Randomised Testing of the Compiler for a Verification-Aware Programming Language

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

Dilan Sheth
Imperial College London
London, UK
0009-0000-9567-5497

Jean-Baptiste Tristan
AWS
Boston, USA
0000-0003-2574-7883

Alex Usher
Imperial College London
London, UK
0009-0007-2416-3962

***Abstract*—We present the design and implementation of two new tools for randomised testing of the compiler of the Dafny programming language. The Dafny language and tool-chain supports formal verification of rich functional properties of programs, and is seeing increasing adoption by industry, being used by companies such as Amazon, Consensys, Microsoft and VMWare for the construction of high assurance software components. Bugs in the Dafny compiler are of critical importance because they have the potential to undermine the safety and security of deployed software that has been formally verified at the source code level. Our new testing tools, fuzz-d and DafnyFuzz, are based on randomised program generation, and overcome the test oracle problem using a combination of differential testing, metamorphic testing and the generation of programs with known expected results. The new tools go significantly further than XDsmith, an existing randomised compiler testing tool for Dafny, in terms of the features of the language that they support. We have used these tools to find and report 24 previously-unknown Dafny compiler bugs that were beyond the reach of XDsmith, of which 9 are soundness issues. Our fuzzing campaign has also led to changes to the Dafny language specification via the identification of ambiguous or under-documented language features. We present a set of controlled experiments looking at statement and mutation coverage on the Dafny compiler code base. The results show that fuzz-d and DafnyFuzz achieve substantial additional coverage on top of that provided by XDsmith, and can cover some areas missed by the Dafny compiler regression test suite. All three of fuzz-d, DafnyFuzz and XDsmith improve upon the number of mutants killed by the Dafny regression test suite.**

***Index Terms*—Fuzzing, compilers, formal verification, Dafny**

I. Introduction

The Dafny programming language and its verifying compiler [9], [23], [24] are seeing increasing adoption in industry for the construction of high assurance software. For example: Dafny was recently used to model the authorisation engine and validator for Amazon's new Cedar authorisation-policy language [19] (used by the the Amazon Verified Permissions and AWS Verified Access managed services [1], [4]); the AWS Cryptographic Material Providers Library is written using Dafny [2] and Amazon provide an AWS Encryption SDK for Dafny [3]; VMWare have used Dafny to build their VeriBetrFS verified file system [51]; the Dafny-EVM project from Consensys uses Dafny to construct a functional specification for the Etherium Virtual Machine [8]; and verification using Dafny was a key component of Microsoft's IronFleet project on proving the correctness of distributed systems [17].

As well as the formal correctness guarantees that Dafny can offer, the Dafny ecosystem is attractive because it features compiler back-ends for multiple target languages: currently C#, Go, Python, Java and JavaScript, with support for C++ and Rust added recently. This can reduce maintenance costs by avoiding the need for multiple implementations of software components written in diverse languages.

The increasing adoption of Dafny for engineering high-assurance software makes it critically important that the Dafny verification engine and the Dafny compiler, including all of its back-ends, are well tested. Bugs in the verification engine threaten to undermine the correctness guarantees that Dafny claims to provide. But arguably worse still, wrong code bugs in the Dafny compiler, where incorrect code is silently generated, evade both code review and formal verification. Presently, the only practical defence against such bugs is to extensively test the Dafny compiler. The focus of this paper is on techniques for automatically testing the Dafny compiler.

Amazon have already made steps towards testing the Dafny compiler using randomised testing, via the XDsmith tool [20]. Focusing on a small subset of the Dafny language, XDsmith generates Dafny programs in a randomised fashion. Each generated program is then compiled to all of the back-end languages that Dafny supports, and downstream tooling for these languages is used to compile and execute the generated code. The the results of execution are then compared, with result mismatches being indicative of compiler bugs. Through this application of *differential testing* [27], XDsmith was able to find a number of Dafny compiler bugs.[1]

While these results are encouraging, the reach of XDsmith is limited: the subset of Dafny that XDsmith focuses on omits key features such as loops, recursive procedure calls and various object-oriented language constructs that are widely-used in practice.

Our contribution. We report on the design, implementation and deployment of two new black-box randomised compiler

This work was supported by EPSRC grant EP/R006865/1, and by an Amazon Research Award.

[1]Another notable feature of XDsmith is its ability to find *verifier* bugs by generating annotated programs whose verification status is known by construction. As the focus of our work is on compiler testing, we do not consider this feature of XDsmith further.

testing tools for Dafny: fuzz-d and DafnyFuzz. Each of these tools supports a substantially larger fragment of the Dafny language compared with XDsmith. Like XDsmith, both fuzz-d and DafnyFuzz generate programs in a randomised fashion that are suitable for differential testing. The tools also support (each in their own way) an alternative oracle whereby at generation time the expected result that the generated program should compute is deduced. This is useful for finding bugs in common parts of the Dafny compiler: such bugs would be missed by standard differential testing if they cause each back-end to computing the *same* wrong result. The DafnyFuzz tool also supports *metamorphic testing* [6]: it can generate families of equivalent Dafny programs such that each program in a family should compute the same result, with result differences between programs being indicative of compiler bugs.

In part, we opted to build two new Dafny compiler testing tools to increase testing *diversity*. In his essay on the uses of diversity in software testing, Groce makes the case that we should think of software testing as "a *scavenger hunt*, where it might be a good idea to split up the team, since finding a teacup with blue flowers and finding a Bunsen burner will probably involve trips to very different locations" [15]. Seen in this light, fuzz-d and DafnyFuzz join XDsmith in the team of randomised testing tools that can be thrown at the Dafny compiler to make it as reliable as possible.

New bugs found using our tools. We have used fuzz-d and DafnyFuzz to find 24 previously-unknown Dafny compiler bugs that could not be found by XDsmith, of which 18 have been confirmed and 9 fixed. Of these bugs, 9 are *soundness* issues, because they cause to the Dafny compiler to emit code that will execute, but will deviate from the semantics of the source program at runtime (either throwing a runtime exception, or computing incorrect results). The remaining bugs are either *crash* bugs that cause the Dafny compiler to abort when processing an input program, or *invalid code* bugs, where the Dafny compiler emits invalid code in one of its target languages that is rejected at compile time by downstream tools. While still important because they may inconvenience developers, these are less critical than soundness bugs because they cannot compromise the correctness of formally verified, deployed code. We report on examples of crash and invalid code bugs that turned out to be "fuzz blockers": they triggered so frequently that they prevented fuzz-d and DafnyFuzz from finding any other (potentially more serious) bugs, and thus had to be urgently fixed or worked-around in order for our fuzzing campaign to proceed.

