Testing Dafny (Experience Paper)

Ahmed Irfan
rfaahm@amazon.com
Amazon Web Services (AWS)
USA

Sorawee Porncharoenwase
sorawee@cs.washington.edu
University of Washington
USA

Zvonimir Rakamarić
zvorak@amazon.com
Amazon Web Services (AWS)
USA

Neha Rungta
rungta@amazon.com
Amazon Web Services (AWS)
USA

Emina Torlak
torlaket@amazon.com
Amazon Web Services (AWS)
USA

ABSTRACT

Verification toolchains are widely used to prove the correctness of critical software systems. To build confidence in their results, it is important to develop testing frameworks that help detect bugs in these toolchains. Inspired by the success of fuzzing in finding bugs in compilers and SMT solvers, we have built the first fuzzing and differential testing framework for Dafny, a high-level programming language with a Floyd-Hoare-style program verifier and compilers to C#, Java, Go, and Javascript.

This paper presents our experience building and using XDsmith, a testing framework that targets the entire Dafny toolchain, from verification to compilation. XDsmith randomly generates *annotated programs* in a subset of Dafny that is free of loops and heap-mutating operations. The generated programs include preconditions, postconditions, and assertions, and they have a known verification outcome. These programs are used to test the soundness and precision of the Dafny verifier, and to perform differential testing on the four Dafny compilers. Using XDsmith, we uncovered 31 bugs across the Dafny verifier and compilers, each of which has been confirmed by the Dafny developers. Moreover, 8 of these bugs have been fixed in the mainline release of Dafny.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

KEYWORDS

program verification, fuzzing, differential testing

ACM Reference Format:
Ahmed Irfan, Sorawee Porncharoenwase, Zvonimir Rakamarić, Neha Rungta, and Emina Torlak. 2022. Testing Dafny (Experience Paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3533767.3534382

ISSTA '22, July 18–22, 2022, Virtual, South Korea

ACM ISBN 978-1-4503-9379-9/22/07.
https://doi.org/10.1145/3533767.3534382

1 INTRODUCTION

The correctness of compilers, static analyzers, and formal verification engines is key to ensuring that the programs they compile, analyze, and verify are correct. Bugs in these tools can have serious consequences: a soundness bug can cause the tool to accept an incorrect program, while a precision bug can cause it to reject too many correct programs. In principle, both kinds of bugs can be eliminated through formal verification. In practice, however, the cost of formal verification remains prohibitive, with teams of experts taking decades to verify a single toolchain (see, e.g., [32]). This cost becomes astronomical when the target is an *ecosystem* of related tools: a verifier together with a set of compilers for a rich general-purpose language. In such a setting, effective testing becomes key to increasing confidence in the correctness of the ecosystem—and all applications that depend on it for their correctness.

This paper presents our experience developing and applying the first fuzzing and differential testing framework for Dafny [12, 30], a high-level programming language equipped with a Floyd-Hoare-style [16, 23] verifier and compilers to C#, Java, Go, and JavaScript. Dafny is used broadly for building verified software. For example, it has been used to prove the correctness of high-level distributed protocols [22, 24], as well as to build low-level verified systems, such as a verified storage system [20] and a verified security monitor [14]. The correctness of all of these systems rests on the correctness of the Dafny verifier and compilers.

Our testing framework, XDsmith, targets the entire Dafny ecosystem, from verification to compilation. The Dafny verifier takes as input a Dafny program annotated with preconditions, postconditions, loop invariants, and assertions, and it checks that the program meets its specification by reducing the verification task to a satisfiability modulo theories (SMT) [4, 5] query. If the verifier accepts the program, the compilers then translate it to their respective target languages. XDsmith tests this workflow end-to-end. It works by randomly generating annotated Dafny programs to test the soundness and precision of the Dafny verifier, and by using these programs to perform differential testing [34] on the Dafny compilers.

The core technical problem we address in XDsmith is how to randomly generate *annotated programs* for testing a Floyd-Hoare-style verifier. The problem of randomly generating well-formed and well-typed programs has been studied extensively in prior work, and we build on the prior tool Xsmith [21] to solve this problem for the Dafny language. The new problem that XDsmith addresses is that of generating *annotations* for these programs that can reveal soundness and precision bugs in the verifier. In particular, XDsmith must be able to annotate a program with random specifications that the program satisfies as well as those that the program violates. If the verifier accepts the latter, the generated test case demonstrates a soundness bug, and if the verifier rejects the former, the test case demonstrates a potential precision bug (or reveals an expected source of incompleteness in the verifier).

However, generating such annotations for arbitrary Dafny programs is undecidable. As a general-purpose language, Dafny includes loops, recursion, non-linear integer arithmetic, quantified formulas, and many other features that make reasoning about Dafny programs undecidable. Moreover, even if we restricted our test generator to a decidable subset of Dafny, the resulting problem would still be intractable for annotations that generalize to all inputs. In fact, it would amount to building a program synthesizer and an oracle verifier for the chosen subset of Dafny—a daunting technical challenge for all but the narrowest subsets of Dafny that are least interesting to test (e.g., loop-free bitvector programs).

To make the annotation problem tractable for a broad subset of Dafny, we make two key design choices. First, we focus on Dafny programs that are free of loops and recursion, but that include arrays, sequences, maps, sets, multisets, integers, Booleans, and strings. Second, we focus on generating annotations that are free of quantifiers and that are *not* general: instead, they are constructed so that the program satisfies or violates them on a specific (randomly chosen) execution. Crucially, these two choices reduce the problem of verifying the resulting annotated program to testing. Because the program is loop-free, it is guaranteed to terminate. Because the annotations apply to a known execution, we can determine whether the Dafny verifier should accept or reject the annotated program simply by running it. Finally, our design enables us to formulate this carefully restricted annotation generation problem as one of example-based syntax-guided synthesis [2, 19], and to solve it using an off-the-shelf synthesis tool [39].

We performed an extensive case study to evaluate our approach. We continuously applied XDsmith on Dafny over a period of three months. XDsmith generated random annotated Dafny programs and used them to test the Dafny verifier and compilers. In the process, we discovered 31 previously unknown bugs, all of which have been confirmed by Dafny developers. The bugs include soundness and precision issues in the verifier, as well as semantics-related issues in the compilers. We discovered 10 of these bugs by testing the verifier and the rest by testing the compilers. Eight bugs have already been fixed in the Dafny mainline release.

In summary, this paper makes the following contributions: (1) an approach for generating annotated programs to test Floyd-Hoare-style verifiers and their accompanying compilers; (2) an implementation of this approach in the XDsmith framework that targets the Dafny verifier and compilers; and (3) an evaluation demonstrating the effectiveness of XDsmith. Our results are encouraging and point to the value of developing tools for testing Floyd-Hoare-style verification ecosystems. While this paper and our implementation target Dafny, the core testing problem and ideas presented here apply more generally to verification toolchains that use Floyd-Hoare-style reasoning. We hope that the paper inspires further research into this important problem.

2 OVERVIEW OF XDsmith

XDsmith aims at increasing confidence in the Dafny ecosystem by finding bugs in the Dafny verifier and compilers. (The source code of XDsmith is publicly available at https://github.com/dafny-lang/xdsmith.) This section presents an overview of the XDsmith design and workflow. We begin with a short introduction to the Dafny language and verifier, then state the problem of testing Dafny components, and finally describe, at a high level, how XDsmith addresses this problem. Section 3 presents the details of the XDsmith's algorithms.

