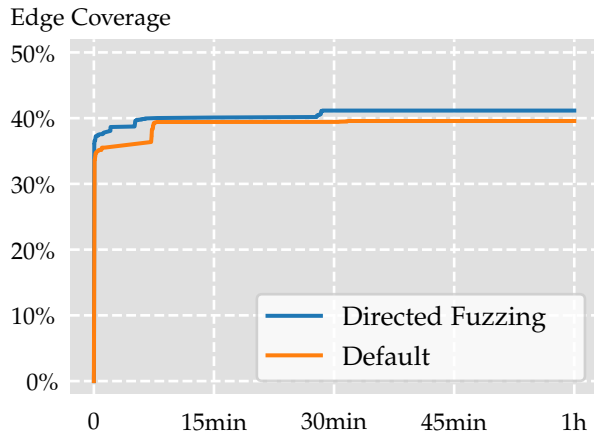


Figure 5.1: SLPVectorizer median of 5 runs.

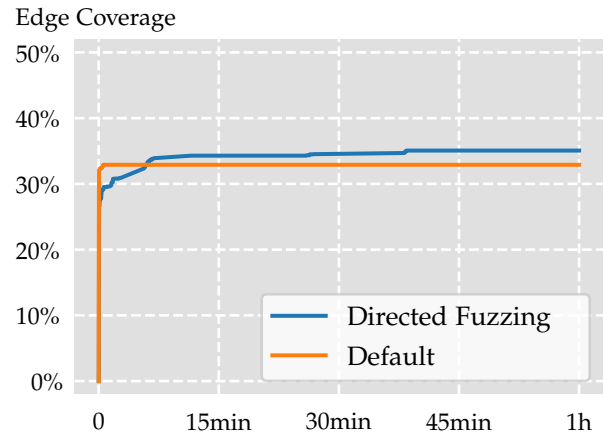


Figure 5.2: Reassociate median of 5 runs.

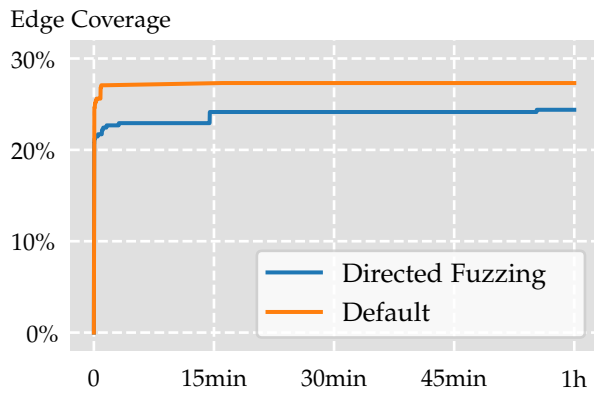


Figure 5.3: VectorCombine median of 5 runs.

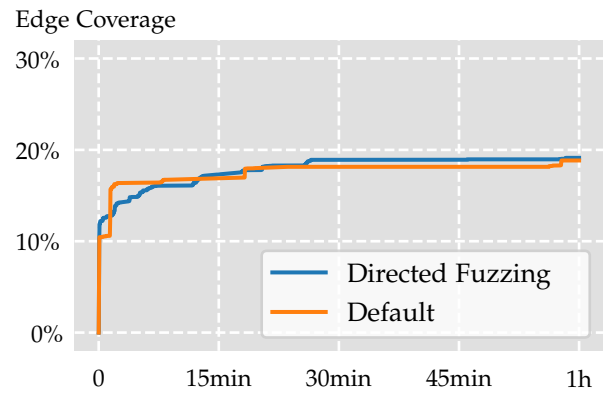


Figure 5.4: InstCombineCompares median of 5 runs.

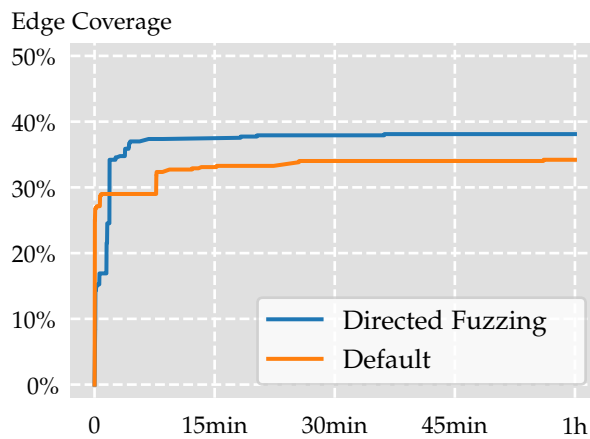


Figure 5.5: MergeCmps median of 5 runs.