In addition to finding bugs, the process of designing and deploying new Dafny program generators highlighted a number of Dafny language design issues, where the intended semantics of constructs was not clear (to us) from the language specification. This led to cases where what we believed to be compiler bugs (due to us having implemented fuzz-d and DafnyFuzz according to *our* interpretation of these language features) turned out to be valuable false alarms: valuable in that they led to the Dafny language specification being clarified. This provides further evidence that randomised testing of programming language implementations can aid in clarifying the semantics of the target programming language, as has been noted in previous work [11].

Increased thoroughness of testing enabled by our tools. In addition to using fuzz-d and DafnyFuzz "in the wild", we report on a set of controlled experiments examining the statement and mutation coverage that the tools achieve on parts of the Dafny compiler in comparison to that achieved by XDsmith and by the Dafny compiler regression test suite. These results demonstrated that our tools offer a significant improvement in code coverage compared to XDsmith, and they can also identify weaknesses in the Dafny compiler regression test suite, using both coverage analysis and mutation testing.

In summary, the contributions of our paper are:

- The design and implementation of two new randomised testing tools for improving the reliability of the compiler for Dafny, a verified programming language that is seeing increasing adoption in industry for the construction of high assurance software. The tools use a combination of differential testing, metamorphic testing, and the generation of self-checking programs, to find bugs.
- A report on the successful use of these tools to improve the Dafny compiler and language specification, leading to the discovery of 24 previously-unknown bugs in the Dafny compiler that could not be found by the existing XDsmith fuzzer, including 9 soundness bugs, as well as a number of language design issues.
- A set of controlled experiments showing that fuzz-d and DafnyFuzz offer improvements over XDsmith with respect to statement coverage, and further using statement and mutation coverage analysis to highlight weaknesses in the Dafny compiler regression test suite.

Paper structure. We present background on the Dafny language, its compiler and verifier, and the XDsmith tool (Section II), details of the fuzz-d and DafnyFuzz tools (Section III), an overview and examples of the compiler bugs and language design issues that we have found and reported using these tools (Section IV), and the results of a set of controlled experiments to compare the test thoroughness of fuzz-d and DafnyFuzz to that of XDsmith and the Dafny regression test suite (Section V). We discuss related work on compiler testing (Section VI) and conclude with a discussion of avenue for future work (Section VII).

II. Background

A. The Dafny language

Dafny is a verification-aware programming language with native support for annotating programs with specifications [9], [23], [24], allowing developers to write code which is provably correct with respect to these specifications. Its language style is imperative in nature, featuring common programming idioms such as control flow constructs, pattern matching, collection types and object-oriented paradigms. Dafny also contains some more unique imperative idioms, such as its `forall` parallel assignment construct. This is used to simultaneously compute and assign values to array indices, as well as having some additional uses in verifier proofs.

At the core of the Dafny language lies a selection of verification constructs that can be used to write functional specifications for methods and functions. Adherence to these functional specifications can then be proven using a Floyd-Hoare-style program verification method. Alongside traditional specification constructs, such as pre- and post-conditions, invariants and framing constructs, Dafny also provides developers with more powerful constructs, including calculational proofs and lemmas.

B. The Dafny compiler and verifier

The tool-chain associated with the Dafny programming language comprises a compiler and verifier, which internally transform a program into a number of abstract syntax trees (ASTs) following the workflow detailed in Fig. 1. The correctness of a Dafny program is established via formal verification, thus Dafny first runs the verifier over a program, and compiles the program only when verification is successful. Verification works in a modular fashion by encoding each procedure in the Dafny program into a corresponding procedure in the Boogie intermediate verification language [5]. The Boogie verification engine then constructs a verification condition for each procedure, which is discharged by an SMT solver (Z3 [10] by default).

Programs can be compiled into one of a number of different target languages: currently C#, Go, Python, Java, JavaScript, C++ and Rust are supported (though support for C++ and Rust was added after we undertook the testing work reported in this paper). Regardless of target language, every Dafny program undergoes AST transformations from the frontend and resolver. Each target language is then handled by a separate back-end, all of which inherit a common AST class—the `SinglePassCompiler` (Fig. 1).

Dafny compiler bugs affecting the code common to all target languages are likely to impact all back-ends, while there is also potential for bugs related to code generation for a particular target language, and that thus affect only a particular back-end.

In this project, we focus on identifying such bugs in the *compiler* stages of the Dafny AST transformations, both in the shared AST classes and those specific to each back-end. When using our tools for compiler testing, we disable the Dafny verifier to allow higher throughput, ensuring that while test programs are not verified, they are created in a manner which makes them correct-by-construction, hence they should correctly compile.

C. XDsmith and its limitations

Existing testing for the Dafny programming language saw the creation of the tool XDsmith [20], which aimed to find bugs in both the Dafny verifier and compiler. It uses an underlying framework, Xsmith [16], to randomly generate AST structures and pretty-prints these into Dafny. When testing the verifier, it uses built-in heuristics to generate compute specifications for the generated AST structures such that the expected verification outcome for each specification is known in advance; deviations from these expectations are indicative of verifier bugs. When used for compiler testing, XDsmith uses differential testing: it invokes the Dafny compiler to generate and execute code for each supported target language, and the outputs of execution are compared—all the back-end outputs should be the same.

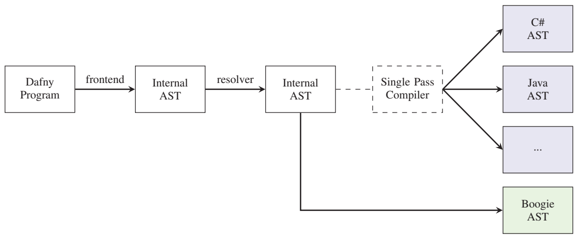

Fig. 1: Dafny Internal AST Transformations

While XDsmith is able to generate for many basic Dafny features, such as primitive types, collections and basic control flow (`if` statements), limitations in the underlying Xsmith framework meant that XDsmith could not support more complex language features, such as loops and object oriented features [20]. Consequently, a large portion of the Dafny language features were left untested, and in our testing we aimed to focus towards these. Furthermore, because XDsmith relies solely on differential testing across back-ends, it lacks mechanisms for identifying bugs in the common parts of the compilers—the AST transformations that are applied regardless of the target language. It is likely such bugs will impact all back-ends in the same way, in which case differential testing across back-ends will not be able to detect them.