2.1 Brief Introduction to Dafny

Dafny is a high-level programming language designed from the ground up with verification in mind. It includes standard constructs for writing object-oriented, functional, and imperative code, and on top of those, it includes dedicated constructs for writing specifications and reasoning about program correctness. The basic specification constructs include preconditions, postconditions, and assertions. These annotations express predicates on program state, written in (a superset) of first-order logic. The Dafny verifier checks that an annotated program meets its specification on all possible inputs.

Methods. To illustrate, consider the following toy procedure that converts integers 1 and 0 to Boolean values `true` and `false`:

```
 1 method BitToBool(i: int)
 2   returns (b: bool)
 3 {
 4   if (i == 1) {
 5     b := true;
 6   } else {
 7     assert i == 0;
 8     b := false;
 9   }
10   print b;
11 }
```

Procedures are called *methods* in Dafny, and they encapsulate imperative code. All method inputs and outputs are given explicit types. Dafny lets us name outputs as well as inputs to a procedure, and the result of the procedure is the final value of the output variable. It is easy to see that our procedure returns `true` when the input is 1 and `false` when the input is 0.

Assertions. While `BitToBool` seems to do what we intended, it is not quite correct. In particular, the Dafny verifier rejects it and flags the *assertion* on line 7. The problem is that this assertion holds only if the method is called with values 0 or 1. If we called it with 2, the assertion would fail. The verifier detects this violation, letting us know that it cannot prove the assertion for all inputs.

Preconditions. To fix the problem, we write a *precondition* to tell the verifier that 0 and 1 are the only allowed inputs to `BitToBool`:

```
method BitToBool(i: int)
  returns (b: bool)
  requires i == 0 || i == 1
{ ... }
```

The verifier can prove the assertion for this restricted set of inputs, and it accepts the program. As usual in Floyd-Hoare-style proofs, the verifier will enforce the precondition for all calls to `BitToBool`.

Modular verification. Now that our method verifies, suppose that we want to use it to convert some bits to Booleans as follows:

```
15 method Main() {
16   var t := BitToBool(1);
17   var f := BitToBool(0);
18   assert t;
19   assert !f;
20 }
```

Having called `BitToBool` with correct inputs, we expect the assertions on lines 18 and 19 to hold. But the verifier cannot prove this, so it rejects the program.

The problem is that Dafny verifies programs in a *modular* fashion. When reasoning about the callers of a method, the verifier ignores the method's body, and instead uses the method's specification—its type signature, preconditions, and postconditions. In our example, the verifier knows only that `BitToBool` produces a Boolean value, which is clearly not enough to prove that the `Main` assertions hold.

Postconditions. To fix the problem, we write a *postcondition* for `BitToBool` that specifies *which* Boolean value is returned:

```
method BitToBool(i: int)
  returns (b: bool)
  requires i == 0 || i == 1
  ensures b == (i == 1)
{ ... }
```

Given this postcondition, verification succeeds for both `BitToBool` and `Main`, guaranteeing that the program as a whole satisfies its specification—modulo any bugs in the Dafny verifier and compilers.

2.2 Testing Dafny Verifier and Compilers

A verification ecosystem like Dafny, which includes a verifier and compilers, can suffer from three broad categories of bugs: compiler bugs, soundness bugs, and precision bugs. While there is a large body of previous work on compiler bugs (see, e.g., [42]), discovering soundness and precision bugs in verifiers is a less explored area. However, soundness and precision bugs are critical. Soundness bugs affect the correctness of the verifier, and they can invalidate the correctness guarantees provided by the Dafny ecosystem. Precision bugs affect the usability of the verifier and can be frustrating to developers. XDsmith aims to test in particular for both of these kinds of bugs, in addition to the more general compiler bugs.

Compiler bugs. Compiler bugs can be broadly categorized as crashes, semantic issues, or malformed target programs. A correct compiler preserves the semantics of the source program in the target code: both the source and target exhibit the same behaviors on all possible inputs. To demonstrate a semantic compiler bug, XDsmith must produce a correctly annotated source program that behaves differently from the compiled target program on some input. Malformed target programs, on the other hand, cannot even be parsed by an off-the-shelf compiler for a target language.

Soundness bugs. Our toy program, as well as real applications of Dafny, trust the Dafny verifier to be *sound*. A sound verifier admits an annotated program only if it can *prove* that the program meets its specification. To demonstrate a soundness bug in the Dafny verifier, XDsmith must produce an annotated program that violates its specification but is accepted by the verifier.

Precision bugs. Like all sound verifiers for non-trivial languages, the Dafny verifier is *incomplete* in that it may reject correctly annotated programs. These false alarms are usually called *precision* issues. Some precision issues are inevitable due to the incompleteness of the underlying theorem prover. But some are considered bugs—for example, rejecting the annotation `assert true`. To demonstrate a potential precision bug, XDsmith must produce a correctly annotated program that is rejected by the verifier.

2.3 XDsmith Components and Workflow

Figure 1 illustrates the workflow that XDsmith employs to search for bugs in Dafny. The framework consists of three main components: the test generator, the verification tester, and the differential tester for the compilers.

Given a source of randomness, the test generator produces a stream of random Dafny tests. Each test takes the form of an annotated program and the expected verification outcome for that program—whether it satisfies or violates its specification. The entry point to the test program is the `Main` method, which invokes other methods in the program with desired concrete values. Because the generator must produce both the annotated program and its expected verification outcome, XDsmith limits the test programs to a carefully chosen subset of the Dafny language. The key challenge is to keep this subset large enough to test the Dafny ecosystem in interesting ways, while keeping it limited enough to be able to decide the verification outcome *without* building an oracle verifier.

Figure 1 shows a test program generated by XDsmith. The program consists of a set of annotated methods, which are transitively called by the `Main` method. Each called method is annotated with *precise* preconditions and postconditions of the form $x == v$, where x is an input or output variable and v is a concrete value. These annotations are maximally precise in that they describe the behavior of a method on a specific input. XDsmith generates them by simply running the program from the `Main` method, and recording the input and output of each called method. Methods can also contain assertions, which are random propositions about the method's state. XDsmith uses program synthesis to generate these assertions so that they have a specific outcome (true or false) in the program's main execution. In our example, all of the generated assertions evaluate to true, and the test program should be accepted by the verifier.

The verification tester consumes test programs and submits them to the Dafny verifier. If the verifier accepts a program that violates its specification, the tester reports a soundness bug. If the verifier rejects a correct program, the tester reports a potential precision bug. Otherwise, the program is both correct and accepted by the verifier, and the verification tester passes it to the compiler tester.

The compiler tester is based on differential testing [34]. It compiles the test program with each of the available compilers, runs the resulting target code, and looks for discrepancies in their (printed) outputs. As we will see in Section 3.2, XDsmith uses one of these compilers as the ground truth for generating decidable program annotations, thereby comparing the verifier semantics to the concrete semantics of the ground truth compiler. Differential testing then tests the compilers against each other. The differential tester reports a bug if at least two of the target programs exhibit different behaviors, or if a target program crashes.

Algorithm 1 **GenerateSpecifications**(P, *valid*)

```
$P_T \leftarrow$ FlattenMethods($P$)
$\langle M_{exec}, Print_{exec}, \mu \rangle \leftarrow$ RunProgram($P_T$)
for all $m \in$ Methods($P_T$) do
    if $m \in M_{exec}$ then
        $P_T \leftarrow$ AddPrePost($P_T, m, \mu$)
    else
        $P_T \leftarrow$ AddPrePost($P_T, m, false$)
    end if
end for
$Asserts \leftarrow$ GenAssertions($\mu, Print_{exec}$, valid)
for all $p \in$ PrintStmt($P_T$) do
    if $p \in Print_{exec}$ then
        $P_T[p] \leftarrow$ 'p; assert $Asserts(p)$;'
    else
        $P_T[p] \leftarrow$ 'p; assert false;'
    end if
end for
return $P_T$
```

3 GENERATING RANDOM TEST PROGRAMS

The XDsmith test generator solves the core technical challenge at the heart of the Dafny testing problem: how do we generate an annotated Dafny program with a known verification outcome, without having to build a verification oracle? Our solution tackles this challenge in two stages: (1) program generation and (2) annotation generation. The program generator uses the Xsmith [21] framework to produce *correct* Dafny programs *without* annotations. The annotation generator then uses a combination of random testing and example-based synthesis to produce both *correct and incorrect annotations* for the resulting program (for precision, soundness, and compiler testing). This section describes the two stages in detail.