(see Section 2.5), we could not find a tool which fit our needs. Therefore, we implemented a script that approaches semantic equivalence similarly to Tinbergen [76] or EMI [66] in 696 lines of Python. We execute each function in every snippet multiple times with random input and recover the output (return value(s), timeout, and exit code). We do this for two versions and verify that they behave the same with respect to the same input. If a tested function calls an undefined function, i.e., one that is only declared, we replace the called function in advance with one that is side effect free and that returns a constant. However, our test script has only support for a limited set of types. For instance, it does not support structs at the moment, which limits the detection scope. During the evaluation additional shortcomings of the script surfaced, which we could not address as part of the thesis, as we did not allocate time for this.

We use the corpus from Table 5.2 which was generated with directed fuzzing. Although, we generated the whole corpus only once, we believe it still highlights the advantages and disadvantages of the generated corpus. Additionally, since we optimize the code coverage, one could argue that one would always use the best result.

5.2.1 LLVM

We found 4 unreported bugs in LLVM that were detected during the snippet creation phase, while fuzzing. All four bugs result in compiler crashes (see Appendix A for submitted bug reports). These kinds of bugs, while annoying, are not as bad as when the generated machine code semantically differs from the source IR code, i.e., when the compiler miscompiles. To investigate miscompilation bugs, we applied the aforementioned script which calls each function of the corpus with multiple random arguments, once on a version compiled with `-O0` and once on a version compiled with `-O2`. The `-O0` acts as truth, since miscompilation bugs in `-O0` are rare [112]. Interestingly, some snippets crash the compiler when compiled with `-O0`, due to an aforementioned unreported bug or some other already reported bug. The bug is not triggered with `-O2` because an optimization can optimize the problematic code away.

Our semantic comparison script found around 5300 cases with mismatches between the `-O0` and `-O2` version. We inspected 100 cases, as we did not have time to go through all reported mismatches. All analyzed mismatches turned out to be false positives. We also suspect that the other results are (mostly) false positives based on the insight from the analyzed false positives. It follows a list of the causes with a short comment on how the false positive could possibly be avoided. Unfortunately, we did not have time to implement the suggestions.

- The insertion mutation inserts `load` instructions that load from random undefined addresses. Depending on the optimization level, this results in different behavior, which is perfectly legal. This could be fixed in the mutation.
- The deletion mutation can remove memory initializations. Uninitialized memory can contain arbitrary garbage, thus it is impossible to compare. This could be fixed in the mutation.
- *llvm-reduce* tends to replace some constants with `undef` which lets the compiler decide on the value. This hinders comparing of different compiled versions. This could be fixed in *llvm-reduce* itself or in the interestingness check. We experimented with a solution, but did not have time to evaluate it properly.
- LLVM IR supports floating points in the IEEE 754 format. IEEE 754 defines special values such as *NaN* (Not a Number) value. However, the propagation of *NaN* is not well-defined, thus depending on the optimization, it is perfectly valid to output *NaN* or *-NaN*. The semantic comparison script could be adapted to ignore these cases.
- LLVM IR's type system does not distinguish between signed and unsigned, instead it uses operation flags that specify whether a operation should wrap when overflowed (unsigned behavior) or not (signed behavior). The later case is undefined behavior, i.e., the compiler is free to optimize however it wishes. Therefore, our semantic comparison script should not create inputs that cause an overflow in any operation that has a *no overflow wrap* flag set. However, this is not trivial to achieve and would require a more elaborate semantic comparison program which is not in our scope. Another solution to prevent the undefined behavior is to preprocess the IR snippet and add checks before each operation which performs an operation that could lead to and overflow. However, this would impact the coverage of the optimizations.

Besides undefined integer overflow, there are a number of similar undefined behavior cases which also exist in C and resulted in false positives:

- Division by zero
- `INT_MIN` modulo `-1`
- Bit shifts by more than the bit size of the integer type

These could be solved similarly to the integer overflow.

- When a function requires a pointer as an argument, the semantic comparison script only reserves little memory for this address. However, some functions access the address at an arbitrary offset, which leads to

buffer overflows which results in undefined behavior. This is difficult to mitigate, but the script already fails with small offsets, thus the problem could be weakened by allocating a larger chunk of memory.

- In some instances a pointer is casted to an integer which is returned. Since the output is different between different compilations even when ASLR is disabled, the semantic comparison script picks it up as mismatch. The simplest solution we could think of is to enable ASLR and check if the function has different output when run multiple times

Addressing the aforementioned issues with the proposed solutions will hopefully bring down the number of false positives such that the cost of manual verification is minimized. Since, we did not manage to go through all reports, it is unclear whether the snippets could help in uncovering miscompilation bugs. However, at this moment the number of false positives is too large to successfully use the snippets to unveil miscompilation errors.

5.2.2 RetroWrite

RetroWrite is a static binary rewriting tool that leverages reassembleable assembly to instrument or modify an executable. The reassembling of the corpus led to the discovery of 2 bugs, one of which was previously unreported. Furthermore, we applied the semantic comparison script to verify that the reassembled versions behave the same as the compiled versions. Unlike in the LLVM case (Subsection 5.2.1), undefined behavior (except in the case of overflows) is not an issue, since *RetroWrite* works with the binary. This limits the number of false positives. There were 37 false positives caused by outputting an address in some form and five where the reassembled version was so much slower that it triggered the timeout meant for detecting infinity loops. 24 cases resulted in different error codes, however we did not consider this a bug as these cases are caused by undefined behavior in the binary anyway. Finally, we found one case in which the binary created by *RetroWrite* resulted in a segmentation fault while the original executable exited normally.