III. The fuzz-d and DafnyFuzz Tools

We now give a brief overview of the fuzz-d [47] and DafnyFuzz [32] tools, both of which are open source on GitHub. We start by describing what the tools have in common in terms of their design (Section III-A), then detail features that are particular to each tool (Sections III-B and III-C).

A. Common features

a) Program generation: Both tools take a grammar-aided, generative approach to producing programs, inspired by the well-known Csmith tool for C compiler testing [52] (and also used by other "-smith" tools, including XDsmith). They start with a template `main` method and generate an AST in a top-down fashion, randomly choosing selections from available language features until the complete AST is formed. Generation maintains a context that stores available variables and top-level constructs, and is also used to determine which language features would be valid to generate at any given point during generation. As well as primitive types, collection types and control flow, the tools both support a number of more complex Dafny features such as inductive datatypes, parallel assignment and pattern matching. The set of features is not entirely the same between fuzz-d and DafnyFuzz; we highlight differences in Sections III-B and III-C.

b) Self-checking oracles: Both fuzz-d and DafnyFuzz implement a notion of a self-checking oracle. This allows the tools to determine the expected output for any arbitrary program that they generate. This is useful for detecting bugs in code that is shared between back-ends (e.g. bugs in the frontend and resolver; see Fig. 1). As noted above, if a bug did exist in such code, it would likely be missed by differential testing across back-ends, since all back-ends would inherit the same behaviour and exhibit the same incorrect result.

c) Test case reduction: To allow diagnosis of the root causes of bugs found by our fuzzers, we have integrated fuzz-d and DafnyFuzz with the Perses test case reducer [43]. To leverage Perses, we first encoded the Dafny grammar in the input format of the Antlr parser generator tool [30]. Driven by this grammar, Perses then allows a large, randomly-generated Dafny program that triggers a compiler bug to be minimised to a small program triggering the same bug that is relatively easy for the Dafny team to inspect in order to investigate the cause of the bug.

d) Common challenges: We discuss two challenges that had to be overcome during the development of both tools. First, the printing of or iterating over unordered Dafny data structures (such as sets and multisets) can lead to non-deterministic output that varies across back-ends. This is because the order in which elements of such structures are considered is not mandated by the Dafny language, and depends on the way that these data structures are modelled via features of the downstream languages that Dafny supports. Nondeterminism across back-ends is incompatible with differential testing as it makes it difficult to distinguish between an erroneous result mismatch due to a compiler bug, and a legitimate result mismatch due to non-determinism. Both tools incorporate logic that allows unordered data structures to be supported in a manner that does not lead to nondeterminism in the results that are printed by generated programs. This logic is also applied for heap based objects such as the `array` type which would otherwise print a memory location.

Second, both tools suffered from the problem of "fuzz blockers": easy-to-trigger bugs in the Dafny compiler that obscured the discovery of new bugs. At various stages in our fuzzing efforts we had to adjust the tools to inhibit them from generating programs likely to trigger common, previously-identified bugs. This is problematic as it inhibits the full testing of the Dafny language as it prevents language features being tested in conjunction which was found as the cause of some bugs discussed below.

B. fuzz-d

a) Program generation: fuzz-d generates arbitrary programs by selecting from features in its supported language set, which extends the commons set listed above to include Dafny's object oriented features: classes, traits and inheritance. The programs produced by the generator may not be valid, and therefore it is necessary to further transform the program to ensure its validity. This approach takes inspiration from re-conditioning [22]—safe wrapper functions are inserted where necessary in a separate pass over the randomly generated AST.

b) Differential testing: Similar to XDsmith, fuzz-d utilises differential testing to identify bugs impacting the Dafny back-ends. Once a program is generated, a built-in test harness invokes all the Dafny back-ends in parallel, and their outputs are compared to identify differences (indicative of bugs).

c) Interpreter: To overcome the limitation of differential testing being unable to identify bugs in common compiler code, fuzz-d implements a Dafny interpreter as a reference oracle, independent from the Dafny codebase. This is a separate component within fuzz-d which can be invoked independently of its program generator.

Given a Dafny program restricted to the subset of features that fuzz-d supports, the interpreter computes the expected output for this program. This provides an independent result to compare the back-end outputs to when performing differential testing, such that if the back-end outputs are then not as expected, either we have found a bug in a shared compiler AST, or in the interpreter itself.

So far, the only bugs we have found in shared code have been invalid code bugs, thus differential testing has proved the most effective oracle within fuzz-d. It is possible that with additional testing we may find such wrong result bugs impacting all back-ends.

C. DafnyFuzz

a) Program generation: Like fuzz-d, DafnyFuzz generates a program in a fairly standard fashion by constructing an AST on the fly, tracking the context in which each program statement is generated so that e.g. a `continue` statement is only generated inside a loop. Upon generating a valid statement, at the top level of the program, it is executed in memory, mutating the statement to avoid defects, such as divide by zero operations, and changing the state space due to the effects of the statement.

To enable comparison of the compiled programs, DafnyFuzz prints all the variable values upon exiting a scope. This enables comparison between executions of Dafny that should produce the same results.

b) Metamorphic testing: When a DafnyFuzz-constructed AST is converted into Dafny syntax, there are several operations that can be emitted in different, equivalent ways. For example, an expression node $e_1 + e_2$ can be emitted equivalently as $e_2 + e_1$. DafnyFuzz exploits such equivalences to generate a set of *equivalent* programs that should all produce the same result. This is useful for testing the internal AST representation of the Dafny compiler to ensure that equivalent programs are not mutated in an unexpected way during the parsing and resolution stage. Such bugs may evade differential testing, if a defect during components of the compiler shared by all back ends leads to all back ends behaving in an identically wrong fashion.

Additionally, equivalence can be proved through the control flow the program. For example, a program statement `if (c) { A } else { B }` can be emitted equivalently as `A` assuming the boolean `c` can be guaranteed to be truthy at every invocation of the statement. This is because `B` is considered as dead-code, allowing for it to be removed with no impact to the execution of the resultant program. This is useful as each statement is executed in memory during the value tracking process allowing for dead-code to be eliminated when converting the program to Dafny syntax. This also tests the Dafny compiler does not perform unexpected mutations due to dead code during the parsing and resolution which would lead to incorrect compiled results in all back-ends.

c) Value tracking: DafnyFuzz uses *value tracking* to ensure that the program that is generated is free from defects such as out-of-bounds array accesses (so that it should indeed produce some well-defined output). This process avoids the Csmith [52] approach of always using safe wrapper replacement operations which would limit the capabilities of the programs generated. Value tracking also provides an alternative test oracle, because the actual output of the program can be compared with the expected output predicted by value tracking. This is similar to how the YARPGen [26] and Orange3 [29] tools work, but DafnyFuzz incorporates variations to allow reduced restrictions upon loops and reuse of methods and functions.