3.1 Generating Programs Without Annotations

The Dafny verifier checks every program against two kinds of correctness constraints: the *explicit specifications* provided as program annotations, and the *implicit specifications* imposed by the Dafny semantics. For example, the semantics requires every program to terminate and to be free of common runtime errors, such as division by zero, index out of bounds accesses, and null dereferences. The verifier rejects programs that cannot be proven to satisfy these constraints. So, to generate correct programs without annotations, our program generator must be able to produce code that is not only syntactically well-formed and well-typed, but that also satisfies the implicit correctness constraints imposed by the Dafny semantics.

To achieve this, we use Xsmith [21] to generate well-formed and well-typed programs in a restricted subset of the Dafny language that satisfies Dafny's implicit semantic constraints by construction. This subset, which we call XDafny, guarantees termination by omitting loops and recursion. We also exclude unsafe operations such as indexing into an arbitrary sequence with an arbitrary index. Instead, XDafny programs use safe wrapper functions to access potentially unsafe operations (Figure 2). Finally, we disallow method parameters with reference types (e.g., arrays), because Dafny prevents most uses of such parameters in the absence of **`reads`** and **`modifies`** annotations. What remains is a rich subset of Dafny that includes control flow, local side effects (e.g., printing), methods, functions, basic types (Booleans, integers, and strings), container types (sets, sequences, and maps), and array types. The program in Figure 1 belongs to this subset, and it was produced by our program generator.

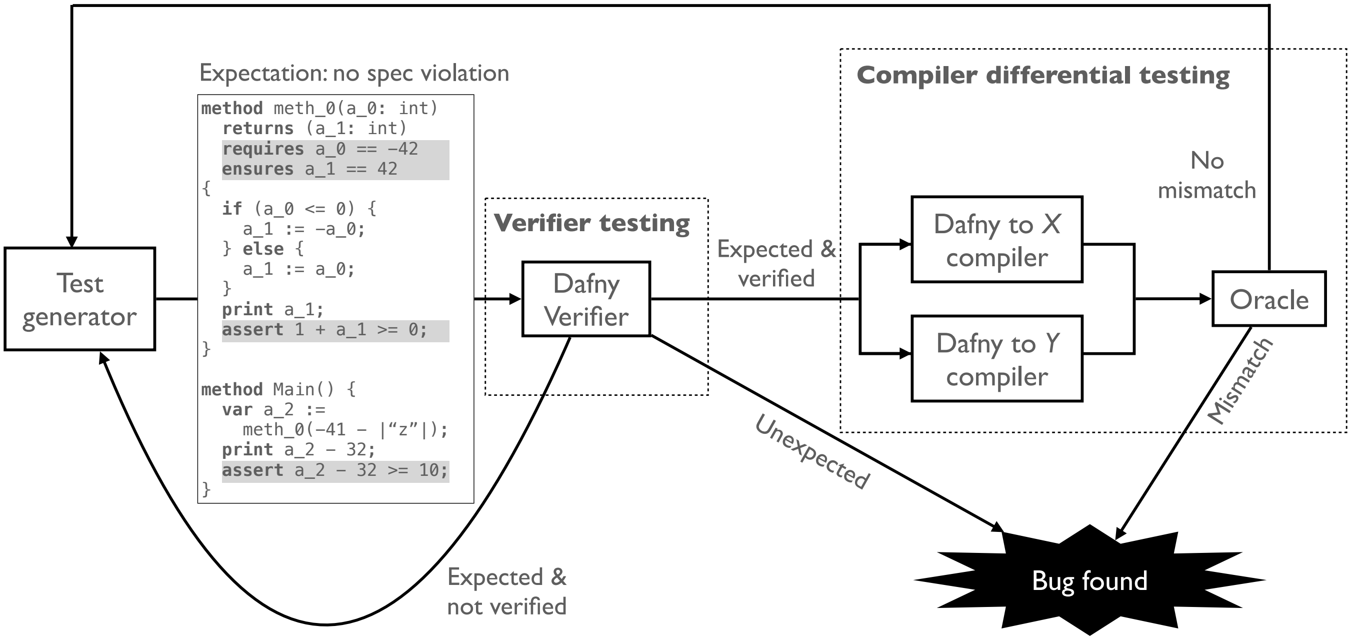
The generator works by instantiating the Xsmith [21] framework with an executable specification of the XDafny grammar and type constraints. Xsmith is a domain-specific language embedded in Racket [13, 15]. It takes as input a language grammar and attributes, such as types, and then randomly generates an AST conforming to the grammar and constraints. The framework operates by adding a *hole* into the grammar, initializing the generated AST to a hole node, and repeatedly mutating the tree by replacing any remaining hole with a node picked from the grammar (which could potentially introduce more holes), subject to attribute constraints. This process continues until there are no holes left. In our case, the final AST is a set of XDafny methods and functions, with a no-argument `Main` method serving as the entry point to the program. The generator extends every method in this AST with a **`print`** statement, which writes the method's output value to standard output. We use the printed output to generate annotations, and to compare the behavior of Dafny compilers during differential testing.

3.2 Generating Program Annotations

This section presents our approach for annotating XDafny programs with assertions, preconditions, and postconditions. Our approach exploits the fact that every XDafny program has exactly one execution. To see why, recall from Section 3.1 that each XDafny program has a `Main` method that takes no arguments and that serves as the program's entry point. Because XDafny is sequential and deterministic, the program has just one execution that starts at its sole entry point. So, generating annotations that are true for all executions of an XDafny program amounts to generating annotations that are true for the sole *main* execution of that program.

Algorithm 1 shows how we employ this observation to annotate XDafny programs with random assertions and with maximally precise preconditions and postconditions of the form $x == v$, where x stands for a variable and v for a value (Figure 3c). We focus on precise preconditions and postconditions because they can be generated without using an oracle verifier, and because they provide a simple way to enable modular verification of the program's main execution. But the assertions that we generate need not be precise, allowing for a wide range of propositions to be used for testing. Our algorithm therefore generates random (general) assertions, and precise preconditions and postconditions.

The algorithm takes two inputs: an XDafny program P and a Boolean flag *valid*. The flag indicates whether to generate a precision (true) or a soundness (false) test. For a precision test, the resulting annotated program should satisfy all of its annotations, and for a soundness test, it should violate at least one annotation. Given these



```
method meth_0(a_0: int)
  returns (a_1: int)
  requires a_0 == -42
  ensures a_1 == 42
{
  if (a_0 <= 0) {
    a_1 := -a_0;
  } else {
    a_1 := a_0;
  }
  print a_1;
  assert 1 + a_1 >= 0;
}

method Main() {
  var a_2 :=
    meth_0(-41 - |"z"|);
  print a_2 - 32;
  assert a_2 - 32 >= 10;
}
```

Figure 1: The XDsmith workflow diagram. The test generator produces annotated Dafny programs, along with the expected verification outcome. We highlight the annotations in gray, to distinguish them from the implementation code. The verifier tester consumes an annotated program and passes it to the compiler tester if the program is both correct and accepted by the Dafny verifier. The compiler tester performs differential testing by checking that the compiled programs print equivalent outputs.

```
// Wrapper for safely accessing sequence xs at index i.
function method ref<T>(xs: seq<T>, i: int, fallback: T): T
{
  if 0 <= i < |xs| then xs[i] else fallback
}

method caller(xs: seq<int>, i: int) {
  // This verifies:
  print ref(xs, i, 123);
  // But this does not:
  // print xs[i];
}
```

Figure 2: An example safe wrapper for a potentially unsafe operator. XDsmith generates such wrappers to ensure that the resulting annotation-free program satisfies the implicit correctness constraints imposed by the Dafny semantics.

inputs, the algorithm annotates P in four steps, guaranteeing that the resulting annotated program is correct if and only if *valid* is true.

