Fuzzing as Editor Feedback

Marcel Garus

Fuzzing as Editor Feedback

Fuzzing als Editor-Feedback

by

Marcel Garus

A thesis submitted to the
Hasso Plattner Institute
at the University of Potsdam, Germany
in partial fulfillment of the requirements for the degree of

Master of Science in IT-Systems Engineering

Supervisors

Prof. Dr. Robert Hirschfeld
Jens Lincke

Software Architecture Group
Hasso Plattner Institute
University of Potsdam, Germany

January 18, 2025

Fuzzing – testing code with random inputs to see if it crashes – is a well-researched technique. In this work, we explore how fuzzing enables new editor tooling. We integrate fuzzing into the compiler pipeline and language tooling of Martinaise, a custom programming language. Our prototype can fuzz individual functions and show results in Visual Studio Code. Unlike other methods for testing code, this requires no extra effort from the developer.

Our tool shows crashing inputs next to the function signature. This shortens the feedback loop compared to traditional fuzzing, as edge cases are shown while the code is written, thereby preventing bugs.

Our tool also displays example inputs, chosen based on code coverage. Examples that execute different parts of a function can help understand the function's behavior without reading the implementation. Examples can be filtered to ones reaching the cursor position.

We evaluate the quality and performance of our prototype and for which kind of code fuzzing can give useful examples. We discuss how our approach can be applied to other programming languages. Fuzzing has the potential to serve as the basis for Babylonian Programming, visualizations, debugging sessions, and other tools that benefit from concrete examples.

Fuzzing – Code mit zufälligen Eingaben testen, um Crashes zu finden – ist ein etabliertes Forschungsfeld. In dieser Arbeit untersuchen wir, ob Fuzzing als Grundlage für Editor-Tooling dienen kann. Wir bauen Fuzzing in den Compiler und das Tooling von Martinaise ein, einer eigenen Programmiersprache. Unser Prototyp kann einzelne Funktionen fuzzen und Ergebnisse in Visual Studio Code anzeigen. Im Gegensatz zu anderen Test-Verfahren müssen Entwickler keinen extra Code schreiben, um unsere Editor-Erweiterung zu nutzen.

Unsere Erweiterung zeigt Fehler verursachende Eingaben neben Funktionssignaturen. Dieses direkte Feedback zu Randfällen während des Code-Schreibens kann Programmierfehler verhindern.

Unsere Erweiterung zeigt außerdem Beispieleingaben für Funktionen, die sie aufgrund von Code-Coverage auswählt. Beispiele, die das komplette Funktionsverhalten abdecken, können dabei helfen, eine Funktion zu verstehen, ohne die Implementierung zu lesen. Beispiele können gefiltert werden.

Wir evaluieren die Qualität und Geschwindigkeit unseres Prototypen. Wir diskutieren, für welche Arten von Coding-Stilen Fuzzing sinnvoll nutzbar ist und wie unser Ansatz auf andere Programmiersprachen übertragen werden kann. Fuzzing hat das Potenzial, die Basis für Babylonian Programming, Visualisierungen, Debugging-Sessions und andere Werkzeuge zu bieten, die von konkreten Beispielen profitieren.

I thank Jens for giving me guidance and advice while working on this thesis. He recognized and supported my idea of fuzzing, asked valuable critical questions, and gave me the space to explore this topic in depth.

I thank Clemens for proofreading this thesis and guiding me in this strange academic world. His no-nonsense approach improved my academic thinking quite a bit.

I thank Jonas for exploring the world of fuzzing with me. Even through numerous challenges, we were captured by the vision of what fuzzing could enable. Our discussions and brainstorming sessions sparked so much inspiration.

I thank Ronja for motivating me on slow days. Our walks and coworking sessions are what kept me going.

I thank my parents for all the support and proofreading. Getting feedback from people who do not spend the majority of their lives in front of computer screens really puts things into perspective.