Originally, we also planed to perform *mutation testing* [56] with *RetroWrite* to assess the snippets expressiveness, i.e., we wanted to induce bugs into *RetroWrite* and see whether the snippets trigger those bugs. This would show for which kind of bugs the corpus is successful. Unfortunately, we did not allocate enough time for the evaluation. However, we believe that the discovered bugs in *RetroWrite* and LLVM give an idea of the capabilities.

5.2.3 Conclusion

Due to time constraints we could not conduct a full-fledged evaluation on all components. For the same reason, we could also not implement the fixes

5.2. Case Study: Finding Bugs

hinted in the evaluation. However, we are confident that with more resources most of these hurdles could be tackled.

Related Work

We start with a survey of *automated test case generation techniques* for compilers, which is followed by work about *coverage-based* and *directed fuzzers*. Finally, we discuss *test case reduction* results.

Automated test case generation for compilers is a long-standing research topic that dates back almost 60 years to Sauder [96], which discussed the efforts by the Air Force Logistics Command to create an automated test data generator for Cobol. Boujarwah and Saleh [28] presented an extensive survey in 1997 about automated test case generation for compilers. In 2020, Chen et al. [37] did a survey on compiler testing which includes recent contributions on automated test case generation. They concluded that the field of compiler testing has greatly evolved and had great impact on real-world compilers. Despite this they still see several interesting challenges, such as generalizing existing approaches or improving effectiveness and efficiency. Burgess and Saidi [30] introduced a test generator for FORTRAN compilers. The test cases are designed to contain specific sequences that compilers are known to optimize and to include self-checking code to verify the correctness of execution. In 2005, Lindig [69] generated random tests to assess the correctness of calling conventions of C programs. Complex data structures are generated, loaded with constant values and passed to function, which perform an assertion check. In 2009, Zhao et al. [116] presented an automated test case generator for embedded C++ compilers. The generation process is constraint by scripts which specify the test requirements, e.g., which optimizations should be tested. In 2011, Yang et al. [112] introduced CSmith, a randomized test-case generation tool for C programs that avoids undefined behavior with static analysis and wrapper functions to prevent unsafe arithmetic functions. Nagai et al. [83, 84, 85] presented the orange test case generator which mainly targets arithmetic expressions and covered different techniques to handle undefined behavior in C programs. The first version from 2012 simply discarded expressions with undefined behavior. The

second version from a year later added constants to turn undefined behavior into well-behaved versions. The third paper from 2013 introduced operation flipping, a technique which replaces operations leading to undefined behavior with related ones that avoid undefined behavior. In 2017 Bastani et al. [23] created GLADE, a tool which uses black-box access to a parser and a set of code examples to infer a target grammar from which tests are generated. In 2020, Livinskii et al. [72] showcased YARPGen, random code generator for C and C++ which builds on the ideas of orange, but supports many additional language features (structures, arrays, pointers. . .) without causing undefined behavior. Furthermore, YARPGen applies generation policies for systematically altering the character of the emitted code. In comparison to all the aforementioned work, we aimed to create a corpus that maximizes the coverage of the compiler instead of having a fast method to generate test cases. Additionally, we work on the IR level of the compiler, whereas previous work focused on generating code for the front-end of the compiler.

Fuzzing, which goes back to Miller et al. [78] in the 1990, has seen a surge in research results in the recent years. Li et al. [68] did a survey in 2018 about state-of-the-art fuzzing techniques, which is complemented by another excellent survey by Manès et al. [75] from 2019. We highlight some influential pieces of work related to coverage-guided and directed fuzzing. In 2014, Rebert et al. [92] showed the importance of a good seed corpus. Bogner [25] introduced a fuzzer for LLVM at the EuroLLVM conference in 2017, on which we built on. In the same year, Böhme et al. [27] modeled coverage-based grey-box fuzzing as Markov chains. This inspired Böhme et al. [26] to directed grey-box fuzzing, which directs coverage to certain code locations. Peng et al. [90] introduced T-Fuzz to break the coverage wall by mutating the PUT. Chen and Chen [38] presented Angora, which proposed a taint-based solution to mitigate the same issue. A year later, Chen et al. [35] improved on the previous directed grey-box fuzzing result of Böhme et al. with seed prioritization, another mutation strategy, and a modified distance metric. Gan et al. [48] refined the coverage accuracy by utilizing a form of path coverage. In 2019, Wang et al. [105] introduced another approximation for path coverage. Wang et al. [106] developed Superion, a grammar-aware grey-box fuzzer which performs mutations on the AST. In 2020, Wüstholtz and Christakis [111] introduced static lookahead analysis, a new approach which improves directed fuzzing. They evaluated their system in state-of-the-art smart contract fuzzer. Österlund et al. [88] developed a directed fuzzer called ParmeSan which builds on Angora. ParmeSan calculates the distance dynamically unlike AFLGo [26] and Hawkeye [35] to account for indirect calls. Furthermore, the targets are derived from the sanitizer.

