Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

# CompFuzzCI: Continuous integration fuzz testing for the Dafny compiler

*Author:*
Karnbongkot Boonriong

*Supervisor:*
Prof. Alastair Donaldson
*Second Marker:*
Prof. Cristian Cadar

Submitted in partial fulfillment of the requirements for the MSc degree in Computing (Security and Reliability) of Imperial College London

September 2024

## Abstract

Compiler fuzzing has been a topic of interest for researchers in recent years. Despite the success of compiler fuzzers in finding previously unknown compiler bugs, it is not regularly used in the industry. In this project, we explore the integration of compiler fuzzing into the continuous integration (CI) pipeline as a solution to regularise compiler fuzzing in the industry. We chose the Dafny language compiler as the target for this research.

We developed CompFuzzCI, a framework to orchestrate fuzzing, test case reduction, result de-duplication, bisection, bug reporting, and bug tracking as part of the compiler's continuous integration. CompFuzzCI brings together CI practice, cloud computing, and existing fuzzers and reducers to provide a generalisable architecture for the integration of continuous fuzz testing in compiler development.

CompFuzzCI has been evaluated in the real world and in simulation. We integrated CompFuzzCI into the Dafny compiler GitHub repository for 50 days, observing the impact and user experience of the integration. During this time, we surfaced potential problems and challenges that may arise for any CI integration of compiler fuzzing. We also performed a simulation by running CompFuzzCI on the changes which introduced known bugs to the Dafny compiler. We evaluated the effectiveness of CompFuzzCI in finding these bugs and the coverage with respect to these changes.

CompFuzzCI has demonstrated the potential of integrating compiler fuzzing into the CI/CD pipeline. We have shown that compiler fuzzing can be fully automated on the CI and can find bugs that are not found by the existing CI test suite. More importantly, we have surfaced areas of improvement for this newly explored problem space and provided insights for future integration of compiler fuzzing into the CI/CD pipeline.

## Acknowledgments

Firstly, I thank my supervisor, Prof. Alastair Donaldson, for his guidance and support throughout this project. His patient approach to problem-solving and encouragement have been extremely helpful to this project and my mental health. His efforts to connect me with others have provided me with a lot of inspiration. His feedback has also improved the quality of this report by a lot.

I would also like to thank my second marker, Prof. Cristian Cadar, for his feedback and suggestions on the background chapter and the project evaluation. His teaching on the software reliability course is what inspired me to take on this project; his enthusiasm for the subject is infectious.

Thank you to Arindam and Amber for showing me different ways of thinking and their help designing and evaluating the project.

A basket of thanks to the Dafny developers from Amazon. Stefan: for his active and consistent feedback and help in integrating CompFuzzCI into the Dafny compiler. Mikael: for his tolerance and patience in handling and clarifying the bugs found by CompFuzzCI. Robin: for many of his suggestions and ideas from the start to the end of this project. Fabio and Remy: for their interest and curiosity, which boosted my motivation to make this project work.

To my family, partner, security brothers, and friends, thank you for your support, love and care. You have been my rock throughout this challenging year. I could not have gone through it without you.

# Contents

# Chapter 1

# Introduction

## 1.1 Motivation

The compiler is a crucial component of software development. Any compiled language must be processed by a compiler before being executed. Compilers are responsible for generating semantically equivalent code, optimizing correctly, and warning the developer of any errors encountered during compilation. This makes any bugs in the compiler critical, as they may lead to incorrectness in all compiled programs that trigger the bug. For instance, a study by LI et al. [1] found 21 bugs in the LLVM compiler that had already been released for 2-8 years, somehow escaping all previous bug-finding efforts until it was found using a newly developed compiler fuzzing technique. It is crucial to perform rigorous testing to find and fix any bugs before they make their way into a release.

Compiler testing, particularly compiler fuzzing, has recently been gaining more interest from researchers, resulting in numerous excellent fuzzers such as CSmith [2], AFL [3], YARPGen [4], and more recently: glslsmith [5], fuzz-d [6], RustSmith [7]. The strength of fuzzers lies in their ability to extensively test software with distinct test cases at minimal time, finding bugs very quickly once one has been introduced. Many compiler bugs have been found and fixed by using these tools.

Unfortunately, while compilers are continuously evolving, most of the tools are used only occasionally during testing campaigns. One consequence of this usage pattern is that in the window of time between one testing campaign and the next, bugs that are most interesting to developers, such as frequently occurring bugs, or new bugs in the area currently under development, have already been found. Compiler fuzzers are left to find old, corner-case bugs that are often put on low priority by developers [8]. This is also evident by the trend of fuzzer-found bugs being ignored and left to live in later software releases [9].

Continuous Integration (CI) is a popular development practice that enforces automated building, testing and deployment of software at every small code change. Automated tests in the CI pipeline often include integration tests, unit tests and CI tests. However, fuzzing has not been routinely incorporated into the CI pipeline, leading to its exclusion from many software development processes. Several factors complicate the integration of fuzzing into the development workflow, which will be discussed in section 2.9.

It becomes apparent that there is a significant gap between compiler fuzzing research and its use in the industry. To explore the problem, we develop a framework, “CompFuzzCI”, to orchestrate fuzzing, test case reduction, result de-duplication, bisection, bug reporting, and bug tracking as part of the compiler’s continuous integration. This framework will allow us to evaluate the effectiveness of compiler fuzzing in the CI and any challenges that may arise from this integration. The Dafny language compiler has been chosen as the target for this research, as it is a relatively new open-source language that is being developed by a small team of developers following CI practice.

## 1.2 Objectives

This project aims to answer the following research questions:
**RQ1** How effective is CI-integrated compiler fuzzing at discovering bugs during development?
**RQ2** What is the developer experience of having compiler fuzzing in the CI pipeline?
**RQ3** How could bug deduplication be done effectively?
**RQ4** What are the challenges of automating bug bisection?
**RQ5** Is the benefit of integrating compiler fuzzing into the CI worth the cost?

## 1.3 Contribution

The following contributions let us answer the research questions above:

1. Design and implement a generalisable architecture for the integration of fuzz testing using black-box fuzzers in compiler development by utilising GitHub continuous integration, multiple AWS cloud services, and existing fuzzers and reducers.

2. Experiment with different strategies for identifying and deduplicating a bug to find a strategy which balances accuracy and cost.

3. Experiment with different strategies of bisecting a bug to optimise for speed. Surface and address the challenges of bisecting a bug in a constantly changing compiler codebase.

4. Develop a system of bug tracking for the compiler fuzzer, which can be used as an interestingness test for the fuzzer and can provide developers with useful information about the bug.

5. Integrate CompFuzzCI into the Dafny compiler GitHub repository for 50 days. Observe the impact and user experience of the integration. Surface realistic problems and challenges that may arise for any CI integration of compiler fuzzing into a relatively new language.

6. Perform simulation by running CompFuzzCI on the changes which introduced known bugs to the Dafny compiler. Evaluate the effectiveness of CompFuzzCI in finding these bugs.

7. Conduct a cost-benefit analysis of using CompFuzzCI in the Dafny compiler development. Provide a guideline for future integration of compiler fuzzing into the CI/CD pipeline.

8. All source code for MutateCSharp are publicly available at:
   `https://GitHub.com/CompFuzzCI/DafnyCompilerFuzzer`

# Chapter 2

# Background

This chapter provides the background knowledge needed to understand the design and evaluation of this project. In section 2.1, we will briefly discuss the Dafny language, focusing on the engineering and development of the Dafny compiler. Section 2.2 and section 2.3 build on top of each other to give a background on compiler fuzzing, a core component of this project. In section 2.4, we talk about test program reduction, why it is needed, and how it could be done. Section 2.5 and section 2.6, review the state of the art of bug deduplication and bug triage. Section 2.7 will walk through the software development concept used in this project. Building on top of everything we learned so far, section 2.8 explores the state-of-the-art tools for continuous integration fuzzing. Finally, in section 2.9, we end the chapter by posing important questions that must be answered before attempting continuous integration fuzzing.

## 2.1 The Dafny Language: Overview

Dafny was created in 2009 by Rustan Leino while working at Microsoft Research. It is a verification-aware language, meaning that it requires verification along with code development. The general proof framework Dafny uses is Hoare logic; the Z3 automated theorem prover [10] is the underlying tool that helps Dafny verify the specifications written by the user. A piece of code written in Dafny is, therefore, "Correct by Construction". The Dafny language is being used at Amazon Web Services (AWS) to develop encryption tools.

Dafny is a high-level language which compiles to other high-level languages, including C#, Java, JavaScript, Go, Python, and Rust (under development). The Dafny language features are similar to the languages it compiles to, with the addition of separation logic-inspired structures such as inductive datatypes and a variation of separation logic known as implicit dynamic frames. Because Dafny promises correctness by construction, it is important that the language implementation itself is correct. A bug in the verifier or in the compiler can make code that is guaranteed to behave correctly behave unexpectedly instead.

The Dafny compiler works by first processing the Dafny code into an abstract syntax tree (AST); this process involves syntactic, semantic and type checking. The AST is used for both verifying and compiling. For verification, the internal AST is converted to Boogie AST, a verification language compatible with Z3. For compilation, the AST is pushed through a single-pass compiler, which transforms it into an internal AST, which can then be converted to the compilation target AST located in the target language backend. The final step is to “pretty print” the target AST by converting its construct into the corresponding representation in that language.

Dafny is being developed as an open-source language on GitHub [11] by a team of developers from AWS. The development process follows the CI/CD practice; the developers of Dafny has created a CI test suite, which continues to grow as the language expands and bugs are discovered.

## 2.2 Compiler testing

Compilers are software that translates code written in the source language into the target language, often from a high-level language to a low-level language such as assembly language or machine code. The typical compiling process includes interaction between several components to preprocess the code, perform lexical analysis, syntactic analysis, semantic analysis, optimising the code, and generating code in the target language. For modern-day compilers, the optimisation step can be inserted at many points in the compilation process. There can also be multiple levels of optimisation. The compiler can also provide support for multiple versions of the language and multiple target platforms. The number of components in a compiler and their interaction makes a compiler a very complex piece of software.

A bug in any compiler component can cause a correct program to produce unexpected results. These results can be classified into three categories:

- **Compiler crash:** the compiler terminates prematurely without producing a valid output. Crashes typically return with some form of error traces, making this type of bug the most straightforward to investigate.

- **Noncompliance generation:** the compiler executes successfully. However, the generated code does not comply with the target language specification. For example, a bug in the Dafny language compiler resulted in the generation of invalid Java code, which raises errors when compiled further by the Java compiler [12]. The root cause for this type of bug can be found by tracing the target language errors back to the compiler code responsible for generating them.

- **Miscompilation:** the compiler executes successfully and generates language compliance code. However, the generated program is semantically different from the source program. This result is likely to occur due to the transformation of the source program during the optimisation phase. The behaviour is silent, making it difficult to identify the root cause.

The input space for a compiler is as large as its source language expressiveness would allow for. This makes it impossible to manually test all possible compiler executions to find bugs. Currently, we know two alternative ways to ensure compiler correctness without going through all possible execution paths: formal verification and fuzzing.

**Formal verification** uses mathematical proofs to ensure the program is correct with respect to its formal specifications. Proving program correctness can be tedious and error-prone if done by hand. Assistant prover such as Coq [13] can be used to help mechanically check the correctness of the proof. Such tools and techniques are used by the developer of CompCert [14], a verified compiler for Clight (a subset of C language recommended for critical embedded software). CompCert is mathematically guaranteed to generate correct, semantically equivalent code. Even though formal verification makes correctness checking for compiler possible, it still requires a lot of manual effort and is very time-consuming. For this reason, a full formal verification of a system is often avoided and is only recommended for safety-critical systems.

**Fuzzing** is an automated randomised testing technique which injects randomised inputs to the software under test (SUT). This is done to observe the behaviour of the software under these inputs, which can help determine the correctness of software implementation. This technique relies on the randomised nature of generated input to cover as many execution scenarios as possible. Fuzzers can be white box, grey box, or black box. The white-box fuzzer has access to the source code of the SUT, the black-box has no information on the workings of the SUT, and the grey-box fuzzer is somewhere in between. Fuzzing works very well for large, complex programs such as compilers. It is much more efficient and scalable than formal verification and, thus, is preferred and recommended for all software.

## 2.3 Compiler fuzzing

The process of fuzzing a compiler for any language is generally as shown below.

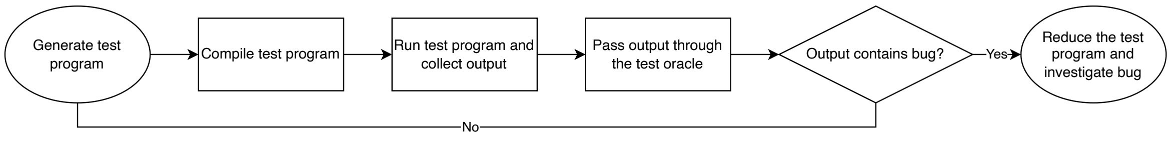

Generate test program
Compile test program
Run test program and collect output
Pass output through the test oracle
Output contains bug?
Yes
Reduce the test program and investigate bug
No

**Figure 2.1:** The general workflow of fuzzing a compiler

The process begins with the generation of the test program. This program in the compiler's source language could be invalid, malformed, unexpected or perfectly valid, depending on the generation strategy of the fuzzer. After generation, the test program is compiled and executed, and its output is captured throughout. The captured output is then analysed to determine if it signifies a bug. This step requires the use of an “oracle”, described in a later subsection. If the output contains a bug, its corresponding test program could be of interest. It is then highly recommended that test program reductions be performed to aid the compiler developer’s investigation once the bug is reported.

## Program generation

The test programs for compilers are programs in the compiler’s source language. As mentioned above, test programs can be used at any level of validation according to the language specification. For compiler testing, it is preferred that the test program pass the compilation step. Thus, invalid and malformed input is not interesting as it will likely be stopped at syntax or semantics analysis. The preferred type of test programs are valid ones that utilise the language in an “interesting” way. Thus, care is needed to ensure the generated program is diverse in the language features and their combinations. Another concern is the termination of the program. It is undesirable to generate programs that terminate prematurely or contain exceptions, as these do not give valuable insight to the compiler.
In general, there are two approaches to program generation:

### Generative program generation

This approach generates the test programs from scratch with the help of language grammar. In a grammar-directed approach, the fuzzer receives context-free grammar (CFG) as input. A syntactically valid program can be generated by traversing the CFG top-down. The resulting program can still be semantically invalid and will likely only be useful for testing the syntax analysis code, as these semantically invalid programs will terminate at the semantic analysis phase.

Another more successful approach is grammar-aided. This approach is used by tools such as CSmith [2] and YARPGen [4]. In this approach, the grammar is being used along with different heuristics. For example, CSmith uses a generation-time check to ensure no dynamic allocation or pointer arithmetic is allowed. Additionally, it uses run-time checks to wrap potentially overflowing operations.

### Mutational program generation

This approach generates programs by applying multiple code mutations to an existing program to produce a new, mutated program. This can be done both in semantic preserving and non-semantic preserving ways. Semantic preserving mutations are based on the idea of equivalence modulo input (EMI) [15]. In EMI, two programs are equivalent under a set of inputs if they exhibit the same behaviour and output under these inputs. Most generation techniques that follow the EMI approach will involve mutations within the input-dead code, which is part of the program that is unexecuted and unaffected by the input. An example of a tool which uses this idea is GLFuzz [16]. It creates a new program by removing input-dead code and generating new code in the input-dead region, combining the power of generation and mutation. On the other hand, the result of non-semantic preserving mutation does not require the same semantics as the input. Nagai et al. [17] uses non-semantic preserving mutation to flip the arithmetic operators that would have caused undefined behaviour.

### Validity of test programs

We have established that the ideal test program is free of syntax and semantic errors, as these could cause compile-time errors. However, another critical factor we need to prevent is undefined behaviour which could result in a run-time error. Undefined behaviour is any behaviour that is not defined by the language specification. Some examples of undefined behaviour are dereferencing a null pointer or decision by zero in C language. When undefined behaviour is present, there are no expectations of what the program might do, devaluing any bugs that might have been found.

With these constraints in mind, researchers have used many approaches to ensure the validity of the generated programs. The simplest approach is to avoid any language structure that might introduce undefined behaviour. However, this approach limits the diversity and expressiveness of the generated program. Alternatively, CSmith [2] uses dynamic checks to wrap any expressions that could generate undefined behaviours. This approach still limits the bug-finding capability of the tools, as evidenced by Even-Mendoza et al. [18] in their study to relax the constraint of CSmith. An approach we will focus on in this study is Program reconditioning by Lecoeur et al. [5]. Program reconditioning decouples validity checking from program generation. The program will first be generated without validity constraints, giving it more expressiveness, and then it will pass through the reconditioning phase, which traverses and transforms the program to avoid invalid code.

## Test oracles

The concept of a test oracle is introduced by Howden [19] as a way to determine if the test output is correct. In manual testing, this oracle can be the developer who pre-determines the output of the test case. In automated testing, determining the correct output for a given input is a hard problem known as the “test oracle problem”. There are two ways to solve this problem for automated compiler testing: differential testing and metamorphic testing.

### Differential testing

The concept of differential testing is proposed by McKeeman [20]. It involves passing the same input through multiple, comparable versions of the program under test, checking for any unexpected differences in the program outputs. If one or more versions of the program exhibit different output, then bugs may be present in some versions of the program. For compiler testing, this means compiling the same pro-

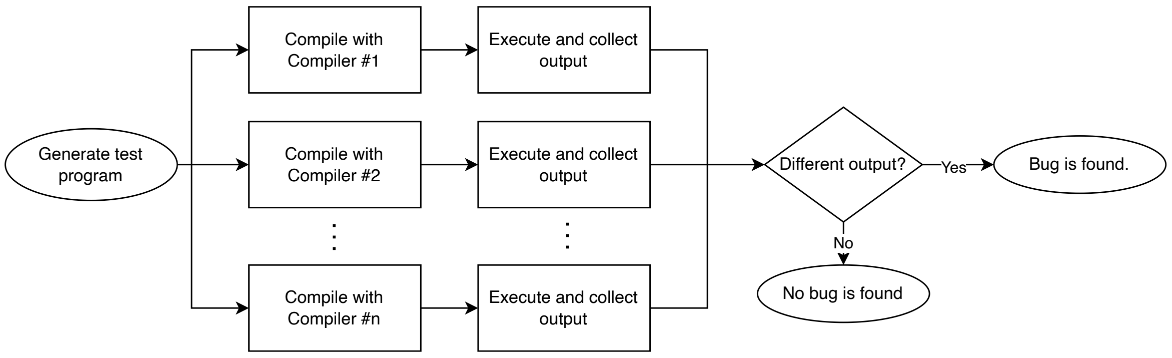


**Figure 2.2:** The general workflow of differential testing

gram in different ways, running the executables and comparing their outputs. The "different ways" in which the program could be compiled include:

- **Cross-Compiler:** The program is compiled using different compiler implementations. This option requires the language to have multiple compiler implementations, which is rarely the case for newer or less popular languages.
- **Cross-Optimisation:** The program is compiled using multiple optimisation levels of the same compiler. This option requires the chosen compiler to have many optimisation levels and enable flags for optimisation control. This is the most widely used option within compiler testing.
- **Cross-Version:** The program is compiled using multiple versions of the same compiler. This option is good for finding CI bugs.

### Metamorphic testing

Metamorphic testing [21] involves finding metamorphic relations, which specify how transformations to the input program should change its output. The most popular method to construct metamorphic relations is using equivalence relations, which establish under some assumptions that two programs are equivalent after transformation. One form of this method is EMI, explained briefly in 2.3.

The workflow of metamorphic testing is as follows: the test program and its metamorphic relations are constructed, the test program and its variants are compiled by one compiler, the executables run with their output captured, and the outputs are compared against their metamorphic relations. If an equivalent relation is used, the last step would involve comparing to find inequivalence in the output, which indicates bugs.

## 2.4 Test program reduction

The next step after finding a compiler bug is to report it to the developer along with the test program that caused the bug. The developer must then investigate the

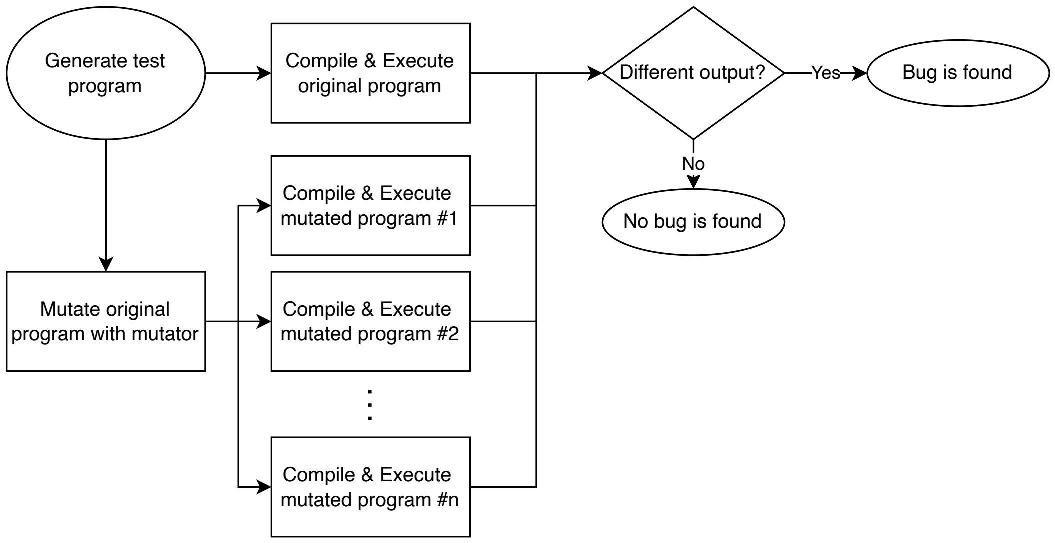

Generate test program
Compile & Execute original program
Different output?
Yes
Bug is found
No
No bug is found
Compile & Execute mutated program #1
Mutate original program with mutator
Compile & Execute mutated program #2
Compile & Execute mutated program #n

**Figure 2.3:** The general workflow of metamorphic testing

program to find the bug-inducing segment. Test programs that find compiler bugs are usually very long and complex. The developer of CSmith [2] discovered that the largest amount of bugs is found in a test program of size 81KB. To investigate a program of this size would be extremely time-consuming for developers. Therefore, it is preferred and sometimes required [22] that the test program is reduced before submitting a bug report.

