Compiler Fuzzing in Continuous Integration: A Case Study on Dafny

Karnbongkot Boonriong
Imperial College London
London, UK
0009-0003-6750-6460

Stefan Zetzsche
Amazon Web Services
London, UK
0009-0001-1304-0613

Alastair F. Donaldson
Imperial College London
London, UK
0000-0002-7448-7961

***Abstract*—We present the design of CompFuzzCI, a framework for incorporating compiler fuzzing into the continuous integration (CI) workflow of the compiler for Dafny, an open-source programming language that is increasingly used in and contributed to by industry. CompFuzzCI explores the idea of running a brief fuzzing campaign as part of the CI workflow of each pull request to a compiler project. Making this effective involved devising solutions for various challenges, including how to deduplicate bugs, how to bisect the project's revision history to find the commit responsible for a regression (challenging when project interfaces change over time), and how to ensure that fuzz testing complements existing regression testing efforts. We explain how we have engaged with the Dafny development team at Amazon to approach these and other problems in the design of CompFuzzCI, and the lessons learned in the process. As a by-product of our work with CompFuzzCI, we found and reported three previously-unknown bugs in the Dafny compiler. We also present a controlled experiment simulating the use of CompFuzzCI over time on a range of Dafny commits, to assess its ability to find historic bugs. CompFuzzCI prioritises support for the Dafny compiler and the fuzz-d fuzzer but has a generalisable design: with modest modification to its internal interfaces, it could be adapted to work with other fuzzers, and the lessons learned from our experience will be relevant to teams considering including fuzzing in the CI of other industrial software projects.**

***Index Terms*—Fuzzing, compilers, continuous integration**

I. Introduction

We present the design of CompFuzzCI,[1] a framework for incorporating compiler fuzzing into the continuous integration (CI) workflow of the compiler for the Dafny language [32], [33], and report on our experience working with Dafny developers in industry to deploy CompFuzzCI in practice.

Dafny is an open-source programming language with first-class support for formal verification that has its origins at Microsoft Research [32]. It is now increasingly being used by a number of companies to construct high assurance software, including Amazon [1]–[5], [27], ConsenSys [8], [15], Microsoft [25], Intel [49], and VMware [47]. In this paper, when we speak of the *Dafny (compiler) developers*, we mean the project's most active contributors at Amazon.

Due to the high-assurance use cases of Dafny, bugs in the Dafny compiler are a major concern. *Miscompilation* bugs—where the compiler emits code in a downstream language that does not respect the semantics of the Dafny source code—are particularly serious. This is because they may lead to the deployment of software that has been proven to meet certain correctness properties at the Dafny source code level but which, when compiled into executable code, no longer meets these properties. Recent work has focused on the use of randomised testing to automatically search for bugs in the Dafny compiler (as well as in the Dafny verification engine) [21], [28], showing that such techniques can be effective in uncovering numerous miscompilation bugs.

Existing work on compiler fuzzing for Dafny (and compiler fuzzing in general) focuses on running fuzzing campaigns periodically, in an ad hoc manner. In this paper we investigate deploying compiler fuzzing as part of a compiler's CI workflow: running a limited fuzzing campaign as part of CI on each pull request[2] (PR) in the hope of finding bugs that were recently introduced to the codebase, and potentially introduced by the current PR. In contrast to running fuzzing periodically, fuzzing during CI may find bugs that are "fresh": the developers responsible for introducing them are likely still members of the development team, and likely recall details of the recent change that introduced the bug, so that they are well-placed to provide a fix.

Our experience designing and deploying CompFuzzCI has involved considering various challenges; for example:

- For how long should fuzzing be applied to a given PR? Fuzzing for too long will lengthen the CI process and consume machine resources, but too short a fuzzing run may miss bugs that a longer run would find.
- How can we ensure that developers are provided with useful bug reports, and avoid providing duplicate reports of the same bug, or re-reporting already-known bugs?
- How can we ensure fuzzing complements standard regression testing?
- When is the right time to engage fuzzing on a project? Early fuzzing may discover important bugs quickly, but can lead to false alarms if a project has known limitations.
- When a bug is found, how do we accurately identify the project revision that introduced the bug, given that project dependencies and interfaces change over time?

Supported by EPSRC grant EP/R006865/1 and gift funding from Amazon.

[1] Pronounced "comp fuzzy", intended to refer to the use of compiler fuzzing during continuous integration (CI).

[2] "Pull request" is the GitHub terminology for a proposed change to a software project, sometimes called a "merge request" or "changelist".

979-8-3315-0814-2/25 © 2025 IEEE

We explain how we have addressed these and other challenges in creating and deploying CompFuzzCI—challenges which other teams would likely face when deploying fuzzing in the CI pipeline of a complex software project such as a compiler—and the lessons we have learned in the process. During our deployment of CompFuzzCI, we found and reported three previously-unknown bugs in the Dafny compiler, which have been confirmed by the Dafny developers. We also report on a controlled experiment simulating the use of CompFuzzCI over time on a range of Dafny commits, to assess its ability to find historic bugs.

Concretely, our CompFuzzCI framework prioritises support for the Dafny compiler and the fuzz-d fuzzing tool, which was shown to be the most effective among three different Dafny fuzzers in recent work [21]. However, CompFuzzCI is comprised of modules that communicate via well-defined interfaces. To adapt the framework for other fuzzers or compilers for other programming languages would merely require writing suitable adapters between the interfaces of CompFuzzCI and the interfaces of these fuzzers and compilers.

Key lessons learned. We summarise the key lessons learned from the design and deployment of CompFuzzCI.

Bisection is hard (Section IV-A). Frequent updates to the Dafny compiler codebase and breaking changes made over time created multiple pitfalls for bisection. Automated bisection tooling must be flexible to handle these changes.

Fuzzing too early can be counter-productive (Section IV-B). Our controlled experiment shows that CompFuzzCI could be useful in detecting faults early in the complete Dafny backends. However, when deployed live on an immature backend, CompFuzzCI detected many false positives that were not useful to the developers.

Duplicate bug reports between fuzzing and regression testing must be avoided (Section IV-C). Bugs were often duplicated between fuzzing and regression testing when both were run simultaneously. We solved this by fuzzing only after regression testing has successfully completed.

There are concrete opportunities to improve current fuzzers (Section V-B). Characteristics of bugs missed by CompFuzzCI during our controlled experiment suggest actionable ways in which existing fuzzers could be improved.

Error message-based deduplication poses challenges (Section V-D). The specificity of error messages varies between different target languages and compiler components. This led to some situations where CompFuzzCI can only be overly conservative or overly lenient in deduplication.

In summary, the contributions of this paper are:

- CompFuzzCI, a framework for incorporating compiler fuzzing into the CI workflow for the Dafny project.
- A report on our experience working with Dafny developers in industry to deploy CompFuzzCI in practice, the lessons learned from this experience, and the previously-unknown Dafny compiler bugs found during this process.
- A controlled experiment using historic revisions of Dafny to assess the ability of CompFuzzCI and its associated fuzzer, fuzz-d, with respect to historic bugs.

Paper structure. After providing relevant background (Section II), we describe the design and implementation of CompFuzzCI (Section III). We then discuss experiences and lessons learned from working with the Dafny developers to integrate CompFuzzCI into their workflow, and compiler bugs found during this process (Section IV). Next, we present our controlled experiment applying CompFuzzCI to historic revisions of Dafny (Section V). After discussing related work (Section VI) we conclude with ideas for future work (Section VII).

Availability. The source code for CompFuzzCI is available online at https://github.com/CompFuzzCI.