Caron and Darnell [34] developed bugfind which reduces compiler bug triggering source files into one file. In 1994, Whalley [109] presented a tool called vpoiso which narrows miscompilation down to a single function. However,

the tools main objective was to synthesize the bug inducing transformation. In 2002, Zeller and Hildebrandt [114] introduced the first generic solution with “lexical” delta debugging. McPeak and Wilkerson [77] developed a tool with line-based delta debugging. Mishserghi and Su [81] examined delta debugging on a structural level of an AST in 2006. Herfert et al. [53] generalized this idea to arbitrary patterns. Sun et al. [100] implemented the perses tool. Perses takes grammars in Backus-Naur form to perform advanced grammar specific transformations. Regehr et al. [93] developed C-Reduce which targets the C language and has language specific transformation. Additionally, C-Reduce avoids the introduction of undefined behavior during the reduction process.

Future Work

Since this is a big project, there are numerous possibilities how this project can be improved or extended. We summarize some suggestions we made previously and cover additional ideas in this chapter.

Coverage

We argued that edge coverage should be a good indicator for the expressiveness and diversity of a snippet when compiled. However, it is unclear how other metrics behave in the setting of compilers. In vulnerability research, the use of different metrics has led to the discovery of more vulnerabilities. Consequently, one could assume that the use of another metric results in a better and more diverse corpus. Therefore, it would be interesting to compare different metrics in the setting of compilers and evaluate with which metric a better corpus is generated. This could be done by applying mutation testing and comparing which corpus leads to the detection of the most bugs.

Mutation

The evaluation showed that certain instruction insertions and deletions can cause undefined behavior which lead to many false positives. This could be addressed in the mutation process. Additionally, other mutations can be introduced. As seen in the evaluation, certain optimizations are not triggered by the current mutations at all. These optimizations most likely require more specific mutations, as for instance the optimizations that target specific library calls. Another idea is to create IR mutations that do not change the semantic meaning of the program. This would facilitate verifying the correctness, but comes with other challenges. In particular triggering a wide set of optimizations will be difficult, because the mutation must act like an optimization, just inverted.

Directed Fuzzing

The snippet generation process builds on the directed fuzzing ideas of AFLGo. We used AFLGo because of its reported effectiveness and because it is open source. However, one weakness of AFLGo is its reliance on a static CG, which does not cover indirect calls in the distance calculation. Österlund et al. [88] recently presented a tool which efficiently updates the distance dynamically to accommodate for this distortion. Improving this aspect in the snippet generation process will probably speed up the coverage discovery, as LLVM is developed in C++ which heavily relies on indirect calls for virtual function invocations.

Generation-based approach

There are generation-based fuzzing approaches on the source code level for finding compiler bugs, but only a limited amount of work has been done on generation-based IR fuzzing. It would be interesting to see how the generation-based approach would impact our work and what kind of coverage could be achieved.

Semantic Comparison

While developing semantic comparison techniques was not a focus of this thesis, it is an essential part in the application of the snippets. Unfortunately, there is hardly any reliable tooling that allows efficient semantic comparison. For binaries, *Tinbergen* [76] achieves a 0.779 average F-Score when evaluated on *coreutils* and is 25% - 53% more accurate as previous state-of-the-art tools. However, for use-cases where the source code is available existing solutions rely too much on syntactic information and the solutions that do not rely on the syntax fall short in covering undefined behavior of languages.

Use-cases of snippets

Besides evaluating other binary analysis tools, one could try to train some machine learning model with the snippets to lift assembly to LLVM IR. In that case, one might have to maximize the coverage of the code generation part in the compiler back-end instead of the mid-end optimizations, but it is certainly an interesting direction to investigate and machine learning has been applied to many translation tasks before with success.

Conclusion

In summary, we introduced a novel tool to generate an IR corpus, which can help in discovering bugs in compilers and binary tools. Our tool leverages directed mutational fuzzing to maximize the edge coverage of each optimization separately and applies test case as well as test suite reduction techniques to minimize the outputted corpus. Since we target all optimizations, we end up with a diverse corpus. This is in contrast to previous research, which focused on testing specific components of a compiler, or used whole projects to assess binary tools. We found 4 previously unreported bugs in LLVM and 2 in RetroWrite. Furthermore, our evaluation shows that additional research on semantic comparison of source code is necessary, as current solutions are lacking in terms of accuracy. They do, in particular, not handle undefined behavior well. Therefore, most code generation tools try to avoid generating code that exhibits undefined behavior and limit the scope to a subset of optimizations.