The test program reduction problem is one where, given an input program and an interestingness test, output a simplified and shortened version of the input program that still passes the interestingness test. The interestingness test in the context of compiler testing answers the question: “Is this program still triggering a bug?”. The idea at the root of test program reduction is Delta Debugging (DD) [23], a systematic way for domain-agnostic test case reduction. To simplify a test case, DD aims to find a subset of the test case such that if any one component is removed from this subset, the test will no longer fail. This is labelled a 1-minimal test case. DD uses an algorithm *ddmin* which repeatedly partitions the test case n-way to find the minimal case within some partitions or the complement of some partitions. ddmin has exponential complexity, making it infeasible to simplify large, real-world programs.

Hierarchical Delta Debugging (HDD) [24] solves the complexity problems in DD by exploiting the structure of the test cases, making HDD a domain-specific test case reducer. In DD, all test cases are treated as flat atomic lists. The *ddmin* algorithm then partitions these lists in a simple way, often creating an invalid input to the software under test. HDD instead view test cases as nested structures, e.g. an abstract syntax tree. Treating test cases as trees with multiple levels, HDD applies the ddmin algorithm at each level of the tree, ensuring the validity of the resulting test case. While the worst-case complexity of HDD is still exponential, the amortised complexity of HDD is much better than DD, enabling its use to simplify real-world test programs.

The state-of-the-art reducers can be divided into two categories: language-agnostic and language-specific. C-Reduce [25] is a language-agnostic reducer that applies pluggable transformation, such as removing unused variables or functions, to the program until it reaches a fixed point. These transformations can be applied to any language that includes the same concepts. PERSES [26] is another language-agnostic reducer. It relies on AST to parse the test program like HDD. However, PERSES also uses AST to guide the transformations, giving its results higher quality than HDD. On the other hand, glsl-reduce [27] is a language-specific reducer for GLSL, and Fast-Reduce [25] is a specific for C.

Language-specific reducers are preferred because of their ability to avoid introducing undefined behaviour in the reduced programs. That is the case until the introduction of Reconditioning [5]. The reconditioner for program generation (section 2.3) can also be used to ensure no undefined behaviour exists after program reduction. This eliminates the disadvantage in terms of the undefined behaviour in language-agnostic reducers, allowing them to be used more freely.

### Bug slippage

One problematic side-effect of test case reduction is *bug slippage*. Introduced by Chen et al. [28], this problem occurs when the reduction of a program containing bug $\alpha$ results in a program containing bug $\beta$. Bug slippage is generally undesirable as bug $\beta$ would prevent the discovery of bug $\alpha$. To make matters worse, bug $\beta$ may be an easy-find bug that has already been reported. However, bug slippage could also be exploited. Holmes et al. [29] suggests that reducers could output multiple reduce programs instead of one so that along with the original bug, any bugs found from a slippage can also be investigated.

## 2.5 Bug deduplication

Bug deduplication is a process of eliminating bugs that have the same cause and exhibit the same behaviour; for example, duplicated bugs are ones whose test program calls the same method, which crashes in the same way. Pieces of information used in bug deduplication are largely from two sources: bug reports and runtime information.

### Run-time information-based approaches

Run time information of a program could be retrieved from many different sources, e.g. stack trace, control flow, register states, etc. When it comes to crash bugs, stack trace is often the go-to source of information. One classical approach is to use information retrieval methods such as Term Frequency – Inverse Document Frequency (TF-IDF) to calculate the similarity between two stack traces. A result from Durfex [30] shows a recall rate of 86% when deduplicating bugs based on package name frequency similarity between stack traces in the Eclipse bug dataset. However, there is an argument that TF-IDF has a drawback due to its ignorance of the order of subroutines in the trace. Therefore, another approach focuses on matching algorithms, including longest common subsequence [31], Levenshtien distance [32]. FaST [33] and TraceSim [34] developed algorithms for a relaxed version of the optimal global alignment problem to align stack traces, then using similarity score to find duplicated crashes. ReBucket [35] proposes a stack-frame position-dependent metric for measuring similarities between crash traces, grouping crashes that are close together in the same bucket, achieving an F-score of 0.876.

The rise of machine learning also brought about more applications of machine learning for bug deduplication. Ebrahimi et al. [36] trained a hidden Markov model with several groups of duplicated stack traces to determine whether a new stack trace is a duplicate. They achieved a mean average accuracy of 76.5% on the Firefox dataset. However, this approach falls short when given a duplicate whose group is not in the training set. DeepCrash by Chao et al. [37] developed a tokenize method to convert stack traces into vector frame representation. These frame representation vectors are then converted into feature vectors using deep learning techniques. Then the feature vectors are bucketed using a clustering algorithm. DeepCrash has been evaluated against Durfex, ReBucket, Tracesim, and FaST. It is found to have the best overall performance for the Netbeans stack traces dataset [38].

Unfortunately, some bugs, such as miscompilation bugs, are silent and do not generate a crash stack trace. In these circumstances, bug similarity can still be determined from their test program context, such as control flow, data flow graph, and program semantics. Chen et al. [28] developed a fuzzer taming tool by using distance functions to measure the diversity/interestingness of a test program, ranking them by Furthest Point First. van Tonder et al. [39] uses an automated program repair technique to identify the root cause of the bug. By patching test programs with approximate fixes determined by rule-based templates and observing their behaviour, the unique bugs in the program can be determined and clustered. This approach works well for common classes of bugs such as null pointer dereferencing and memory corruption bugs, achieving much better deduplication results than built-in deduplicator from AFL [3], CERT BFF [40], and HongFuzz [41]. The drawback of this approach is the manual effort needed to define the rules and fix templates for different types of bugs.

### Bug report-based approaches

Information Retrieval (IR) and Machine Learning (ML) techniques dominate the area of bug report-based deduplication/triage because of their ability to handle natural language. Both of the techniques have the same workflow: collect information, perform feature extraction, and calculate similarity and grouping. It is in the similarity calculation step that IR and ML use different sets of methods.

Yang et al. [42] introduced a knowledge-based duplicate detector (K-Detector). K-Detector takes source code, stack trace and historical crash as input. It strips all stop words from the stack trace, leaving only a sequence of function calls. Then acquiring knowledge of function-to-component mapping from the source program AST built using Clang and converted the stack trace into a component sequence. K-Detector uses a mathematical model to calculate pairwise similarities between two-component sequences. This method achieved an AUC of 0.986 on the SAP HANA dataset. However, false positives are still a problem for K-Detectors. Because its mathematical models put high weight on the components on top of the stack, two stack traces from different bugs with similar beginning sequences will likely become false positives.

Jiang et al. [43] proposed a new framework: Test Report Fuzzy Clustering Framework (TERFUR). Taking bug reports as inputs, TERFUR uses feature extraction techniques such as word segmentation and stop word removal. Natural language processing (NLP) technique is also used to enhance the word description of a bug. Then the bug report similarity is calculated using a vector space model, and a cluster merging algorithm is used for clustering. TERFUR is evaluated against a dataset of bug reports from 5 different applications, achieving an F-score of 0.758.

Rodrigues et al. [44] presented a novel deep learning network: Soft Alignment Model for Bug Deduplication (SABD). The model takes a new query bug report $q$ and a candidate existing report $c$ as input and outputs the probability that $q$ is a duplicate of $c$. SABD has two sub-networks that independently process categorical features and textual features of the bug reports, where the textual network uses specific techniques like word embeddings, and output from both sub-networks is used to produce the final probability. This method achieved the highest recall rate when compared to other state-of-the-art methods on several benchmarking datasets.

Xie et al. [45] proposes DBR-CNN, a domain-specific convolutional neural network for bug reports. DBR-CNN's data preprocessing steps involve basic NLP techniques such as word tokenizing, stemming and stop word removal. The pre-processed bug reports are then transformed into feature vectors using word embedding. Taking two embedded bug reports as inputs, these reports go through a convolutional layer and max pooling layer, then have their domain-specific features concatenated; finally, logistic CI is used for their classification. Evaluation of DBR-CNN on four open-source bug report datasets resulted in an F-score of 0.903.

### Test program reduction and bug deduplication

It is important to point out that test case reduction is still a crucial step, even if the process of bug investigation can be partially automated. Test case reduction reduces the amount of irrelevant information (noise) from the bug report, which reduces the probability of IR or ML techniques focusing on the wrong terms. Some approaches directly rely on the test program to analyse a bug. Chen et al. [28] compared the result of their fuzzer taming tool on the same set of bug-triggering programs with and without reduction. The experiment showed the result from the non-reduced set is significantly worse than the reduced set. The author commented that one should not attempt deduplication without reduction.

## 2.6 Bug triage

Bug triaging is the process of prioritising and assigning a bug to the appropriate developer. It is usually done manually due to the complexity of the software. Recently, a lot more work has been put into using machine learning to understand the severity of a bug and the historical relationship between developers and different types of bugs.

**This section is not necessary to understand the project.** Advanced bug triaging has not been implemented in CompFuzzCI due to time constraints. However, it is still an interesting topic to explore.

Guo et al. [46] propose a method to assign a bug report to a developer using text processing and a Convolutional Neural Network (CNN). This method takes the textual bug report with its assigned developer as input. The text preprocessing follows the regular pattern of segmentation, stop word removal, stemming and One-Hot encoding to convert words to a feature vector. This feature vector is fed into CNN to predict the name of the developer. Then the method cross-checks the predicted developer's recent activity if they are still active. Evaluation of this method on Eclipse, Mozilla and NetBeans bug report datasets resulted in a top-10 accuracy of 0.749. While this result is not high enough for the method to be used in production, it still has an interesting approach that later research could benefit from.

Latent Dirichlet Allocation (LDA) is a popular algorithm used for statistical topic modelling. It can discover topics in a document corpora in an unsupervised manner, outputting the list of topics and their word distribution. Xia et al. [47] proposes a novel algorithm which extends LDA: multi-feature topic model (MTM). MTM extends LDA by also considering information outside of the bug report, including the related software product, the software component, and the developer information. This allows MTM to extract specialised topics related to some product or component. MTM is part of the method called *TopicMiner* for incremental learning so that TopicMiner can adapt to bug report changes over time. TopicMiner trains on a dataset of bug reports that developers have fixed and predicts candidate developers to assign a new bug report. The candidate developers are chosen by the highest *affiliation score*, representing how many topics the developer has in common with the bug report. TopicMiner is evaluated against baseline methods against baseline methods at the time on GCC, OpenOffice, Mozilla, Netbeans, and Eclipse bug report datasets. Results showed that TopicMiner significantly improved from other methods, achieving a top-5 precision average of 0.851.

Yadav et al. [48] introduce a system to rank appropriate developer on bug reports based on their expertise score. They aim for the system to help reduce the iterations of bug tossing between developers. The developer expertise score (DES) is calculated based on three matrices: priority-weighted fixed issues, versatility and breadth index[19] and developer average bug fixing time, which can be received from bug report priority, product and timestamp. The developer recommendation for a new bug report works as follows: Features of the developer are learned from historical bug reports using text processing similar to existing methods. When a new bug report arrives, its feature is extracted and calculated for similarities with developer features. DES is calculated for each developer. Using similarity score and DES, choose the highest-ranking developer to assign the new bug report. The evaluation of the DES approach on Mozilla, Eclipse, Netbeans, Firefox, and Freedesktop datasets shows an average accuracy of 0.895. The wrong assignments made by DES are mainly from incomplete information from the bug report, lack of root cause, or lack of information on new developers. DES is not capable of assessing developers' workloads. If one developer has high DES, then they can be assigned more bug reports than others, which will overload their work and increase bug fixing time or bug tossing.

Alazzam et al. [49] proposes the RFSH method to classify bug report priority using a graph-based approach. Before constructing a graph of terms, the terms in bug reports are assigned relative frequency using TF-IDF, then using *frequency cutoff* to discard outlier terms. Topics are also extracted using LDA. The key part of this method is to construct a graph with terms as nodes and cooccurrence as weighted edges. When a new bug report is received as input, its feature vector is augmented with the terms that have heavy edges to the terms in the bug report. Then the resulting feature vector can be fed into classification algorithms. The method shows different accuracies for each priority bug, with p1 being highest at 0.734 when using SVM for classification, p2 at 0.868 when using Random forest classification, and p3 at 0.942 when using SVM for classification. The limitation of this approach is its dependency on the graph structure, which can be complex or poorly connected. Both cases will degrade the performance of RFSH.

Kukkar et al. [50] suggests that the techniques commonly used for bug triage, such as term frequency, often misclassify bugs because of their lack of consideration of word order. They proposed BCR: a method to classify bug severity using a CNN and Random Forest with Boosting based on n-grams. BCR require only the title and descriptions of bug reports as input. It extracts n-grams from the text, pushing them through CNN to extract the non-linear, high-representative features, then using random forest with boosting to classify the severity of the bug report. Evaluations of Mozilla, Eclipse, JBoss, OpenFOAM, and Firefox showed that this approach was effective, with an average accuracy of 0.963.

Many of the proposed methods in bug triaging have very interesting prospects. However, it will be difficult to apply them to newly developed software since it would require sizable historical data to train the Machine Learning models. Many of the methods above also assume the availability of some information (software component, severity, platform, etc.) which might not be the case in real life. There are also complexity concerns because we aim to run bug triaging every fuzz session. There will be multiple bugs to triage within a short time. On the positive side, bug triaging can possibly be done alongside bug deduplication because of the common nature of the information and text preprocessing for both processes. In our research, we can use these methods to inspire our approach, which will need to be tailored to the information and computational resources we have available.

## 2.7 Software development

This section will cover some software development concepts used by the Dafny compiler development team, which are a big part of CompFuzzCI's design.

### Version control

Version control is a software engineering practice of tracking changes to a repository over time. The history of each file in the repository is kept so that it can always be referred to at any point in time. A version control system (VCS) is a system that automates version control, and it can be localised, centralised, or distributed. In this section, we focus on the distributed version control system (DVCS) as it is the most popular and is used by the Dafny compiler development team.

Distributed version control system can be simplistically explained as a peer-to-peer approach to version control. Each developer has a repository with its full history on their local server, and they have the flexibility to make any changes in their local repository without affecting their peers. This is especially useful when a team of developers works on the same repository but with different features. Once the changes are ready, they can be shared with other developers, who will pull the changes to their local repository. In this way, there are many copies of the repository, so if any servers go down, the repository and its history can be restored from any other location.

### Git

Git [51] is the most used DVCS. In Git, the remote repository acts somewhat like a central repository, where developers can push and pull changes to/from their local repository. Some of the key concepts in Git that are relevant to this project are:
**Commit** is a unit of change in a repository. It is a snapshot of the repository at a point in time. The commit stores a compressed version of the files that have been changed. The commit is based on one or more parent commits. Each commit has a unique hash, which is a SHA-1 hash of the commit's content. A head commit is the most recent commit in the repository. The commit is a node in a chain of commits

that make a history of the branch.
**Branch** is a separate line of development (a chain of commits). Every repository has a default branch called the master branch. Other branches can exist alongside the master branch and be used to develop a feature or a fix in parallel with the master branch without affecting it.
**Fork** is a copy of a repository that is under the forker's ownership. This makes it easier for the developer to work on the repository on their own without having to handle authorisation issues.
**Pull Request (PR)** is a request to merge changes from the head branch to the base branch. The head or base branch can come from fork repositories or branches within the same repository. Generally, pull requests are where code reviews and tests are done before merging the changes into the base branch.

### Continuous integration

Continuous integration (CI) is a software development practice that helps integrate code changes from multiple developers into a central software repository. Before CI, developers often worked on a local repository until completion before merging their change into the main repository. This makes code change very difficult to integrate because the code would have diverged largely between each developer, making it difficult to resolve conflicts and increasing the risk of introducing bugs. In CI, code changes from developers are regularly integrated into the central repository and tested. The goal is to find and fix any breaking changes as soon as possible.

Automated testing with CI usually involves linting, unit testing, integration testing, and CI testing. Developers mostly craft these tests. They are fast and can help discover changes that introduce bugs early on. However, human-made tests often do not have high path coverage and tend to suffer from bias and human limitations [52, 53]. The CI test suites also need a lot of effort to maintain and improve. For this reason, many companies started paying more attention to using fuzzing as a complement to classical CI testing.

### GitHub Actions platform

Most platforms for version control, such as GitHub [54], provide support for CI processes. GitHub Actions Platform allow users to automate a CI process that gets triggered by events on the code repository, such as a developer pushing new changes or pull request creation. The process is defined in a workflow and consists of event triggers, actions, and jobs (sequence of actions).

Many unit test frameworks, like Jest or Pytest, can be integrated with GitHub Actions. When workflow gets triggered, actions defined within it will be executed on a GitHub Runner, a virtual machine provided by GitHub. These runners work well with lightweight tests, such as the ones mentioned above. However, fuzz testing can be a computation-intensive job, especially if one needs to fuzz with high intensity to preserve short testing time while maximising coverage. Therefore, existing CI fuzz testing tools often utilise cloud computing. In section 2.8, we will review Google's solution for continuous fuzzing of open-source software.

## 2.8 Continuous fuzz testing solutions

### OSS-Fuzz by Google

OSS-Fuzz is Google's open-source project that aims to fuzz open-source software [55]. It currently supports 3 fuzzers: libFuzzer [56], AFL [3], and HongFuzz [41]. Therefore it can fuzz software written in C/C++, Rust, Go, Python, Java/JVM, and JavaScript. OSS-Fuzz allows developers to integrate their projects by writing their build script, configuration script, and fuzzer test target. After the integration, OSS-Fuzz will automatically build, fuzz, and create a bug report on the OSS-Fuzz tracker for any bugs found. The choice to file bugs to its own tracker is for information control since security bugs could be sensitive information. However, there is also an option to file bugs to GitHub Issues. OSS-Fuzz can be viewed as a front-end to ClusterFuzz, another project that is handling the fuzzing [57]. ClusterFuzz is a scalable fuzzing infrastructure that can run on any size cluster using the Google compute engine, making it perfect for intensive fuzz jobs. ClusterFuzz can also perform test case reduction, deduplication and triage.

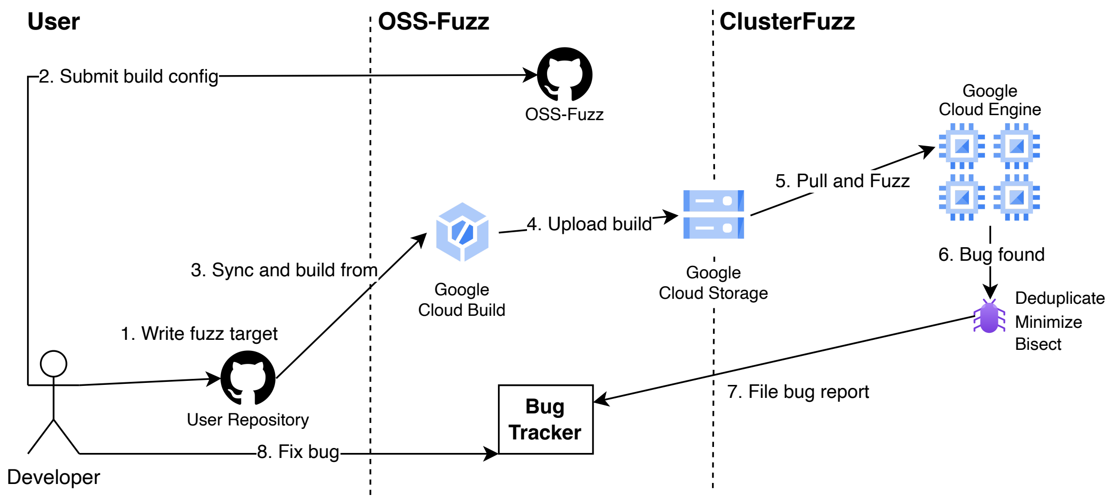


**Figure 2.4:** High-level workflow of OSS-Fuzz

ClusterFuzz uses Google's Cloud NDB to store bug signatures that are used in bug deduplication. The bug signature is created from the crash type, address, state and information parsed from the stack trace. Parsing information from stack trace can be a challenging task, as the stack trace can contain a lot of information specific to the test case, for example, addresses and variable names. ClusterFuzz dealt with this challenge by approaching the stack trace as structured information. A stack trace is a sequence of frames, with each frame containing filenames, function names, addresses, etc. All of which are matched using regular expressions. The fuzzer that found the bug also plays a role in deduplicating the bugs since different fuzzers are good at finding different types of bugs.

Triaging a bug is fairly simple when dealing with crash bugs only. ClusterFuzz analyses a bug by regular expression matching the crash with dictionaries of common issues such as heap buffer overflow, stack use after return, etc. Some regular expressions are specific to the language, and some are specific to the software under test. Using these terms, ClusterFuzz determines the severity of the bug with respect to the potential security vulnerability it could cause. When finding a bug which cannot be triaged using a text-matching script, ClusterFuzz assigns high severity to these bugs by default.