Flattening. The first step (line 1) is to *flatten* P's call graph into a tree so that we can annotate its methods with preconditions and postconditions of the form $x == v$. Flattening is similar to inlining: it replaces each method call with a call to a fresh copy of that method. To see why this step is necessary, consider the method `test` in Figure 3a, which is called twice with different inputs. No single specification of the form $x == v$ works for both of these calls, so we use flattening to provide each invocation with its own precise specification. It is easy to see that the resulting flat program P_T has a single main execution, and that this execution produces the same printed output as the main execution of P.

Execution. After flattening, we run P_T (line 2) to collect a *trace* of its main execution. This step uses the **RunProgram** procedure, which compiles P_T to a target language (using any Dafny compiler), executes the target `Main` code, and maps the printed output back to the source program P_T. The result is a tuple with three components: the set M_{exec} of all executed methods in P_T; the set $Print_{exec}$ of all executed **`print`** statements; and the map μ that stores the values of the input and output variables for each executed method $m \in M_{exec}$.

Generating preconditions and postconditions. Given the trace of P_T, we use the procedure **AddPrePost** to annotate each method m in P_T with preconditions and postconditions. If m is executed in the main trace (line 5), **AddPrePost** annotates it with equality specifications that reflect the values of m's input and output variables in μ. In particular, each input variable $i \in inputs(m)$ generates the annotation **`requires`** i `==` $\mu(i, m)$, and each output variable $o \in outputs(m)$ generates the annotation **`ensures`** o `==` $\mu(o, m)$. If m is not executed (line 7), it is annotated with `false` preconditions and postconditions, denoting dead code. The resulting annotations are the most precise preconditions and postconditions for P_T, and P_T satisfies them by construction.

Generating assertions. In the last step, we use **GenAssertions** to generate random assertions for all print statements in P_T. If the statement p is executed, **GenAssertions** produces an assertion

```
 1 method test(x: int)
 2   returns (y: int)
 3 {
 4   y := x + 2;
 5   print y;
 6 }
 7
 8 method Main() {
 9   var y1 := test(1);
10   var y2 := test(2);
11   print y1 - y2;
12 }
```

(a) Original program

```
 1 method test_1(x: int)
 2   returns (y: int)
 3 {
 4   y := x + 2;
 5   print y;
 6 }
 7
 8
 9
10
11 method test_2(x: int)
12   returns (y: int)
13 {
14   y := x + 2;
15   print y;
16 }
17
18
19
20
21 method Main() {
22   var y1 := test_1(1);
23   var y2 := test_2(2);
24   print y1 - y2;
25
26 }
```

(b) Transformed program

```
 1 method test_1(x: int)
 2   returns (y: int)
 3   requires x == 1
 4   ensures y == 3
 5 {
 6   y := x + 2;
 7   print y;
 8   assert y + x < 5;
 9 }
10
11 method test_2(x: int)
12   returns (y: int)
13   requires x == 2
14   ensures y == 4
15 {
16   y := x + 2;
17   print y;
18   assert y * 2 > 7;
19 }
20
21 method Main() {
22   var y1 := test_1(1);
23   var y2 := test_2(2);
24   print y1 - y2;
25   assert -2 * (y1 - y2) > 0;
26 }
```

(c) Annotated program

Figure 3: Annotating an XDafny program using GenerateSpecifications(P, *true*). The original program (a) is flattened so that each method is called exactly once (b). The flattened program is annotated (c) with random assertions (one for each print statement), as well as preconditions and postconditions of the form $x == v$, where x is an input or output variable and v is a concrete value.

$$\begin{array}{lll} \text{Expression} & e ::= & z \mid i \mid \mid e + e \mid e \times e \mid \\ & & e < e \mid e = e \mid \\ & & e \text{ and } e \mid e \text{ or } e \mid \text{not } e \\ \text{Integer constant} & z ::= & \ldots, -1, 0, 1, \ldots \\ \text{Integer variable} & i & \end{array}$$

Figure 4: Dafny grammar used for synthesizing assertions.

$Asserts(p)$ for it, and line 13 adds this assertion immediately after p in P_T. Otherwise, line 15 adds `assert false` after p, denoting dead code. **GenAssertions** guarantees that the main execution of P_T satisfies every assertion $Asserts(p)$ if and only if *valid* is true. Figure 3c illustrates a set of *valid* assertions for the program in Figure 3b.

GenAssertions relies on a simple form of syntax-guided synthesis [2, 19] to generate assertions for the statements in $Print_{exec}$. To produce a true (resp. false) assertion for a statement p, we synthesize a proposition in a particular grammar that evaluates to *true* (resp. *false*) in the main execution of P_T. Our implementation uses the grammar in Figure 4, which allows integer and Boolean operations, but in principle, any subset of the XDafny grammar can be used for this purpose. The leaves of the grammar include a single variable i, which represents the value printed by the statement p. We feed this grammar to the Rosette synthesis engine [39], along with the constraint that the synthesized expression must evaluate to *true* (resp. *false*) when i evaluates to the value printed by p. This constraint always has some solution in our grammar. When the synthesizer returns a solution e, we turn it into an assertion by replacing each occurrence of i in e with the expression printed by p.

To illustrate, consider the `print` statement on line 24 of Figure 3b. This statement p prints the value of the expression `y1 - y2`, which is -1 in the main execution of P_T. To generate a true assertion for p, **GenAssertions** invokes the synthesizer with the grammar in Figure 4, constraining the synthesized expression to evaluate to *true* when i evaluates to -1. Suppose that the synthesizer returns the expression $-2 * i > 0$. **GenAssertions** turns this expression into the assertion on line 25 of Figure 3c by replacing i with the expression `y1 - y2` printed by p. By construction, the resulting assertion is always true in (the main and only execution of) P_T, as desired.

4 EVALUATION

4.1 Opportunistic Bug Finding

While developing XDsmith, we performed an opportunistic bug finding case study over a three-months period that was uncontrolled and unstructured, and similar to the one from the Csmith paper that was performed over three years [42]. We ran XDsmith on Dafny continuously on AWS Batch [1] with 100 concurrent threads: 50 threads were testing the Dafny verifier and 50 threads were testing the Dafny compilers. In such a setup, XDsmith was generating approximately 100,000 random annotated Dafny programs per day.

During the case study, we reported any Dafny crash as a bug. If XDsmith annotated a program with a correct specification and the verifier rejected it, we reported a potential bug. Similarly, if XDsmith annotated a program with an incorrect specification and the verifier accepted it, we reported a bug. We employed differential testing on the Dafny compilers. Because compilation strips away annotations, we streamline the differential testing of compilers by directly compiling Dafny programs *without* annotations to multiple

Table 1: Summary of bugs found.