Appendix

A.1 Open Source Contributions

We list all contributions in form of commits and bug reports to existing open source projects here.

A.1.1 AFLGo

- Improved distance computation algorithmically, through implementing it in C++, and parallelization (<https://github.com/aflgo/aflgo/pull/83>).
- Ported AFLGo to LLVM 11 (<https://github.com/aflgo/aflgo/pull/85>).
- Fixed discrepancy between paper and implementation (<https://github.com/aflgo/aflgo/pull/95>).

A.1.2 LLVM

- Bug Reports
 - crash: https://bugs.llvm.org/show_bug.cgi?id=48836
 - crash: https://bugs.llvm.org/show_bug.cgi?id=50027
 - crash: https://bugs.llvm.org/show_bug.cgi?id=50029
 - crash: https://bugs.llvm.org/show_bug.cgi?id=50032
- Fixed bug in instruction deletion mutation (<https://reviews.llvm.org/D96207>)

A.1.3 RetroWrite

- Bug Reports
 - Assertion Error
(<https://github.com/HexHive/retrowrite/issues/22>)
 - Segmentation fault in reassembled binary
(<https://github.com/HexHive/retrowrite/issues/23>)

Bibliography

- [1] How to submit an LLVM bug report. <https://llvm.org/docs/HowToSubmitABug.html>. Accessed: 2021-03-21.
- [2] AddressSanitizer. <https://clang.llvm.org/docs/AddressSanitizer.html>. Accessed: 2021-04-07.
- [3] Clang is now used to build chrome for windows. <https://blog.llvm.org/2018/03/clang-is-now-used-to-build-chrome-for.html>, . Accessed: 2021-03-15.
- [4] Google chromium security. <https://www.chromium.org/Home/chromium-security/bugs#TOC-Security-fuzzing>, . Accessed: 2021-04-07.
- [5] CTMark. <https://github.com/llvm/llvm-test-suite/tree/main/CTMark>. Accessed: 2021-04-26.
- [6] Frama-C gitlab project. <https://git.frama-c.com/pub/frama-c>. Accessed: 2021-03-26.
- [7] Project Zero: Fuzzing ImageIO. <https://googleprojectzero.blogspot.com/2020/04/fuzzing-imageio.html>. Accessed: 2021-04-07.
- [8] GCC documentation: How to minimize test cases for bugs. <https://gcc.gnu.org/bugs/minimize.html>, . Accessed: 2021-03-21.
- [9] GCC testsuites. <https://gcc.gnu.org/onlinedocs/gccint/C-Tests.html>, . Accessed: 2021-03-17.
- [10] The real story behind the java 7 ga bugs affecting apache lucene / solr. <https://blog.thetaphi.de/2011/07/real-story-behind-java-7-ga-bugs.html>. Accessed: 2021-04-22.
- [11] KCC github project. <https://github.com/kframework/c-semantics>. Accessed: 2021-03-26.

-
- [12] The LLVM project. <https://llvm.org/>, . Accessed: 2021-03-15.
 - [13] Android NDK. https://developer.android.com/ndk/downloads/revision_history.html#expandable-10, . Accessed: 2021-03-15.
 - [14] Chromium uses clang. https://chromium.googlesource.com/chromium/src/+/_master/docs/clang.md, . Accessed: 2021-03-15.
 - [15] LLVM users. <https://llvm.org/Users.html>, . Accessed: 2021-03-15.
 - [16] LLVM testing infrastructure guide. <https://llvm.org/docs/TestingGuide.html>, . Accessed: 2021-03-17.
 - [17] OSS-Fuzz: continuous fuzzing for open source software. <https://github.com/google/oss-fuzz>. Accessed: 2021-04-16.
 - [18] Rust issue: Generation of unoptimizable llvm ir. <https://github.com/rust-lang/rust/issues/83623>. Accessed: 2021-04-22.
 - [19] Cisco Talos: exploitable or not exploitable. <https://blog.talosintelligence.com/2018/08/exploitable-or-not-exploitable-using.html>. Accessed: 2021-04-07.
 - [20] ThreadSanitizer. <https://clang.llvm.org/docs/ThreadSanitizer.html>. Accessed: 2021-04-07.
 - [21] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 1–10. IEEE, 2011.
 - [22] Roberto Baldoni, Emilio Coppa, Daniele Cono D’elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), May 2018. ISSN 0360-0300. doi: 10.1145/3182657. URL <https://doi.org/10.1145/3182657>.
 - [23] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing program input grammars. *ACM SIGPLAN Notices*, 52(6):95–110, 2017.
 - [24] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. In *NDSS*, volume 9, pages 8–11. Citeseer, 2009.
 - [25] Justin Bogner. EuroLLVM 2017: Adventures in fuzzing instruction selection. http://llvm.org/devmtg/2017-03/assets/slides/adventures_in_fuzzing_instruction_selection.pdf, 2017. Accessed: 2021-04-09.
 - [26] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017*