The strength of OSS-Fuzz is its computational power, fuzz strategy, and excellent fuzzer it utilises. As of August 2023, OSS-Fuzz has identified over 10,000 vulnerabilities and 36,000 bugs across 1,000 projects. ClusterFuzz's use of cloud services has inspired CompFuzzCI's architecture design.

### CI Fuzz by Code Intelligence

Unlike OSS-Fuzz, CI Fuzz is a closed-source commercial product that is designed to be integrated into the software development pipeline. Code Intelligence offers its own platform to interact with the fuzzer result. CI Fuzz supports projects written in C/C++, Java and JavaScript using specific build systems. Compared to OSS-Fuzz, CI Fuzz is more integrated into the GitHub CI. It can be incorporated into the CI pipeline as if it were a unit test. CI Fuzz provides a bot which will report the bug findings on the pull request, blocking the merge until the developer assesses the bug as not critical. The CI Fuzz dashboard also offers coverage reports and other metrics.

## 2.9 The 4Ws and 1H of continuous fuzz testing

There are many things one needs to consider before integrating an activity into one's routine. By default, we would not want to do costly things that give us little benefits. In order to decide if fuzzing in the CI is suitable for a project, we must reflect on the resources available, the potential benefits, and the potential costs. In this section, we will discuss the questions that should be answered before integrating fuzzing into the CI.

- **Why do we need to fuzz?** Integrating fuzzing into a software development routine requires a lot of planning and consideration. It is good to reflect on whether the software under test needs fuzzing. Fuzzing is good for finding bugs that all human-made tests missed. However, if a bug in the software under test has no security implications or is insignificant, then the costs of fuzzing could outweigh the benefits.

- **What to fuzz?** Fuzzers require the software under test to be executable. Building the whole software every time changes are made can be very costly. In the ideal case, the developer could target only the part of the code that changed by writing fuzz targets and using white-box fuzzers such as libFuzzer. However, another problem that could arise in this approach is how to determine which part of the code is affected by the change. For example, a change in one function could affect another that depends on it, and the bug to be found is not in the changed function itself but rather the interaction with another. This solution is also not probable in some software, specifically ones with very complex build systems [58]. For these software, between changing the way the SUT is built and building the whole SUT every time, there seem to be no effective options.

- **When to fuzz?** It is well known in testing that not every change is breaking. Therefore it will be extremely inefficient to fuzz every single change in the code. The problem of finding the right change to test occurs not only in fuzz testing but with any large-scale testing in general. A study by Memon et al. [59] provides valuable insights on situations where code is more likely to break, for example, codes that are modified by 3 or more developers within a short period. Interestingly, language also has a factor in how likely code would break. Their experiment showed that C++ is more likely to break after a change than Java. Insights from this type of study can be applied to fuzzing. However, code-breaking behaviour will likely differ from one software to another, and the SUT will require some monitoring to determine which strategy is appropriate.

- **Where to fuzz?** Fuzzing, as we mentioned, can be a computationally expensive job. Most software development platforms provide customers with virtual machines within which to run tests, but the capabilities of these runners are often limited. It is likely not enough computational power or not scalable enough for fuzzing. Nowadays, the solution to this problem is to subscribe to cloud computing services, such as AWS or Google Cloud. This solution comes with a subscription cost and the responsibility of properly orchestrating the rented power. Orchestration is also being provided as a service. Some examples of the providers are Kubernetes, Openshift and Amazon ECS. The availability of cloud and orchestration services takes away a lot of manual effort needed to configure the fuzzing environment, partially solving the problem. The developer would still have to figure out how to integrate this into the CI pipeline, provide the fuzzer and the SUT with all its dependencies, and maintain the integration.

- **How (long) to fuzz?** In a normal test campaign setting, it is shown that the best duration to fuzz is 24 hours [60]. However, if we were to fuzz every code change, the duration would need to be much shorter than that. The

> recommended build and test duration for continuous integration is 10 minutes. However, some are concerned that running the fuzzer for such a short time would prevent the discovery of bugs. Klooster et al. [61] has investigated this concern and found that fuzzing for a shorter time can be as effective as a multi-hour testing campaign. The short but frequent approach to fuzzing works especially well when each change is fuzzed for 30 minutes, and longer campaigns are run nightly when no changes occur. This duration will not apply to all software and will require some monitoring to determine the best duration. It is also worth observing the usual workflow of the software development team and trying to balance it with the ideal duration of running the fuzzer.

Tools such as OSS-Fuzz or CI Fuzz help solve these problems for many open-source software, making continuous fuzzing more accessible. Unfortunately, these tools cannot be used for the Dafny compiler, which is written in an unsupported language. The Dafny compiler also contains noncompliance generation bugs and miscompilation bugs, which are more complex to handle than crash bugs. CompFuzzCI is designed to specialise in finding and handling the types of bugs present in compilers, filling in the gap of other CI-fuzzing solutions.

# Chapter 3

# Design and Implementation

This chapter will cover the design of CompFuzzCI. The structure of the chapter is as follows. In section 3.1, we provide answers to the 4Ws and 1H of continuous fuzz testing, which will clarify some of the decisions made with the Dafny team prior to designing CompFuzzCI. Section 3.2 will walk through the system requirements of CompFuzzCI, how they were fulfilled, and justify some of the choices that were made. Section 3.3 moves from the system requirements to the functional requirements of CompFuzzCI, it details the interaction between the fuzzer, deduplication, reduction, bisection and bug-tracking modules, which were used to fulfil the functional requirements. The rest of the chapter will go into more detail on each of the modules.

In section 3.4, we look into the fuzzer module and briefly discuss its workflow. Section 3.5 focuses on the deduplication module and on the different bug deduplication strategies that we tried. Section 3.6 talks about the test case reducer we have used, their strength and weaknesses, and our chosen strategy for test program reduction. In section 3.7, we get into detail regarding the challenges faced while making a bisection module and how and when we run bisection. Finally, section 3.8 explains briefly how CompFuzzCi keeps track of the bugs it has found and the bugs found by Dafny users.

## 3.1 Answer to the 4Ws and 1H

The first step in designing CompFuzzCI is to answer the 4Ws and 1H of continuous fuzz testing. To answer these questions, we researched the available technologies and tools and how to use them. We also initially met with the Dafny compiler development team from AWS to learn about their development workflow and discuss their preferences for interacting with CompFuzzCI.

- **Why do we need to fuzz?** The Dafny language is being used to write and ensure the correctness of code. It is being used to ensure the correctness of Amazon cryptography tools, which could be used in a security-critical context. Therefore bugs in the Dafny compiler can have severe consequences. This justifies the attempt to integrate fuzzing into the Dafny CI pipeline; whether it is worth the cost will be evaluated in chapter 4.6.

- **What to fuzz?** In this project, our aim is to fuzz the Dafny compiler. The Dafny language codebase contains not only its compiler but also its verifier, test generator, formatter, etc. The ideal situation would be to isolate the compiler and only build and test it. To make this possible would require us to patch the Dafny build scripts every time, which could be problematic during bisection (further explanation in section 3.7). The fuzzer that we have is also a black box fuzzer, which means we will not have the ability to target specific parts of the code. The final decision of what to fuzz is, therefore, to build the whole package and fuzz the entire compiler code at once.

- **When to fuzz?** The Dafny compiler is being developed on GitHub following CI practices. A CI pipeline for testing changes in a pull request already exists. We had 3 options on when to fuzz: on changes being pushed, on pull request creation, or nightly.
  - **Fuzz on push:** Originally, we wanted to fuzz on changes being pushed, with the motivation that if the bug is found right after it was introduced, it will be much easier for the developer to revert the commit. If we choose this option, CompFuzzCI will be triggered very often and for a shorter period of time. We are concerned about whether the fuzzer will be able to find any bugs in such a short time. We also have concerns about the computational cost of running the fuzzer so often. Fortunately, we did not have to deal with these concerns as the Dafny team prefers tests and code reviews to be done on pull requests.
  - **Fuzz on pull request:** The Dafny CI tests are already being run on the CI pipeline on pull request creation and synchronisation. The Dafny team prefers to have the fuzzer run on pull request creation so that the developer can review the fuzzer result along with the CI test result. This option allows the fuzzer to run less often but for a longer duration.
  - **Fuzz nightly:** Fuzzing nightly is also an option which is still not out of the question. It is definitely possible to schedule fuzzing nightly, and it is something we want to consider in future work.

- **Where to fuzz?** The Dafny test suite is currently running on the GitHub Actions platform. The test result is reported on the pull request, and the test logs can be viewed directly on GitHub. Fuzzing on the same platform would be ideal. However, it was not possible, due to the need for intensive computation and integration with other functionality, as we explain later in section 3.2.

- **How (long) to fuzz?** During the initial meeting, the Dafny team informed us that the average lifetime time for a pull request is 1 hour. This means that CompFuzzCI should be able to fuzz, reduce and bisect within 1 hour. We have decided to run the fuzzer for 1 hour, but in chapter 5.5, we will explore the possibility of running the fuzzer for a shorter or longer duration.

## 3.2 Architecture

This section will describe the environment and the services that CompFuzzCI will use. We start by listing the system requirements and then describe the services that will be used to meet these requirements.

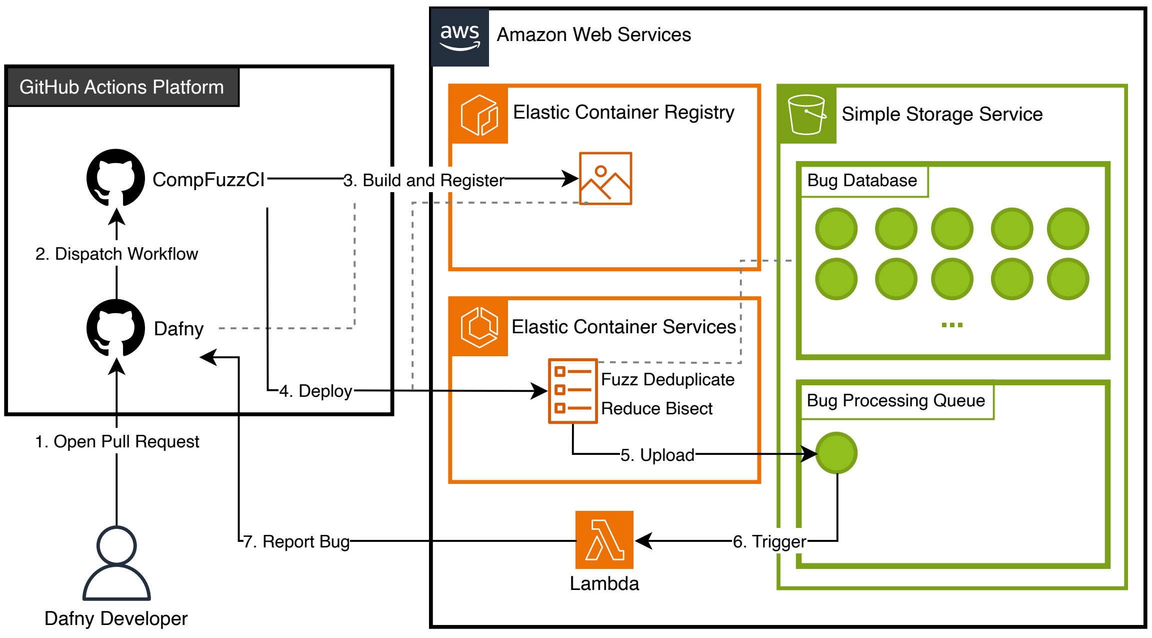


**Figure 3.1:** The architecture of CompFuzzCI and the interaction between the services

The starting system requirements for CompFuzzCI are:

- **GitHub workflow:** CompFuzzCI must be able to run as part of the CI pipeline on the GitHub Actions Platform. The simplest way for CompFuzzCI to be part of the Dafny CI is for it to be one of the **GitHub workflows** in the Dafny repository that gets triggered by a pull request event.
- **AWS cloud services:** CompFuzzCI will be running in a time-constrained manner compared to a generic fuzzing campaign. CompFuzzCI will run many instances of the fuzzer simultaneously to increase the probability of finding a bug and cover as much of the Dafny compiler as possible. We know that CompFuzzCI will need more resources than a single GitHub runner, which the GitHub Actions Platform provides. Since the fuzzer tends to utilise a lot of CPU, we decided to use **AWS cloud services** to provide the computing power.
- **Containerisaion:** CompFuzzCI must have a copy of the Dafny repository and be able to build the Dafny compiler for fuzz testing. We know that the environment where the fuzzer tests the compiler will need to have a specific set

of dependencies installed. To avoid having to install these dependencies every time, we decided to define a container image that will have all the dependencies installed and keep the environment consistent. The image will be used as a base for installing different versions of the Dafny compiler. Because of that, we also need a **container registry** to store the images containing different versions of the Dafny compiler, and fuzzing will be run in a **container**.

- **Auto-scaling:** CompFuzzCI must always run whenever the GitHub action is triggered (must be able to acquire compute power spontaneously). We need the computing power to be acquired on demand and released when unnecessary. Therefore, we need it to automatically scale; we use **AWS auto-scaling groups**.

- **S3 bucket:** CompFuzzCI need a database to store the bug signatures. As of now, we do not need the data to be relational, so we decided to use **AWS S3 bucket storage** because it is cheap and easy to use.

By fulfilling the starting requirements, we have a very basic architecture of CompFuzzCI. Our next step is to design the interaction between the services. At this stage, the service interaction requirements are as follows:

- **Building and running the correct Dafny version:** The GitHub workflow must be able to point the compute instances to the container image with the correct version of the Dafny compiler (right branch, right commit). We decided that the workflow will have an action which builds the container image containing the head and base branch of the Dafny compiler from the pull request. The image will be pushed to the container registry. Later, the workflow will signal AWS to deploy CompFuzzCI using the right image.

- **Signalling to the auto-scaling group:** The GitHub workflow must be able to signal to the AWS auto-scaling group to acquire compute instances to run the container. We know that it will be difficult for the workflow to directly communicate with the auto-scaling group, so we decided to use **Amazon ECS (Elastic Container Service)** with the auto-scaling group to manage the compute instances. The rationale for this decision can be found in appendix 6.3.

- **Container access to the storage:** The compute instance must be able to access the S3 bucket storage. We know that the compute instances will need to have permission to read and modify the storage. We decided to use **AWS IAM role** to give the compute instances a role with access to the S3 bucket.

- **Reporting bug back to GitHub:** The compute instance must have a way to report bugs back to GitHub. We know that the compute instances will need to have permission to write to the GitHub repository. This is only possible if the compute instances have a GitHub token. However, a token is sensitive information and should not be widely distributed to computing instances. We decided to take this responsibility away from the compute instances and instead have a **Lambda function** with a connection to **AWS secret manager**. The Lambda functions will be triggered by the compute instances to report the bugs back to GitHub.

After fulfilling the second set of system requirements, we have a more detailed architecture of CompFuzzCI, as shown in figure 3.1.

## 3.3 High-level workflow

With the architecture in place, we can now move on to the functional requirements of CompFuzzCI. The functional requirements for CompFuzzCI and the modules that fulfil them are as follows:

- CompFuzzCI must be able to fuzz the Dafny compiler. Fulfilled by the Fuzzer module.
- CompFuzzCI must be able to determine if the error is a bug and if it is a new bug or a duplicate. Fulfilled by the Deduplication module.
- CompFuzzCI must be able to reduce the test case that caused the bug. Fulfilled by the Reduction module.
- CompFuzzCI must be able to bisect the commit that caused the bug. Fulfilled by the Bisection module.
- CompFuzzCI must be able to report the bug back to GitHub. Fulfilled by the Bug Tracking module.
- CompFuzzCI must be able to keep track of the bugs found, both by itself and by users. Fulfilled by the Bug Tracking module.

Before looking into the details of each module, we will first discuss the high-level workflow of CompFuzzCI when handling different types of bugs.

### Workflow for crash and non-compliance generation bugs

We will review the workflow in figure 3.2.

1. **Fuzzer:** The fuzzer will fuzz the Dafny compiler until it finds an output that contains one or more error messages in one or more backends.
2. **Deduplication:** The bug deduplication module takes the output of the fuzzer and parses it, acquiring the error and checking if it already exists in the bug database. If the bug is new, its signature will be stored in the database. If the bug is a duplicate, the bug will be discarded.

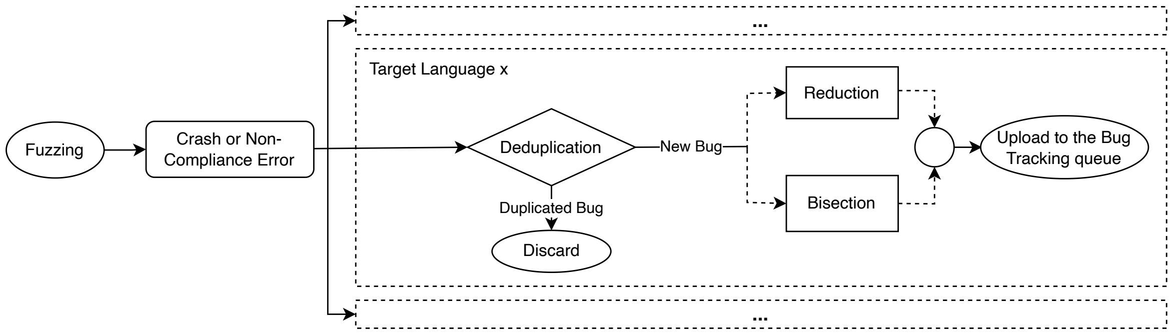


**Figure 3.2:** The workflow for processing crash and non-compliance generation bugs

3. **Reduction & Bisection:** The test program reduction and bisection are done asynchronously. The reason for this is justified later in this section.

4. **Bug Tracking:** Once the bug is reduced and bisected, it will be uploaded to the bug tracking module's processing queue, which will then report the bug back to GitHub.

## Workflow for miscompilation bugs

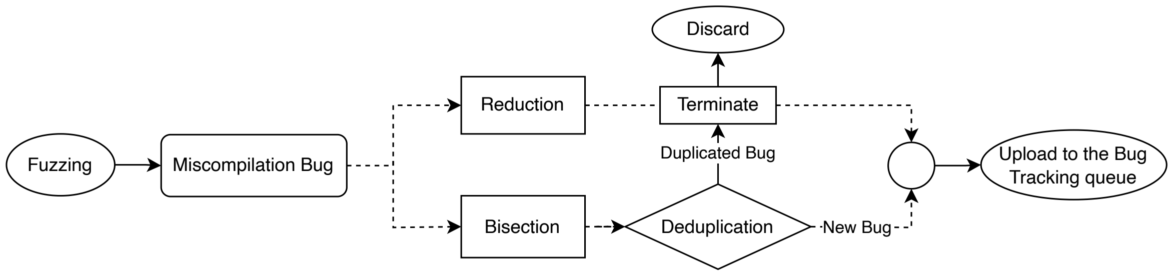


**Figure 3.3:** The workflow for processing miscompilation bugs

We will review the workflow in figure 3.3.

1. **Fuzzer:** The fuzzer will fuzz the Dafny compiler until it finds an output that contains differences between the backends.

2. **Reduction & Bisection** Due to the lack of a clear error message, the signature for the miscompilation bug cannot be created immediately. However, the differences between the backends are a clear indication that a bug is present. Therefore, we skip the bug deduplication step and go straight to the reduction and bisection step.

3. **Deduplication** The output of bisection (the first bad commit) is used to create the bug signature for miscompilation bugs. If the bug signature exists in the database, the bug is a duplicate and will be discarded. The reduction module, which is unlikely to be finished, will also be terminated. If the bug is new, the reduction module will continue until completion.

4. **Bug Tracking:** Once the bug is reduced, it will be uploaded to the bug tracking module's processing queue, which will then report the bug back to GitHub.

## Justification for workflow design decisions

**Why are the errors in different backends processed separately?**
We define processing as the deduplication, reduction, bisection, and reporting of the bug. We decided to process errors separately because we observed that errors present in different backends might not have the same root cause. To clarify this, let us explore the possible cases of finding errors in multiple backends simultaneously.

1. The error comes from a bug that exists in the Dafny compiler itself; the crash happens during the processing of the internal AST, and the error message is uniform when executing all the backends. For example, a bug in the Dafny parser. In this case, the error will be in Dafny error format and will be present in all backends. We handled this case by separating the Dafny error into a class of its own.

2. The error comes from a bug that exists in the Dafny compiler itself; the crash could happen during the translation of internal AST to the target language, during the compilation of the target code, or during the execution of the target code. The error might not exist in all backends and will have different formatting. For example, pull request number 2734 (appendix 1) changes the timing of the flattening of a particular nested structure. This change is reflected in the Dafny Core utility. The bug in issue 3472 (appendix 4) shows up when the user tries to compile the translated code using C#, Java and Go compilers. The error message is now in the format of the target language compiler. The developer knows these seemingly different errors are from the same root cause. However, the bug deduplication module will not be able to determine this. In this particular case, CompFuzzCI will spend unnecessary time trying to process the bug separately for C#, Java, and Go. Only to end up with the same results. We have not yet found a way to solve this problem, and we leave it as future work.