A challenge associated with value tracking is to ensure that the semantics of Dafny are strictly adhered to. An example that surprised us here is that arithmetic in Dafny is performed according to Euclidean algorithms, rather than geometrical arithmetic used in most modern programming languages. With Euclidean arithmetic, the result of a division, a/b should produce an integer quotient q and a natural number remainder r strictly smaller than the absolute value of the denominator (i.e. $0 < r < |b|$). This is done such that $a = b \times q + r$ holds. Accurate value tracking in DafnyFuzz required implementing division and modulo operations in this way in the value tracker.

The value tracking is used to create the self-checking oracle by outputting the value of each variable at upon executing a print statement, allowing the output of executing the program to be compared with an independently formed expectation. Most bugs were identified using differential testing with the self-checking oracle; however, one bug [36] was also identified by the metamorphic testing oracle above.

IV. Practical Impact on the Dafny Compiler and Language Specification

Our testing work so far has led to the reporting of 14 issues found by fuzz-d, of which 6 have been fixed, and 12 issues found by DafnyFuzz, of which 5 have been fixed. Among these, 5 have been tagged by the Dafny team as soundness issues, indicating that they have the potential to compromise the reliability of deployed, formally verified software by causing the software to behave incorrectly at runtime. We have also identified a further 4 issues which are not yet categorised, but we believe to be soundness related.

We provide a detailed overview of the GitHub issues that correspond to these in Table I. For any issues which are not categorised, we provide a predicted categorisation marked by an asterisk (*).

As discussed in the introduction, bugs can be broadly categorised as: *soundness* bugs, where the Dafny compiler generates incorrect code in a downstream programming language, and where the fact that this code is incorrect only becomes clear when the generated code is executed; *invalid code* bugs, where the Dafny compiler generates code that is statically rejected by the compiler of the downstream programming language (e.g. because it is syntactically incorrect or violates static typing rules); and *crash* bugs, where the Dafny compiler crashes (e.g. due to an assertion failure).

Soundness bugs are the most serious class of bugs, since they have the potential to cause applications that have been formally verified at the Dafny level to violate their specifications when they are deployed. Invalid code and crash bugs may be an impediment to deploying Dafny applications, but are less severe than soundness bugs because they are caught before execution time. When the result produced by Dafny cannot be executed, through invalid code or compiler crashing, the offending feature poses a "fuzz blocker" as they prevent the output of the compiler being tested and any miscompilation errors being detected.

Our fuzzing campaign also highlighted a number of clarity problems with features of the Dafny language, leading to improved documentation.

We now detail a selection of soundness bugs (Section IV-A), invalid code and crash bugs (Section IV-B) and language design issues (Section IV-C) that were brought to light by the use of our tools.

A. Soundness bugs

Forall expression inside match statement [49] (found by fuzz-d). Of the language features tested by fuzz-d, pattern-matching-related features were among those which caused the most issues. Fig. 2 shows a minimised version of a program generated by fuzz-d. This features a `forall` parallel assignment inside a match statement, and proved particularly problematic, causing issues across all 5 back-ends: for three back-ends (C#, Java and Go), Dafny produced invalid code that caused exceptions during compilation by the relevant back-end. However, for the interpreted back-ends (Python and JavaScript), this manifested as a soundness bug because it resulted in *runtime* exceptions. Because this bug affected all back-ends it was likely to be contained in common compiler code—this was demonstrated in the fix the developers released for this issue, which related to improving the deep copying logic of the AST class corresponding to the forall feature.

Runtime cardinality limit of multisets [50] (found by fuzz-d). An interesting edge case in the C# and Python implementations of multisets was identified by fuzz-d. When trying to take the modulus of a multiset whose size is greater than the maximum supported integer value, runtime exceptions

Issue	Status	Categorisation	Component	Description
#4894	Crash	Invalid Code	Resolver	Nested match within loop with assignment after break.
#4358	Unconfirmed	Documentation*	Compiler (all back-ends)	Inconsistent printing of strings across back-ends
#4141	Fixed	Wrong Result (Soundness)	Compiler (Java)	Incorrect use of deep equality comparing arrays
#4130	Confirmed	Code Generation	Compiler (Python)	Memory issues with nested lambdas
#4061	Unconfirmed	Invalid Code*	Compiler (Java)	Generated code with invalid use of variables
#4032	Fixed	Wrong Result (Soundness)*	Compiler (Java)	Incorrect use of deep equality comparing arrays
#4011	Unconfirmed	Wrong Result (Soundness)*	Compiler (C#)	Incorrect equality of multisets with 0-cardinality elements
#4007	Fixed	Invalid Code	Compiler (Python)	Invalid syntax in generated code
#4004	Fixed	Crash	Parser	Match with parallel assignment
#3988	Fixed	Crash (Soundness)*	Compiler (C#, Python)	Runtime cardinality limit for multisets
#3987	Fixed	Crash (Soundness)	Compiler (Python, JS)	Referencing undeclared variable in generated code
#3978	Fixed	Invalid Code	Compiler (Go)	Miscompilation related to use of `continue`
#3969	Confirmed	Crash	Verifier	Assertion failure during translation
#3966	Fixed	Crash	Parser	Assertion failures with nested match statements
#3952	Confirmed	Invalid Code	Compiler (Java)	Miscompiling combinations of tertiary and comparison operators
#3950	Confirmed	Incorrect rejection	Resolver	Incomplete type checking for multiset operations
#3949	Fixed	Documentation	-	Unexpected handling of variables in match cases
#3932	Won't Fix	Error Reporting	Parser	Handling of a clash in the grammar
#3910	Unconfirmed	Invalid Code	Compiler (Java, Go, C#)	Multiple issues using variables in pattern matches
#3906	Unconfirmed	Crash	Verifier	Boogie – Internal translation error
#3887	Unconfirmed	Invalid Code	Compiler (Java)	Type representation issues for maps using chars
#3874	Fixed	Invalid Code (Soundness)	Compiler (Python)	Multiset equality issues
#3873	Confirmed	Invalid Code (Soundness)	Compiler (JS)	Type representation issues for maps with array keys
#3871	Confirmed	Wrong Result (Soundness)	Compiler (Java)	Incorrect cardinality of sets and multisets
#3856	Duplicate	Wrong Result (Soundness)	Compiler (JS, Go)	Incorrect internal representation of maps
#3854	Confirmed	Invalid Code	Compiler (Java)	Multiple issues related to type representation and class casting.