II. Background

A. The Dafny Language and Compiler

Dafny is a multi-paradigm programming language with first-class support for contract-based formal verification [16], [32], [33]. Combining a full-fledged programming language with a rich specification language, allows developers to write industrial-strength programs and specify the functional correctness properties of those programs, via features such as pre- and post-conditions for procedures and invariants for loops.

Dafny is implemented via a verifier and a compiler. The verifier uses automated Floyd-Hoare-style reasoning to check whether a given program meets its formal specifications. This works by translating the procedures of the Dafny program into the Boogie intermediate verification language [7], and using the Boogie verification engine to create a verification condition for each procedure. These verification conditions are then discharged by an SMT solver (the Z3 solver [17] is used by default). If a procedure cannot be proven to meet its specification, a warning is issued to the user. Otherwise, the program is compiled into a target language, via backends for various target languages including C#, Go, Python, Java, JavaScript, and Rust (the Rust backend is currently a work in progress). Once code has been generated for a particular target language, downstream tooling for that language is used to further compile and eventually execute the program.

Dafny has been used in numerous industry projects. For example, Dafny was used to model the authorisation engine and validator for Cedar, an authorisation-policy language from Amazon [27]; Dafny was used to write the AWS Cryptographic Material Providers Library [2]; Dafny was used by VMWare to build their VeriBetrFS verified file system [47]; a project from Consensys, called Dafny-EVM, used Dafny to construct a functional specification for the Ethereum Virtual Machine [15]; and verification using Dafny was a key component of Microsoft's IronFleet project on proving the correctness of distributed systems [25].

Very recently, it was announced that Dafny has been used to rewrite and prove correct AWS's policy authorisation engine, which makes more than a billion decisions per second [5], and that the AWS Clean Rooms Differential Privacy service uses SampCert [45], a library of verified randomised algorithms that utilises Dafny to extract its implementations in target languages (via the DafnyVMC project [50]).

B. Kinds of Compiler Bugs

Dafny compiler bugs can be broadly categorised as follows:

Compiler crash. Given a valid program, the compiler terminates prematurely without producing a valid output, typically yielding an error trace or message indicating the location at which the crash occurred in the compiler codebase.

Non-compliant code generation. The compiler emits code that does not comply with the syntax or static typing rules of the target language. This is detected when code generated by the Dafny compiler yields an error when compiled by a compiler for the target language. Comparing the original Dafny code with the non-compliant generated code usually provides a hint as to the nature of the bug in the Dafny compiler.

Miscompilation. The compiler emits code in the target language that compiles without error, but that is semantically inequivalent: the behaviour of the generated code is different to the behaviour that should be expected from the original source code according to the Dafny semantics, e.g. terminating abnormally or yielding different output. Miscompilations are *silent*: they do not lead to compile-time errors during compilation of the Dafny code or downstream compilation of generated code. This makes it hard to distinguish between miscompilation bugs.

C. The fuzz-d Compiler Fuzzing Tool

CompFuzzCI has been designed to work with fuzz-d [21], [46], a black box fuzzer for the Dafny compiler. As shown in Fig. 1, fuzz-d works by generating test programs in a randomised fashion, and putting them through a test oracle to check whether they trigger compiler bugs.

Test programs are generated by fuzz-d from scratch using a grammar-guided approach. Because the main focus of fuzz-d is on compiler testing, rather than verifier testing, and because the programs that it generates are typically too large and complex to be amenable to automated verification, fuzz-d invokes Dafny in a mode where formal verification is disabled. To make up for this, fuzz-d applies program reconditioning [31] to generated programs to conservatively ensure that they are free from problems that might lead to runtime errors.

To detect miscompilation bugs, fuzz-d relies on differential testing [36], comparing the result computed by the Dafny program after compilation to each target language with the result computed by a custom interpreter built into fuzz-d.

For each generated program, fuzz-d captures the output of the Dafny compiler, the target language compilers, and the execution of the compiled program for each target language. An error during compilation or a mismatch arising from differential testing indicates a bug in the Dafny compiler. The captured output and the summary of bug indicators are piped out to a file for further analysis.

III. Design of CompFuzzCI

We discuss the various modules that comprise CompFuzzCI (Section III-A), how these are used in the overall CompFuzzCI workflow (Section III-B), and provide details of how CompFuzzCI is deployed in practice using GitHub actions and Amazon Web Services (Section III-C).

A. Modules of CompFuzzCI

CompFuzzCI comprises five modules that work together to run a mini fuzzing campaign on a Dafny compiler pull request.

Fuzzing. The fuzzing module generates test programs and runs them against the Dafny compiler. The module uses a simple interface to interact with the chosen fuzzer, and currently only the fuzz-d tool (see Section II-C) is supported. The module outputs a file containing captured outputs and a summary of bug indicators from the fuzzer.

Deduplication. The deduplication module aims to determine whether a bug found by the fuzzing module is new or a duplicate of a known bug. This is essential to avoid cluttering the Dafny bug tracker with duplicate bug reports. The module has access to a database of known bugs and uses this to determine whether a bug found by the fuzzing module is new.

For crash and non-compliant code generation bugs, each database entry includes a *bug signature*: a set of normalised and hashed error messages that can be tested against the output of the Dafny compiler (crash) or a downstream compiler (non-compliant code generation) to determine whether the output exhibits symptoms of the bug. We have designed a script that extracts relevant error messages and details of stack traces from outputs captured by the fuzzing module. When a bug is found, relevant error messages and stack trace details are extracted via this script, hashed, and compared with the hashes stored in the database. If all the hashes are found (even if they only appear across multiple entries in the database) the bug is considered a duplicate. If any hashes are not found, the bug is potentially new and will be processed further. The difficulty of putting this into practice comes from the challenge of creating signatures that are strict enough to distinguish between genuinely distinct bugs, yet lenient enough to allow for natural variation in the output associated with a particular bug. We discuss this further in Section V-D.

Recall from Section II-B that miscompilation bugs do not exhibit any bug-specific symptoms. To deduplicate such bugs, our first idea was to use the minimised test program as a signature. Newly-found miscompilations could then be deduplicated based on the distance between its signature and the signatures of existing miscompilations [12]. However, we found that this was not feasible due to the time required to run test case reduction, and because (unless test case reduction is highly normalising) irrelevant syntactic differences between reduced programs can easily fool this kind of deduplication. Instead, inspired by the *correcting commit* metric used in an empirical study on compiler testing [9], CompFuzzCI uses the commit that *introduced* the bug as the signature for a miscompilation: we conservatively assume that a single commit introduces at

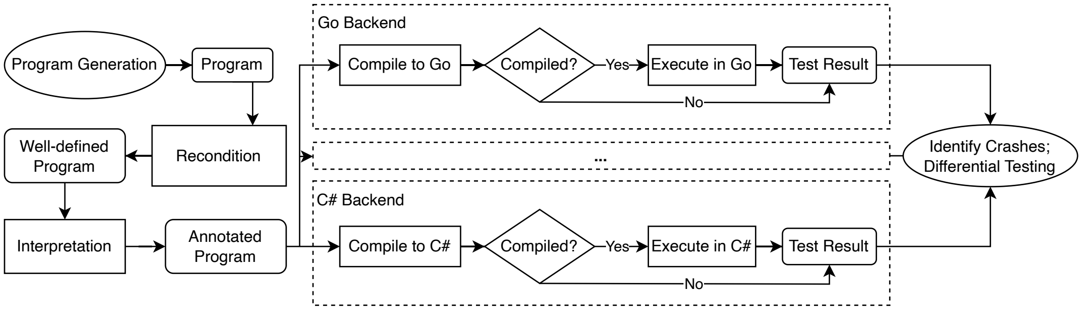

Fig. 1: Illustration of the workflow of the fuzz-d fuzzing tool for Dafny. The illustration shows two of Dafny's backends in action; in practice fuzz-d exercises all available backends.

most one miscompilation bug. This will not always be true, but is a reasonable approximation. A miscompilation bug database entry thus contains the ID of the commit that introduced the bug. Miscompilations are deduplicated by using the bisection module (discussed next) to find the commit that introduced the bug, and comparing this against the commits of known miscompilations in the database.