3. The error comes from a bug that exists in the target language backends from a code that is trying to achieve the same functionality. The error might not exist for all backends and will have differences in the stack trace. For example, issue number 5677 reported incorrect handling of a special character string. The bug is present in the C# and Go backend (appendix 5b). However, the bug was introduced into both backends at a different time. The bug existed in the Go backend first (appendix 5a) during an attempt to fix an issue related to the handling of the UTF-16 escape character in pull request number 2926 (appendix 2). The bug was then introduced into the C# backend during a backend-wide redesign of the string handling in pull request number 3016 (appendix 3). Because the bug in C# and Go backend are done separately, CompFuzzCI was able to find the first bad commit for each backend. This

would not have been possible if the errors had been processed together, and we would only see that the bug was introduced by pull request 3016.

4. The error is not related at all. The Dafny Java backend contains a lot of errors related to datatype casting and the way it uses lambda. These errors frequently show up simultaneously with the errors in other backends. By separating the processing of the errors in different backends, we can avoid introducing complexity into the deduplication, reduction and bisection modules that are caused by the Java errors. As a result, all the modules became much more accurate. Most of the time, the Java errors are duplicated and discarded by the deduplication module.

While there is a disadvantage of processing and reporting the same bug multiple times for each target language backend affected in case 2, we gain accuracy in cases 3 and 4. We prioritise accuracy over efficiency, and therefore we decided to process the errors in different backends separately.

**Why do reduction and bisection run asynchronously?**
First, we consider the bisection process. Bisection requires the Dafny compiler to be rebuilt repeatedly from different commits to find where the bug first appears. Second, errors are processed in different backends separately and in parallel. Both of these facts combined mean that if we bisect each error in parallel, there will be many threads trying to write to the local Dafny repository at the same time, causing race conditions. This will not only affect the processing of the current bug but could also affect the state of the Dafny code during fuzzing in the next iteration. Therefore, we decided that bisection should be done in a different container to avoid any modification to the local Dafny.

For this reason, the bisection module on the fuzzing instance will only be sleep waiting for the result from the bisection container to arrive. It makes sense to run the reduction module asynchronously while the bisection module is sleep waiting.

To be explicit, only the bisection module will be running on its own container, which we call the “bisection container”. The fuzzer, deduplicator, and reducer are running in the same container, which we call the “fuzzing container”.

## 3.4 Fuzzer

To fuzz the Dafny compiler, we must first have a fuzzer. The choices of fuzzer available in this research are XDsmith [62], fuzz-d [6], and DafnyFuzz [63]. All of these are black box generative fuzzers. XDsmith and fuzz-d rely on differential testing, and DafnyFuzz relies on metamorphic testing. The coverage of the fuzzers was evaluated in research by Donaldson et al. [63], who found that fuzz-d has the highest coverage among the three. In our research, we decided to integrate fuzz-d into CompFuzzCI first. However, we should also be mindful that different fuzzers have different strengths and weaknesses, and it is possible that fuzz-d will not find all the bugs that DafnyFuzz or XDsmith can find. We have designed CompFuzzCI's internal interface in a way that is more suitable for fuzz-d. However, the interface is also generic enough that only small modifications would need to be applied to the DafnyFuzz and XDsmith output format to integrate it with CompFuzzCI.

Since fuzz-d is an open source, we were able to extend it with the ability to test the Rust backend as part of the differential testing. This was not a part of fuzz-d before. We install fuzz-d on every container that will run the fuzzer. fuzz-d does not need any input, and it outputs a generated test program and a log file containing the output of running the generated program on all backends. We use this log file in the deduplication module to determine if we have found a new bug.

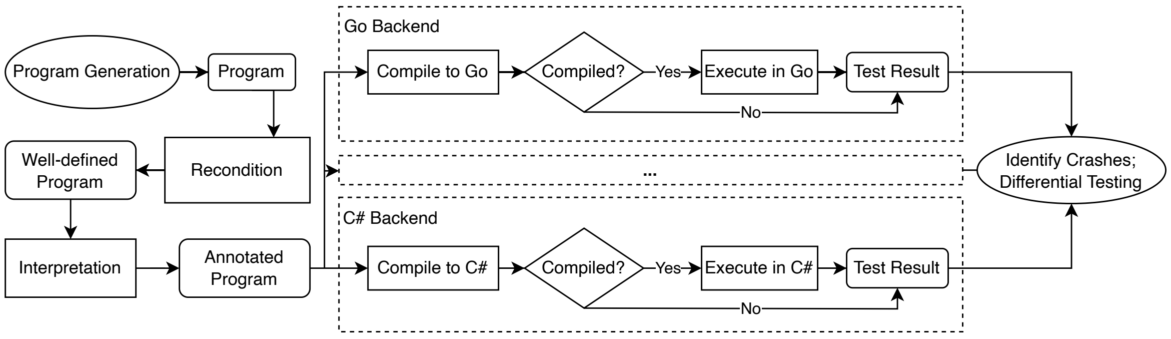


**Figure 3.4:** The workflow of fuzz-d from generation to output

The workflow of fuzz-d is as follows: The generator generates a test program using a grammar-aided approach. This program might not be well-defined and might contain undefined behaviour. The program is passed through the reconditioner (section 2.3) to produce a well-defined program. The well-defined program then undergoes interpretation, a process which simulates the program's execution to determine its expected output. Using this knowledge, the program can be annotated with print statements for program variables which fuzz-d know its expected value. The final annotated program will then be compiled and executed in the target languages. Its output will be captured, compared and piped to the output log file.

More details on fuzz-d can be found in [6].

## 3.5 Deduplication

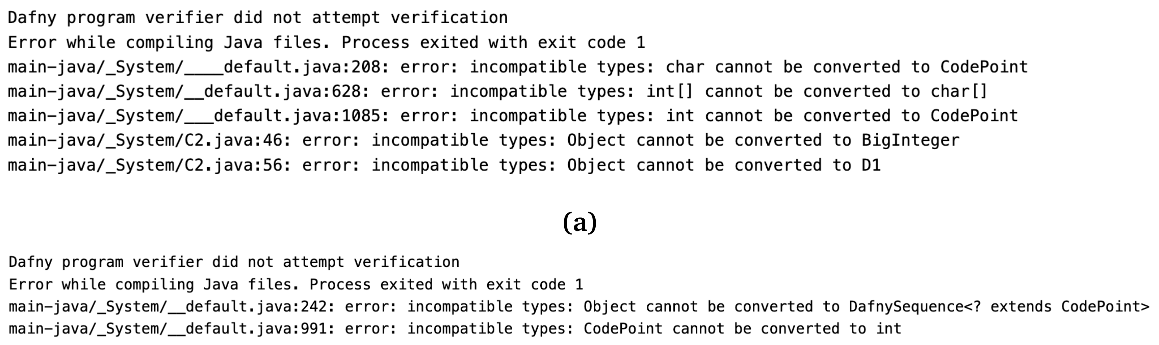
If a bug exists on the Dafny compiler, the fuzzer module can find the same bug multiple times until it is fixed. This bug might produce the exact same error or different ones in each encounter. We cannot report the bug every time we find one because it will clutter the bug tracker and distract the developers. We need to be able to distinguish between a known and unknown bug. This is the responsibility of the deduplication module.

When we first started designing the deduplication, we knew that the existing bugs would be stored in the database somehow. The design here is focused on how the bug signature will be created and stored so that the deduplication for all types of bugs can be done efficiently.

## Deduplicating crash and non-compliance generation bugs

The signature for crash and non-compliance generation bugs is created from the error messages, which are parsed from the output of the fuzzer. Parsing the error message is done using regular expression matching, for the lack of better ways. This requires making a large file containing all the error formats output by Dafny, C#, Java, JavaScript, Python, Go, and Rust. The error message itself is usually enough to identify the bug. If a bug affects multiple backends, signatures will be created for each backend based on its respective error message.

Our first strategy is to create a signature from a HashSet for all error messages. This strategy is too strict and makes duplicated bugs seem unique more often than is acceptable. For example: the error messages from the same bug in figure 3.5 and figure 3.6.



```
Dafny program verifier did not attempt verification
Error while compiling Java files. Process exited with exit code 1
main-java/_System/____default.java:208: error: incompatible types: char cannot be converted to CodePoint
main-java/_System/__default.java:628: error: incompatible types: int[] cannot be converted to char[]
main-java/_System/___default.java:1085: error: incompatible types: int cannot be converted to CodePoint
main-java/_System/C2.java:46: error: incompatible types: Object cannot be converted to BigInteger
main-java/_System/C2.java:56: error: incompatible types: Object cannot be converted to D1
```

**(a)**

```
Dafny program verifier did not attempt verification
Error while compiling Java files. Process exited with exit code 1
main-java/_System/__default.java:242: error: incompatible types: Object cannot be converted to DafnySequence<? extends CodePoint>
main-java/_System/__default.java:991: error: incompatible types: CodePoint cannot be converted to int
```

**(b)**

**Figure 3.5:** Bug in Java that should be detected as duplicates

We then added a list of error messages that are known to be duplicated, show up frequently, and are too complex to parse, such as the “incompatible type” error in Java, which has infinite variants. Let us call these error messages the ones known by CompFuzzCI. If the error message is in the list, the bug will be discarded and excluded from the HashSet. This helps reduce the noise from the Java backend errors, but it does not solve the case in the parsing error example.

Finally, we decided to deduplicate at parsing time. We will no longer use HashSet and will store error messages individually on the database, tagging it with the bug issue number. During parsing of the fuzzer output, each error message is looked up first in the known list and then in the database. If it is found, it will be discarded. If all the error messages are found in the database or the list, the bug is duplicated

```
main.dfy(3094,75): Error: rbracket expected
main.dfy(3094,81): Error: invalid Suffix
main.dfy(3094,76): Error: missing semicolon at end of statement
main.dfy(3094,319): Error: invalid UpdateStmt
4 parse errors detected in main.dfy
```

**(a)**

```
main.dfy(1846,30): Error: closeparen expected
main.dfy(1846,128): Error: invalid UpdateStmt
2 parse errors detected in main.dfy
```

**(b)**

**Figure 3.6:** Bug in the parser that should be detected as duplicates

and will be discarded. If there are some unique error messages left, then the bug is interesting. This does not completely solve the problem like the parsing error example, but it does reduce the number of times the same bug would be reported. In the worst-case scenario, the deduplication module would have to store all the different messages that could be triggered by the parser bug and register them in the database. At that point, the bug would no longer possibly be considered unique. This approach also allows bugs with errors in the known list to be considered as long as they trigger at least one unique error message. The final workflow for deduplication of crash and non-compliance generation bugs can be viewed in figure 3.7.

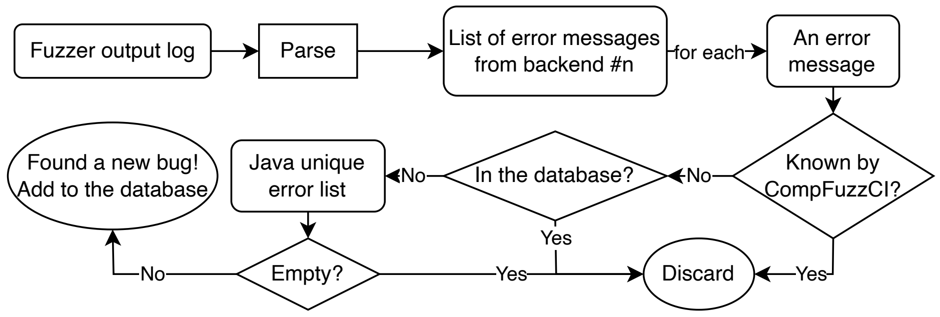


**Figure 3.7:** The workflow for deduplication of crash and non-compliance generation bugs

One limitation that we could not overcome is how to handle variable names and other test program-specific information in the error message. It would be good for the deduplication module to be able to symbolise the error message so that the names can be uniform. This is partially possible when names or types are put in a quote in the error message, but it is not always the case. For example, the error message in figure3.8.

```
# command-line-arguments
/main-go/src/main.go:49:12: goto Co jumps over declaration of _1_v_int_93 at main-go/src/main.go:51:9
```

**Figure 3.8:** Example of error message with variable name

## Deduplicating miscompilation bugs

Miscompilation bugs are very different from crash and non-compliance generation bugs. The program that triggers miscompilation does not contain any faults and, therefore, provides no error messages. The only way we could detect a miscompilation is by manual investigation or by using differential testing and comparing the results from different backends, as fuzz-d has done. We know that there is a miscompilation when there is a mismatch of results from the backends or between the backend and the fuzz-d interpreter. Depending on where the miscompilation bug is located in the compiler, different mismatches can come from the same bug. For example, a miscompilation bug located within a function that is responsible for flattening datatypes could cause nested arrays, multi-level maps, and other structures of the same nature to have mismatching results. We cannot automatically determine the root cause of the miscompilation bug by looking at the output.

The first idea on how to deduplicate miscompilation bugs is to reduce the test program and use the distance function, similar to what is done by Chen et al. [28], to calculate the similarity of the reduced test program to the other programs that cause miscompilation bugs in the database. However, this approach soon proved to be ineffective. The reason is that reduction of miscompilation bug programs always takes extremely long. The complete reduction is often not possible within the 1-hour time frame. Even if we use a timeout on the reducer, we will end up with a large program containing unrelated lines of code, which will throw off the distance function.

Following this realisation, we decided to rely on the bug-introducing commit to be the signature of a miscompilation bug. This comes from an assumption that one commit will likely not introduce more than one miscompilation bug. This assumption might not always be true, but it is a good enough approximation. As we will see in section 3.7, bisection also takes around 40 minutes in the worst case. This gives us more chance to identify a bug before the 1-hour time constraint runs out.

## The interestingness test

By general definition, the interestingness test [25] is the test used in automated test program reduction to determine if the bug still exists in the reduced program. In CompFuzzCI, we also use the interestingness test in the bisection script to determine if the bug exists at the current commit. Different interestingness tests are needed for different bugs. In CompFuzzCI, the test for each bug is generated after it has been deduplicated and contains the checks for unique error messages that survived deduplication. CompFuzzCI also uses the same interestingness test for test program reduction and bisection of the same bug. The fact that the interestingness test contains only error messages that survived deduplication means that the test is focused only on the new error, so reduction can be made smaller, and bisection will have less noise to handle.

Within the interestingness tests, we need to repeat the later part of the fuzzing process: compiling to the target language, executing the compiled files, capturing the output, and comparing the result for the mismatch. To centralise these functionalities, we extended fuzz-d with the “validate” option. The validator reuses fuzz-d’s test harness in a pickier and prettier way. This option will take as input the Dafny program and the target (C#, Java, JavaScript, Python, Go, Rust, Dafny and miscompilation). The target will determine which test harness will be used; this helps eliminate unnecessary work. The validator will also prettify the output of the test harness, printing it in a format that the bug-tracking module will later use to report the bug. Validation is always performed as the first step of the interestingness test.

## 3.6 Reduction

Our work on test program reduction in CompFuzzCI mainly experiments with different strategies to optimise for size, correctness and time. There is no language-specific reducer for Dafny, therefore our only option is to use language-agnostic reducers. We chose two reducers to work with: Perses and C-Reduce. Perses is chosen because it is a language agnostic reducer that has proved to perform well in the work by Usher et al. [6]. Perses also takes in ANTLR grammar, which is the grammar already used by the fuzz-d program generator, making it easier to integrate and update. C-Reduce is chosen because it is known to be fast, and it has many different heuristics that can be used in a language-agnostic way. This section will discuss the different ways we have used these reducers and the results we have obtained.

- **Using Only Perses:** Perses is a good language agnostic reducer because it is syntax-guided and will always produce minimal code that is syntactically correct. The problem with Perses is the fact that it can be quite slow, a lot of times exceeding the 1-hour time frame, and the worst case we have experienced is 3 hours.

- **Using Only C-Reduce:** C-Reduce heuristics for language agnostic reduction are removals of objects in the code, such as lines and tokens. C-Reduce is very fast, but it is not syntactically-aware. The minimal program from C-Reduce often contains the bug trigger but also a lot of syntax errors underneath. This might seem counter-intuitive because the syntax error would have shown up during the interestingness test, but the compiler often stops and crashes due to the bug before it even reaches the syntax error. Our concern is that this could be misleading for the developer who is trying to understand the bug.

- **Using Perses and C-Reduce together:** Given the strength of both reducers, we tried to combine them by alternating them. The result in the good case is that C-Reduce can remove a lot of unrelated code, and Perses can then reduce the smaller code to its syntactically correct minimum in a much shorter time. However, this is rarely the case. Most of the time, C-Reduce will produce a code that Perses cannot reduce further because of parsing errors. We could fix this by saving multiple versions of the reduced program from C-Reduce and trying to use the syntactically correct one. However, the overhead of starting Perses and trying to parse multiple programs is still high, although the result is still better than using Perses alone. We keep this as a backup option and continue to explore other approaches.

- **Using Perses with timeout for crash and non-compliance generation bugs:** From our observation, we find that Perses can reduce crash bugs in a decent time, consistently less than 30 minutes. But reduces very slowly on miscompilation bugs. Perses handles crash bugs faster than the two reducers combined, and the minimal program will be syntactically correct. We have decided to use Perses with a timeout of 30 minutes for crash and non-compliance generation bugs. If the reduction is timed out before completion, the reduced programs are often small enough to be reduced further by the developer manually.

- **Using C-Reduce with timeout for miscompilation bugs:** The final problem we still have is miscompilation bugs. We observed that C-Reduce could handle miscompilation much faster than Perses due to its aggressiveness. The problem C-Reduce has with syntax error is not present when it comes to miscompilation bugs because miscompilation bugs requires the programs to be compiled and executed properly. Any reduction candidate containing syntax error will cause a crash and be discarded. Although, the timing is not always consistent due to the randomness of test programs. Circling back to fuzz-d's test programs, the miscompilation is surfaced by the print statements that are added to the program. C-Reduce consistently discards all print statements unrelated to the miscompilation within 30 minutes. The resulting program, although not always minimal, is small enough for the developer to trace the program variable in the print statement and reduce the program further manually. We have decided to use C-Reduce with a timeout of 30 minutes for miscompilation bugs.

Finally, we have a strategy for reduction that can produce an acceptably minimal program within the 1-hour time frame for every type of bug. As we have seen in reduction and deduplication, the crash and miscompilation bugs are very different in nature and require different handling. This further highlights the need for a CI testing framework specialising in compiler bugs.

## 3.7 Bisection

Bug bisection is a process of finding the commit that introduced the bug. This is a functionality offered by Git. It takes in the input of the known bad commit (has bugs) and known good commit (no bugs). It then finds the commit and introduces the bug by checking out, building, and testing the commits in a binary search manner. The process is done in log(n) time, where n is the number of commits between the known good and bad commits.

Bisection is needed for the bugs in Dafny because 1) the Dafny compiler is complicated software and 2) the bug can be very old. Hence, the context might already have been lost, and is hard to be gained right away. Knowing which chance introduced the bug will greatly help developers in recovering the context of the bug. This section will discuss the challenges we faced in implementing automated bisection and how we designed its solution.

## When to run bisection?

Initially, we want to bisect every bug we find. However, we soon realised that the bisection process is very time-consuming. One iteration of testing will take about 4 minutes; most of the time is spent building the Dafny compiler from the commit. The number of commits between the latest commit (October 2024) and the earliest suitable commit (section 3.7) in the Dafny history is 1784 commits. The bisection algorithm could run, in the worst case, roughly 10 times to find the first bad commit. Thus, bisection will take 40 minutes in the worst-case scenario. We have taken this problem to the Dafny team, and they have informed us that not every bug will need bisection.

The pull requests on the Dafny repository are created so that one pull request will only contain one functionality change/bug fix. If the bug is found in a pull request and it is from the head branch (usually, the Dafny master branch is the base branch), then the developer could find the erroneous code fairly easily. In this case, we will not need to do unnecessary work on bisection. Nevertheless, in the case where a miscompilation bug is found on the head branch, we will still need to bisect it because there are no other ways to make a bug signature.

The workflow deciding if bisection needs to be done is shown in the figure below. The “bisection limit” mentioned in the workflow will be discussed in section 3.7.
To better understand the workflow, we will discuss the 5 possible end states that the bisection module can fall into.

1. The bug is on the master branch and is introduced before the bisection limit. In this case, it is not possible to bisect the bug.

2. The bug is on the master branch and is introduced after the bisection limit. In this case, the bug will be bisected.

3. The bug is on the head branch, but it is inherited from the master branch and is outdated. This state is possible when the head branch inherits the bug from the master branch, but the bug is fixed on the master branch in a later commit, and the head branch is outdated with the master branch. In this case, the bug will be discarded because it has already been fixed.