- ACM SIGSAC Conference on Computer and Communications Security, CCS '17, page 2329–2344, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349468. doi: 10.1145/3133956.3134020. URL <https://doi.org/10.1145/3133956.3134020>.
- [27] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.
- [28] Abdulazeez S Boujarwah and Kassem Saleh. Compiler test case generation methods: a survey and assessment. *Information and software technology*, 39(9):617–625, 1997.
- [29] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proceedings of the 2013 International Conference on Software Engineering, ICSE '13*, page 122–131. IEEE Press, 2013. ISBN 9781467330763.
- [30] Colin J Burgess and M Saidi. The automatic generation of test cases for optimizing fortran compilers. *Information and Software Technology*, 38(2): 111–119, 1996.
- [31] Jose Campos and Rui Abreu. Leveraging a constraint solver for minimizing test suites. In *2013 13th International Conference on Quality Software*, pages 253–259. IEEE, 2013.
- [32] Alberto Caprara, Matteo Fischetti, and Paolo Toth. A heuristic method for the set covering problem. *Operations research*, 47(5):730–743, 1999.
- [33] Ryan Carlson, Hyunsook Do, and Anne Denton. A clustering approach to improving test case prioritization: An industrial case study. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance, ICSM '11*, page 382–391, USA, 2011. IEEE Computer Society. ISBN 9781457706639. doi: 10.1109/ICSM.2011.6080805. URL <https://doi.org/10.1109/ICSM.2011.6080805>.
- [34] J. M. Caron and P. A. Darnell. Bugfind: A tool for debugging optimizing compilers. *SIGPLAN Not.*, 25(1):17–22, January 1990. ISSN 0362-1340. doi: 10.1145/74105.74106. URL <https://doi.org/10.1145/74105.74106>.
- [35] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18*, page 2095–2108, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243849. URL <https://doi.org/10.1145/3243734.3243849>.

-
- [36] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. An empirical comparison of compiler testing techniques. In *Proceedings of the 38th International Conference on Software Engineering*, pages 180–190, 2016.
 - [37] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A survey of compiler testing. *ACM Computing Surveys (CSUR)*, 53(1):1–36, 2020.
 - [38] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.
 - [39] Chris Lattner and Vikram Adve. The LLVM Instruction Set and Compilation Strategy. Tech. Report UIUCDCS-R-2002-2292, CS Dept., Univ. of Illinois at Urbana-Champaign, Aug 2002.
 - [40] Pascal Cuoq, Julien Signoles, Patrick Baudin, Richard Bonichon, Géraud Canet, Loïc Correnson, Benjamin Monate, Virgile Prevosto, and Armand Puccetti. Experience report: Ocaml for an industrial-strength static analysis framework. *SIGPLAN Not.*, 44(9):281–286, August 2009. ISSN 0362-1340. doi: 10.1145/1631687.1596591. URL <https://doi.org/10.1145/1631687.1596591>.
 - [41] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1497–1511. IEEE, 2020.
 - [42] S. Elbaum, D. Gable, and G. Rothermel. The impact of software evolution on code coverage information. In *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*, pages 170–179, 2001. doi: 10.1109/ICSM.2001.972727.
 - [43] Chucky Ellison and Grigore Rosu. An executable formal semantics of c with applications. *SIGPLAN Not.*, 47(1):533–544, January 2012. ISSN 0362-1340. doi: 10.1145/2103621.2103719. URL <https://doi.org/10.1145/2103621.2103719>.
 - [44] Thomas A Feo and Mauricio GC Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations research letters*, 8(2):67–71, 1989.
 - [45] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.
 - [46] G. Fraser and A. Gargantini. Generating minimal fault detecting test suites for boolean expressions. In *2010 Third International Conference on*

- Software Testing, Verification, and Validation Workshops*, pages 37–45. doi: 10.1109/ICSTW.2010.51.
- [47] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proceedings of the 30th International Conference on Software Engineering, ICSE '08*, page 321–330, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580791. doi: 10.1145/1368088.1368132. URL <https://doi.org/10.1145/1368088.1368132>.
- [48] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.
- [49] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *2009 IEEE 31st International Conference on Software Engineering*, pages 474–484, 2009. doi: 10.1109/ICSE.2009.5070546.
- [50] Debin Gao, Michael K Reiter, and Dawn Song. Binhunt: Automatically finding semantic differences in binary programs. In *International Conference on Information and Communications Security*, pages 238–255. Springer, 2008.
- [51] David Gelperin and Bill Hetzel. The growth of software testing. *Communications of the ACM*, 31(6):687–695, 1988.
- [52] CISQ Herb Krasner. The cost of poor software quality in the us: A 2020 report. 2020. URL <https://www.it-cisq.org/pdf/CPSQ-2020-report.pdf>.
- [53] Satia Herfert, Jibesh Patra, and Michael Pradel. Automatically reducing tree-structured test inputs. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017*, page 861–871. IEEE Press, 2017. ISBN 9781538626849.
- [54] Jiyong Jang, Maverick Woo, and David Brumley. Towards automatic software lineage inference. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 81–96, Washington, D.C., August 2013. USENIX Association. ISBN 978-1-931971-03-4. URL <https://www.usenix.org/conference/usenixsecurity13/technical-sessions/papers/jang>.
- [55] D. Jeffrey and N. Gupta. Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Transactions on Software Engineering*, 33(2):108–123, 2007. doi: 10.1109/TSE.2007.18.
- [56] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5): 649–678, 2010.