Regardless of bug type, if the deduplication module deems a bug to be new, it generates an *interestingness test* for the bug (a term introduced by the C-Reduce project [11]): a bug-specific script that determines whether a given Dafny program triggers the bug of interest. The content of this test is different for each crash and non-compliant code generation bug, being based on the bug's signature. For miscompilation bugs, a *differential* interestingness test is predefined by CompFuzzCI. This compares the output obtained for a Dafny program by the fuzz-d interpreter with the output obtained via each target backend. While theoretically, the interestingness test for a miscompilation bug might allow for bug slippage [12], we have not encountered this issue in practice.

Bisection. Given a bug-triggering test case, the bisection module aims to determine the commit that introduced the bug. While the key aim of CompFuzzCI is to find newly-introduced bugs, the randomised nature of fuzzing means that it may discover bugs that were introduced in earlier changes to the project, rather than by the pull request on which CI is being run. Bisection is useful for all bug types because (a) the codebase of the Dafny compiler is complex, and (b) when the fuzzer finds older bugs, the context associated with the code that is relevant to the bug may not be obvious to the compiler developers. Knowing which change introduced the bug can help in determining the relevant part of the codebase, and understanding the context of the bug.

Bisection is always needed for miscompilation bugs, to create a signature for bug deduplication. For other bugs, bisection is only needed when the bug is located on the master branch because it is likely to be old and difficult to gain context for. When a bug is located on the merge head of a PR, the developer can quickly gain context by looking at the PR.

CompFuzzCI leverages the `git-bisect` command for bisection [23]. This command takes a known bad commit (exhibits the bug) and a known good commit (does not exhibit the bug), where the known good commit must be an ancestor of the known bad commit. It also takes a script that determines whether the bug is present for a given commit; in the case of CompFuzzCI this is the interestingness test associated with the bug. The commit that introduces the bug is then determined via binary search: checking out, building, and running the interestingness test on commits in a systematic manner to find a pair of adjacent commits: the newest good commit and the oldest bad commit. This requires considering $\log(n)$ commits in the worst case, where n is the number of commits between the known good and bad commits.

In Section IV-A we discuss why finding the known good commit turned out to be a challenging problem, leading to us settling on a specific historic commit, the 'bisection limit' to use as the known commit, meaning that we cannot bisect bugs that existed prior to this commit.

Despite the efficiency of binary search, bisection can still be time-consuming because of the time associated with checking out, building, and testing a commit. In practice, this takes around 4 minutes for the Dafny project on the AWS infrastructure that we used (see Section III-C). We rely on various build tools used by the Dafny compiler to cache build artefacts to reduce the time taken to build the Dafny compiler for each commit. Bisecting from the 'bisection limit' commit requires roughly 10 iterations, taking around 40 minutes in the worst case. Therefore, we prefer to run bisection only when it is needed. With input from the Dafny developers, we devised a decision tree (Fig. 2) to determine when to run bisection.

Reduction. This module reduces the size of a bug-triggering test program to make it easier for compiler developers to work with. Like the bisection module, it takes the bug-triggering test program and its associated interestingness test as input. It outputs a reduced version of the test program that (according to the interestingness test) still triggers the bug. Reduction of crash and non-compliant code generation bugs is supported by the language-agnostic program reducer, Perses [43]. For miscompilation bugs, the reduction is supported by C-Reduce [11], [37], a reducer for C/C++ programs that has a "not C" mode in which C/C++-specific reduction steps are disabled; in this mode, C-Reduce has been shown to provide

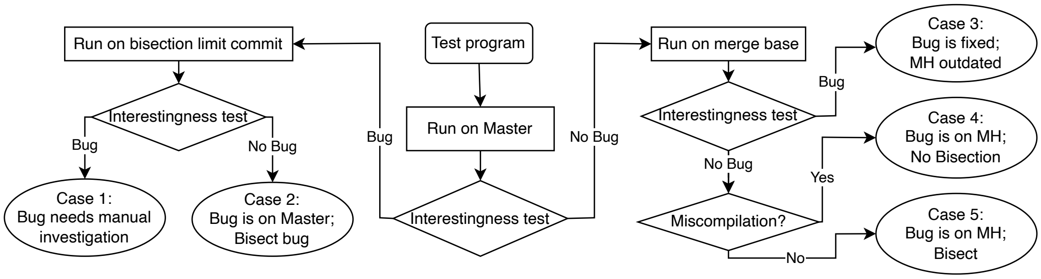
Run on bisection limit commit
Interestingness test
Bug
No Bug
Case 1: Bug needs manual investigation
Case 2: Bug is on Master; Bisect bug
Test program
Run on Master
Bug
No Bug
Interestingness test
Run on merge base
Interestingness test
Bug
No Bug
Miscompilation?
Yes
No
Case 3: Bug is fixed; MH outdated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

Fig. 2: Bisection decision tree. MH refers to the merge head of a pull request.

effective test case reduction for a range of languages.

Ignoring efficiency, we find that Perses yields higher quality reduction than C-Reduce because it uses a grammar for Dafny and is able to perform grammatically-correct reduction steps that are out of scope for C-Reduce. However, we found that C-Reduce runs more efficiently than Perses, so we use it to handle miscompilation bugs which we find often take a long time to reduce in practice.

Bug tracking. This module performs various tasks such as automatically reporting bugs via the Dafny GitHub issue tracker, automatically commenting on PRs, updating the status of bugs in the database, and adding user-reported bugs to the database. CompFuzzCI can report bugs in two ways: by creating a new issue on the Dafny GitHub repository when a bug is found on the master branch (such bugs being existing bugs, rather than bugs introduced by the PR), or by commenting on the PR that triggered the bug when a bug is found on the merge head of a PR (such bugs being new problems introduced by the PR).

When a GitHub issue is closed, the bug tracking module removes the associated bug entry from the database. This ensures that if a future fuzzer-found bug exhibits symptoms of the closed issue (indicating a regression) this will not be flagged as a duplicate: CompFuzzCI report the regression.

When a user opens an issue, the bug tracking module scrapes the issue for a bug-triggering program, compiles and executes the program, captures the output, and stores its signature in the database. Some user-created issues might not comply with the Dafny bug report format, these issues will not be processed by the bug tracking module.

B. Workflow of CompFuzzCI

We now explain the overall workflow of CompFuzzCI in terms of the five modules discussed in Section III-A.

When CompFuzzCI is applied to a pull request, the fuzzing module is invoked to generate a test program.

If the test program triggers a crash or non-compliant code generation bug, the workflow of Fig. 3 is used. Deduplication determines whether the bug is new. If so, bisection and reduction are run simultaneously, to provide a reduced test program and the bug-introducing commit. The bug tracking module gathers the bug's details and reports it to the developers.

If instead, the test program triggers a miscompilation bug, the workflow of Fig. 4 is used. Bisection and reduction run simultaneously with a differential interestingness test. Once bisection is complete, the ID of the commit that introduced the miscompilation bug is fed to the deduplication module to determine whether the miscompilation bug is new. If the bug is deemed to be new, the bug tracking module adds details of the bug (including the reduced test case emitted by the reduction module) and reports it to the Dafny developers.