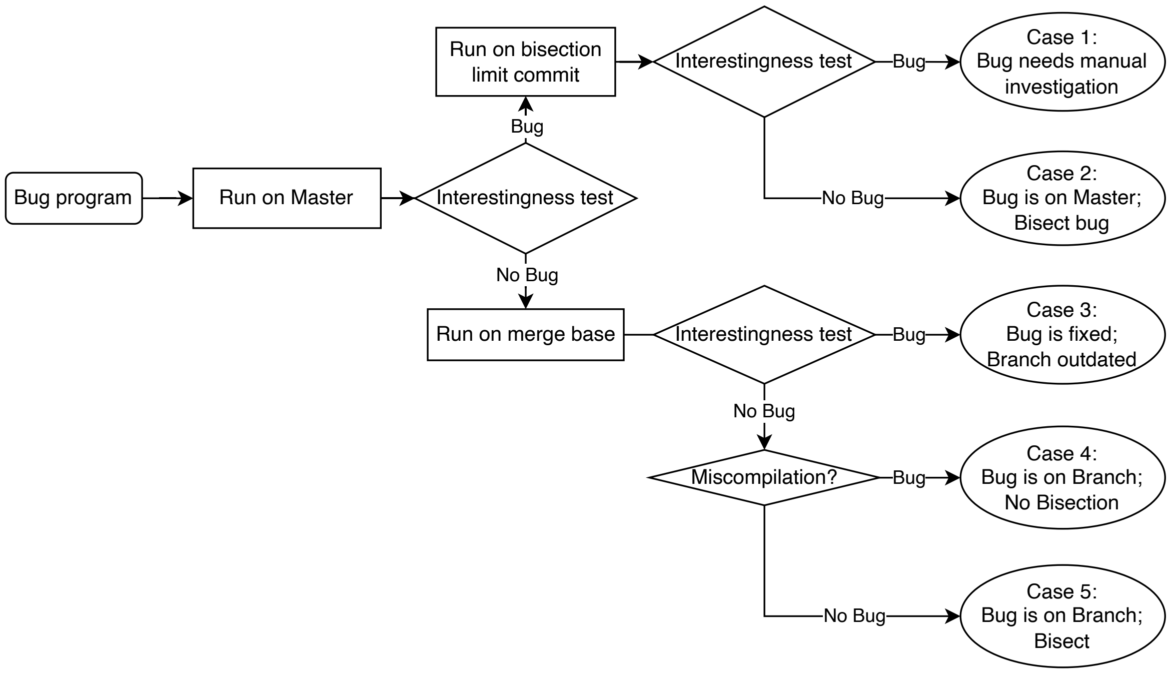

Bug program
Run on Master
Interestingness test
Bug
Run on bisection limit commit
Interestingness test
Bug
Case 1: Bug needs manual investigation
No Bug
Case 2: Bug is on Master; Bisect bug
No Bug
Run on merge base
Interestingness test
Bug
Case 3: Bug is fixed; Branch outdated
No Bug
Miscompilation?
Bug
Case 4: Bug is on Branch; No Bisection
No Bug
Case 5: Bug is on Branch; Bisect

**Figure 3.9:** The workflow of deciding if bisection needs to be done

4. The bug is on the head branch, is introduced on the branch and is a crash or non-compliance generation bug. In this case, the bug will not be bisected.

5. The bug is on the head branch, is introduced on the branch and is a miscompilation bug. In this case, the bug will be bisected.

The “merge base” mentioned in the workflow is the last common commit between two branches. For example, in figure 3.10, the merge base of the head branch and the master branch is commit B. It is possible that commit B has a bug that is fixed by commit C. Because the head branch is outdated, the bug would still be alive in the head branch. This is the reason why we need to check the merge base when the bug is on the head branch but not the master branch.

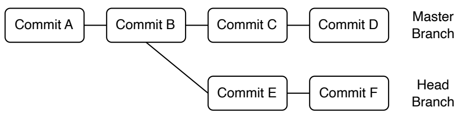


**Figure 3.10:** The merge base of the head branch and the master branch

## How to run bisection?

As mentioned in section 3.3, bisection needed to be done in a different container from the fuzzing container. We could run bisection on the one container, where Dafny is checked out and built multiple times, taking 4 minutes each. Another option is to try to cache each different version of Dafny in an image and run each step of bisection on different containers. To clarify this, we will look at the workflow for both approaches.

### Running bisection on one container

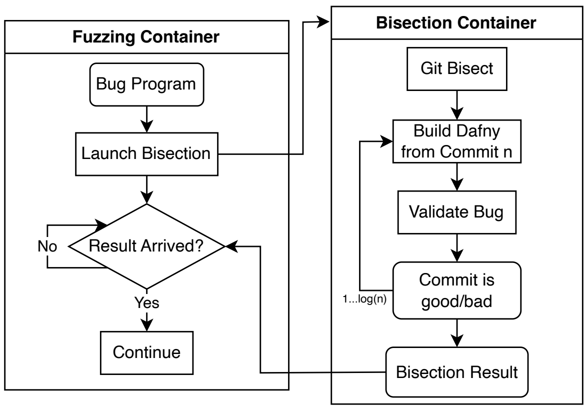


**Figure 3.11:** The workflow of bisection on the same container

This approach is very simple; it follows the general bisection process of repeatedly checking out, building and testing the commits. We wanted to try to optimise its speed by caching the built Dafny. Which leads us to the next approach.

### Running bisection on ready-built Dafny images

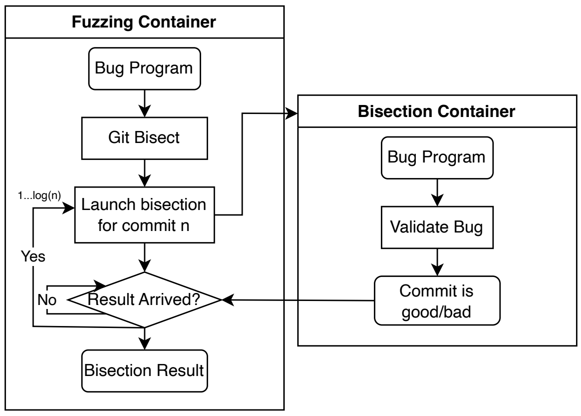


**Figure 3.12:** The workflow of bisection on ready-built Dafny images

Initially, we thought this approach would be faster because we would not have to build Dafny from the source every time. However, we soon realised that the overhead of downloading the Dafny image from the registry and provisioning and initialising the compute instance for the container could take longer than building Dafny. Only when multiple fuzzing containers launch bisection containers on similar sets of commits simultaneously does this approach yield an advantage. This is because the images would already be stored on the compute instance block storage, and there is no more overhead of downloading, provisioning and initialising. However, this advantage comes with a potential issue where the compute instance block storage runs out of space and crashes. We have tried to configure the compute instances to delete the least used image from the storage when the instance is not active. This helped with the storage issue, but not as much as we needed since instances are rarely inactive.

Our observation from several test runs showed that the run duration for the same container bisection is consistently less than 40 minutes. The timing for the ready-built Dafny image bisection is inconsistent, relying on many other factors. The advantage of ready-built Dafny images seldom outweighs the overhead. We have decided to run a bisection in the same container.

Another approach that might be better is to store the built Dafny in the S3 bucket and download the Dafny version that is needed for the bisection. This will reduce the time it takes to build Dafny but will have the overhead of downloading the Dafny version. We might also have to be careful since the information stored by Git during bisection on the Dafny repository could get overwritten and confuse the bisection process. Alternatively, we could have two Dafny repositories on the container, one running bisection and the other storing built Dafny. The system path will point to the Dafny repository containing the built Dafny. This approach could decrease bisection time further and should be explored in future work.

### The known good commit and the bisection limit

Now that we are ready to start bisection, our first difficulty in bisecting automatically is to find the known good commit. The known good commit will be used as the first lower boundary of Git Bisect's binary search. Currently, we do not have a way to determine the known good commit automatically. Choosing the earliest suitable commit has also turned out to be a complicated task.

While the process of bisection is simple in theory, the reality is that the Dafny repository has changed significantly over the years. These changes involved the build scripts, the command line interface, the output format of the compiler, and the options within the compiler itself. The challenge of automating bisection is to make sure the Dafny compiler is invoked in an equivalent way at every commit, no matter how far back.

Bisection is driven by the Git bisect algorithm and a script. The script will be run at every commit and will typically involve checking out the commit, building the Dafny compiler, and running the tests. In our case, that test is the interestingness test, which contains a call to the fuzz-d validator. Once we hit a significant change in Dafny, the output of the validator will be unpredictable, and the interestingness test will likely fail. This random failure will convince the Git bisect algorithm that the commit which introduced this significant change is the first bad commit, making bisection inaccurate and misleading.

We solved this problem by running multiple iterations of bisection for different bugs and investigating the first bad commit manually. If the first bad commit is incorrect, we investigate the changes made within that commit and modify the fuzz-d validator to adapt to these changes. We repeated this process until bisection became mostly accurate. However, there could still be more changes we have not yet encountered because they did not affect the set of bugs we used to test bisection.

We defined the bisection limit as the earliest commit where the fuzz-d validator output is predictable. We assign the bisection limit as the known good commit, as it is the earliest commit where we can run bisection accurately.

From our iterative investigation, we have identified 8 different ways to invoke the Dafny compiler as a result of the following changes in the Dafny repository:

**Version 4.5.0: Warning becomes error**

On version 4.5.0, the Dafny team decided to start treating compiler warnings as errors, which instantly terminate compilation. They have introduced a new flag –allow-warnings, to allow the compiler to continue compilation even when there are warnings. This means that from version 4.4.0 downwards, the flag will be unrecognised, and we must remove it. The validator is the component in the bisection module that is responsible for invoking the Dafny compiler. Therefore, we modify the validator to emit the build command without the "–allow-warnings" flag when the Dafny version is less than 4.5.0.

**Version 4.2.0: Python compilation output name change**

Pull request number 4345, which was merged into Dafny on version 4.2.0, fixes the problem where the Dafny file name is the same as a Python module. The fix changes the name of the Python output file, which has always been the same for the validator. To fix this problem, we modify the validator emit command executing file "main.py" instead of "_main_.py" from 4.1.0 downwards.

**Version 4.2.0: Default function syntax change**

Pull request number 3623, which was merged into Dafny on version 4.2.0, changes the default syntax version for functions. Dafny has two function syntax versions, namely 3 and 4. Functions are treated very differently in each version. The validator is modified to emit a command with the flag "–syntaxVersion:4" from 4.1.0 downwards.

### Version 3.11.0: Java compilation output format change

Pull request number 3355, which was merged into Dafny on version 3.11.0, changes the format of the Java output file. The Java output file is now a jar file instead of a class file. The validator is modified to emit a command to execute the jar file instead of the class file from 3.10.0 downwards.

### Version 3.10.0: New command line interface

The new CLI was introduced in version 3.9.0 but would not become usable until version 3.10.0. Soon after the new CLI was introduced, the old CLI was deprecated. The validator is modified to emit commands using the old CLI from 3.9.0 downwards.

### Version 3.8.0: Version went missing

After pull request number 2787, which was merged into Dafny on version 3.8.0, the version number went missing from the Dafny version command output for 21 commits. This is a problem because the validator uses the version number to determine how to invoke Dafny. Fortunately, this is the only occurrence where the version command is broken. We hardcoded the version number to 3.8.0 whenever the version command returns blank.

### Version 3.6.0: Gradle version upgrade

Pull request number 2060, which was merged into Dafny on version 3.6.0, upgrades the Gradle version for the Dafny Java backend. This is a problem because the Java version installed on the bisection container is incompatible with the older Gradle version. Changing the Java version would be problematic because the validator also uses Java. It is possible to patch the Gradle version from 3.5.0 downwards, but it will require modifying the Dafny repository at every step of the bisection process before 3.6.0.

### Version 3.3.0: .NET version upgrade

Pull request number 1642, which was merged into Dafny on version 3.3.0, upgrades the .NET version for the whole project. Patching the .NET version will be more tedious than the Gradle version because it is also used in every component of Dafny. Once again, it would require modifying the Dafny repository for every step of the bisection process before 3.3.0.

### Version 3.3.0 Downwards: Building Dafny from source gets tricky

The build scripts for Dafny change significantly from version 3.3.0 downwards. Making it increasingly difficult to write a script that can build Dafny from the source. We have decided that this is the bisection limit.

Although we decided that 3.3.0 was the limit, we stopped at 3.6.0 due to time constraints.

### Improving bisectability

The changes that were mentioned in the previous chapter all happened within 2.5 years of the Dafny compiler's development. This has raised a concern about the automated bisection of any codebase that is actively developed. If many significant changes to user interaction with the tool are made within such a short time, automated bisection will soon be overwhelmed by the number of conditions it needs to check before running the test.

Making automated bisection work in the long run will require maintenance effort from the developers. This could come in many forms depending on the codebase, such as maintaining the older version of the software, documenting the changes made that affect user interaction, making sure that the changes are backwards compatible, providing dependency patches, and many more. The developer will need to decide how much effort they are willing to put into maintaining the bisectability of the codebase. It could be that most bugs are obvious enough that their root cause can be determined with less accumulated effort than maintaining the bisectability of the codebase.

## 3.8 Bug tracking

The bug-tracking module performs multiple simple tasks that are crucial to CompFuzzCI's success. In this section, we will discuss each responsibility of the bug-tracking module and how we have implemented them.

#### Bug reporting

The bug tracking module is responsible for reporting the bug back to the Dafny repository on GitHub. It achieves this by having a lambda function that is triggered when bugs get uploaded to the bug report queue. The information available to the lambda function is the bug location (master/branch), the first bad commit (if available), the reduced test program, and a pretty printed program output log. This information will be structured into a markdown string in the standard format of the Dafny bug report. It is then filed as an issue if the bug is on the master branch or commented on the pull request if it is on the branch. This is done using GitHub API. After reporting the bug, the lambda function moves the bug from the queue into the bug database, tagging it with either a pull request number or an issue number.

#### Issue tracking

Sometimes the user or developer could find the bug before CompFuzzCI does. To keep track of the bugs found by the user and possibly find out more information about them, CompFuzzCI has a workflow that gets triggered by issues created by others on the Dafny repository. The workflow will parse the issue to get the problematic code and then launch a variation of the fuzzing container where the fuzzing is disabled. The problematic code is used as the bug program, which is then reduced and bisected. Our future plan is for the information gathered from the issue to be commented back on the issue itself on GitHub.

CompFuzzCI also monitors for issue closing events on the Dafny repository so the bug signatures with issue number tagged can be moved from active bugs to closed bugs in the bug database.

#### Pull request tracking

Similar to issue tracking, CompFuzzCI monitors pull request closing events on the Dafny repository. Whenever a pull request is closed, all the bugs tagged with the pull request number will be deleted. This is because the bug is either fixed or no longer relevant.

# Chapter 4

# Real world integration

We have integrated CompFuzzCI into the Dafny repository for 50 days. This chapter will discuss our achievements and the challenges we faced throughout the integration. The content of this chapter is the following. Section 4.1 outlines the collaboration with the Dafny team. Section 4.2 discusses the challenges we faced in fuzzing the Rust backend. Section 4.3 discusses the relationship between CompFuzzCI and the CI test suite. Section 4.4 lists the bugs we have found during the integration. Section 4.5 talks about the need to maintain the Dafny grammar for the reducer. Finally, section 4.6 analyse the cost and benefit of running CompFuzzCI.

Before exploring the content of this chapter, let us first clarify the notion of bugs. Technically, a bug is defined as a flaw in the software that causes it to behave unexpectedly or produce undesirable results. A flaw in the software that produces an error would be considered a bug, as it produces undesirable results. In CompFuzzCI, all errors and miscompilation are considered bugs, as they are undesirable results. Nevertheless, the definition of a bug in real software rather depends on the context and the developer's judgement. In this chapter, we will use the term 'bug' to describe the errors and miscompilation that are unexpected or undesirable to the developers. We use the term non-bug to describe the behaviour that the developers expect.

## 4.1 Interaction with the Dafny Team

This chapter would not have been possible if not for the collaboration and support of the Dafny AWS team. Before the integration, we held a meeting with the Dafny team to discuss expectations and preferences for an ideal CI compiler fuzzer. The insight gained from the meeting was invaluable and helped shape the design of CompFuzzCI. The Dafny team was also very supportive during the integration process, providing us with code reviews and helping with debugging the integration.

During CompFuzzCI integration with the Dafny repository, we held most of our interactions through discussion via GitHub issues and pull requests. They were very responsive and helpful in helping us understand the bugs that CompFuzzCI found. They helped us distinguish the (technical) bugs that were found in section 4.2 from what would be regarded as a real bug. They also provided us with suggestions on how to solve the problem we faced in section 4.3, which we later implemented. Every topic in this chapter has been discussed with the Dafny developers before being written. Therefore, many of the ideas and suggestions in this chapter are provided by the Dafny developers.

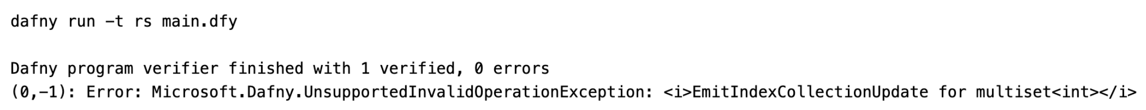
At the time of writing this report, we are still in the process of deciding the future of CompFuzzCI integration within the Dafny repository, which might involve further redesigning of CompFuzzCI architecture. Despite all the limitations of CompFuzzCI, the team is still interested and full of ideas on how to improve CompFuzzCI. One also expressed their positive attitude towards fuzzing Dafny on the CI as a helper of the CI test suite.

## 4.2 Testing the incomplete backend

At the time of the CompFuzzCI integration, the Rust backend was being actively developed. A lot of fuzzing is triggered on the Rust backend, which led us to find a lot of errors, none of which were truly regarded as bugs by the developers. The errors were mostly due to the incompleteness of implementation, of which the developer is aware. We know there could be a bug in the implemented part of the Rust backend, but we couldn't find it. Essentially, the unimplemented language features became our fuzz-blocker. This section will discuss some of the non-bugs that CompFuzzCI regarded as bugs.

### Unsupported Invalid Operation

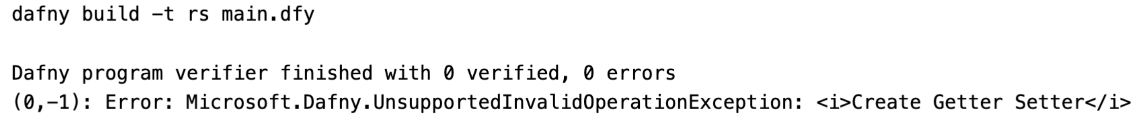
Unsupported invalid operation error is the first family of errors we found while fuzzing the Rust backend. This error is on the very surface level; it is thrown for every operation that has not been implemented in Rust.



```
dafny run -t rs main.dfy

Dafny program verifier finished with 1 verified, 0 errors
(0,-1): Error: Microsoft.Dafny.UnsupportedInvalidOperationException: <i>EmitIndexCollectionUpdate for multiset<int></i>
```

**Figure 4.1:** Unsupported invalid operation error for multiset in Rust



```
dafny build -t rs main.dfy

Dafny program verifier finished with 0 verified, 0 errors
(0,-1): Error: Microsoft.Dafny.UnsupportedInvalidOperationException: <i>Create Getter Setter</i>
```

**Figure 4.2:** Unsupported invalid operation error for getter/setter in Rust

After reporting these errors to Dafny developers, we were told that these are not bugs but rather features that have not been implemented yet. The Rust backend is still in development, and the developer is working on implementing the missing features. After receiving the feedback, we ignored any unsupported invalid operation errors and noted them as non-bugs. However, this type of error is what we would find most of the time when fuzzing the Rust backend. It blocks the fuzzer from finding any real bugs in the Rust backend.

## Missing implementation

After ignoring the unimplemented operations, we still found errors that acted like technical bugs but were simply missing implementation. One error was triggered while fuzzing the pull request that attempts to fix bug $\alpha$ that occurs when the user tries to assign a value to a constant twice. When the program is run in the master branch, it shows the same error message as the bug $\alpha$. However, when it is run on the pull request branch, it shows a different error (appendix 6).

This led us to think that this unexpected behaviour was introduced by some changes that were made in the pull request and is a bug. However, after reporting it to the Dafny developer, we were told that the error was also due to a different missing implementation. It seems that fixing one unimplemented feature has exposed another unimplemented feature.

## Lessons learned from testing the Rust backend

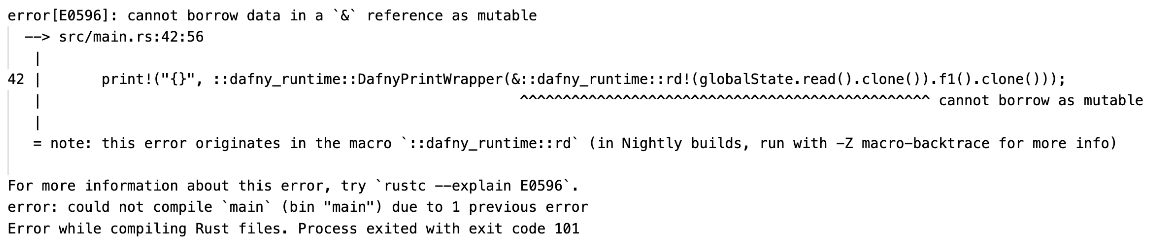
Depending on the situation, it might be appropriate to request the developer to ignore the bug reports. However, that does not solve the problem of the fuzzer being blocked by unimplemented features. This problem has raised the question, "When is the right time to start fuzzing?". Logically, we would want to test software while it is being developed. If the bugs can be found early, they can be fixed quickly, as the developer is still working on the code. Additionally, as mentioned by Donaldson et al. [64], fuzzing early can influence the language design and implementation choice. Fuzzing could identify potentially problematic implementation or missing cases in the language specification early on. However, testing too early could also be a waste of resources and distract the developer with unnecessary bug reports. The right time to start fuzzing will vary depending on the nature of the software. The Dafny compiler will need more exploration and discussion with the developers to reach a conclusion.

There are also things that can be done to the fuzzer and the compiler to ease the problem of unimplemented features. The fuzzer can be made to disable the unimplemented features during program generation. The developer could store a program generation configuration within their branch. This configuration file will contain probabilities for each language feature the fuzzer supports. The fuzzer will then use this configuration file to generate the program. This way, the developers can disable the unimplemented features. Additionally, developers working on specific features of the language can also increase the probability of that feature. This will make the fuzzer more customisable for developers without fuzzing knowledge. A solution on the compiler side, suggested by the Dafny developers during our discussions, is to make a uniform error message for all unimplemented features. This way, CompFuzzCI will not be misled by different error messages for different unimplemented features.

Even after implementing the fuzzer-side and compiler-side solution to unimplemented blockers, we should evaluate again whether the fuzzing would be worth the cost. As far as we currently know, the unit tests in the CI pipeline for the Rust backend have done a good job of catching the errors without having problems with unimplemented features at a lower computational cost.