-
- [57] Lingxiao Jiang and Zhendong Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA '09*, page 81–92, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583389. doi: 10.1145/1572272.1572283. URL <https://doi.org/10.1145/1572272.1572283>.
 - [58] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105. IEEE, 2007.
 - [59] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.
 - [60] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972. ISBN 978-1-4684-2001-2. doi: 10.1007/978-1-4684-2001-2_9. URL https://doi.org/10.1007/978-1-4684-2001-2_9.
 - [61] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018.
 - [62] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *International static analysis symposium*, pages 40–56. Springer, 2001.
 - [63] Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer Publishing Company, Incorporated, 5th edition, 2012. ISBN 3642244874.
 - [64] Alexander S Kossatchev and MA Posypkin. Survey of compiler testing methods. *Programming and Computer Software*, 31(1):10–19, 2005.
 - [65] Arun Lakhotia, Mila Dalla Preda, and Roberto Giacobazzi. Fast location of similar code fragments using semantic ‘juice’. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, pages 1–6, 2013.
 - [66] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *ACM SIGPLAN Notices*, 49(6):216–226, 2014.
 - [67] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert-a formally verified

- optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.
- [68] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):1–13, 2018.
- [69] Christian Lindig. Random testing of c calling conventions. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 3–12, 2005.
- [70] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. α diff: cross-version binary code similarity detection with dnn. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 667–678, 2018.
- [71] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. Gplag: Detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '06*, page 872–881, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933395. doi: 10.1145/1150402.1150522. URL <https://doi.org/10.1145/1150402.1150522>.
- [72] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for c and c++ compilers with yarpngen. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–25, 2020.
- [73] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. *IEEE Transactions on Software Engineering*, 43(12):1157–1177, 2017.
- [74] Xue-ying Ma, Bin-kui Sheng, and Cheng-qing Ye. Test-suite reduction using genetic algorithm. In Jiannong Cao, Wolfgang Nejdl, and Ming Xu, editors, *Advanced Parallel Processing Technologies*, pages 253–262, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-32107-1.
- [75] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.
- [76] Derrick McKee, Nathan Burow, and Mathias Payer. Software ethology: An accurate, resilient, and cross-architecture binary analysis framework, 2020.
- [77] McPeak and Wilkerson. Delta tool. <https://web.archive.org/web/20200301150914/http://delta.tigris.org/>. Accessed (archieved link): 2021-03-26.

-
- [78] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, December 1990. ISSN 0001-0782. doi: 10.1145/96267.96279. URL <https://doi.org/10.1145/96267.96279>.
- [79] C. Miller. Fuzz by number: More data about fuzzing than you ever wanted to know. In *Proceedings of the CanSecWest*, 2008.
- [80] Matt Miller. Trends, challenge, and shifts in software vulnerability mitigation. https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf, 2019. Accessed: 2021-04-09.
- [81] Ghassan Misherghi and Zhendong Su. Hdd: Hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering, ICSE '06*, page 142–151, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933751. doi: 10.1145/1134285.1134307. URL <https://doi.org/10.1145/1134285.1134307>.
- [82] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *TACAS'08/ETAPS'08 Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, pages 337–340, 2008.
- [83] Eriko Nagai, Hironobu Awazu, Nagisa Ishiura, and Naoya Takeda. Random testing of c compilers targeting arithmetic optimization. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012)*, pages 48–53, 2012.
- [84] Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. Scaling up size and number of expressions in random testing of arithmetic optimization of c compilers. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2013)*, pages 88–93, 2013.
- [85] 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 LSI Design Methodology*, 7:91–100, 2014.
- [86] NIST U.S Department of Commerce. The economic impacts of inadequate infrastructure for software testing. 2002. URL <https://www.nist.gov/system/files/documents/director/planning/report02-3.pdf>.
- [87] A. Jefferson Offutt, Jie Pan, and Jeffrey M. Voas. Procedures for reducing the size of coverage-based test sets. In *In Proc. Twelfth Int'l. Conf. Testing Computer Softw*, pages 111–123, 1995.