Otherwise, if the test program does not trigger a bug, no action is required.

CompFuzzCI then starts again with the fuzzer module. This execution sequence is repeated until the time budget dedicated to fuzzing for the pull request is exhausted.

C. Deployment Details

CompFuzzCI is integrated into the Dafny continuous integration pipeline as a GitHub Actions workflow, supported by Amazon Web Services to provide the computing power required for fuzzing, bisection, etc. to scale. The infrastructure underlying CompFuzzCI and their interactions once invoked is shown in Fig. 5 and can be described as follows:

1) The Dafny developer opens or synchronises a PR on GitHub, triggering the CompFuzzCI workflow in the Dafny CI pipeline.
2) CompFuzzCI builds the Dafny compiler from the PR's merge head and registers the container image to the Amazon Elastic Container Registry (ECR).
3) CompFuzzCI deploys multiple containers from the registered image to Amazon Elastic Container Service (ECS) to run the fuzzing campaign.
4) The fuzzing campaign runs, executing the fuzzing, deduplication, bisection, reduction, and bug tracking modules as needed.
5) The bug tracking module reports bugs to the developers via GitHub issues or comments on the PR.

IV. Experiences and Lessons Learned From Initial Deployment

Over the last four months, we have worked with the Dafny developers on integrating CompFuzzCI into their continuous integration workflow. Engagement with the development team was through a series of meetings, as well as via frequent interaction on GitHub issues and pull requests.

We now describe a number of challenges and lessons learned from this deployment experience (Sections IV-A to IV-C), and discuss a previously-unknown bugs that we

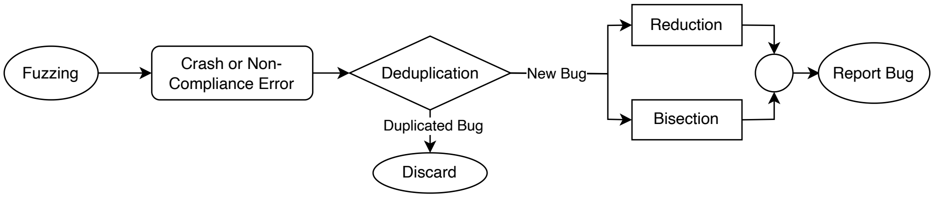

Fig. 3: The CompFuzzCI workflow for processing crash and non-compliant code generation bugs

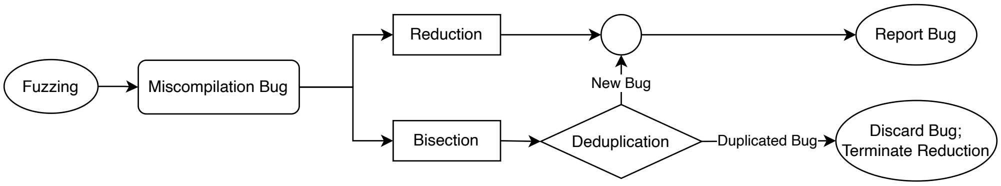

Fig. 4: The CompFuzzCI workflow for processing miscompilation bugs

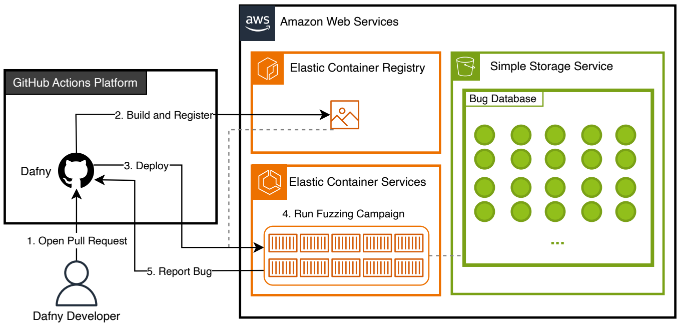

Fig. 5: CompFuzzCI infrastructure and interactions

found and reported to the Dafny team during the process of deploying CompFuzzCI (Section IV-D).

A. *Bisection: a Moving Target*

Bisection is an easy process in theory, but the reality is that the Dafny compiler is a moving target. The Dafny developers continuously make changes to the compiler to improve its performance and add new features. Mostly, such changes only affect the internal workings of the compiler. But sometimes they change the *interface* to the compiler, e.g. by affecting its user interface, or the format of its output.

Interface changes can render naive bisection inaccurate and misleading. The interestingness test, which determines whether a bug in the compiler is present (see Section III-A) must invoke the compiler. Using an interestingness test written against the interface of the current Dafny compiler will cause problems if applied to an older version of the compiler that expects arguments to be provided in a different form, or that prints error messages in a different manner. These problems can lead to `git-bisect`, which relies on the interestingness test for a bug, finding the wrong first bad commit.

From our experience deploying CompFuzzCI, we found that bisection can be misled by the following problems.

Changes to the compiler's command-line interface. The Dafny compiler's command-line interface underwent a redesign in October 2022, changing the available commands and introducing a different command-line argument format. To allow bisection to work with older commits, the interestingness test must be version-aware, issuing a suitable compiler command line depending on the Dafny version associated with the commit under consideration.

Changes to default flag values. Certain Dafny compiler flags have default values that the user can override, dictating compiler behaviour such as the function syntax that is used and how compiler warnings should be treated. The default values for these settings can vary over time. For instance, in February 2023, the default for the function syntax flag was updated from '3' to '4'. Similar to the problem with command line arguments, if a bug depends on compiler options whose defaults have changed over time, an interestingness test that does not account for this may incorrectly identify the first bad commit associated with a bug.

Changes to interaction with downstream tools. Over time, the way the Dafny compiler supports interaction with downstream compilers for target languages has changed. For example, after a successful Dafny-to-Java compilation, Dafny used to invoke the `javac` compiler in a manner that would generate `.class` files; in January 2023 this changed so that a `.jar` file would be generated instead. Additionally, in July 2023, the Python backend was changed so that the entry point file for a compiled Dafny program was renamed from `name.py` to `__name__.py`. All changes of this nature require attention for bisection to work correctly, so that the correct commands to attempt to reproduce a bug are issued depending on the version of Dafny under consideration.

Temporary breakages. The Dafny compiler has a 'version' option, which returns the compiler version. This is used as part of the interestingness test to identify the correct command to invoke Dafny and the expected output. We found that this option was broken for a series of 21 commits in the

Dafny revision history, resulting in no output being returned. This caused bisection to fail, as the interestingness test was expecting the version number as part of the compiler's output. The Dafny developers were able to fix this issue quickly, but it highlights the fragility of the bisection process when it relies on the output of the compiler.

Changes to build dependencies. Recall from Section III-A that bisection involves building historic commits of Dafny. The Dafny compiler is written in multiple languages including C# and Java. Over time, the required Java and .Net versions have changed, affecting the Gradle and MSBuild versions required by the build process. Without accounting for these older versions in the container image used during CI, older Dafny commits cannot be built. In principle, this could be solved by patching the Java and .Net versions in the Dafny compiler source code for each iteration of bisection, but this would add overhead to the (already expensive) bisection process. We have not investigated support for this yet.

To overcome many of the above problems, we had to adapt CompFuzzCI so that interestingness tests are *version-aware*: the version of the compiler is queried, and the commands to build and invoke Dafny and associated downstream compilers are issued accordingly. We have provided support for building and invoking versions of Dafny from May 2022 onwards (since Dafny version 3.6). This currently requires the management of two major versions and 18 minor versions.