## 4.3 Fuzzer vs CI testsuite

Another alteration we had to make to the CompFuzzCI workflow was to run the fuzzer after the CI tests had passed. The Dafny CI test suite includes various tests such as build tests, integration tests, and unit tests. For every single pull request that has failed CI tests, the fuzzer would also find a subset of the bugs found by CI tests. This is a waste of resources and time, as CompFuzzCI's role is to find bugs that complement the CI test suite.



```
error[E0596]: cannot borrow data in a `&` reference as mutable
  --> src/main.rs:42:56
   |
42 |       print!("{}", ::dafny_runtime::DafnyPrintWrapper(&::dafny_runtime::rd!(globalState.read().clone()).f1().clone()));
   |                                                       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ cannot borrow as mutable
   |
   = note: this error originates in the macro `::dafny_runtime::rd` (in Nightly builds, run with -Z macro-backtrace for more info)

For more information about this error, try `rustc --explain E0596`.
error: could not compile `main` (bin "main") due to 1 previous error
Error while compiling Rust files. Process exited with exit code 101
```

**Figure 4.3:** The bug is caught by both the fuzzer and the CI testsuite

The bug in figure 4.3 is caught by both the fuzzer and the CI test suite. The CI test suite found it in 20 minutes, while CompFuzzCI took around 45 minutes, with the reduction timed out before completion. This results in a non-minimum test program from the CompFuzzCI, containing many unnecessary statements. The test from the suite is simple and clear to understand. In this case, we conclude that the CI tests are doing a much better job at catching these easy-find bugs. Running the fuzzer to catch the same easy-find bug is not worth the time and resources.

The problem here is different from what the deduplication module is trying to solve. For deduplication, CompFuzzCI is working with bugs that were previously known. The bug that fails the CI test suite is unknown until test time. It is possible to check the CI after the fuzzer finds a bug. However, it is unpredictable whether the bug will be found by the fuzzer or by the CI tests first. We decided that it was not feasible to run the fuzzer alongside the CI test suite.

Our implementation now is to schedule a check 30 minutes after the pull request event. If, after 30 minutes, all CI tests have passed, CompFuzzCI will run. The flaw of this solution is that it relies on the CI test suite to finish within 30 minutes and that the test suite may not flake. Unfortunately, it could be that the code that was supposed to pass the test suite failed due to a flaky test or a random failure that occurred during the building step. In these cases, CompFuzzCI will not run, even if it was supposed to. This is the risk that we have accepted in order to avoid duplicating bugs with the CI tests.

## 4.4 Bugs found

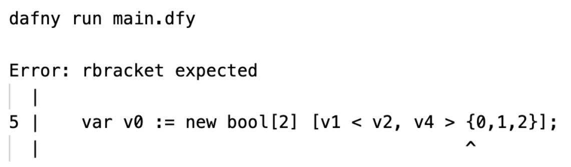
This section looks at the bugs found by CompFuzzCI during the integration with the Dafny repository, focusing on the bugs that were confirmed to be useful by the Dafny developers.

### Parser error

The first bug found by CompFuzzCI is a parser error. It is triggered by the program in figure 4.4. This is an edge-case bug. The bug also requires a very specific expression in a specific order to trigger, which makes it unlikely that the user or the CI tests will find it. The fuzzer found this bug multiple times and could not deduplicate it well. The effect of parsing error could have an avalanche effect on the rest of the program, resulting in many different combinations of parsing error messages (unexpected symbol, invalid operation, invalid suffix, etc.).

```
method Main(){
   var v1 : int := 0;
   var v2 : int := 1;
   var v4 : set<int> := {0};
   var v0 := new bool[2] [v1 < v2, v4 > {0,1,2}];
}
```

**(a)** The test program



```
dafny run main.dfy

Error: rbracket expected
  |
5 |    var v0 := new bool[2] [v1 < v2, v4 > {0,1,2}];
  |                                     ^
```

**(b)** The resulting error message

**Figure 4.4:** The parser bug

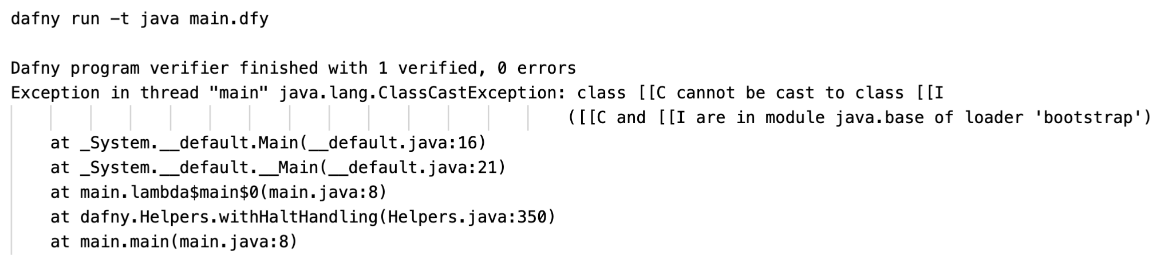
The developer confirmed that it is indeed a parsing error, where the element should be parsed as a relational expression; it was parsed as type parameters. The bug also existed before Dafny version 3.3.0, the bisection limit. This means that the bug is likely not a regression and the developer would have to find the cause of the bug manually.

## Nested array translation to Java

The second bug found by CompFuzzCI is related to an older bug report. This bug, and the older bug, concerns the translation of nested arrays to Java. However, the array creation expression in our bug is different from the older bug. The bug is triggered by the program in figure 4.5.

```
method Main() {
  var v2 := new char[1];
  var v3 := new array<char>[1][v2];
}
```

**(a)** The test program



```
dafny run -t java main.dfy

Dafny program verifier finished with 1 verified, 0 errors
Exception in thread "main" java.lang.ClassCastException: class [[C cannot be cast to class [[I
                                                           ([[C and [[I are in module java.base of loader 'bootstrap')
    at _System.__default.Main(__default.java:16)
    at _System.__default.__Main(__default.java:21)
    at main.lambda$main$0(main.java:8)
    at dafny.Helpers.withHaltHandling(Helpers.java:350)
    at main.main(main.java:8)
```

**(b)** The resulting error message

**Figure 4.5:** The nested array translation bug in Java

The developer confirmed the bug and suspected it might be a subset of the older bug. However, all bug-triggering statements from the older bug report are currently not producing any errors. We are still waiting for the developer to confirm its relationship to the older bug.

## Variable identifier comparison

The third bug found by CompFuzzCI is a unique identifier comparison bug. The bug is triggered by the program in figure 4.6,. This bug is introduced fairly recently by pull request number 5669. This changes the scope of the generated variable identifier used for compilation.

The developer has not confirmed the bug yet. However, the bug program behaved normally before this pull request, and the stack trace shows that the problem occurred during the comparison of the new variable identifier. Therefore, it is likely that the pull request causes the bug.

```
method Main(){
    var i8 := 4;
    var v1 := multiset{1,2,3,4};
    var v2 := map v3 : int | v3 in v1 ::
                if (| seq(1, i8 =>0) | in v1)
                    then v1[| seq(1, i8=>8) |] else 0 := 0;
}
```

**(a)** The test program

```
dafny run crash.dfy

Process terminated. Assumption failed.
      at Microsoft.Dafny.Triggers.ExprExtensions.ShallowEq(LambdaExpr expr1, LambdaExpr expr2)
      at Microsoft.Dafny.Triggers.ExprExtensions.ShallowEq(ComprehensionExpr expr1, ComprehensionExpr expr2)
      at Microsoft.Dafny.Triggers.ExprExtensions.ShallowEq_Top(Expression expr1, Expression expr2)
      at Microsoft.Dafny.Triggers.ExprExtensions.ExpressionEq(Expression expr1, Expression expr2)
      ...
      at System.Threading.Thread.StartCallback()
   /dafny/Scripts/dafny: line 35: 87808 Abort trap: 6          "$DOTNET" "$DAFNY" "$@"
```

**(b)** The error from the bug program

**Figure 4.6:** Variable identifier comparison causing a crash

## 4.5 Outdated grammar

One problem that became apparent once we started processing user-reported bugs for the bug tracking module is that Perses often struggled to parse the program. The Dafny ANTLR grammar was last updated in May 2023 by the developer of fuzz-d. It is also tailored to the fuzz-d supported feature, making it an incomplete grammar for dealing with real-world programs.

A very good suggestion from the Dafny developers is to reuse the Coco/R Wirth syntax notation that the Dafny developers regularly update. Coco/R is a compiler generator which takes in Wirth syntax notation and generates a parser for the language. In the future, it would be better for a compiler fuzzer or program reducer to be integrated with part of the compiler that is responsible for the grammar of that language. This would lessen the cost and effort of updating the fuzzer and reducer following language updates.

In our case, Coco/R is impossible because fuzz-d and Perses need ANTLR grammar. It would also be good to look into grammar translators that can convert Coco/R notations to ANTLR notations. Since both are deterministic context-free grammar, it should be possible to convert one to the other.

On the other hand, regularly updating ANTLR grammar for the fuzzer and the reducer, despite requiring more effort, also has its benefits. It encourages the developer to review the grammar, which could lead to the discovery of bugs in the grammar or ideas for improvement.

## 4.6 Cost-benefit analysis

In the previous sections, we have discovered many hidden costs of integrating CompFuzzCI into the Dafny repository. The hidden cost comes in terms of efforts needed to decide the right time to start fuzzing, maintain the compiler for automated bisection and maintain the grammar for the reducer. There could also be more need for maintenance in other aspects of CompFuzzCI that we have not yet encountered. It is currently unclear how much effort is needed to do these tasks and how much it would cost. Due to this lack of clarity, we have decided to do a cost-benefit analysis based on the information we have currently. However, these less tangible costs are still **extremely important** to consider when deciding whether to integrate fuzzing into the CI pipeline.

In the 50 days of CompFuzzCI integration with Dafny, there have been a lot of changes to CompFuzzCI itself, including a period where CompFuzzCI was disabled to fix bugs. However, the number of times Dafny invoked CompFuzzCI is recorded throughout. This information, along with AWS cloud pricing, is used for the cost estimate.

The information we used to calculate the cost estimate is the following:

- CompFuzzCI was invoked by the Dafny developers approximately 6 times a day (workday only).
- CompFuzzCI runs on Amazon EC2 t2.medium instances with 2 vCPUs and 4GB RAM. The cost of running this instance on-demand is $0.0464 per hour.
- The EC2 instances need internet access to download/upload artefacts. The cost of the virtual private cloud (VPC) is $0.005 per hour.
- CompFuzzCI runs 20 instances of the fuzzing container for each invocation.
- Each instance of the fuzzing container occupies 2 vCPUs and 4GB RAM (1 t2.medium).
- Each instance of the fuzzing container runs for 1 hour.
- On average, CompFuzzCI will launch another 4 bisection containers for each invocation.
- Each instance of the bisection container occupies 1 vCPU and 2GB RAM (2 bisection containers per 1 t2.medium).
- Each instance of the bisection container runs for 40 minutes, worst case.
- Other uses of AWS services, such as S3 bucket and EBS, are within the free tier.

From this information, we can calculate the daily cost of CompFuzzCI as shown in table 4.1. Assuming that there are 20 workdays in a month, the monthly cost estimate of CompFuzzCI will be $130.80. However, the cost could be lower if the Dafny team could invest in reserved instances or long-term saving plans.

| **Category** | **Explanation** | **Cost** |
|---|---|---|
| Fuzzing | 6 invocations x 20 instances x 1 hr x $0.0464/hr | $5.57 |
| Bisection | 6 invocations x 2 instances x 0.67 hr * $0.0464/hr | $0.37 |
| VPC | 6 invocations x 20 instances x 1 hr x $0.005/hr | $0.60 |
| Total | Total cost per workday | $6.54 |

**Table 4.1:** Daily cost estimate of CompFuzzCI

At present, we lack sufficient information to determine the monetary benefit of CompFuzzCI, as it would vary based on the compiler's use case. If Dafny is employed to develop critical software, where an uncovered edge case could result in a bug lawsuit, then integrating CompFuzzCI would be advisable.

# Chapter 5

# Simulation with Known Bugs

Our experience with the integration of CompFuzzCI into the Dafny repository has given us a lot of insight into the real-world value of CompFuzzCI. However, the bugs found during the integration are not enough to evaluate its effectiveness. We have decided to simulate CompFuzzCI with known bugs to quantify its effectiveness.

In this evaluation, we want to answer the following questions:

1. How many known bugs can CompFuzzCI find? Are the bugs found consistently? Answered in section 5.3.
2. What percentage of the lines changed by bug-introducing pull requests were covered by fuzz testing? Answered in section 5.4.
3. How long should the fuzzer run on the CI pipeline? Answered in section 5.5.
4. How effective is the CompFuzzCI bug deduplication strategy? Answered in section 5.7.
5. How effective is the CompFuzzCI reduction strategy? Answered in section 5.8.

Additionally, section 5.1 walks through the process of selecting bugs for the simulation. Section 5.2 describes the evaluation setup. Section 5.6 analyses the result from section 5.3 and 5.4, looking deeper into the factors that determine if a bug would be found during fuzzing.

## 5.1 Selecting the known bugs

The process of selecting a known bug involved both automated script and manual investigation. There were 2727 issues in the Dafny repository, and out of this, we ended up with 20 known bugs for the simulation. The process of selecting bugs is as follows:

1. We first discarded the open issues, only focusing on the closed issues. This is because developers have confirmed bugs in the closed issues. This left 1643 issues.

2. Using an automated script, we ignored issues related to another part of Dafny that were out of the scope of CompFuzzCI. For example, VSCode integration, Dafny test generation, etc. We also discarded issues that existed before Dafny version 3.10.0 because the fuzzer could not generate programs that were compatible with it. This left approximately 250 issues.
3. Using the fuzz-d's Dafny ANTLR grammar to automatically parse the code in these issues, we filtered out the issues that cannot be parsed, assuming it will be impossible for the fuzzer to generate these codes. This left approximately 70 issues.
4. We manually investigated and filtered out the issues that were caused by the same bugs but put in separate issues for visibility. We also filtered out issues related to file names. This left 50 issues.
5. Using an automated bisection script, we filtered out the bugs that we could find in the introducing PR. The reason for filtering out these issues is that we need to know the changes made in the bug-introducing commit in order to profile the change coverage. This left us with 20 bugs corresponding to 13 bug-introducing pull requests.

## 5.2 Evaluation setup

The setup for the simulation is as follows:

- We will run CompFuzzCI on the 13 bug-introducing pull requests to see if it can find the bugs.
- CompFuzzCI will run 10 instances of the fuzzing container for each pull request.
- We will run the simulation with 10 repetitions for each pull request to test the consistency of CompFuzzCI.
- We will run each repetition for 2 hours in total, measuring the result every 30 minutes.
- All CompFuzzCI instances will run on Amazon EC2 t2.medium instances with 2 vCPUs and 4GB RAM, the same as in the real world.

**Limitation 1:** Due to the way evaluation is set up, there is no way for us to also evaluate the bisection module. This is because all of the bugs if found, will fall into case 4 in figure 3.9 and will not be bisected.

**Limitation 2:** None of the 20 bugs selected for evaluation are miscompilation bugs. This is not a design choice but rather a limitation of the bugs found in the Dafny repository. There were a few miscompilation bugs left in the set of bugs at step 4 of the bug selection process; unfortunately, all of them existed before 3.6.0. This means that we have no automatic way of confirming that the bugs found during the simulation are the bugs we are looking for. Doing manual checks for all the miscompilation bugs found is not feasible.

**Limitation 3:** Not having any miscompilation bugs in the known bug set implies that the evaluation of the reduction module and the deduplication module will be incomplete since it is missing half of the use case. This is a limitation we accepted and noted.

## 5.3 Bugs found

During the simulation, CompFuzzCI would run normally, except for the final step of reporting the bug. Instead of reporting the bug to the Dafny repository, the bug will be stored in a database. After the simulation is done, we will compare the bugs found by CompFuzzCI with the known bugs. The result of the simulation in terms of bugs found is shown in table 5.1.

When the bug's status is "Not always found", this means that out of 10 repetitions of the simulation, the bug was not found in all repetitions. The time to discovery is the longest time it took for CompFuzzCI to find the bug for the first time. We can see that CompFuzzCI only found 6 out of 20 bugs. This does not come as a surprise, as in section 5.6, we will combine the coverage information and the bug characteristics to explain why many of the bugs were not found.

## 5.4 Change coverage

When we talk about the change coverage, we are referring to the line coverage of the changes made in the bug-introducing pull request. Ideally, we would also want to measure the branch coverage, but getting the branch information from the PR is not as straightforward as getting the line information. We have decided to measure the line coverage only for now.

To gather the line coverage information during CompFuzzCI runs, we will need to instrument the Dafny binary files; we have achieved this using Coverlet tool [65]. Due to the overhead of instrumentation, running coverage along with CompFuzzCI will affect its performance and, therefore, the time it takes to find the bugs. We have decided to let CompFuzzCI run without coverage and upload all the generated test programs to a bucket. After the simulation is done, the bucket will be sampled for coverage. We had to sample the bucket because the number of test programs generated by CompFuzzCI is too large to be processed by Coverlet in a reasonable amount of time.

| **Issue number** | **PR introduced** | **Status** | **Time to discovery** (within) |
|---|---|---|---|
| 3343 | 2734 | Not found | |
| 3472 | 2734 | Not found | |
| 3658 | 2734 | Not found | |
| 3691 | 3479 | Not found | |
| 3922 | 2734 | Not found | |
| 3987 | 2734 | Not found | |
| 4000 | 3909 | Not found | |
| 4004 | 3886 | Not found | |
| 4007 | 2646 | Always found | 30 minutes |
| 4686 | 4591 | Always found | 30 minutes |
| 4894 | 3886 | Not found | |
| 5238 | 3623 | Not found | |
| 5283 | 4136 | Not found | |
| 5523 | 5474 | Not found | |
| 5569 | 2241 | Not found | |
| 5572 | 5528 | Not found | |
| 5642 | 5591 | Not always found | 90 minutes |
| 5643 | 5390 | Always found | 30 minutes |
| 5700 | 5390 | Always found | 30 minutes |
| 5701 | 5390 | Not always found | 60 minutes |

**Table 5.1:** Bugs used in the simulation and their status

The lines changed for each pull request are retrieved from the GitHub API. Then, the lines changed, and line coverage was compared to get the percentage of changes covered by CompFuzzCI. The percentage of changes covered by the fuzzer at each 30-minute time interval is shown in table 5.2.

## 5.5 Fuzzing duration

The information gathered from the two previous sections supports the idea that the optimal duration for fuzzing in the CI pipeline is 30 minutes. In table 5.1, we can see that the bugs that have "Always found" status were also always found within 30 minutes. This is further supported by the fact that the change coverage percentage rarely increases after 30 minutes, and there was no increase in coverage after 90 minutes.

This is not to say that no more bugs can be found after 90 minutes. As mentioned in the background chapter, the recommended fuzz testing duration is 24 hours. It is possible that if we had run CompFuzzCI for longer, some of the bugs that were not found could have been found, and the change coverage could have increased. However, CompFuzzCI is constrained because it is a part of the CI pipeline. Running

| PR number | Coverage at 30m | 60m | 90m | 120m |
|---|---|---|---|---|
| 2241 | 11.96% | | | |
| 2646 | 18.40% | | | |
| 2734 | 26.40% | | | |
| 3479 | 19.52% | | | |
| 3623 | 0.27% | | | |
| 3886 | 18.31% | | | |
| 3909 | 17.15% | + 0.07% | | |
| 4136 | 7.79% | | | |
| 4591 | 43.34% | | | |
| 5390 | 2.37% | | + 0.04% | |
| 5474 | 11.27% | | | |
| 5528 | 25.91% | + 0.10% | + 0.09% | |
| 5591 | 12.76% | | + 0.04% | |

**Table 5.2:** The percentage of changes covered by the fuzzer at each 30-minute time interval

for longer than 2 hours would be unacceptable for the developers since it would slow down the development process. Therefore, it makes sense to run CompFuzzCI for 30 minutes and then stop, leaving the rest of the bugs to be found by possibly running CompFuzzCI nightly.

Our fuzzing of the Dafny compiler is not the only case where fuzzing for 30 minutes turns out to be the optimal duration. As mentioned briefly in the Background chapter, Klooster et al. [61] had also tested for the optimal fuzzing duration for the CI pipeline. Using AFL++, Honggfuzz, and libFuzzer on the Magma fuzzing benchmark [66], their result also showed that the number of bugs triggered increased significantly in the first 10 and 30 minutes. The next significant increase in the number of bugs triggered is at the 4-hour mark, which is already too long for the CI pipeline. Therefore, Klooster et al. recommended fuzzing for 10 minutes for time-constrained CI pipelines and, ideally, 30 minutes if the CI pipeline can afford it. This is consistent with our result.

## 5.6 Bug characteristics

From the previous sections, it seems that the change coverage cannot be a direct indicator of whether a bug would be found. None of the five bugs that were introduced by PR 2734 were found, even though the change coverage was the second highest (26.40%). Contradictorily, all three bugs introduced by PR 5390 were found, even though the change coverage was the second lowest (2.37%).

This contradiction can be explained by the fact that all five bugs introduced by PR

2734 were much more complex than the bugs introduced by PR 5390 (that we know of). The changes in PR 2734 were made on a specific language structure, and the bugs were triggered by a combination of nested structures involving the changed structure. The pull request 5390, however, introduced a set of new features for the newly added Rust backend. The bugs we knew of and found from this PR were all related to the lack of implementation of some part of the new feature, which is considerably easy to find.