	Verifier		Compiler	Other	Total
	Snd.	Prec.			
Verifier Testing	0	8	1	1	10
Compiler Testing	2	0	18	0	20
Other	1	0	0	0	1
Total	3	8	19	1	31
Confirmed	3	8	19	1	31
Fixed	0	0	7	1	8

target languages supported by Dafny. We reported any miscompilation, when a Dafny compiler generates a wrong target program, as a bug. Recall that as part of our program generation, we inject print statements that write out program state to the standard output. We ran each compiled target program, and compared the output with the others; we reported any discrepancy in the results as a bug.

Whenever XDsmith uncovered a potential Dafny bug, we paused the fuzzing process to investigate the root cause of the bug with input from Dafny developers. We minimized the randomly generated program to make root cause analysis easier. After reporting the bug, we modified XDsmith to block it from generating programs with features that led to the bug, so as to avoid discovering the same bug over and over again. We then resumed the fuzzing process. Following the agile development process, we continuously improved XDsmith with new features during the three months period.

We discovered 31 bugs in Dafny using XDsmith, and Table 1 provides a summary. First, we show whether we discovered a bug during verifier testing, compiler testing, or using other means. Here, we note that our verifier testing, where the synthesis of annotations is our key contribution, accounts for almost 1/3 of all the bugs we found. We manually investigated each reported bug, its root cause, and the associated bug fix, whenever it was available. Based on this manual investigation, we further categorize the bugs into three categories based on where their root cause is located: verifier, compilers, and other bugs. Based on the effect of verifier bugs, we partition them into soundness (column Snd.) and precision (column Prec.) bugs. Finally, Dafny developers confirmed all of the reported bugs to be real and valid. Moreover, 8 of these confirmed bugs have been fixed in the Dafny mainline. We describe example bugs in more detail next.

4.1.1 *Verifier Bugs.*

We found 11 bugs (3 soundness and 8 precision) in the Dafny verifier. Among those, verifier testing found 8 bugs, all of which are related to precision. Compiler testing found 2 bugs related to soundness. Finally, we manually found one verifier bug while formalizing the Dafny language in XDsmith.

Sequence prefix precision bug (found with verifier testing). Figure 5a shows a minimized precision-related Dafny bug. Evidently, the list containing 1 is not a prefix of a list containing 2. Therefore, `[1] <= [2]` should be false, where <= is the sequence prefix testing operator in Dafny. This means `!([1] <= [2])` should be true, which is indeed what the compiled program outputs (if we ignore the verification error). However, the verifier is unable to prove this simple fact.

```
method Main() {
  print !([1] <= [2]);
  assert !([1] <= [2]);
}
```

(a) Sequence prefix precision bug

```
method Main() {
  var a := new array<int>;
}
```

(b) Soundness bug related to unspecified array length

```
method Main () {
  var a := multiset{12}[12 := 0];
  var b := multiset{42};
  print 12 in a, " ", a == b, "\n";
}
```

(c) Multiset semantic bug

```
method Main() {
    print {1} >= {1, 2};
    assert {1} >= {1, 2};
}
```

(d) Superset operator bug

Figure 5: Example Dafny bugs found by XDsmith.

Soundness bug related to unspecified array length (found with compiler testing). While this is a verifier soundness bug, we discovered it during differential compiler testing. Figure 5b shows a faulty program that creates an array of integers whose length is unspecified. This is illegal in Dafny, yet the verifier does not report an error. When the program is compiled with various Dafny compilers, the output programs are malformed. For example, when compiled with the C# compiler, the output program throws the error *"Array creation must have array size or array initializer"*.

Map data structure soundness bug (found during Dafny formalization). We discovered this bug while formalizing the Dafny language in XDsmith. In Dafny, the map data structure requires the keys to be equality-supporting. For instance, it is not possible to create a map keyed by function values, since function values cannot be tested for equality. However, Dafny imposes no such restriction on the values. Yet, the operation `.Value` on a map returns a set of values, where the set data structure requires its elements to be equality-supporting. Thus, for example, operation `.Value` on a map from integers to function values creates a set of function values, thereby violating its equality-supporting constraint. The Dafny verifier fails to report an error in this case.

4.1.2 *Compiler Bugs.*

We found 19 bugs in the Dafny compilers. Among those, compiler testing found 10 semantic bugs and 8 bugs that caused compiled programs to be malformed. Note that semantics bugs required differential testing to be found, while malformed program bugs can be discovered by simply running the compiled programs. Finally, verifier testing found one compiler bug.

Multiset semantic bug (found with compiler testing, fixed). Figure 5c shows a program that creates multiset data structures and performs operations on them. The multiset data structure supports multiplicity setting. For instance, `multiset{12}[12 := 3]` would be equivalent to `multiset{12, 12, 12}`. Evidently, the program in Figure 5c is supposed to output `false false`. However, of all the Dafny target languages, only Java target program outputs the correct result. C#, Go, and JavaScript output `true false`, `false true`, and `true true`, respectively. Upon inspection, we discovered that most runtime of target languages represent the multiset data structure with a dictionary from elements to numbers of occurrences. However, they incorrectly assume an invariant that the number of occurrences must be positive, which in fact does not hold. We submitted a patch that fixed this bug by correctly maintaining the multiset invariant in every operation.

Superset operator bug (found with verifier testing, fixed). Figure 5d shows a simplified compiler bug, where the superset operator is compiled to the subset operator. Therefore, the print statement outputs `true`, even though it should output `false`. Notably, every compiler is incorrect in the same way, so it is impossible for differential testing to discover the bug — this is a well known limitation of differential testing. Verification testing, on the other hand, is able to detect the discrepancy: the print statement incorrectly outputs `true`, our annotation generator then asserts the expression to be true, but the verifier correctly reports an assertion violation. We submitted a patch that fixed this bug by compiling the operator correctly.

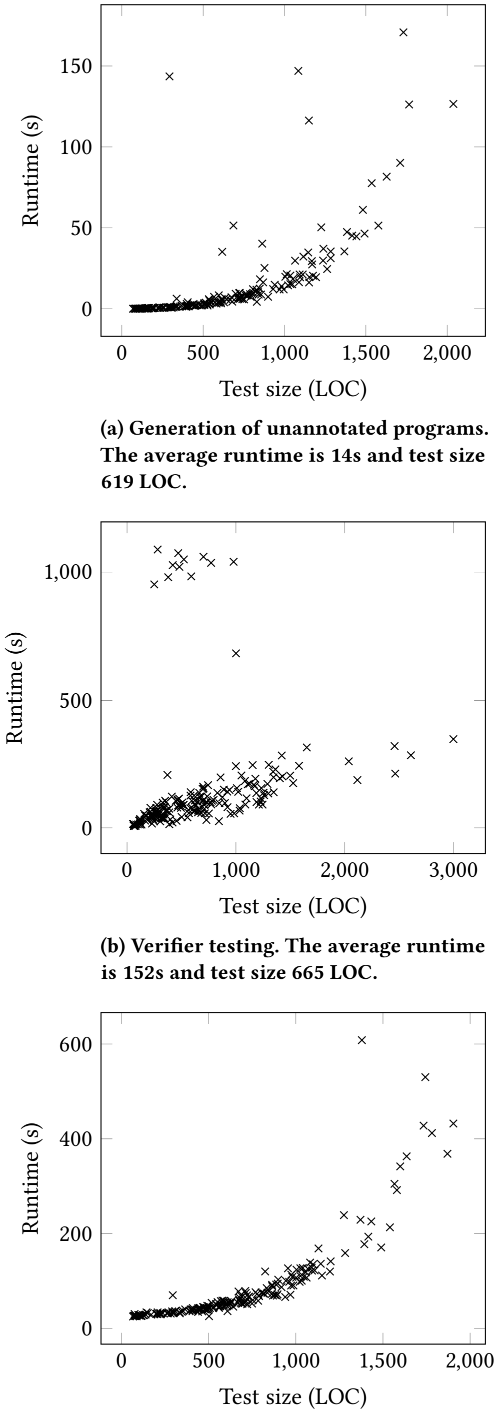
Incorrect subset implementation for Go (found with compiler testing, fixed). The Dafny's runtime for Go came with a subset for multiset implementation such that for every element the number of occurrences in one multiset must be strictly less than the number of occurrences in the other multiset. The definition was incorrect as it considered $\{1\}$ to *not* be a subset of $\{1, 2\}$. We submitted a patch that fixed this bug by implementing the correct semantics of the subset operation in the Dafny Go runtime.