As changes to interfaces and dependencies are common in industrial software projects, this problem of historic version management is not specific to Dafny: it is fundamental to supporting bisection-based bug triage.

B. Too Soon to Fuzz: the Dafny Rust backend

Support for Rust as a target language has only recently been added to the Dafny compiler. At the time of CompFuzzCI's deployment, the Dafny compiler's Rust backend was still under active development. As a result, when CompFuzzCI was triggered on Rust-related pull requests, it often found errors related to currently unimplemented or partially-supported features. From a formal standpoint, such errors are bugs: failures to compile valid Dafny programs. However, from a developer standpoint, these reports were false alarms: providing or completing support for relevant Rust backend features was already on the development roadmap.

In an attempt to avoid bothering developers with such false alarm reports, we added a simple text matching feature to CompFuzzCI to ignore failures where the term 'UnsupportedInvalidOperation' appeared in the compiler output. This sufficed for simple cases, but some partially-supported features proved more challenging to detect via text-based matching, since they led to Dafny emitting an invalid Rust code. It was non-obvious how to adapt CompFuzzCI to distinguish between a genuine non-compliant code generation bug, vs. a case where invalid code was generated due to a known limitation of the work-in-progress Rust backend.

For example, Dafny's support for traits has been implemented in the Rust backend. However, having a constant attribute within a trait was not yet supported at the time of our integration. When the fuzzer generated a program containing a class that extends a trait with a constant attribute, the Dafny compiler emitted invalid Rust due to this known limitation. This caused the Rust compiler to fail with an error message. From the perspective of CompFuzzCI this looks like a bug, rather than an unsupported feature, leading to a false positive.

```
method Main() {
  var v1 := new char[1];
  var v2 := new array<char>[1][v1];
}
```

Fig. 6: Program that triggers a Java backend miscompilation

The challenges faced when fuzzing the Rust backend raise a question about the right time to apply fuzzing. In principle, it makes sense to intensively test software while it is under development: if bugs can be found early, they can be fixed quickly by developers who are still actively working on the associated features. However, our experience with the Dafny Rust backend suggests that fuzzing too early can waste computational resources and distract developers if the feature under test is not yet mature enough for the scrutiny of fuzzing to be worthwhile. This echoes recent experience with the deployment of compiler fuzzing tools for the WebGPU shading language, where pros and cons associated with early deployment of fuzzing were identified [18].

C. Fuzzing vs. Regression Testing in Continuous Integration

The Dafny CI pipeline has a comprehensive regression test suite that covers most of the compiler code. Initially, CompFuzzCI was configured to run alongside the test suite. We observed that when shallow bugs are present in a newly created PR, they will often be caught by *both* the regression test suite and CompFuzzCI. This would lead to CompFuzzCI posting comments on the PR about a problem that the developer would already be alerted to due to a regression test suite failure. Furthermore, the reduced program in the CompFuzzCI bug report would rarely be as small or easy to understand as the failing test in the manually-crafted regression test suite.

Instead of devising a complex strategy to deduplicate bugs found through fuzzing from those identified by the regression test suite, we decided to delay the execution of CompFuzzCI until after the regression test suite has successfully passed. This simple approach ensures that any issues identified by fuzzing are genuinely new.

D. Previously-unknown Bugs

During the deployment of CompFuzzCI, we found three previously-unknown bugs in the Dafny compiler, all of which have been confirmed by the developers.

Issue #5741: Java miscompilation with nested arrays. The Dafny program of Fig. 6, involving nested arrays, triggers a miscompilation when the Java backend is used. The generated Java code compiles successfully, but raises a runtime exception when executed. CompFuzzCI bisected this bug and determined that it was a regression introduced by commit 5758205 that aimed to implement coercion for collection operations.

Issue #5736: Crash due to sequence comparison inside map comprehension. This problem occurs when a sequence comparison is used inside a map comprehension, causing the Dafny resolver, a component of the Dafny compiler responsible for the resolution of names and types, to crash. CompFuzzCI's bisection showed that this was a regression introduced by PR #5669, which was written to fix a different bug.

Issue #5698: Parser error on set comparison. This bug occurs when an array initialiser or a datatype constructor is given the following as input: a relational expression, followed by another relational expression involving two sets. The Dafny compiler incorrectly rejected the test programs due to a parser error (which CompFuzzCI regards as a crash bug). Our bisection module was unable to pinpoint the commit that introduced the bug because it was already present in the earliest commit we could bisect. Further investigation showed that the bug has been around since at least November 2021.

V. Controlled Evaluation on Historic Commits

We now present the results of an evaluation where we run CompFuzzCI on a series of pull requests associated with historic commits known to have introduced compiler bugs into the Dafny codebase. We are interested in whether, via CompFuzzCI, these bugs can be found on the commits that we know introduced them, how well the bug-inducing test programs are automatically reduced, and whether they are deduplicated successfully in relation to known bug reports.

Because the evaluation focuses on bugs that have been introduced on the merge head of a PR, it does *not* serve to evaluate the bisection module. Conducting an evaluation of this module would be possible by applying CompFuzzCI to a selection of more recent commits for each bug, where the bug is still present; we leave this for future work.

For simplicity, this evaluation focuses on crash and non-compliant code generation bugs, because it is easier to obtain ground-truth information about whether tests that expose such bugs are duplicates of one another. In contrast, this is hard for miscompilations due to the inherent difficulty associated with deduplicating them.

A. Evaluation Setup

We meticulously studied the Dafny issue tracker on GitHub to select known bugs in the Dafny compiler for our evaluation of CompFuzzCI. From 2,727 total issue reports, we filtered down to 20 relevant bugs by excluding open issues, non-compiler-related issues, issues without known introducing commits, and miscompilation bugs. The 20 selected bugs were introduced by 13 distinct pull requests.

The controlled evaluation involved running CompFuzzCI on the 13 bug-introducing pull requests. In each run, we deployed 10 instances of CompFuzzCI in parallel, each running for 2 hours. This is similar to the way CompFuzzCI is deployed live in the Dafny CI pipeline.

To simulate a realistic "steady state" environment for CompFuzzCI, as if it were in regular use, we used the bug tracking module to pre-populate the bug database with signatures of crash and non-compliant code generation bugs known to exist before the oldest PR in the evaluation. We then ran CompFuzzCI on the PRs in order from oldest to newest, adding any new bugs found to the bug database before running CompFuzzCI on the next PR.

To account for variance, we ran this experiment 10 times for each PR, allowing us to determine whether bugs could be reliably found or not. All instances ran on Amazon EC2 t2.medium instances with 2 vCPUs and 4GB RAM, ensuring consistency with real-world conditions.

B. Ability of CompFuzzCI to Find Historic Bugs

CompFuzzCI was able to identify 4 out of 20 known bugs fully reliably. That is, these bugs were found on every one of the 10 repeat runs that were carried out for the associated PR. In this case, the bug was always found within a 30-minute timeframe. There were 2 additional bugs that CompFuzzCI could *sometimes* find: these bugs were not found during every repeat run, but were found during at least 5 repetitions. Being harder to find, these bugs took longer to detect but when they were found this was within 90 minutes of fuzzing.

The remaining 14 bugs were not discovered during fuzzing. We attribute this to the complexity of their triggers and the limitations of the fuzz-d fuzzer used by CompFuzzCI. We discuss the reasons for this, demonstrating the value of evaluation in shedding light on the limitations of fuzz-d (shared by other Dafny fuzzing tools), which provide inspiration for future improvements in this area.