A lot of the bugs that CompFuzzCI never found fall into the category of having complex triggers, like the bugs introduced by PR 2734. These bugs could still be found if we had run CompFuzzCI for longer. However, CompFuzzCI will never find some bugs until it integrates a fuzzer that can handle different bug characteristics. We discuss some of these bugs in the following subsections.

### Issue #5283: Go reserved keyword not properly escaped

The bug from issue number 5283 (figure 5.1) is caused by the Go reserved keyword ‘fmt’ being used as a module name. The fmt is a standard library in Go that is used for formatting. CompFuzzCI is not likely to find this bug because the names that can trigger this bug are very specific. If the naming scheme of fuzz-d generated test programs were more random, there could be a chance of finding this bug. However, the naming scheme of fuzz-d is structured to be more readable. Modules generated by fuzz-d would be named ‘M0’, ‘M1’, ‘M2’, etc.

```
module fmt {}

method Main(){
    print "done\n";
}
```

**Figure 5.1:** The bug-triggering program from issue 5283

This bug will never be found due to the design of the fuzzes’ variable naming scheme. For fuzzers with different naming schemes, this bug will have little probability of being found due to the specificity of the trigger.

### Issue #5523: Mishandling of ‘this’ outside of a class

The bug from issue number 5523 (figure 5.2) is caused by the change in pull request number 5474 to guard the keyword ‘this’ in a class from being modified in a tail-recursive function inside a class. The developer forgot to handle the case where the tail-recursive function could be outside the class and ended up guarding a non-existent ‘this’. CompFuzzCI will never find this bug because none of the fuzzers we currently have for Dafny can handle a program with any type of recursion. Generally, a program with recursion is hard to handle for a fuzzer, as it could lead to an infinite loop.

```
function Loop(xs: seq<()>): ()
{
  if |xs| == 0
  then ()
  else
    var _: bool -> bool := e => e;
    Loop(xs[1..])
}

method Main(){
    print Loop([(), ()]), "\n";
}
```

**Figure 5.2:** The bug-triggering program from issue 5523

This bug will never be found due to the fuzzes' inability to handle recursion. It could be found by fuzzers with recursion handling or fuzzers that do not expect program termination.

## Issue #5569: Order of declaration problem in generated Python code

The bug from issue number 5569 (figure 5.3) is caused by the declaration order in the generated Python code. In the bug-triggering code, the class C that extends trait Trait is declared before Trait. When this Dafny code is translated to Python, the declaration order stays the same, and the Python interpreter cannot find the Trait.

CompFuzzCI is not likely to find this bug, even though it is both simple and generic in nature. This is because fuzz-d has a strict declaration order in the generated code. The order is always datatype, trait, class, function, and method. While this gives the generated code a more readable structure, it also stops the fuzzer from finding this bug.

```
method Main() {}

class C extends Trait {}
trait Trait {}
```

**Figure 5.3:** The bug-triggering program from issue 5569

This is a bug that will never be found due to the design of the fuzzer's program generation. It could easily be found for fuzzers with different orders of declaration.

## 5.7 Deduplication

As mentioned above, we have achieved an incomplete evaluation of the deduplication and reduction modules. Evaluating the deduplication rate completely is challenging due to the lack of time and the nature of our deduplication strategy. Because our deduplication relies solely on the text processing of the error message, there are three different outcomes based on three different bug behaviours:

1. The bug triggers the same error message every time. CompFuzzCI can deduplicate this perfectly. For example, error messages with assigned code like E0308 in Rust can only mean that the argument to a function that crashes is not the right type.

2. The bug triggers different error messages every time. CompFuzzCI will store all the possible error messages that the bug can trigger over time and will finally be able to deduplicate them at some point in the future. For example, the parser error, which, depending on where the unexpected token is, can trigger different error messages.

3. The bug triggers error messages with either complex messages or very simple messages. CompFuzzCI is currently not able to deduplicate these bugs. For example, the bug in the Java backend generates code with incompatible type conversions. The error message has the types embedded in it, and the types can be infinitely different; being able to deduplicate this will be more complex. Another example of a bug that triggers a very simple message is the bug in the Go backend that generates code that with a problematic variable; the error message is often “undefined: variable name”, regardless of the cause that led to the variable being unknown at runtime. In the case where the bug triggers very simple error messages, we will need another type of information to deduplicate the bug.

The deduplication module performance could range from detecting all duplicates to detecting none, depending on the bugs present and the error message it generates. There are also multiple error formatting schemes that CompFuzzCI must handle, including the error message in the format of Dafny, C#, Java, Rust, JavaScript, Python, and Go. Making it even more challenge for the deduplication module.

While we cannot quantify the effectiveness of the deduplication module at the moment, we can safely state that the strategy used for deduplication is flawed. We will need more information than just the error messages to deduplicate the bugs consistently.

In our simulation, we calculate the successful deduplication rate shown in table 5.3 from the number of bugs successfully deduplicated out of all the bugs found. We evaluate the successful deduplication rate for each backend separately. This means that the rate for the C# backend would include any bugs found while compiling Dafny to C# and executing the C# code. Due to the relatively consistent error message formatting of C# and Python, the successful deduplication rate is high for these languages. This would also be the case for Rust once the Rust backend is free from the Dafny format error message related to unimplemented features. The language backend that is most problematic for the deduplication module is Go due to its short and simple error messages. The entry for JavaScript in the table is empty because no JavaScript backend bug exists in the range of pull requests we used for evaluation.

| Backend | Successful deduplication |
|---|---:|
| C# | 100.00% |
| Java | 90.13% |
| JavaScript | - |
| Python | 100.00% |
| Go | 48.33% |
| Rust | 53.00% |

**Table 5.3:** Successful deduplication rate for each backend

## 5.8 Reduction

Finally, we also collected information on the performance of the reduction module. The fact that only crash and non-compliance generation bugs were selected means that the reduction will only involve running Perses on the test programs generated by the fuzzer. CompFuzzCI always runs the reduction module with a 30-minute timeout. This resulted in the reduction consistently finishing in a reasonable time, regardless of the quality of the reduced program. The quality of the reduced program is judged in two ways: size and readability. However, because we are using only Perses (which is syntax-guided) in this simulation, the readability is not a concern. Readability would be a concern in the case where we use C-Reduce with miscompilation bugs. Therefore, we will only focus on the size of the reduced program in this evaluation.

We first categorise test programs into 3 groups based on their size:

- Small: Smaller than 30 KB in size
- Medium: 30-100 KB in size
- Large: Larger than 100 KB in size

The table 5.4 shows the average performance of the reduction module for each group. Before discussing the result, let us put the average size of the programs that were used to report bugs in the Dafny repositories in context. The average line count of the programs in the Dafny bug report is 13 lines, which is very small and readable. It would require the reduction to always be done until completion to achieve the same quality, which is not reasonable for CompFuzzCI, a slightly larger program could be allowed here for the bug report. The average reduced size of small programs, although not ideal, is still acceptable as it is close to the average size of the programs in the real-world bug report. The same could not be said for medium and large programs. The average reduced size of medium and large programs is not acceptable for filing a bug report, and reduced large programs are too big for manual inspection too.

| Group | Original line no. | Reduced line no. | Reduction % | Frequency % |
|---|---:|---:|---:|---:|
| Small | 468 | 58 | 88.06% | 45.28% |
| Medium | 1105 | 108 | 90.24% | 37.75% |
| Large | 5281 | 1399 | 73.52% | 16.97% |

**Table 5.4:** Averaged performance of the reduction module

Perhaps one solution here is to limit the size of the test programs generated by the fuzzer, as we can see that the fuzzer generates medium-large programs 54.72% of the time. Limiting the size might affect our chance of triggering a bug in one program, but it will increase the rate of fuzzing and processing, giving us more chance to generate a program that triggers the bug. We could also learn from ClusterFuzz by delaying bug reports until we find a bug-triggering program that is appropriate in size. This would help us maintain the quality of the bug report while we explore further options to make program reduction more efficient.

# Chapter 6

# Conclusion and Future Work

CompFuzzCI is a framework to automate fuzz testing of the Dafny compiler within the CI pipeline. We have successfully integrated CompFuzzCI into the Dafny CI pipeline, discovering many areas for improvement. We have also simulated CompFuzzCI with known bugs and quantified its effectiveness. Most importantly, we have shown the potential of compiler fuzzing in the CI pipeline and put into perspective the practical challenges and costs that come with it. In section 6.1, we will summarise our findings with respect to the questions posted in section 1.2. We will also discuss the improvements that can be made to CompFuzzCI and the future work around CompFuzzCI in section 6.2 and 6.3.

## 6.1 Summary of findings

### RQ1: How effective is CI-integrated compiler fuzzing at discovering bugs during development?

We answered this question in chapter 5 by simulating CompFuzzCI with known bugs. We found that CompFuzzCI could find 6 out of 20 bugs from 13 pull requests. The bugs that were found were found consistently within 30 minutes. CompFuzzCI can cover, at minimum, some part of the changes made in the bug-introducing pull request. However, coverage is not a direct indicator of whether the bugs will be found; bug characteristics and fuzzer capability play a big role in whether the bugs will be found. From our experience, CI-integrated compiler fuzzing is somewhat effective at finding bugs during a pull request, but it would become even more effective if it could also run nightly with fewer time constraints.

### RQ2: What is the developer experience of having compiler fuzzing in the CI pipeline?

In chapter 4, we find that the answer to this question is not simple. There are aspects of CompFuzzCI that the developer finds useful: the fuzzing helped find real production bugs that were previously unknown, and CompFuzzCI can also find the starting point of the bug. However, there are also aspects of CompFuzzCI that are not as helpful: the reporting of issues that were not considered real bugs in the new backend and the early struggle to deduplicate bugs with the integration tests. In the end, the attitude of the developer towards compiler fuzzing in the CI pipeline can be summarised as inspired. The highlight of this experience is discovering the complications that come with the realistic use of compiler fuzzing in the CI pipeline and finding solutions with the help of the Dafny team.

## RQ3: How could bug deduplication be done effectively?

In section 3.6, we showed that the deduplication strategy used in CompFuzzCI is not consistently effective. The deduplication module is only able to deduplicate bugs with error messages of a simpler, consistent format. The deduplication rate is high for C# and Python but low for Go. The evaluation was also partial, and the deduplication rate of miscompilation bugs would also rely on the effectiveness of the bisection module. We have learned that error messages alone are not enough to duplicate effectively, and will need more research to answer this question.

## RQ4: What are the challenges of automating bug bisection?

Although not involved during the evaluation due to the evaluation design, the bisection module has also gone through many iterations of testing and debugging. This iterative process is how we discover and improve the challenges of automated bisection. The challenges we found are mostly related to the changes in the way Dafny is invoked and the format of its output. We have overcome these changes by adapting the bisection module's interaction with Dafny. This, in turn, highlights the need for significant maintenance of the older versions of Dafny and the need to keep track of the changes within the repository to keep the bisection module up to date. In the end, we are very satisfied with the bisection module, and it is the first module to be made a standalone tool on the GitHub Actions platform for the Dafny repository.

## RQ5: Is the benefit of integrating compiler fuzzing into the CI worth the cost?

In section 4.6, we have estimated the daily cost and best estimated the benefit of running CompFuzzCI. The cost of running CompFuzzCI is currently $6.54 a day. It is possible to reduce the cost further, and it could also become clearer how much cost would be associated with the occasional maintenance of CompFuzzCI and compiler artefacts. We could not give a definite answer to this question at the moment, and the answer would depend on the compiler's use case. CompFuzzCI has shown that it is capable of finding bugs during development in the real world and in the simulation. If the compiler's use case is to compile critical software, where every edge case is valuable, then integrating compiler fuzzing into the CI pipeline would be highly advised.

## 6.2 Improvements to CompFuzzCI

- **Fuzzer:** Although CompFuzzCI is designed to suit fuzz-d, it is still possible to integrate XDsmith and DafnyFuzz. CompFuzzCI uses formatted information throughout its process; we can make small modifications to the way other fuzzers generate output reports or write a parser to convert the output to the format that CompFuzzCI can understand. Using multiple fuzzers will increase the chance of finding different types of bugs and increasing change coverage.

- **Deduplication:** The deduplication module is currently flawed, utilising only error message information. It could be interesting to experiment with including control flow information of the reduced test program in the bug signature. This could help deduplicate bugs that are caused by the same control flow but different error messages. We could also experiment with using machine learning to deduplicate bugs based on stack traces. This would require a lot of data, but we could start by using the data we have from the issues on the Dafny repository.

- **Bisection:** We could increase the range of bisection by implementing version patching; this will let us go down to 3.3.0. More investigation is also needed to find out what other changes were made below 3.3.0. This will help us find the root cause of more bugs. We could also explore more ways to cache the Dafny executables to speed up bisection.

- **Bug triage:** We have explored a lot of bug triage methods in the background chapter. However, we did not get to implement any of these methods. From our observations, each developer in the Dafny team has their own area of expertise; for example, one person is responsible for the Rust backend, another person is responsible for the Java backend, etc. We could use some text processing techniques to automatically assign the bug to the right person based on the error message.

- **Offer standalone modules:** The reduction and especially bisection module could be useful for the Dafny developers to use on their own. We could offer these modules as standalone tools that can be used on any Dafny program. We have agreed with the Dafny developers to make a standalone bisection module that will run on GitHub Actions since it does not need many computing resources and actions are more accessible to the developers.

## 6.3 Future around CompFuzzCI

### Customisable fuzzer

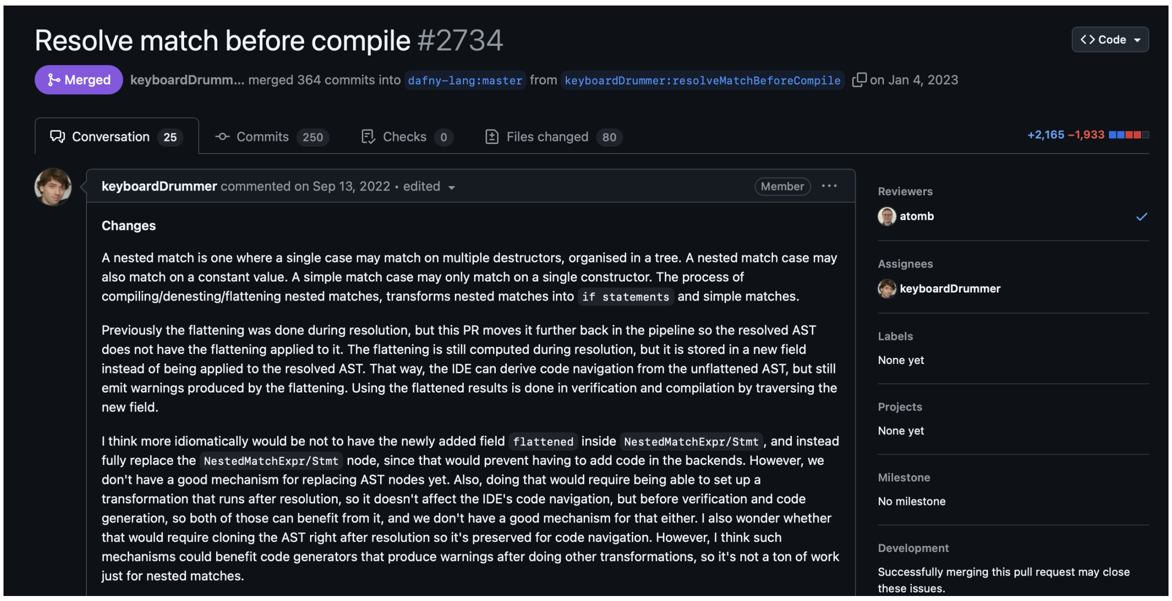
The problem faced during the fuzz testing of the incomplete Rust backend highlights the need for a fuzzer that can be somewhat controlled by the developer. It would not be difficult to implement such a feature in fuzz-d and any fuzzers that use probability configuration during program generation. The developer could store the probability configuration file within their development branch; they can decide which part of the language to enable and how often they want the fuzzer to generate a program with that feature. This will be beneficial in two ways:

- **Testing partially implemented feature or backend:** Recall that a lot of unimplemented errors in the Rust backend have been very easy to find for the fuzzer, stopping it from finding other bugs. If the developer could disable these unimplemented features by setting their probability to zero, it would increase the chance of the fuzzer finding real bugs in the implemented part of the backend. Disabling a feature should not cause problems within the fuzzer.
- **Testing a change to a specific feature:** When the developer is improving, redesigning, or fixing a specific feature, they could increase the probability of the fuzzer generating that feature within the program. This would possibly increase the change coverage and increase the number of test programs that could trigger the bug. However, we cannot guarantee that the modification of random probability will be handled well by the fuzzer. Unreasonable modification could lead to more problems than it solves.

### Change-coverage guided fuzzer

An idea that came from evaluating the change coverage of CompFuzzCI is to use the change coverage information to guide the fuzzer. For fuzz testing that is done within the CI pipeline, the goal should be to test all the changes made in the pull request. This is because the changes are the most likely the place where bugs will be introduced. Following change coverage coverage could also be a less demanding task than following overall coverage.

# Appendix



Resolve match before compile #2734

Merged keyboardDrumm... merged 364 commits into `dafny-lang:master` from `keyboardDrummer:resolveMatchBeforeCompile` on Jan 4, 2023

keyboardDrummer commented on Sep 13, 2022 • edited

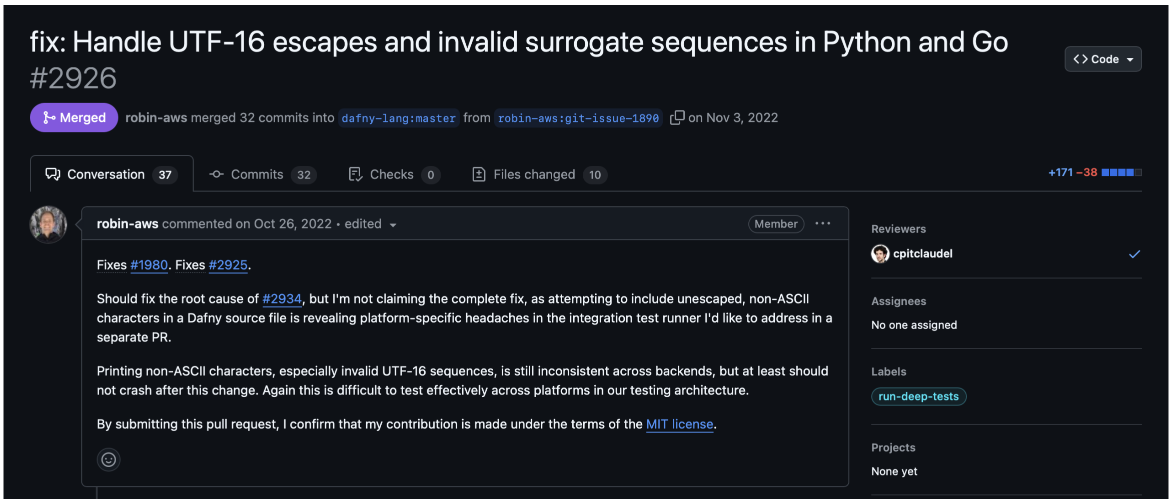
**Changes**

A nested match is one where a single case may match on multiple destructors, organised in a tree. A nested match case may also match on a constant value. A simple match case may only match on a single constructor. The process of compiling/denesting/flattening nested matches, transforms nested matches into `if statements` and simple matches.

Previously the flattening was done during resolution, but this PR moves it further back in the pipeline so the resolved AST does not have the flattening applied to it. The flattening is still computed during resolution, but it is stored in a new field instead of being applied to the resolved AST. That way, the IDE can derive code navigation from the unflattened AST, but still emit warnings produced by the flattening. Using the flattened results is done in verification and compilation by traversing the new field.

I think more idiomatically would be not to have the newly added field `flattened` inside `NestedMatchExpr/Stmt`, and instead fully replace the `NestedMatchExpr/Stmt` node, since that would prevent having to add code in the backends. However, we don't have a good mechanism for replacing AST nodes yet. Also, doing that would require being able to set up a transformation that runs after resolution, so it doesn't affect the IDE's code navigation, but before verification and code generation, so both of those can benefit from it, and we don't have a good mechanism for that either. I also wonder whether that would require cloning the AST right after resolution so it's preserved for code navigation. However, I think such mechanisms could benefit code generators that produce warnings after doing other transformations, so it's not a ton of work just for nested matches.

**Figure 1:** Pull request number 2734



fix: Handle UTF-16 escapes and invalid surrogate sequences in Python and Go #2926

Merged robin-aws merged 32 commits into `dafny-lang:master` from `robin-aws:git-issue-1890` on Nov 3, 2022

robin-aws commented on Oct 26, 2022 • edited

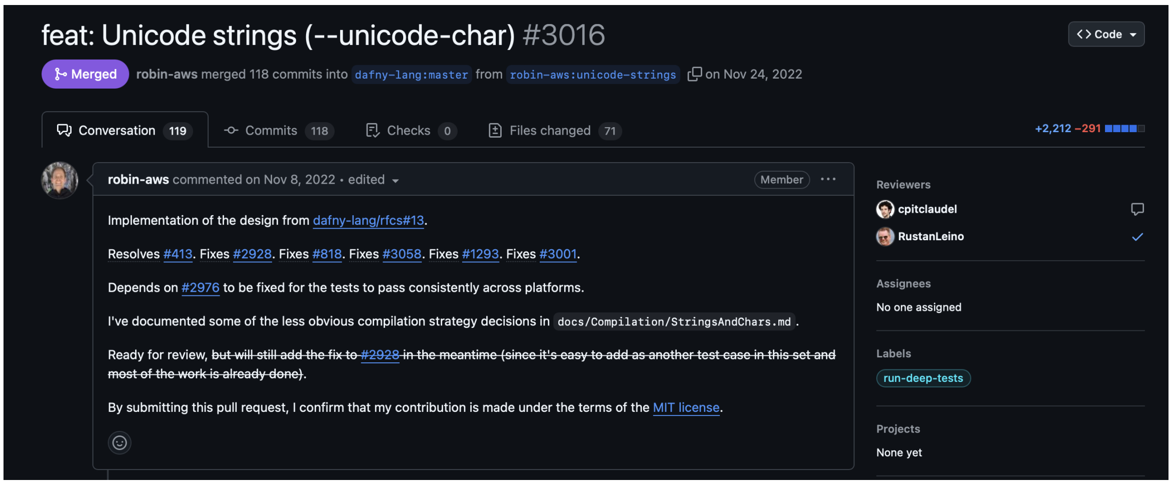
Fixes #1980. Fixes #2925.