Finally, I thank the members of the Casual Coding Club. The amount of deranged ideas, feedback, and joy I received during our cozy weekly meetings is invaluable.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

AFL	American Fuzzy Lop
API	Application Programming Interface
AST	Abstract Syntax Tree
CPU	Central Processing Unit
I/O	input/output
JSON	JavaScript Object Notation
LLM	Large Language Model
LSP	Language Server Protocol
QA	Quality Assurance
STDIN	standard input
STDOUT	standard output
VM	Virtual Machine
VS CODE	Visual Studio Code

1 Introduction

When writing code, it is common to start by writing a draft that handles most inputs correctly ("the happy path") and only then implement the behavior for edge cases. For example, a developer might calculate the average of a list of numbers by dividing the list's sum by the number of list elements. While this works for most lists, empty lists cause a division by zero – a bug was introduced! Could tooling have prevented this?

To fix the code, there are two approaches: Either, you communicate assumptions about the input in clear terms – for example, an empty list might produce a human-readable error like "You cannot average empty lists" and the documentation for the function may explicitly state that lists should not be empty. Or you change the behavior of your code so that it handles edge cases – for example, it may return 0 or `null` for empty lists.

Both of these approaches require the developer to think about possible edge cases. If they do not, such unhandled edge cases might produce wrong values propagating further through the program. As a result, the root cause and the observed failure can be far apart. Finding and fixing such bugs can be time-consuming, especially if developers are unfamiliar with the implementation. Can we highlight edge cases while the code is written, preventing expensive bug hunts later?

Now consider a developer trying to understand code. As Rauch et al. [28] showed, this process gets easier if you have concrete values. Getting such values can be difficult for code deep inside the inner workings of a project. If the code contains complicated nested conditions, what kinds of inputs even reach a particular part of the code? If the code is thoroughly tested, you might find such inputs by changing the code and seeing which tests fail. However, if the code is not tested, what do you do?

In these scenarios, examples can help. What if the editor showed you edge cases that cause your code to crash? What if the editor gave you a bunch of inputs and explained how they flowed through the program? What if the editor could give you a set of inputs that reach a specific place in the code?

The infinite monkey theorem states:

> If an army of monkeys were strumming on typewriters they might write all the books in the British Museum. [6]

What if we applied this principle to programming to find bugs in our code? We do not have infinite time. But we have fast CPUs and can observe how code behaves for a given input.

Fuzzing does exactly that: A *fuzzer* generates random inputs, runs your program with them, and observes how it behaves. It then picks promising inputs that reach critical parts of your code and changes them slightly, exploring even more of the codebase. Figure 1.1 shows a command line running American Fuzzy Lop (AFL)[1], a popular fuzzer, on a program that takes numbers as arguments and outputs the average. In the screenshot, AFL already found several inputs that caused the tested program to crash.

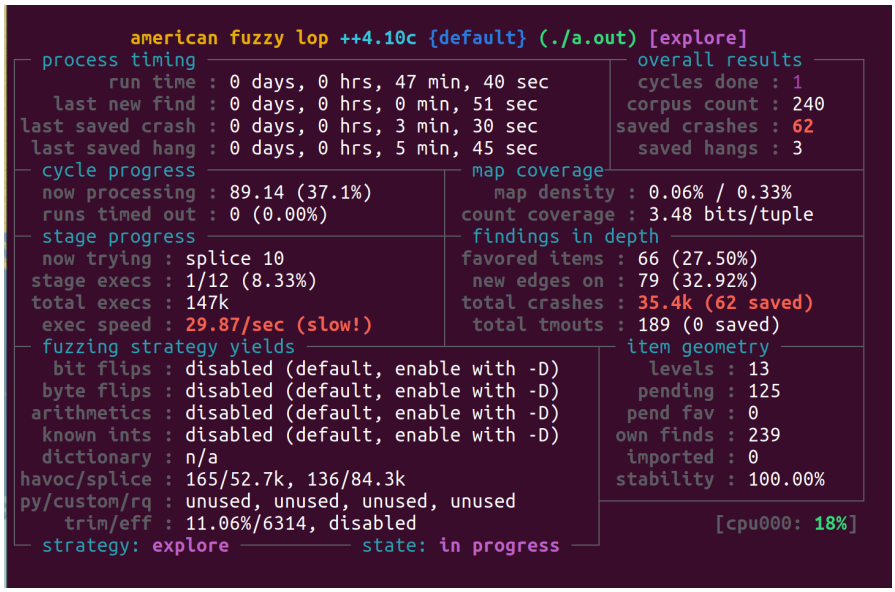


Figure 1.1: American Fuzzy Lop running in the command line

Fuzzing sounds simple, but because of the numerous runs and its evolutionary approach, it can find edge cases and security vulnerabilities in real-world software projects. Fuzzing found bugs in Firefox[2], LibreOffice[3] and many more software projects. Most bugs are triggered by inputs that look very unusual,