-
- [88] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Parmesan: Sanitizer-guided greybox fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2289–2306, 2020.
 - [89] Pádraig O’sullivan, Kapil Anand, Aparna Kotha, Matthew Smithson, Rajeev Barua, and Angelos D Keromytis. Retrofitting security in cots software with binary rewriting. In *IFIP International Information Security Conference*, pages 154–172. Springer, 2011.
 - [90] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: Fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.
 - [91] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy*, pages 709–724. IEEE, 2015.
 - [92] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 861–875. USENIX Association, 2014.
 - [93] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for c compiler bugs. *SIGPLAN Not.*, 47(6):335–346, June 2012. ISSN 0362-1340. doi: 10.1145/2345156.2254104. URL <https://doi.org/10.1145/2345156.2254104>.
 - [94] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 34–43, 1998. doi: 10.1109/ICSM.1998.738487.
 - [95] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. Detecting code clones in binary executables. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA ’09*, page 117–128, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583389. doi: 10.1145/1572272.1572287. URL <https://doi.org/10.1145/1572272.1572287>.
 - [96] Richard L. Sauder. A general test data generator for cobol. In *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference, AIEE-IRE ’62 (Spring)*, page 317–323, New York, NY, USA, 1962. Association for Computing Machinery. ISBN 9781450378758. doi: 10.1145/1460833.1460869. URL <https://doi.org/10.1145/1460833.1460869>.
 - [97] August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. Balancing trade-offs in test-suite reduction. In *Proceedings*

- of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, page 246–256, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330565. doi: 10.1145/2635868.2635921. URL <https://doi.org/10.1145/2635868.2635921>.
- [98] Shilpi Singh and Raj Shree. A combined approach to optimize the test suite size in regression testing. *CSI Transactions on ICT*, 4(2):73–78, Dec 2016. ISSN 2277-9086. doi: 10.1007/s40012-016-0098-8. URL <https://doi.org/10.1007/s40012-016-0098-8>.
- [99] 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.
- [100] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. Perses: Syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering, ICSE '18*, page 361–371, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356381. doi: 10.1145/3180155.3180236. URL <https://doi.org/10.1145/3180155.3180236>.
- [101] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, 2013. doi: 10.1109/SP.2013.13.
- [102] Yixuan Tang, Zhilei Ren, Weiqiang Kong, and He Jiang. Compiler testing: a systematic literature analysis. *Frontiers of Computer Science*, 14(1):1–20, 2020.
- [103] Ramtine Tofighi-Shirazi, Maria Christofi, Philippe Elbaz-Vincent, and Thanh-ha Le. Dose: Deobfuscation based on semantic equivalence. In *Proceedings of the 8th Software Security, Protection, and Reverse Engineering Workshop, SSPREW-8*, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450360968. doi: 10.1145/3289239.3289243. URL <https://doi.org/10.1145/3289239.3289243>.
- [104] Francis J Vasko. An efficient heuristic for large set covering problems. *Naval Research Logistics Quarterly*, 31(1):163–171, 1984.
- [105] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 1–15, 2019.
- [106] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 724–735. IEEE, 2019.

-
- [107] Shuai Wang and Dinghao Wu. In-memory fuzzing for binary code similarity analysis. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 319–330. IEEE, 2017.
- [108] T. Wang, T. Wei, G. Gu, and W. Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*, pages 497–512, 2010. doi: 10.1109/SP.2010.37.
- [109] David B. Whalley. Automatic isolation of compiler errors. *ACM Trans. Program. Lang. Syst.*, 16(5):1648–1659, September 1994. ISSN 0164-0925. doi: 10.1145/186025.186103. URL <https://doi.org/10.1145/186025.186103>.
- [110] W. Eric Wong, Joseph R. Horgan, Saul London, and Aditya P. Mathur. Effect of test set minimization on fault detection effectiveness. In *Proceedings of the 17th International Conference on Software Engineering, ICSE '95*, page 41–50, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897917081. doi: 10.1145/225014.225018. URL <https://doi.org/10.1145/225014.225018>.
- [111] Valentin Wüstholtz and Maria Christakis. Targeted greybox fuzzing with static lookahead analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE '20*, page 789–800, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371216. doi: 10.1145/3377811.3380388. URL <https://doi.org/10.1145/3377811.3380388>.
- [112] 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.
- [113] Shin Yoo and Mark Harman. Pareto efficient multi-objective test case selection. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis, ISSTA '07*, page 140–150, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595937346. doi: 10.1145/1273463.1273483. URL <https://doi.org/10.1145/1273463.1273483>.
- [114] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002. doi: 10.1109/32.988498.
- [115] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In Oscar Nierstrasz and Michel Lemoine, editors, *Software Engineering — ESEC/FSE '99*, pages 253–267, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-48166-9.

- [116] Chen Zhao, Yunzhi Xue, Qiuming Tao, Liang Guo, and Zhaohui Wang. Automated test program generation for an industrial optimizing compiler. In *2009 ICSE Workshop on Automation of Software Test*, pages 36–43. IEEE, 2009.



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

COMPLETE COMPILER CODE CREATION

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):

REIFF

First name(s):

LORIS

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the '[Citation etiquette](#)' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

2. May 2021, Zürich

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.