Incorrect compilation of strings (found with compiler testing, fixed). Dafny compilers represent strings differently in their target languages. The low-level differences between internal representations of strings across multiple target languages can lead to corner cases in which strings are printed out differently. XDsmith detected such an issue, and we submitted a patch that fixed it by compiling Dafny strings to the same representation in all compilers.

4.1.3 Other bugs.

We found one bug in the Dafny pretty printer.

Incorrect Dafny pretty printing (found with verifier testing, fixed). Dafny's function type allows users to drop parentheses when there is exactly one argument, *except* when the argument is of a tuple type to prevent ambiguous parsing. XDsmith, extended with a pretty printer testing functionality, uncovered a bug in the Dafny pretty printer where it would unconditionally drop parentheses, thereby generating malformed programs. Dafny developers fixed the bug after we reported it by introducing a check in the pretty printer for tuple arguments.



(a) Generation of unannotated programs. The average runtime is 14s and test size 619 LOC.

(b) Verifier testing. The average runtime is 152s and test size 665 LOC.

(c) Differential compiler testing. The average runtime is 83s and test size 674 LOC.

Figure 6: XDsmith runtimes.

4.2 Controlled Quantitative Evaluation

To more precisely measure the performance and effectiveness of XDsmith, we performed several controlled quantitative evaluation experiments. In the experiments, we measure the runtime of XDsmith and the code coverage achieved on Dafny. We set the Xsmith AST depth, which controls the size of the generated tests, to its default value. We ran all the experiments on a Macbook Pro (M1, 13-inch) with 16 GB of memory in a controlled environment.

Figure 6 presents the runtimes of XDsmith plotted against the size of the generated test cases. In each experiment, we generated 200 test cases with three different flows through XDsmith: generation of unannotated programs, verifier testing, and differential compiler testing. All the reported runtimes are end-to-end, from starting XDsmith to retrieving the output after it terminates.

Figure 6a shows the barebone runtimes of XDsmith for generating test programs without annotations. XDsmith generates majority of tests quickly, with the average runtime of 14 seconds per test, and with the average test size of 619 lines.

Figure 6b shows the runtimes of the verifier testing flow, which include generating random programs with annotations and verifying them using Dafny. As expected, the runtimes increased when compared to the barebones test generation since most of them are dominated by the invocations of the annotation synthesis step. There are a number of outliers that have high runtimes (around 1000 seconds) for test programs of modest size (under 1000 LOC). These outliers contain many print statements, leading to large annotation synthesis workloads. Other outliers some have complex call graphs, leading to a large increase in size after flattening.

Figure 6c shows the runtimes of the differential compiler testing flow, which includes random generation of programs without annotations and the differential compiler testing step. We perform the compilation and concrete execution steps with each Dafny compiler one at a time. Moreover, while the random programs we use for compiler testing do not contain annotations, the Dafny verifier is still run to check the implicit specifications (e.g., no overflows, no array out of bounds accesses) before the programs can be compiled. When compared to the previous two plots, the runtimes fall in-between—they are higher than the barebones test generation since we invoke the compilers, but they are also lower than the verifier testing since we omit the more expensive annotation synthesis step.

Figure 7 presents the code coverage (i.e. the coverage of the Dafny verifier code in C#) that XDsmith achieves on Dafny over 100 runs, and compares the resulting coverage against the coverage achieved by verifying a baseline program. The baseline program is a solution to a challenge from the verification competition VSComp-2010 [27], taken from the Dafny documentation website [31]. We can see that the coverage of the first test program already exceeds the coverage of the baseline program, indicating that our tests cover a wide range of commonly used Dafny features. As is common with fuzzing, we quickly reach saturation, where majority of the test cases that follow contribute little to improving coverage. However, we can still occasionally notice sizable jumps in coverage (around test 20 and test 80). These are the tests that exercise new parts of code, and that are more likely to lead to new bugs being found. Hence, as supported by our opportunistic bug finding case study, it is worthwhile to run XDsmith over multiple weeks, or even months, to improve our chances of uncovering such test cases. In fact, continuously performing fuzzing for long periods of time is a well-established practice [42].

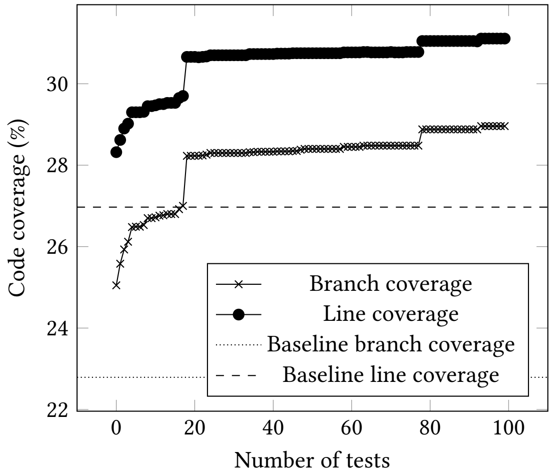


Figure 7: Code coverage from XDsmith verifier testing. The baseline branch coverage is at 23%, and the baseline line coverage is at 27%.

5 LESSONS LEARNED

In this section, we discuss our experience and lessons learned while developing and applying XDsmith.

Software infrastructure. We chose to build XDsmith in the Racket language [15] for two reasons. First, Racket is designed for building new programming languages and tools, and as such, it offers powerful facilities for syntactic abstraction and for writing code that manipulates ASTs. Second, using Racket let us easily adopt two key pieces of infrastructure that we needed for XDsmith: a fuzzing framework, Xsmith [21], and a program synthesis framework, Rosette [39].

Thanks to Xsmith, we implemented a working prototype of the program generator in just a few weeks. The initial fuzzer generated random programs in a small but useful subset of the XDafny grammar. This rapid start enabled us to begin testing Dafny immediately, find gaps in our grammar coverage, and determine what additional features to support next. Xsmith made it easy to extend our grammar incrementally, and to go from a small core grammar to the full XDafny subset (Section 3.1) in three person-months.

On the flip side, we also had to work around two core limitations of Xsmith. The first limitation was in Xsmith’s type system, which currently provides no support for expressing classes, traits, and inheritance. Xsmith authors acknowledged that improving Xsmith to support classes is future work [38]. We side-stepped this problem by excluding the object-oriented features of Dafny from the XDafny grammar. The second limitation was in Xsmith’s *canned components*, which are grammar abstractions for expressing common language constructs, such as let expressions and conditionals. Xsmith encourages developers to build new fuzzing grammars by combining canned components. In our case, the default definitions of these components were insufficient to express the XDafny grammar, and there was no easy way to customize them. We ended up copying their implementation and modifying it to suit our needs.