Complex triggers. Bugs with complex triggers, such as those requiring specific nested structures or particular sequences of operations, were less likely to be found within the 2-hour testing period. For example, issue #3987 was caused by the use of an array operation within a **`forall`** loop, inside a **`match`** statement. This bug was previously found by fuzz-d during a 12-hour fuzzing campaign, indicating that the complexity of the trigger, and the time budget of the fuzzing campaign, were factors in the bug not being found.

Specific names. The bug in issue #5283 was caused by the Go reserved keyword `fmt` being used as a module name, which will never arise in programs generated by fuzz-d due to its naming scheme. It would be worth extending fuzz-d to deliberately use identifiers that correspond to keywords of the languages that Dafny targets, to ensure that Dafny avoids naming conflicts when generating code.

Recursion. Bugs involving recursion were not found because fuzz-d does not generate recursive programs. This includes issue #5523 where the **`this`** keyword is mishandled in the compilation of a tail-recursive function. It would be worthwhile to add support for recursion to Dafny fuzzers.

Order of declarations. Bugs related to the order of declaration in generated code were also missed. For instance, issue #5569 was caused by a problematic declaration order of class

TABLE I: Percentage of changes in each PR covered by fuzzing after 30 minutes. In most cases, coverage had saturated by this time. In a small number of cases, coverage increased further during the 2-hour run, but never by more than 0.2%.

PR	Coverage at 30m
2241	11.96%
2646	18.40%
2734	26.40%
3479	19.52%
3623	0.27%
3886	18.31%
3909	17.15%
4136	7.79%
4591	43.34%
5390	2.37%
5474	11.27%
5528	25.91%
5591	12.76%

and methods in the generated Python code. The fuzz-d tool always generates declarations in a specific order, which does not match the order required to trigger the bug. Since Dafny is permissive regarding the ordering of declarations, a fruitful addition to fuzzing for Dafny would be a post-processing pass that randomises the order of declarations.

C. Coverage Analysis to Inform Fuzzing Duration

To investigate a suitable time limit for fuzzing, we computed coverage data achieved on the Dafny compiler by CompFuzzCI during our evaluation. For each PR and each repeat run, we calculated the coverage achieved on the changed lines of code, combined over the 10 fuzzing container instances at 30-minute intervals. We then averaged these numbers across the 10 repeat runs. Table I shows the average percentage of changed lines of code covered for each PR within 30 minutes. In almost all cases, coverage had saturated by this time, while for a small number of PRs, a marginal coverage increase was observed during the remainder of the 2-hour run (but never more than an additional 0.2%). Combined with the fact that the bugs that were found reliably were found within 30 minutes, we conclude that a 30-minute duration is a suitable time limit for running a 10-way parallel fuzzing session during CI.

This finding aligns with a previous study [29], which also recommends a 30-minute fuzzing duration for CI pipelines. Their research showed that the number of bugs triggered increased significantly in the first 10 to 30 minutes, with the next significant increase only occurring at the 4-hour mark, which is too long for CI pipelines.

D. Rate of Successful Deduplication

We calculated the successful deduplication rate shown in Table II based on the number of bugs successfully deduplicated out of all the bugs found during the evaluation. The deduplication module was mostly effective in handling crash bugs that included error traces and non-compliant generation bugs that had informative error messages. However, the deduplication module struggled with crash bugs that had varying error messages, and non-compliant generation bugs that had uninformative error messages.

We illustrate the difficulty of error message-based deduplication with three examples from our controlled experiment.

First, CompFuzzCI failed to deduplicate distinct test cases exposing a single bug where Dafny would incorrectly reject a valid program. This is because the details of the errors would vary significantly depending on the features of the test program. The following errors were regarded by CompFuzzCI as distinct despite being duplicates:

TABLE II: Deduplication success rates for different bug types

Type of bugs	Successful deduplication rate
Crash	68.36 %
Non-compliant generation	87.47 %

Error Message 1: Error Message 2:

```
Error: semicolon expected       Error: closeparen expected
Error: invalid UpdateStmt       Error: missing semicolon
Error: rbrace expected           at end of statement
3 parse errors detected in      Error: rbrace expected
  main.dfy                      3 parse errors detected in
                                  main.dfy
```

Second, CompFuzzCI is too coarse-grained in its deduplication of some non-compliant code generation bugs. For example, two different bugs affecting the Go backend led to the downstream Go compiler rejecting generated code with similar, uninformative messages:

Error Message 1: Error Message 2:

```
undefined: _2_a                 undefined: fmt.Dummy__
```

Because the only differences in these messages relate to identifiers in the generated Go code, which are expected to vary based on features of the input Dafny program, without further context CompFuzzCI has no way to tell whether these messages correspond to the same or different underlying bugs.

Third and finally, it is hard to find a sweet spot whereby the deduplication of non-compliant code generation bugs is neither too coarse-grained nor too fine-grained. Consider the following three error message snippets which arose from the non-compliant generation of Java code:

Error Message 1: Error Message 2: Error Message 3:

```
incompatible types: incompatible types: incompatible types:
Object cannot be    Object cannot be    CodePoint cannot be
converted to        converted to        converted to int
BigInteger          CodePoint
```

Based on manual investigation, the first and second error messages are due to a common issue and should be regarded as duplicates, whereas the third error arises due to a distinct issue and should not be regarded as a duplicate. If CompFuzzCI were to deduplicate based on the non-program-specific error message alone, it would incorrectly regard all three as duplicates. However, if CompFuzzCI would deduplicate based on the types involved, it would regard all three as distinct. Both approaches are inaccurate.

In difficult cases like these, CompFuzzCI would require a more intelligent deduplication mechanism that can identify the underlying issue by understanding the context of the generated program. We designed the deduplication module to err on the side of being coarse-grained, to avoid alerting developers to the same bug multiple times.

E. Size of Reduced Test Programs

We studied the size of the reduced test programs output by the reduction module. Based on their initial size, we cat-

TABLE III: Average number of lines of code before and after reduction, and the reduction percentage achieved for small, medium, and large programs.

Group	Original no. lines	Reduced no. lines	Reduction %
Small	468	58	88.06%
Medium	1,105	108	90.24%
Large	5,281	1,399	73.52%

egorised original, unreduced test programs as small (original size <30 KB), medium (original size 30-100KB), and large (original size >100KB). Table III shows the average number of lines of test programs in these categories before and after reduction, and indicates the reduction factor achieved for these categories based on these averages.

On average, the test programs attached to the GitHub issues for the 20 bug reports used in our evaluation contained 13 lines of code. The reduction module effectively minimised small programs to a size of the same order of magnitude as those in bug reports. However, medium and large programs often remained impractically large. In future, delaying bug reports until a suitably small bug-triggering program is found could help maintain the quality of bug reports while exploring further options to improve program reduction efficiency.

VI. Related Work

Compiler fuzzing. The fuzz-d tool [21] used by CompFuzzCI principally uses *differential testing* [36] as a test oracle, where the execution of compiled programs is compared across compilers, compiler backends, or optimisation levels. This is a widely-used oracle in compiler testing, having been used to test compilers for languages such as C [48], OpenCL [34], Java [13], Verilog [26] and Rust [41]. It is also used by the XDsmith fuzzer for Dafny [28].

An alternative test oracle, *metamorphic testing* [10], is used by the DafnyFuzz fuzzer for Dafny [21]. This involves comparing behaviour across compiled programs that, by construction, should behave identically. Metamorphic testing is the basis of various previous approaches to randomised compiler testing [19], [20], [22], [30], [42], [44].