Should fix the root cause of #2934, but I'm not claiming the complete fix, as attempting to include unescaped, non-ASCII characters in a Dafny source file is revealing platform-specific headaches in the integration test runner I'd like to address in a separate PR.

Printing non-ASCII characters, especially invalid UTF-16 sequences, is still inconsistent across backends, but at least should not crash after this change. Again this is difficult to test effectively across platforms in our testing architecture.

By submitting this pull request, I confirm that my contribution is made under the terms of the MIT license.

**Figure 2:** Pull request number 2926


feat: Unicode strings (--unicode-char) #3016

Merged robin-aws merged 118 commits into `dafny-lang:master` from `robin-aws:unicode-strings` on Nov 24, 2022

Conversation 119 · Commits 118 · Checks 0 · Files changed 71 · +2,212 −291

robin-aws commented on Nov 8, 2022 • edited · Member

Implementation of the design from dafny-lang/rfcs#13.

Resolves #413. Fixes #2928. Fixes #818. Fixes #3058. Fixes #1293. Fixes #3001.

Depends on #2976 to be fixed for the tests to pass consistently across platforms.

I've documented some of the less obvious compilation strategy decisions in `docs/Compilation/StringsAndChars.md`.

Ready for review, ~~but will still add the fix to #2928 in the meantime (since it's easy to add as another test case in this set and most of the work is already done)~~.

By submitting this pull request, I confirm that my contribution is made under the terms of the MIT license.

Reviewers: cpitclaudel, RustanLeino ✓

Assignees: No one assigned

Labels: run-deep-tests

Projects: None yet

**Figure 3:** Pull request number 3016

```
Errors compiling program into main
(5062,53): error CS0103: The name '_4_s' does not exist in the current context
```

**(a)** C# Error

```
# command-line-arguments
main-go/src/main.go:143:35: undefined: _4_s
```

**(b)** Go Error

```
main-java/_System/____default.java:19: error: cannot find symbol for (java.math.BigInteger _assign_such_that_0: (_4_s).Elements()) {
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | ^
symbol: variable _4_s
location: class ____default 1 error
```

**(c)** Java Error

**Figure 4:** Bug in issue 3472

```
Unhandled exception. System.AggregateException: One or more errors occurred. (Index was outside the bounds of the array.)
| | | | | | | | | | | | |   System.IndexOutOfRangeException: Index was outside the bounds of the array.
| | | | | ...
dafny/Scripts/dafny: line 34: 88266 Abort trap: 6
"$DOTNET" "$DAFNY" "$@"
```

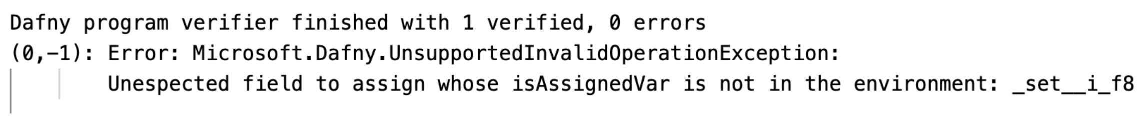
**(a)** The original Go backend error after pull request 2926

```
CS:
StdErr: Unhandled exception. System.AggregateException: One or more errors occurred. (Index was outside the bounds of the array.)
| | | | | | | | | | | | | |   System.IndexOutOfRangeException: Index was outside the bounds of the array.
| | | | | | | ...
|
dafny/Scripts/dafny: line 34: 87637 Abort trap: 6
"$DOTNET" "$DAFNY" "$@"

go:
StdErr: Unhandled exception. System.AggregateException: One or more errors occurred. (Index was outside the bounds of the array.)
| | | | | | | | | | | | | |   System.IndexOutOfRangeException: Index was outside the bounds of the array.
| | | | | | | ...
dafny/Scripts/dafny: line 34: 87632 Abort trap: 6
$DOTNET" "$DAFNY" "$@"
```

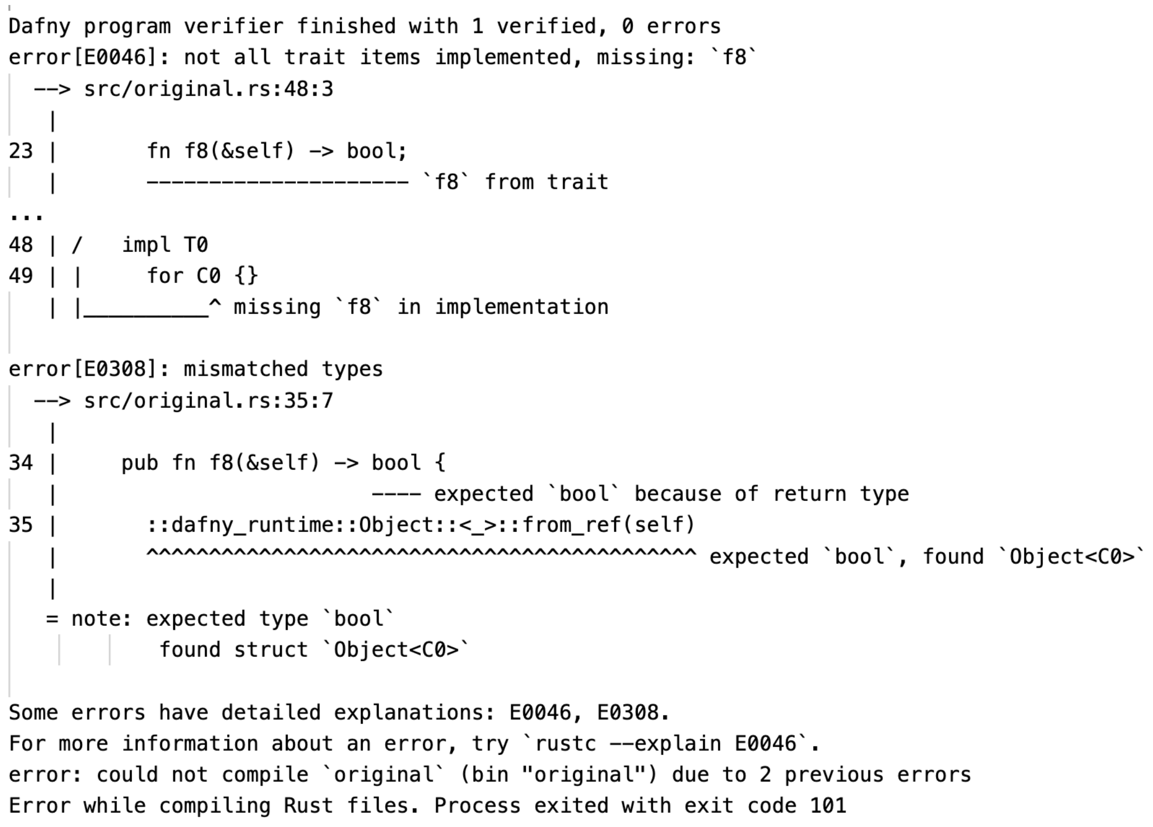
**(b)** The error from C# and Go backend from pull request 3016

**Figure 5:** Bug from two different issues with similar consequence



```
Dafny program verifier finished with 1 verified, 0 errors
(0,-1): Error: Microsoft.Dafny.UnsupportedInvalidOperationException:
        Unespected field to assign whose isAssignedVar is not in the environment: _set__i_f8
```

**(a)** The error message on the master branch



```
Dafny program verifier finished with 1 verified, 0 errors
error[E0046]: not all trait items implemented, missing: `f8`
  --> src/original.rs:48:3
   |
23 |       fn f8(&self) -> bool;
   |       --------------------- `f8` from trait
...
48 | /   impl T0
49 | |     for C0 {}
   | |__________^ missing `f8` in implementation

error[E0308]: mismatched types
  --> src/original.rs:35:7
   |
34 |     pub fn f8(&self) -> bool {
   |                         ---- expected `bool` because of return type
35 |       ::dafny_runtime::Object::<_>::from_ref(self)
   |       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ expected `bool`, found `Object<C0>`
   |
   = note: expected type `bool`
              found struct `Object<C0>`

Some errors have detailed explanations: E0046, E0308.
For more information about an error, try `rustc --explain E0046`.
error: could not compile `original` (bin "original") due to 2 previous errors
Error while compiling Rust files. Process exited with exit code 101
```

**(b)** The error message on the pull request branch

**Figure 6:** Missing implementation error in Rust

# Rationale for choosing Amazon ECS

There are multiple options on AWS for container orchestration, and batch processing services can be used to manage the compute instances. We will list the options and justify our decision to use **Amazon ECS (Elastic Container Service)** for container orchestration.

- **AWS Batch:** AWS Batch is a service design for batch processing. AWS Batch has a job queue and job definition, where priority can be assigned to each job. Overall, the AWS batch is very well suited for our use. The downside of using Batch is that it allows a limited choice of compute instance type. The instance types used by Batch are optimised types, such as compute-optimised instances, which are more costly than generic instances. Fuzzing jobs can be computing intensive and could benefit from a compute-optimised instance but not to the level needed by AI/ML jobs, which these instances are optimised for. Fuzzing jobs could already do well with general-purpose instances with more virtual CPUs. If we use AWS Batch, CompFuzzCI will be running on instances that are more expensive than necessary. This is the main reason we decided not to use AWS Batch.

- **Amazon EKS (Elastic Kubernetes Service):** AWS EKS is a managed Kubernetes service for running Kubernetes on AWS. Kubernetes is a container orchestration service that is very popular and widely used. It provides a lot of flexibility and control over the container deployments and communication. The reason we decided not to use EKS is that we can achieve the same result with ECS, which does not have cluster management costs. EKS is more suited for work with more complex requirements or for work that already uses Kubernetes.

- **Amazon ECS (Elastic Container Service):** AWS ECS is a managed container service. It is easy to set up and manage. ECS also allow us to write a template for the instances that would be launched by the autoscaling group, which gives us more control over the instance type and the resources.
  The CompFuzzCI workflow, using ECS, would be as follows: GitHub workflow gets triggered, it builds/updates a container image, it then submits a task definition with the image and IAM role for S3 Bucket to ECS, GitHub workflow signal to ECS to deploy the task definition with the desired number of tasks. ECS will take care of provisioning resources and running the fuzz jobs. Thus, fulfilling the **signalling to the auto-scaling group** requirement with the lowest cost. For this reason, we decided to use ECS.

# Bibliography

[1] SHAOHUA LI, THEODOROS THEODORIDIS, and ZHENDONG SU. Boosting Compiler Testing by Injecting Real-world Code. 2023.

[2] 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*, 2011.

[3] AFL developers. American Fuzzy Lop, . URL `https://lcamtuf.coredump.cx/afl/`.

[4] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for C and C++ compilers with YARPGen. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), 2020.

[5] Bastien Lecoeur, Hasan Mohsin, and Alastair F Donaldson. Program reconditioning: Avoiding undefined behaviour when finding and reducing compiler bugs. *Proceedings of the ACM on Programming Languages*, 7(PLDI), 2023.

[6] Alex Usher, Alastair Donaldson, and Cristian Cadar. fuzz-d: Random Program Generation for Testing Dafny.

[7] Mayank Sharma, Pingshi Yu, and Alastair F. Donaldson. RustSmith: Random Differential Compiler Testing for Rust. ISSTA 2023, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702211. doi: 10.1145/3597926.3604919. URL `https://doi.org/10.1145/3597926.3604919`.

[8] Michaël Marcozzi, Qiyi Tang, Alastair F Donaldson, and Cristian Cadar. Compiler fuzzing: How much does it matter? *Proceedings of the ACM on Programming Languages*, 3(OOPSLA), 2019.

[9] Barton P Miller, David Koski, Cjin Pheow Lee, Vivekandanda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1995.

[10] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3540787992.

[11] dafny lang. GitHub - dafny-lang/dafny: Dafny is a verification-aware programming language, Sep 2024. URL `https://github.com/dafny-lang/dafny/tree/master`.

[12] Dafny-Lang. Bad cast in java target code from type-parameter trickiness · issue 3956 · Dafny-Lang/dafny. URL `https://github.com/dafny-lang/dafny/issues/3956`.

[13] The Coq development team. *The Coq proof assistant*. LogiCal Project, 2004. URL `http://coq.inria.fr`. Version 8.0.

[14] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7), 2009.

[15] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices*, 49(6), 2014.

[16] Alastair F Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. Automated testing of graphics shader compilers. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), 2017.

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

[18] Karine Even-Mendoza, Cristian Cadar, and Alastair F Donaldson. Closer to the edge: Testing compilers more thoroughly by being less conservative about undefined behaviour. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020.

[19] William E Howden. Theoretical and empirical studies of program testing. *IEEE Transactions on Software Engineering*, (4), 1978.

[20] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1), 1998.

[21] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. Metamorphic testing: a new approach for generating next test cases. *arXiv preprint arXiv:2002.12543*, 2020.

[22] LLVM Project. How to submit an LLVM bug report. URL `https://llvm.org/docs/HowToSubmitABug.html`.

[23] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on software engineering*, 28(2), 2002.

[24] Ghassan Misherghi and Zhendong Su. HDD: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*, 2006.

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

[26] 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*, 2018.

[27] Alastair F Donaldson, Hugues Evrard, and Paul Thomson. Putting randomized compiler testing into production (experience report). In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2020.

[28] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013.

[29] Josie Holmes, Alex Groce, and Mohammad Amin Alipour. Mitigating (and exploiting) test reduction slippage. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*, 2016.

[30] Korosh Koochekian Sabor, Abdelwahab Hamou-Lhadj, and Alf Larsson. Durfex: a feature extraction technique for efficient detection of duplicate bug reports. In *2017 IEEE international conference on software quality, reliability and security (QRS)*. IEEE, 2017.

[31] Sean Banerjee, Bojan Cukic, and Donald Adjeroh. Automated Duplicate Bug Report Classification Using Subsequence Matching. In *2012 IEEE 14th International Symposium on High-Assurance Systems Engineering*, 2012. doi: 10.1109/HASE.2012.38.

[32] Tejinder Dhaliwal, Foutse Khomh, and Ying Zou. Classifying field crash reports for fixing bugs: A case study of Mozilla Firefox. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2011.

[33] Irving Muller Rodrigues, Daniel Aloise, and Eraldo Rezende Fernandes. FaST: a linear time stack trace alignment heuristic for crash report deduplication. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393034. doi: 10.1145/3524842.3527951. URL `https://doi.org/10.1145/3524842.3527951`.

[34] Irving Muller Rodrigues, Aleksandr Khvorov, Daniel Aloise, Roman Vasiliev, Dmitrij Koznov, Eraldo Rezende Fernandes, George Chernishev, Dmitry Luciv, and Nikita Povarov. Tracesim: An alignment method for computing stack trace similarity. *Empirical Software Engineering*, 27(2), 2022.

[35] Yingnong Dang, Rongxin Wu, Hongyu Zhang, Dongmei Zhang, and Peter Nobel. Rebucket: A method for clustering duplicate crash reports based on call stack similarity. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012.

[36] Neda Ebrahimi, Abdelaziz Trabelsi, Md Shariful Islam, Abdelwahab Hamou-Lhadj, and Kobra Khanmohammadi. An HMM-based approach for automatic detection and classification of duplicate bug reports. *Information and Software Technology*, 113, 2019.

[37] Liu Chao, Xie Qiaoluan, Li Yong, Xu Yang, and Choi Hyun-Deok. DeepCrash: deep metric learning for crash bucketing based on stack trace. In *Proceedings of the 6th International Workshop on Machine Learning Techniques for Software Quality Evaluation*, 2022.

[38] Aleksandr Khvorov. Netbeans Stacktraces, Mar 2021. URL `https://figshare.com/articles/dataset/netbeans_stacktraces_json/14135003`.

[39] Rijnard van Tonder, John Kotheimer, and Claire Le Goues. Semantic crash bucketing. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[40] Software Engineering Institute. CERT BFF, Oct 2016. URL `https://insights.sei.cmu.edu/library/cert-bff/`.

[41] HonggFuzz developers. HongFuzz, . URL `https://honggfuzz.dev/`.

[42] Hao Yang, Yang Xu, Yong Li, and Hyun-Deok Choi. K-Detector: Identifying Duplicate Crash Failures in Large-Scale Software Delivery. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2020.

[43] He Jiang, Xin Chen, Tieke He, Zhenyu Chen, and Xiaochen Li. Fuzzy clustering of crowdsourced test reports for apps. *ACM Transactions on Internet Technology (TOIT)*, 18(2), 2018.

[44] Irving Muller Rodrigues, Daniel Aloise, Eraldo Rezende Fernandes, and Michel Dagenais. A soft alignment model for bug deduplication. In *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020.

[45] Qi Xie, Zhiyuan Wen, Jieming Zhu, Cuiyun Gao, and Zibin Zheng. Detecting duplicate bug reports with convolutional neural networks. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2018.

[46] Shikai Guo, Xinyi Zhang, Xi Yang, Rong Chen, Chen Guo, Hui Li, and Tingting Li. Developer activity motivated bug triaging: via convolutional neural network. *Neural Processing Letters*, 51, 2020.

[47] Xin Xia, David Lo, Ying Ding, Jafar M Al-Kofahi, Tien N Nguyen, and Xinyu Wang. Improving automated bug triaging with specialized topic model. *IEEE Transactions on Software Engineering*, 43(3), 2016.

[48] Asmita Yadav, Sandeep Kumar Singh, and Jasjit S Suri. Ranking of software developers based on expertise score for bug triaging. *Information and Software Technology*, 112, 2019.

[49] Iyad Alazzam, Ahmed Aleroud, Zainab Al Latifah, and George Karabatis. Automatic bug triage in software systems using graph neighborhood relations for feature augmentation. *IEEE Transactions on Computational Social Systems*, 7 (5), 2020.

[50] Ashima Kukkar, Rajni Mohana, Anand Nayyar, Jeamin Kim, Byeong-Gwon Kang, and Naveen Chilamkurti. A novel deep-learning-based bug severity classification technique using convolutional neural networks and random forest with boosting. *Sensors*, 19(13), 2019.

[51] others Junio Hamano. Git. URL `https://git-scm.com/`.

[52] Laura Marie Leventhal, Barbee M Teasley, Diane S Rohlman, and Keith Instone. Positive test bias in software testing among professionals: A review. In *Human-Computer Interaction: Third International Conference, EWHCI'93 Moscow, Russia, August 3–7, 1993 Selected Papers 3*. Springer, 1993.

[53] Gul Calikli, Berna Aslan, and Ayse Bener. Confirmation bias in software development and testing: An analysis of the effects of company size, experience and reasoning skills. 2010.

[54] Tom Preston-Werner Chris Wanstrath, Scott Chacon. GitHub. URL `https://github.com/`.

[55] Google. Oss-fuzz: Continuous fuzzing for open source software, May 2022. URL `https://github.com/google/oss-fuzz`.

[56] LLVM project. libFuzzer – a library for coverage-guided fuzz testing. — LLVM 19.0.0git documentation. `https://llvm.org/docs/LibFuzzer.html`. (Accessed on 06/01/2024).

[57] Google. Github - google/clusterfuzz: Scalable fuzzing infrastructure., Feb 2023. URL `https://github.com/google/clusterfuzz`.

[58] Jie Liang, Mingzhe Wang, Yuanliang Chen, Yu Jiang, and Renwei Zhang. Fuzz testing in practice: Obstacles and solutions. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018.

[59] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. Taming Google-scale continuous testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2017.

[60] 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*, 2018.

[61] Thijs Klooster, Fatih Turkmen, Gerben Broenink, Ruben Ten Hove, and Marcel Böhme. Continuous Fuzzing: A Study of the Effectiveness and Scalability of Fuzzing in CI/CD Pipelines. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*. IEEE, 2023.

[62] Ahmed Irfan, Sorawee Porncharoenwase, Zvonimir Rakamarić, Neha Rungta, and Emina Torlak. Testing Dafny (experience paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022.

[63] Alastair F Donaldson, Dilan Sheth, Jean-Baptiste Tristan, and Alex Usher. Randomised Testing of the Compiler for a Verification-Aware Programming Language. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2024.

[64] Alastair F Donaldson, Ben Clayton, Ryan Harrison, Hasan Mohsin, David Neto, Vasyl Teliman, and Hana Watson. Industrial deployment of compiler fuzzing techniques for two GPU shading languages. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 374–385. IEEE, 2023.

[65] Coverlet. coverlet-coverage/coverlet, Jun 2021. URL `https://github.com/coverlet-coverage/coverlet`.

[66] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A Ground-Truth Fuzzing Benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3), December 2020. doi: 10.1145/3428334. URL `https://doi.org/10.1145/3428334`.