TABLE I: Summary of issues reported to Dafny developers

```
datatype D = A | B

method Main() {
    match A {
        case A ⇒
            var a: array<int> := new int[24](i1 ⇒ i1);
            forall i2 | 0 ≤ i2 < a.Length {
                a[i2] := i2;
            }
        case _ ⇒ {}
    }
}
```

Fig. 2: Test case inserting a parallel assignment inside a match statement, resulting in invalid generated code that triggered compile-time and runtime exceptions in the back-ends.

are thrown due to an arithmetic overflow from trying to fit the modulus into an integer type. The other three back-ends are able to handle this case using Dafny's `BigInteger` implementation, which allows Dafny to have (theoretically) unbounded integer values, therefore this is clearly a missed implementation detail in the C# and Python back-ends.

Fig. 3 shows the test case which caused this error for Python—it sets multiplicity of the value `1` in multiset `x` as 2^{64}, which is greater than the maximum allowed value for an index-type integer. Consequently, when the test case was run, the error `OverflowError: cannot fit 'int' into an index-sized integer` was thrown.

Multi-level Multisets Wrong Result [45] (found by fuzz-d). Running the program of Fig. 4 will trigger a wrong result bug in the C# back-end. It is clear that the output of the program should be `true`, since `a[true := 0]` is equivalent to `multiset{false}`. However, the C# back-

```
method Main() {
    var x: multiset<int> := multiset{};
    x := x[1 := 18446744073709551616];
    print |x|;
}
```

Fig. 3: Test case demonstrating the runtime cardinality limit of Python multisets. Generated code tries to place the value 2^{64} inside an integer type, which triggers a runtime overflow exception.

```
method Main() {
  var a: multiset<bool> := multiset{false, true};
  var b: multiset<multiset<bool>> := multiset{a[true :=
      0]};

  print b = multiset{multiset{false}}, "\n";
}
```

Fig. 4: Test case which triggers a wrong result in the C# back-end, which outputs `false` instead of `true`.

end outputs `false` at runtime, while all other tested back-ends output `true`. Although the Dafny developers have not confirmed this bug, we believe it to be a soundness issue as it impacts the runtime safety of deployed programs using the C# back-end. It is likely to be caused by logical issues in the C# definition of multiset equality.

Incorrect compilation of set and multiset cardinality operations [38] (found by Dafny-Fuzz). The program of Fig. 5, reduced from a program generated by DafnyFuzz, triggered a miscompilation in the Java back-end of Dafny. This program prints the cardinality of (a) a set containing a single array, and (b) a multiset containing a single array. In both cases it is

```
method Main() {
    var v_array: array<int> := new int[] [1, 2];

    var v_int_s: int := |{v_array}|;
    assert(v_int_s == 1);
    print v_int_s, "\n";

    var v_int_m: int := |multiset{v_array}|;
    assert(v_int_m == 1);
    print v_int_m, "\n";
}
```

Fig. 5: Miscompilation by Java back-end: incorrect cardinality of `set` and `multiset<array>`

```
method m_method_12() returns (ret_1: seq<array<int>>)
{
  var v_array_1: array<int> := new int[2] [25, 2];
  var v_array_2: array<int> := new int[3] [16, 9, 17];
  var v_seq: seq<array<int>> := [v_array_1, v_array_2];
  return v_seq;
}

method Main() returns ()
{
  var v_seq_1: seq<array<int>> := m_method_12();
  var v_seq_2: seq<array<int>> := m_method_12();
  print v_seq_34 == v_seq_36;
}
```

Fig. 6: Miscompilation by Java back-end: incorrect return equality check

clear that the cardinality should be 1. However, the Java code emitted by the Dafny Java back-end yields 2 as the cardinality in both cases. This bug was found thanks to the value tracking oracle of DafnyFuzz. Additionally, differential testing confirmed that only the Java back-end behaves incorrectly here; the JavaScript, C#, Python and Go back-ends compute correct results for these cardinalities.

Miscompilation by Java back-end: Inconsistent equality between return values of methods [37] (found by Dafny-Fuzz). The program shown in Fig. 6, reduced from a program generated by DafnyFuzz, triggered a miscompilation in the Java back-end of Dafny. This program prints the equality of the return values from subsequent calls to the same method. Given each array creates a new object in memory, the equality should return `false`. However, the Java code emitted by the Dafny Java back-end yields `true` as the result of the comparison. This bug was found thanks to the value tracking oracle of DafnyFuzz. Additionally, differential testing confirmed that only the Java back-end behaves incorrectly here; the JavaScript, C#, Python and Go back-ends compute correct results for these equality check.

This root cause of this issue has been identified and fixed, being an incorrect use of deep-equality to check the equality of each element within the sequence. This leads the performing an equality check between the individual elements of the arrays and hence the incorrect result being returned.

B. Invalid code and crash bugs

Match statement crash bug [39] (found by fuzz-d and DafnyFuzz). The program shown in Fig. 7 demonstrates a

```
method Main() returns ()
{
  match 8 {
    case _ => {
      var v_bool: bool, v_real: real := true, match 15.06
          {
        case _ => 6.58
      };
      print v_bool, " ", v_real, "\n";
    }
  }
}
```

Fig. 7: Match parallel assignment crash bug

```
datatype D0 = DC1(cf1: int, cf2: int)

method Main() {
    for i := 1 to 2 {
        print DC1(1, 2).(cf1 := i);
    }
}
```

Fig. 8: Test case causing a Java compilation crash due to non-final variables referenced in lambdas in generated code.

crash bug that was discovered during the development process of the fuzzers. This occurred due to performing a parallel assignment within a match statement. This bug was introduced during a fix of a different bug, and proved to be a fuzz blocker, with the throughput of successfully compiled programs reducing to 3% of the previous throughput of random programs produced by DafnyFuzz. This emphasised the importance of testing against the latest version of the Dafny compiler, as this may have gone unnoticed and released as an official stable version.