Incorporating fuzzing in CI. Google's OSS-Fuzz [24] project for fuzzing open-source software supports multiple fuzzers, including libFuzzer, AFL, and HongFuzz. Once a project is integrated, OSS-Fuzz regularly builds the project, runs the fuzzers over it, and reports bugs that are found. However, OSS-Fuzz does not work directly as part of the CI for a project: it runs "after the fact" on commits that have been pushed to the main branch of a project, rather than running before pull requests have been merged. OSS-Fuzz uses ClusterFuzz, a scalable fuzzing infrastructure that runs on Google Cloud infrastructure. CompFuzzCI's use of AWS is similar to this. OSS-Fuzz is primarily designed to detect generic issues such as memory corruption, buffer overflows, and use-after-free bugs. In contrast, CompFuzzCI is targeted towards finding compiler bugs specifically.

CI Fuzz [14], developed by Code Intelligence, is a commercial product which is designed for projects written in C/C++, Java, and JavaScript that use specific build tools. CI Fuzz integrates into GitHub CI, allowing it to be incorporated into the CI pipeline as if it were a unit test. Compared to CompFuzzCI, CI Fuzz is better integrated into the CI pipeline. Due to its limited language support, it is not suitable for projects like Dafny, which are written in multiple unsupported languages. CI Fuzz is also not designed for finding bugs in compilers, which require a specialised fuzzer like fuzz-d.

Bug deduplication. Existing work on bug deduplication typically utilises runtime data or information from bug reports. Our focus is on runtime information-based approaches, which leverage data such as stack traces, control flow, and register states to identify duplicates. These approaches employ methods like TF-IDF for term frequency analysis [40], sequence alignment algorithms [6], [38], [39], and deep learning techniques [35]. Currently, none of the approaches mentioned can be applied to deduplicate miscompilation bugs. The deduplication in CompFuzzCI is limited to using stack traces from crash bugs and is not as sophisticated as the existing approaches.

An existing approach for deduplicating miscompilation bugs is the *furthest point first* ranking approach [12], based on the Levenshtein distance between the test cases' content and (optionally) runtime information. This approach requires the test cases to be fully minimised. CompFuzzCI's reduction module has not yet been optimised to fully minimise test cases within a reasonable time frame.

VII. Conclusions and Future Work

We have reported on the design and deployment of CompFuzzCI, a framework for running compiler fuzzing as part of continuous integration in the Dafny project. Our experience working with the Dafny developers to put this into practice has identified a number of challenges associated with CI-based fuzzing, including the challenges of accurately bisecting bugs when the environment of a project changes over time, the pros and cons of applying fuzzing early in the development of a new feature (the Rust backend), and the issue of ensuring that fuzzing and regression testing complement one another. A controlled study on historic Dafny commits suggests that a 30-minute fuzzing campaign using 10 fuzzing containers in parallel works well for finding various bugs introduced by PRs, while some bugs are out of scope due to limitations of the fuzz-d fuzzer, identifying concrete ways in which fuzzing could be improved.

Ideas for future work include integrating the DafnyFuzz and XDsmith fuzzers into CompFuzzCI, investigating whether fuzzing on a given PR could be directed based on the changes introduced by the PR, and improving error message-based deduplication by cross-checking error messages arising from distinct Dafny backends when a compiler bug is found to affect multiple backends simultaneously.

Acknowledgements

We thank the Dafny developers Fabio Madge, Mikael Mayer, Robin Salkeld, and Remy Willems for their input, and Cristian Cadar for feedback on an earlier draft of this work.

References

[1] Amazon Web Services, "Amazon verified permissions," 2023, https://aws.amazon.com/verified-permissions/, last accessed 2025-01-18.

[2] ——, "AWS cryptographic material providers library," 2023, https://github.com/aws/aws-cryptographic-material-providers-library-dafny, last accessed 2025-01-18.

[3] ——, "AWS encryption SDK for Dafny," 2023, https://github.com/aws/aws-encryption-sdk-dafny, last accessed 2025-01-18.

[4] ——, "AWS verified access," 2023, https://aws.amazon.com/verified-access/, last accessed 2025-01-18.

[5] ——, "AWS re:Inforce 2024 - Proving the correctness of AWS authorization (IAM401)," 2024, https://www.youtube.com/watch?v=oshxAJGrwMU.

[6] S. Banerjee, B. Cukic, and D. A. Adjeroh, "Automated duplicate bug report classification using subsequence matching," in *14th International IEEE Symposium on High-Assurance Systems Engineering, HASE 2012, Omaha, NE, USA, October 25-27, 2012*. IEEE Computer Society, 2012, pp. 74–81. [Online]. Available: https://doi.org/10.1109/HASE.2012.38

[7] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, ser. Lecture Notes in Computer Science, F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, Eds., vol. 4111. Springer, 2005, pp. 364–387. [Online]. Available: https://doi.org/10.1007/11804192_17

[8] F. Cassez, J. Fuller, M. K. Ghale, D. J. Pearce, and H. M. A. Quiles, "Formal and executable semantics of the Ethereum virtual machine in Dafny," in *Formal Methods - 25th International Symposium, FM 2023, Lübeck, Germany, March 6-10, 2023, Proceedings*, ser. Lecture Notes in Computer Science, M. Chechik, J. Katoen, and M. Leucker, Eds., vol. 14000. Springer, 2023, pp. 571–583. [Online]. Available: https://doi.org/10.1007/978-3-031-27481-7_32

[9] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "An empirical comparison of compiler testing techniques," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, L. K. Dillon, W. Visser, and L. A. Williams, Eds. ACM, 2016, pp. 180–190. [Online]. Available: https://doi.org/10.1145/2884781.2884878

[10] T. Chen, S. Cheung, and S. Yiu, "Metamorphic testing: a new approach for generating next test cases," Department of Computer Science, The Hong Kong University of Science and Technology, Tech. Rep. HKUST-CS98-01, 1998.

[11] Y. Chen, E. Eide, and J. Regehr, "C-Reduce," 2024, https://github.com/csmith-project/creduce, last accessed 2025-01-18.

[12] Y. Chen, A. Groce, C. Zhang, W. Wong, X. Z. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, H. Boehm and C. Flanagan, Eds. ACM, 2013, pp. 197–208. [Online]. Available: https://doi.org/10.1145/2491956.2462173

[13] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of JVM implementations," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, C. Krintz and E. D. Berger, Eds. ACM, 2016, pp. 85–99. [Online]. Available: https://doi.org/10.1145/2908080.2908095

[14] Code Intelligence, "CI Fuzz. efficient dynamic testing." 2024, https://www.code-intelligence.com/product-ci-fuzz, last accessed 2025-01-18.

[15] Consensys, "evm-dafny," 2023, https://github.com/Consensys/evm-dafny, last accessed 2025-01-18.

[16] Dafny Project, "Dafny GitHub repository," 2023, https://github.com/dafny-lang/dafny, last accessed 2025-01-18.