We use Rosette to generate assertions for annotating the random Dafny programs generated by the fuzzer. Like Xsmith, Rosette helped us to quickly implement the assertion generator. Also, like Xsmith, it presented limitations that we had to work around. For example, Rosette's synthesizer is deterministic: given a synthesis problem, it always produces the same expression (if one exists) as the answer. However, XDsmith needs to synthesize a variety of *random* assertions. We solved this problem by randomly concretizing [25] the grammar passed to Rosette's synthesizer.

Computing infrastructure. We had access to AWS resources for our experiments, allowing us to launch multiple threads of XDsmith with ease. Using this infrastructure, we ran our testing framework continuously on 100 machines for 3 months. As a consequence, we were able to obtain results much faster than we could have with the traditional desktop or local cluster setups.

Domain experts. Like prior testing efforts [11, 42], we benefited from having good rapport with the developers of the tools we were using and testing. In our case, we were able to connect with the Xsmith developers and quickly understand Xsmith limitations. Similarly, we regularly connected with the Dafny experts and developers. They helped us understand the semantics of Dafny, and they quickly acknowledged and triaged our bug reports. These interactions were essential in helping us refine our approach and converge on the design presented in this paper.

Fuzzing finds new bugs. Dafny is a mature software infrastructure that follows good software development practices, including continuous integration, regression and unit testing, and code reviews. For example, Dafny repository [12] contains about 1233 integration tests and 258 unit tests as of May, 2022. These tests are regularly run on every code change as a part of the continuous integration process. Using XDsmith we were still able to uncover a number of new bugs in Dafny, which shows that fuzzing is complimentary to the existing Dafny tests.

Importance of annotated programs. As Table 1 shows, using verifier testing with annotated programs we found almost 1/3 of all the bugs we reported. This supports our intuition we had early on that performing just the more traditional (differential) compiler fuzzing is not sufficient in the domain of testing of verification ecosystems such as Dafny. While our novel approach of automatically generating annotated programs using automatic synthesis is already effective in finding bugs, more research should be done in this space to further improve the effectiveness of the annotated program generation; we leave this for future work.

6 RELATED WORK

There are numerous approaches and tools for testing of compilers, and our work falls into this broader area of compiler testing [10, 29]. More specifically, Cadar and Donaldson [8] noted the importance of improving our confidence in program analyzers using testing several years ago. We describe recent efforts to address this problem by developing approaches for testing of verifiers and static analyzers.

Cuoq et al. [11] leverage an interpreter mode of the static analyzer Frama-C to perform differential testing between the analyzer's interpreter and a concrete execution on randomly generated C programs. Additionally, they modify the static analyzer to output the internal inferred facts, and then run the test program to see if the inferred facts agree with the concrete execution. They use an existing tool called Csmith to generate random C programs, while we developed a new generator of random Dafny programs by leveraging Xsmith. Moreover, unlike them, we avoided having to modify Dafny by automatically synthesizing interesting assertions during the random program generation process. Finally, their focus is on testing of the Frama-C interpreter and static analysis modules, while we are focused on testing of a Floyd-Hoare-style verifier. Recently, Casso et al. [9] proposed a testing method for static analyzers that is similar to the one proposed by Cuoq et al.

Klinger et al. [28] perform differential testing of static analyzers for C programs to uncover soundness and precision issues. While we generate both random Dafny programs and assertions from scratch, their technique takes a set of seed C programs as input, and it randomly injects assertions into them. Moreover, they establish the likely ground truth for each injected assertion by performing differential testing across several analyzers. Since Dafny has only one available verifier, we leverage concrete executions to synthesize assertions with known truth values ahead of time.

Kapus and Cadar [26] focus on automatic testing of symbolic execution engines on C programs that they randomly generate using Csmith. The key novelty of this work is the generation of program versions and the accompanying oracles that are particularly suitable for symbolic execution engines (e.g., based on path coverage). These are not easily transferable to our Hoare-style verification setting since, for example, path coverage is not readily available. Hence, we proposed a more suitable oracle for our setting based on automatic synthesis of assertion with known truth values.

Taneja et al. [37] use an SMT solver to infer sound and precise dataflow facts about program fragments. Then, they use the generated facts to test the LLVM's static analyzer for soundness and precision bugs. Like us, they generate facts that are sound and precise by construction, thus avoiding the oracle problem. Unlike us, they focus on inferring facts suitable for testing of dataflow analyses, while we synthesize assertions targeting Hoare-style verifiers.

Recently, Groce et al. [18] devised a method for testing of static analyzers that combines differential and mutation testing. Their testing method is specifically geared towards abstract-interpretation-based analysis engines since it focuses on effectively dealing with potentially high rates of false alarms, which is a common problem in such tools. They also leverage differential testing across several static analyzers. Similarly, Ami et al. [3] employ mutation testing to uncover soundness issues in static analyzers for Android applications. While we show that our approach based on generating random Dafny programs from scratch is successful in finding bugs in the Dafny verifier and compilers, as future work, it would be interesting to explore mutation testing for Dafny as well.

Apart from static analyzers and verifiers, fuzzing techniques have also been successfully applied to find bugs in other parts of a typical software verification toolchain, namely SMT solvers [6, 7, 33, 35, 36, 40, 41]. These techniques are complementary to our approach since they work on the backend SMT solvers instead of on the frontend verifier like us. Finally, there are studies [17, 42] complementary to ours that focus on assessing the correctness of formally verified artifacts instead of the actual verifiers. In particular, as the byproduct of testing of software systems developed using Dafny, Fonseca et al. [17] uncovered several soundness bugs in the Dafny verifier as well.

7 CONCLUSIONS AND FUTURE WORK

Program verifiers such as Dafny are powerful tools that allow us to prove the correctness of systems. To build confidence in these proofs, it is important to test all parts of the Dafny ecosystem, from the verifier to the compilers. The core challenge we solve in this work is how to automatically generate annotated programs with a known verification outcome. We present our experience in building XDsmith, an automated testing framework for the Dafny verifier and compilers that uses a combination of random program generation and example-based generation of assertions. The source code of XDsmith is publicly available at https://github.com/dafny-lang/xdsmith.

Our guiding principle was to use simple and obvious techniques that work in detecting bugs. We leverage the Xsmith framework to randomly generate well-formed and well-typed programs in a subset of Dafny. We configure the program generation to add print statements that write program state to standard out for differential testing and annotation generation. We generate Dafny programs that are free of loops and heap mutation; this ensures termination and lets us avoid having to generate loop invariants and frame conditions. We then concretely execute the Dafny program to record the values of the input and output variables; using these, we annotate the Dafny program with precise preconditions and postconditions. Based on the precise preconditions and postconditions, we transform the print statements to assertions in the annotated Dafny program, where the assertions are weaker than the concrete value observed on the standard out. Over a three-month period, where we ran 100 concurrent threads generating annotated Dafny programs and testing the verifier and the underlying compilers, we detected 31 bugs that were all confirmed by the Dafny developers. Eight of those have already been fixed and pushed to the Dafny mainline. This demonstrates that our simple approach is effective in testing the Dafny ecosystem.

As part of future work, we will integrate it in the Dafny ecosystem. We are currently working with the Dafny developers on this effort. The choices we made for generating specifications and assertions were based on Occam's razor and were the simplest approach given the tools and techniques available. In future work, we will compare different assertion generation techniques, and support more of the Dafny constructs when generating the random Dafny programs.

REFERENCES

[1] [n. d.]. AWS Batch: Fully managed processing at any scale. https://aws.amazon.com/batch/. Accessed: 2022-01-07.