Datatype Update inside For Loop [48] (found by fuzz-d). We experienced a lot of issues testing Dafny's Java back-end, finding that it was particularly vulnerable to generating invalid code which would trigger errors in the Java compiler. These issues were so common that they became fuzz blockers, preventing us from testing the Java back-end thoroughly as only very simple generated programs could successfully pass through the compiler. Fig. 8 shows an example that causes invalid Java code to be output. Internally, the generated code uses lambdas for the datatype operations; however, this causes issues when it tries to use the loop counter `i` inside these lambdas, despite `i` not being final. Hence, the Java compiler rejects the generated code.

Error in the handling of `continue` statements [33] (found by Dafny-Fuzz). The program of Fig. 9, reduced from a program generated by DafnyFuzz, triggered a bug in the Dafny Go back-end. The Go code generated by Dafny implemented the semantics of `continue` using a `goto` statement, but inadvertently violated a rule of the Go language that prohibits a `goto` from jumping over a declaration of a variable that is still in scope at the target label for the `goto`. Thus the generated Go code did not compile. This is an example of a bug that did not require the metamorphic or value tracking oracle to be detected, since the problem manifests by the Dafny-generated

```
method Main()
{
 for v_int_7 := 3 to 18
 {
  if (false) {
    continue;
  }
  var v_int_93 := 1;
  print v_int_93, "\n";
 }
}
```

Fig. 9: A Dafny program for which the Go back-end generated invalid code

code failing to compile (rather than compiling successfully but into a form that does not respect the semantics of the input Dafny program). This issue has been fixed by the Dafny compiler developers.

C. Language design issues

Lack of clarity about interchangeability between strings and character sequences [34] (found by DafnyFuzz). The program detailed in Fig. 10 was unexpectedly rejected by Dafny's parsing stage during testings, caused due to a type mismatch between `string` and `seq<char>`. According to the Dafny documentation [9], these types should be synonymous, stating "A special case of a sequence type is seq<char>, for which Dafny provides a synonym: string". This suggests that the types should be interchangeable, as can be seen on line 3 which produces the result `true`. However, the program is rejected during the parsing stage as matching on the sequence type is invalid. This was detected by DafnyFuzz where the it interchangeably uses the synonymous types. From testing the subset of the language, this was the the only case found where the types were not interchangeable. Dafny's developers have not yet responded to this issue.

```
1 method Main() returns ()
2 {
3   print "str" = ['s', 't', 'r'];
4   var v_string_9: string := (match "GX" {
5     case ['G', 'X'] => "123"
6     case _ => "456"
7   });
8   print v_string_9;
9 }
```

Fig. 10: Test case demonstrating inconsistency in the Dafny language specification

This leads to a related issue when printing strings and seq<char> types, shown in Fig. 11 where the printing of the string varies based on the surrounding context. When printing the sequence of characters generates an equivalent output to printing the equivalent string representation, displayed as juxtaposed characters. However when the string is contained within another structure, in this case a tuple, it is then printed as separate characters. While the Dafny language maintainers commented that this was an intentional, the change in behaviour when printing makes verifying the output of the program have an additional challenge which had to be worked around by both fuzz-d and DafnyFuzz.

```
method Main() {
  print "abcd", "\n";
  print ['a', 'b', 'c', 'd'], "\n";
  print (1, "abcd"), "\n";
}

// Output:
// abcd
// abcd
// (1, ['a', 'b', 'c', 'd'])
```

Fig. 11: Test case demonstrating inconsistency in the printing of strings

Parsing ambiguity: clash between generics and comparison operators [46] (found by fuzz-d). To our surprise, the program of Fig. 12 was rejected by Dafny's resolver. This was identified to be caused by a clash in the language grammar—line 3 could either be parsed as a parallel assignment with two comma-separated expressions on the right-hand side, or as a parallel assignment where the values are returned from a parameterised function call. Dafny parsed this as the latter, whereas the program requires the former and therefore the program was rejected. This design decision could be better documented and rejections following this case could provide better diagnostic error messages; however, these changes have yet to be made. It might also be that more context-aware parsing could allow Dafny to handle both cases; however, the Dafny developers do not feel that this would be worthwhile since there already exists a workaround to the problem.

```
1 method Main() {
2     var x := 1;
3     var z1, z2 := x < x, x > (1);
4 }
```

Fig. 12: Test case demonstrating a clash between generics and comparison operators

Semantics of match statements with variables [35] (found by DafnyFuzz). The program detailed in Fig. 13 was compiled correctly and executed by all backends, each producing the identical result of `Unexpected`. Match statement documentation [9] had stated "The cases must be exhaustive, but you can use a wild variable ('_') or an as yet unused simple identifier to indicate 'match anything' ", which suggested that the case would compare the existing variables value. This also produced the warning `this branch is redundant` in relation to line 7 which was unclear given the reduced test case would not enter the first case. This lead to a change in the documentation, to increase clarity on how identifiers can be used in cases, now stating "The cases must be exhaustive, but you can use a wild variable ('_') or a simple identifier to indicate 'match anything' ". The bug, detected due to value tracking by DafnyFuzz, lead to clearer documentation and a better defined language. Ideally, this would have shown a

TABLE II: Coverage experiment results, showing line and branch coverage percentages

Run	Line (%)	Branch (%)
1	32.59	30.51
2	32.54	30.50
3	32.50	30.46
Avg	32.54	30.49

(a) Coverage results for XD-Smith

Run	Line (%)	Branch (%)
1	46.56	40.83
2	46.51	40.81
3	46.38	40.75
Avg	46.48	40.80

(b) Coverage results for fuzz-d

Run	Line (%)	Branch (%)
1	38.55	28.99
2	38.48	35.30
3	38.59	35.36
Avg	38.54	33.22

(c) Coverage results for Dafny-Fuzz

Line (%)	Branch (%)
74.00	69.24

(d) Coverage Results for Dafny Compiler Regression Test Suite

warning to indicate that a variable previously declared in the scope would be shadowed in the match body.

```
method Main() returns () {
  var v: int := 1;
  match 0 {
    case v ⇒ {
      print "Unexpected";
    }
    case _ ⇒ {
      print "Expected";
    }
  }
}
```

Fig. 13: Test case demonstrating a lack of clarity in documentation of variables in match statements

V. Controlled Experiments

To gain further insights into the overlap and complementarity between fuzz-d, DafnyFuzz, XDsmith and the Dafny test suite, we conducted controlled experiments to evaluate code coverage and mutation coverage on the Dafny compiler.