[17] L. M. de Moura and N. S. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24

[18] A. F. Donaldson, B. Clayton, R. Harrison, H. Mohsin, D. Neto, V. Teliman, and H. Watson, "Industrial deployment of compiler fuzzing techniques for two GPU shading languages," in *IEEE Conference on Software Testing, Verification and Validation, ICST 2023, Dublin, Ireland, April 16-20, 2023*. IEEE, 2023, pp. 374–385. [Online]. Available: https://doi.org/10.1109/ICST57152.2023.00042

[19] A. F. Donaldson, H. Evrard, A. Lascu, and P. Thomson, "Automated testing of graphics shader compilers," *PACMPL*, vol. 1, no. OOPSLA, pp. 93:1–93:29, 2017. [Online]. Available: https://doi.org/10.1145/3133917

[20] A. F. Donaldson and A. Lascu, "Metamorphic testing for (graphics) compilers," in *Proceedings of the 1st International Workshop on Metamorphic Testing, MET@ICSE 2016, Austin, Texas, USA, May 16, 2016*. ACM, 2016, pp. 44–47. [Online]. Available: https://doi.org/10.1145/2896971.2896978

[21] A. F. Donaldson, D. Sheth, J. Tristan, and A. Usher, "Randomised testing of the compiler for a verification-aware programming language," in *IEEE Conference on Software Testing, Verification and Validation, ICST 2024, Toronto, ON, Canada, May 27-31, 2024*. IEEE, 2024, pp. 407–418. [Online]. Available: https://doi.org/10.1109/ICST60714.2024.00044

[22] A. F. Donaldson, P. Thomson, V. Teliman, S. Milizia, A. P. Maselco, and A. Karpinski, "Test-case reduction and deduplication almost for free with transformation-based compiler testing," in *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, S. N. Freund and E. Yahav, Eds. ACM, 2021, pp. 1017–1032. [Online]. Available: https://doi.org/10.1145/3453483.3454092

[23] Git Project, "Documentation for git-bisect," 2024, https://git-scm.com/docs/git-bisect, last accessed 2025-01-18.

[24] Google, "OSS-Fuzz: Continuous fuzzing for open source software," 2024, https://github.com/google/oss-fuzz, last accessed 2025-01-18.

[25] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill, "Ironfleet: proving safety and liveness of practical distributed systems," *Commun. ACM*, vol. 60, no. 7, pp. 83–92, 2017. [Online]. Available: https://doi.org/10.1145/3068608

[26] Y. Herklotz and J. Wickerson, "Finding and understanding bugs in FPGA synthesis tools," in *FPGA '20: The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside, CA, USA, February 23-25, 2020*, S. Neuendorffer and L. Shannon, Eds. ACM, 2020, pp. 277–287. [Online]. Available: https://doi.org/10.1145/3373087.3375310

[27] M. Hicks, "How we built Cedar with automated reasoning and differential testing," 2023, https://www.amazon.science/blog/how-we-built-cedar-with-automated-reasoning-and-differential-testing, last accessed 2025-01-18.

[28] A. Irfan, S. Porncharoenwase, Z. Rakamaric, N. Rungta, and E. Torlak, "Testing Dafny (experience paper)," in *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, S. Ryu and Y. Smaragdakis, Eds. ACM, 2022, pp. 556–567. [Online]. Available: https://doi.org/10.1145/3533767.3534382

[29] T. Klooster, F. Turkmen, G. Broenink, R. ten Hove, and M. Böhme, "Continuous fuzzing: A study of the effectiveness and scalability of fuzzing in CI/CD pipelines," in *IEEE/ACM International Workshop on Search-Based and Fuzz Testing, SBFT@ICSE 2023, Melbourne, Australia, May 14, 2023*. IEEE, 2023, pp. 25–32. [Online]. Available: https://doi.org/10.1109/SBFT59156.2023.00015

[30] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, M. F. P. O'Boyle and K. Pingali, Eds. ACM, 2014, pp. 216–226. [Online]. Available: https://doi.org/10.1145/2594291.2594334

[31] B. Lecoeur, H. Mohsin, and A. F. Donaldson, "Program reconditioning: Avoiding undefined behaviour when finding and reducing compiler bugs," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, pp. 1801–1825, 2023. [Online]. Available: https://doi.org/10.1145/3591294

[32] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, ser. Lecture Notes in Computer Science, E. M. Clarke and A. Voronkov, Eds., vol. 6355. Springer, 2010, pp. 348–370. [Online]. Available: https://doi.org/10.1007/978-3-642-17511-4_20

[33] ——, "Accessible software verification with Dafny," *IEEE Softw.*,

vol. 34, no. 6, pp. 94–97, 2017. [Online]. Available: https://doi.org/10.1109/MS.2017.4121212

[34] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, D. Grove and S. Blackburn, Eds. ACM, 2015, pp. 65–76. [Online]. Available: https://doi.org/10.1145/2737924.2737986

[35] C. Liu, Q. Xie, Y. Li, Y. Xu, and H. Choi, "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, MaLTeSQuE 2022, Singapore, Singapore, 18 November 2022*, M. Cordy, X. Xie, B. Xu, and S. Bibi, Eds. ACM, 2022, pp. 29–34. [Online]. Available: https://doi.org/10.1145/3549034.3561179

[36] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998. [Online]. Available: http://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf

[37] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, J. Vitek, H. Lin, and F. Tip, Eds. ACM, 2012, pp. 335–346. [Online]. Available: https://doi.org/10.1145/2254064.2254104

[38] I. M. Rodrigues, D. Aloise, and E. R. Fernandes, "FaST: A linear time stack trace alignment heuristic for crash report deduplication," in *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*. ACM, 2022, pp. 549–560. [Online]. Available: https://doi.org/10.1145/3524842.3527951

[39] I. M. Rodrigues, A. Khvorov, D. Aloise, R. Vasiliev, D. V. Koznov, E. R. Fernandes, G. A. Chernishev, D. V. Luciv, and N. Povarov, "Tracesim: An alignment method for computing stack trace similarity," *Empir. Softw. Eng.*, vol. 27, no. 2, p. 53, 2022. [Online]. Available: https://doi.org/10.1007/s10664-021-10070-w

[40] K. K. Sabor, A. Hamou-Lhadj, and A. 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 2017, Prague, Czech Republic, July 25-29, 2017*. IEEE, 2017, pp. 240–250. [Online]. Available: https://doi.org/10.1109/QRS.2017.35

[41] M. Sharma, P. Yu, and A. F. Donaldson, "Rustsmith: Random differential compiler testing for rust," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, R. Just and G. Fraser, Eds. ACM, 2023, pp. 1483–1486. [Online]. Available: https://doi.org/10.1145/3597926.3604919

[42] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, E. Visser and Y. Smaragdakis, Eds. ACM, 2016, pp. 849–863. [Online]. Available: https://doi.org/10.1145/2983990.2984038

[43] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, "Perses: syntax-guided program reduction," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 361–371. [Online]. Available: https://doi.org/10.1145/3180155.3180236

[44] Q. Tao, W. Wu, C. Zhao, and W. Shen, "An automatic testing approach for compiler based on metamorphic testing technique," in *17th Asia Pacific Software Engineering Conference, APSEC 2010, Sydney, Australia, November 30 - December 3, 2010*, J. Han and T. D. Thu, Eds. IEEE Computer Society, 2010, pp. 270–279. [Online]. Available: https://doi.org/10.1109/APSEC.2010.39

[45] J.-B. Tristan, "SampCert version 1.0.0," 2024, https://github.com/leanprover/SampCert, last accessed 2025-01-18.

[46] A. Usher, "fuzz-d GitHub repository," 2023, https://github.com/fuzz-d/fuzz-d, last accessed 2025-01-18.

[47] VMware, "Verified BetrFS," 2024, https://github.com/vmware-labs/verified-betrfs.

[48] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, M. W. Hall and D. A. Padua, Eds. ACM, 2011, pp. 283–294. [Online]. Available: https://doi.org/10.1145/1993498.1993532

[49] Z. Yang, W. Wang, J. Casas, P. Cocchini, and J. Yang, "Towards a correct-by-construction FHE model," Cryptology ePrint Archive, Paper 2023/281, 2023. [Online]. Available: https://eprint.iacr.org/2023/281

[50] S. Zetzsche and J.-B. Tristan, "Dafny-VMC: a library for verified Monte Carlo algorithms," 2023, https://github.com/dafny-lang/Dafny-VMC, last accessed 2025-01-18.