[2] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. *Syntax-guided synthesis.* IEEE. 1–8 pages. https://ieeexplore.ieee.org/document/6679385/

[3] Amit Seal Ami, Kaushal Kafle, Kevin Moran, Adwait Nadkarni, and Denys Poshyvanyk. 2021. Systematic Mutation-Based Evaluation of the Soundness of Security-Focused Android Static Analysis Techniques. *ACM Trans. Priv. Secur.* 24, 3 (2021), 15:1–15:37. https://doi.org/10.1145/3439802

[4] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. 2009. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). Frontiers in Artificial Intelligence and Applications, Vol. 185. IOS Press, 825–885. https://doi.org/10.3233/978-1-58603-929-5-825

[5] Clark W. Barrett and Cesare Tinelli. 2018. Satisfiability Modulo Theories. In *Handbook of Model Checking*, Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). Springer, 305–343. https://doi.org/10.1007/978-3-319-10575-8_11

[6] Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. 2018. StringFuzz: A Fuzzer for String Solvers. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10982)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 45–51. https://doi.org/10.1007/978-3-319-96142-2_6

[7] Robert Brummayer and Armin Biere. 2009. Fuzzing and delta-debugging SMT solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*. 1–5.

[8] Cristian Cadar and Alastair F. Donaldson. 2016. Analysing the program analyser. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 765–768. https://doi.org/10.1145/2889160.2889206

[9] Ignacio Casso, José F. Morales, Pedro López-García, and Manuel V. Hermenegildo. 2020. Testing Your (Static Analysis) Truths. In *Logic-Based Program Synthesis and Transformation - 30th International Symposium, LOPSTR 2020, Bologna, Italy, September 7-9, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12561)*, Maribel Fernández (Ed.). Springer, 271–292. https://doi.org/10.1007/978-3-030-68446-4_14

[10] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1 (2020), 4:1–4:36. https://doi.org/10.1145/3363562

[11] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. 2012. Testing Static Analyzers with Randomly Generated Programs. In *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7226)*, Alwyn Goodloe and Suzette Person (Eds.). Springer, 120–125. https://doi.org/10.1007/978-3-642-28891-3_12

[12] dafny [n. d.]. Dafny. https://github.com/dafny-lang/dafny

[13] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2018. A Programmable Programming Language. *Commun. ACM* 61, 3 (March 2018), 62–71.

[14] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 287–305. https://doi.org/10.1145/3132747.3132782

[15] Matthew Flatt and PLT. 2010. *Reference: Racket*. Technical Report PLT-TR-2010-1. PLT Design Inc.

[16] Robert W. Floyd. 1993. *Assigning Meanings to Programs*. Springer Netherlands, Dordrecht, 65–81. https://doi.org/10.1007/978-94-011-1793-7_4

[17] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. 2017. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic (Eds.). ACM, 328–343. https://doi.org/10.1145/3064176.3064183

[18] Alex Groce, Iftekhar Ahmed, Josselin Feist, Gustavo Grieco, Jiri Gesi, Mehran Meidani, and Qihong Chen. 2021. Evaluating and Improving Static Analysis Tools Via Differential Mutation Analysis. In *21st IEEE International Conference on Software Quality, Reliability and Security, QRS 2021, Hainan, China, December 6-10, 2021*. IEEE, 207–218. https://doi.org/10.1109/QRS54544.2021.00032

[19] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Found. Trends Program. Lang.* 4, 1-2 (2017), 1–119. https://doi.org/10.1561/2500000010

[20] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. 2020. Storage Systems are Distributed Systems (So Verify Them That Way!). In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 99–115. https://www.usenix.org/conference/osdi20/presentation/hance

[21] William Gallard Hatch, Pierce Darragh, Guy Watson, and Eric Eide. 2020. Xsmith software repository. https://gitlab.flux.utah.edu/xsmith/xsmith

[22] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, Ethan L. Miller and Steven Hand (Eds.). ACM, 1–17. https://doi.org/10.1145/2815400.2815428

[23] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

[24] Chiao Hsieh and Sayan Mitra. 2019. Dione: A Protocol Verification System Built with Dafny for I/O Automata. In *Integrated Formal Methods - 15th International Conference, IFM 2019, Bergen, Norway, December 2-6, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11918)*, Wolfgang Ahrendt and Silvia Lizeth Tapia Tarifa (Eds.). Springer, 227–245. https://doi.org/10.1007/978-3-030-34968-4_13

[25] Jinseong Jeon, Xiaokang Qiu, Armando Solar-Lezama, and Jeffrey S. Foster. 2015. Adaptive Concretization for Parallel Program Synthesis. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9207)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 377–394. https://doi.org/10.1007/978-3-319-21668-3_22

[26] Timotej Kapus and Cristian Cadar. 2017. Automatic testing of symbolic execution engines via program generation and differential testing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 590–600. https://doi.org/10.1109/ASE.2017.8115669

[27] Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T. Leavens, Valentin Wüstholz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark A. Hillebrand, Bart Jacobs, K. Rustan M. Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan Tobies, Thomas Tuerk, Mattias Ulbrich, and Benjamin Weiß. 2011. The 1st Verified Software Competition: Experience Report. In *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6664)*, Michael J. Butler and Wolfram Schulte (Eds.). Springer, 154–168. https://doi.org/10.1007/978-3-642-21437-0_14

[28] Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2019. Differentially testing soundness and precision of program analyzers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 239–250. https://doi.org/10.1145/3293882.3330553

[29] Alexander S. Kossatchev and Mikhail Posypkin. 2005. Survey of compiler testing methods. *Program. Comput. Softw.* 31, 1 (2005), 10–19. https://doi.org/10.1007/s11086-005-0008-6

[30] K. Rustan M. Leino. 2010. 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 (Lecture Notes in Computer Science, Vol. 6355)*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

[31] K. Rustan M. Leino, Richard L. Ford, and David R. Cok. [n. d.]. Dafny Reference Manual. https://dafny-lang.github.io/dafny/DafnyRef/DafnyRef

[32] Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *J. Autom. Reason.* 43, 4 (2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

[33] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 701–712. https://doi.org/10.1145/3368089.3409763

[34] William M. McKeeman. 1998. Differential Testing for Software. *Digit. Tech. J.* 10, 1 (1998), 100–107. http://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf

[35] Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2021. Generative type-aware mutation for testing SMT solvers. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–19. https://doi.org/10.1145/3485529

[36] Joseph Scott, Federico Mora, and Vijay Ganesh. 2020. BanditFuzz: A Reinforcement-Learning Based Performance Fuzzer for SMT Solvers. In *Software Verification - 12th International Conference, VSTTE 2020, and 13th International Workshop, NSV 2020, Los Angeles, CA, USA, July 20-21, 2020, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 12549)*, Maria Christakis, Nadia Polikarpova, Parasara Sridhar Duggirala, and Peter Schrammel (Eds.). Springer, 68–86. https://doi.org/10.1007/978-3-030-63618-0_5

[37] Jubi Taneja, Zhengyang Liu, and John Regehr. 2020. Testing static analyses for precision and soundness. In *CGO '20: 18th ACM/IEEE International Symposium on Code Generation and Optimization, San Diego, CA, USA, February, 2020*. ACM, 81–93. https://doi.org/10.1145/3368826.3377927

[38] The Xsmith team. [n. d.]. Personal communication.

[39] Emina Torlak and Rastislav Bodík. 2014. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 530–541. https://doi.org/10.1145/2594291.2594340

[40] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. On the unusual effectiveness of type-aware operator mutations for testing SMT solvers. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 193:1–193:25. https://doi.org/10.1145/3428261

[41] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT solvers via semantic fusion. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 718–730. https://doi.org/10.1145/3385412.3385985

[42] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. 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*, Mary W. Hall and David A. Padua (Eds.). ACM, 283–294. https://doi.org/10.1145/1993498.1993532