A. Statement coverage

We conducted experiments for statement coverage over an instrumented version of the Dafny codebase created using the `coverlet` tool [40]. For each of the fuzzing tools XDsmith, fuzz-d and DafnyFuzz, we performed 8 hour testing campaigns and calculated the coverage that their generated programs could achieve over the instrumented codebase. To account for the stochastic nature of fuzzing, we performed three repeat runs. As a baseline, we compare with the coverage achieved by the existing Dafny compiler regression test suite.

The more flexible generation approach taken by fuzz-d and DafnyFuzz clearly shows promising improvements over the existing tool XDsmith (Table II). We attribute the difference in coverage between our two tools to fuzz-d supporting a more complete subset of the features of Dafny, in particular through its support for Dafny's object-oriented features. There are, however, some areas that DafnyFuzz covers which fuzz-d cannot—notably for generic datatypes.

There is evidently still quite a large portion of the codebase left uncovered by the tools, but covered by the Dafny compiler regression test suite, including verification features which were not the focus of the designed fuzzers. In spite of this, we were able to identify edge cases within the Dafny language which were not covered by the regression test suite, but were covered by our tools' generated test programs. We present two examples below featuring code which could be generated by our tools, but is not included in Dafny's test suite.

Example 1: Cloning of Sequence/Multiset Bounded Pools (detected by fuzz-d)

```
method Main() {
    var a := [1, 2, 3];
    match true {
        case _ ⇒ var b := map x | x in a :: x := x * x;
    }
}
```

Fig. 14: Smallest test case covering cloning of the `SeqBoundedPool` class

Within Dafny, comprehensions are considered as being formed of three parts: a list of bound variables, a *range* which confines these variables to a finite range of values, and a *term* which represents the expression used to evaluate the comprehension's elements. The range can take multiple different forms, and each of these results in the possible range values being represented as a *bounded pool*—for example, providing an int range (e.g. `0 <= i < 10`) results in an `IntBoundedPool` while providing a data structure (e.g. `x in [1, 2, 3]`) results in a bounded pool corresponding to that data structure. The lines omitted by Dafny's compiler regression test suite result from an edge case related to the bounded pools for sequences and multisets.

Each bounded pool implements a function `Clone()` providing a deep copy of itself. This is necessary since the Dafny internal AST representation is mutable, meaning that if the current AST state needs to be cached or maintained for later use, it must be cloned so as not to be changed by later transformations. However, the Dafny regression tests omit testing the clone functions for comprehensions bounded by the contents of sequences or multisets. This is covered fuzz-d following its support of match statements and comprehensions, and a reduced test case obtained from fuzz-d covering cloning of sequence bounded pools is shown in Figure 14.

Example 2: Missed Binary Operators (detected by fuzz-d and DafnyFuzz)

```
method Main() {
    print true ⟸ false;
    print true ⟺ true;
    print map[1 := 1] ≠ map[1 := 2];
    print multiset{1} ≠ multiset{2};
}
```

Fig. 15: A small test case with statements to achieve coverage for missing binary operators

The Dafny test suite notably has a number of missing cases for binary operators: for explies (<==), iff (<==>) and map/multiset not equals (`!=`). Missing these operators could be argued as a potential weakness in Dafny, since binary operators are among the most commonly used language features and therefore it is important the tests are able to identify any compile issues related to all possible types. Figure 15 shows a simple test case where each line would introduce coverage for one of the above operators. These statements are representative of code generated by our tools.

B. Mutation coverage

We used a C# mutation testing tool, Stryker [41], to perform four controlled experiments evaluating mutation coverage using fuzz-d, DafnyFuzz, XDsmith and the Dafny compiler regression test suite respectively. Stryker identifies and injects mutations—including arithmetic, logical, initialisation and assignment-based—into the Dafny codebase, with the test suite then being invoked to see if at least one test fails, in which case the mutant is *killed* by the test suite. Due to the compute-intensive nature of mutation testing—Stryker generates over 80,000 mutants over all of the DafnyCore module—it is necessary to limit the scope of the experiment to mutations affecting only the core compiler file, `SinglePassCompiler`, for which Stryker generates 2967 mutants.

Since randomised compiler testing usually involves generating an indefinitely-large sequence of test programs, rather than working with a fixed test suite, while Stryker requires a fixed test suite, it was necessary for us to use fuzz-d, DafnyFuzz and XDsmith to form test suites representative of the capabilities of each tool. These consist of programs generated by the tool and their expected output over each back-end. To make this as fair as possible in comparison with the Dafny compiler regression test suite, we compiled the test suites by generating and running programs with the fuzzers for the time taken to execute the Dafny regression tests, with the aim of all four test suites therefore having approximately the same amount of time to test Dafny.

TABLE III: Mutation coverage results over the `SinglePassCompiler`.

Test Suite	Killed	Survived	Timed Out
fuzz-d	2960	1	6
DafnyFuzz	2956	11	0
XDsmith	2957	0	10
Dafny Compiler Regression Tests	2939	28	0

The results of this experiment are shown in Table III. A mutant classified as "timed out" is still detected [41], but not killed—for example, a test suite cannot kill a mutant that induces an infinite loop; instead Stryker would mark this as "timed out" after the test suite failed to complete within a certain time budget.

While the Dafny compiler regression tests kill the majority of the mutants, a number of mutants survived. Of these, 21 mutants are located in functions responsible for compiling the direct comparison of integer types with zero. This is treated as a special case aimed at simplifying the comparison, for example from `x < 0` to `x.Sign == -1`. All three fuzzers were able to kill these 21 mutants (those which survived in the case of fuzz-d and DafnyFuzz were located in other parts of the codebase). This demonstrates that, similarly to our findings with statement coverage, fuzzing is able to identify missing cases in the Dafny compiler regression test suite.

We were surprised that our tools did not outperform XDsmith in this experiment, given how many more features are supported in comparison. Understanding this result further will require additional investigation.

VI. Related Work

Our fuzz-d tool principally uses *differential testing* [27] as a test oracle. Differential testing is widely used for compiler testing: it was popularised by the Csmith project for C compilers [52], and has since been used in the testing of compilers for languages such as OpenCL [25], Java [7], Verilog [18] and Rust [31]. The XDsmith tool was the first to apply differential testing to compilers for the Dafny language [20]. Compared with XDsmith, our fuzz-d and DafnyFuzz tools handle a substantially larger fragment of the Dafny language.

DafnyFuzz uses *metamorphic testing* [6], which has also been widely applied in the domain of compiler testing: an early approach involved generating equivalent programs [44]; a family of techniques termed "Equivalence Modulo Inputs" testing involve creating equivalent versions of existing programs by applying mutations that do not affect the expected output of a program for a given input [21], [42]; and this idea has been extended to the more general notion of applying semantics-preserving transformations to an existing program to obtain families of equivalent programs [12]–[14].

Another means for obtaining a test oracle for compiler testing is to produce self-checking programs. This is the approach taken by the Orange family of C compiler testing tools [28], [29], and it is also a facility that the YARPGen C/C++ compiler testing tool offers [26] (although in practice YARPGen is reported to have been used for differential testing). Both fuzz-d and DafnyFuzz offer the ability to produce self-checking programs, and the XDsmith tool uses generation-time analysis to produce programs with known expected verification outcomes [20].

A useful by-product of our work has been the clarification of some aspects of the Dafny language specification, as discussed in Section IV-C. The potential of compiler fuzzing to inform programming language design was noted in a report on the industrial deployment of randomised testing techniques for GPU compilers, which led to clarifications being made to the specification of the WebGPU shading language [11].

VII. Conclusions and Future Work

We have presented details of the design and implementation of fuzz-d and DafnyFuzz, two new tools for automatically testing the reliability of the Dafny compiler via random generation of programs. These tools have allowed the discovery of a substantial number of new compiler bugs, several of which have been fixed, and they go beyond the capabilities of an existing automated testing tool for Dafny, XDsmith.

Avenues for future work include: incorporating fuzz-d and DafnyFuzz into the continuous integration infrastructure of the Dafny project, so that they are either run continuously, or so that a limited amount of fuzzing is done on a per-commit basis; testing a new Dafny back-end that emits Rust code, that was not in place when we conducted our bug-finding campaign; and using a combination of fuzzing and mutation testing to synthesise additional regression test suites for Dafny that fill gaps in the coverage that the current regression test suite achieves.

REFERENCES

[1] Amazon Web Services, "Amazon verified permissions," 2023, https://aws.amazon.com/verified-permissions/.

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

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

[4] ——, "AWS verified access," 2023, https://aws.amazon.com/verified-access/.

[5] 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

[6] 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.

[7] 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

[8] Consensys, "evm-dafny," 2023, https://github.com/Consensys/evm-dafny.

[9] Dafny Project, "Dafny GitHub repository," 2023, https://github.com/dafny-lang/dafny.

[10] 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

[11] 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

[12] 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

[13] 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

[14] 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

[15] A. Groce, "Let a thousand flowers bloom: on the uses of diversity in software testing," in *Onward! 2021: Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Virtual Event / Chicago, IL, USA, October 20-22, 2021*, W. D. Meuter and E. L. A. Baniassad, Eds. ACM, 2021, pp. 136–144. [Online]. Available: https://doi.org/10.1145/3486607.3486772

[16] W. Hatch, P. Darragh, and E. Eide, "Xsmith," 2023, https://docs.racket-lang.org/xsmith/index.html, accessed 18 October 2023.

[17] 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

[18] 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

[19] 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.

[20] 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

[21] 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

[22] 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

[23] 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

[24] ——, "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

[25] 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

[26] V. Livinskii, D. Babokin, and J. Regehr, "Random testing for C and C++ compilers with YARPGen," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 196:1–196:25, 2020. [Online]. Available: https://doi.org/10.1145/3428264

[27] 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

[28] E. Nagai, A. Hashimoto, and N. Ishiura, "Reinforcing random testing of arithmetic optimization of C compilers by scaling up size and number of expressions," *IPSJ Trans. Syst. LSI Des. Methodol.*, vol. 7, pp. 91–100, 2014. [Online]. Available: https://doi.org/10.2197/ipsjtsldm.7.91

[29] K. Nakamura and N. Ishiura, "Introducing loop statements in random testing of C compilers based on expected value calculation," in *Proceedings of the Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2015)*, 2015, p. 226–227.

[30] T. Parr, "ANTLR," 2023, https://www.antlr.org/, last accessed 2023-10-24.

[31] 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

[32] D. Sheth, “Dafnyfuzz github repository,” 2023, accessed: 25th October 2023. [Online]. Available: https://github.com/dilan-s/dafny-verifier

[33] ——, “Go continue miscompilation error,” 2023, https://github.com/dafny-lang/dafny/issues/3978.

[34] ——, “Inconsistencies in handling of string and sequence of char,” 2023, https://github.com/dafny-lang/dafny/issues/4672.

[35] ——, “Inconsistencies in handling of string and sequence of char,” 2023, https://github.com/dafny-lang/dafny/issues/3949.

[36] ——, “Incorrect cardinality of set and multiset¡array¿ - java miscompilation error,” 2023, https://github.com/dafny-lang/dafny/issues/3871#issuecomment-1520854345.

[37] ——, “Java inconsistent equality between return values of methods,” 2023, https://github.com/dafny-lang/dafny/issues/4141.

[38] ——, “Miscompilation for java set and multiset cardinality,” 2023, https://github.com/dafny-lang/dafny/issues/3871.

[39] ——, “Parallel assignment including match epression within match statement crash,” 2023, https://github.com/dafny-lang/dafny/issues/4004.

[40] T. Solarin-Sodara, “Coverlet: Cross-platform code coverage,” 2023, accessed: 19th October 2023. [Online]. Available: https://github.com/coverlet-coverage/coverlet

[41] Stryker Project, “Stryker Mutator,” 2023, accessed: 19th October 2023. [Online]. Available: https://stryker-mutator.io/docs/stryker-net/introduction/

[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] A. Usher, “C# wrong result: Multi-level multisets,” 2023, https://github.com/dafny-lang/dafny/issues/4011.

[46] ——, “Comma seperated expressions parsed incorrectly as generics,” 2023, https://github.com/dafny-lang/dafny/issues/3932.

[47] ——, “fuzz-d GitHub repository,” 2023, https://github.com/fuzz-d/fuzz-d.

[48] ——, “Java compilation crash: Datatype update inside for loop,” 2023, https://github.com/dafny-lang/dafny/issues/4061.

[49] ——, “JS, Python runtime exception: Forall inside match statement,” 2023, https://github.com/dafny-lang/dafny/issues/3987.

[50] ——, “Runtime exceptions for modulus of large multiset,” 2023, https://github.com/dafny-lang/dafny/issues/3988.

[51] VMware Labs, “Verified BetrFS,” 2023, https://github.com/vmware-labs/verified-betrfs.

[52] 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