[1] https://github.com/google/AFL

[2] https://lcamtuf.blogspot.com/2014/09/cve-2014-1564-uninitialized-memory-when.html
https://lcamtuf.blogspot.com/2014/10/two-more-browser-memory-disclosure-bugs.html

[3] https://sourceforge.net/p/libwpd/tickets/3/

so they are difficult to discover by accident. Fuzzers create weird, inhuman inputs, and use this as an advantage.

If you write a program for averaging numbers, modern fuzzers will tell you within seconds that the function fails for empty inputs. If you set a fuzzer loose on `libjpeg`, a library for parsing JPEGs, the fuzzer will learn to generate valid JPEGs – some of them are shown in fig. 1.2. You see nothing useful in the generated pictures, but they use uncommon JPEG features such as black-and-white or arithmetic-coded data. [44]

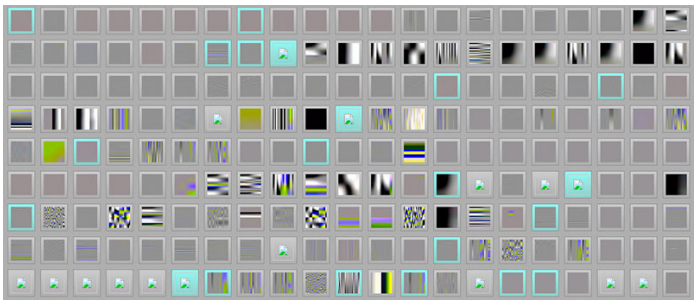


Figure 1.2: JPEGs generated by American Fuzzy Lop

While finding security vulnerabilities, a fuzzer gains interesting insights: It finds concrete inputs for the main path and edge cases. For each input, it knows which part of the code was executed ("the coverage"). [19, 48] Finally, the fuzzer knows which parts of the program were not covered during its tests. All of this may be interesting runtime information to have when writing code.

Over the years, code editors have been extended with lots of tooling: syntax highlighting, type hints, lints, and sometimes even concrete examples via Babylonian Programming. [28] This work explores whether it makes sense to add fuzzing to that list. In particular, we look at the following questions:

- Can fuzzing highlight edge cases while writing code?
- Can fuzzing help understand code with example inputs?

The remainder of this thesis is structured as follows: First, we describe how static type systems, linters, and tests make code more robust. We also show how immediate examples in the code help developers (chapter 2). We sketch out how the editor might show insights from fuzzing, an application of fuzzing that has not yet been explored (chapter 3). Next, we describe how we integrated fuzzing into the compiler pipeline and language tooling of Martinaise, a custom programming language (chapter 4). We evaluate our tool's example quality, performance, and communication, and compare it to existing tools (chapter 5). Finally, we discuss which areas need further improvements and how similar techniques could be applied to different programming languages (chapter 6).

2 Background and Related Work

We want to bring insights from fuzzing into the editor. Fuzzing is not unique in its ability to discover program flaws. Over the years, the software engineering industry has adopted several practices that help you write more robust code.

Software that is not robust, is expensive: A function that behaves unexpectedly for some combination of inputs might return a wrong value or crash. In big software systems built from many smaller abstractions, the wrong behavior of a single part can affect the entire system's stability. Figure 2.1 shows a system architecture with a fault in a transitive dependency.

This is not just a concern for runtime reliability: Fixing bugs deep in your software stack is also time-consuming for developers. Finding and correcting the misbehaving code might involve discovering transitive dependencies you did not know you had,[1] thinking in new domains on completely different levels of abstraction, and contacting and coordinating with maintainers.

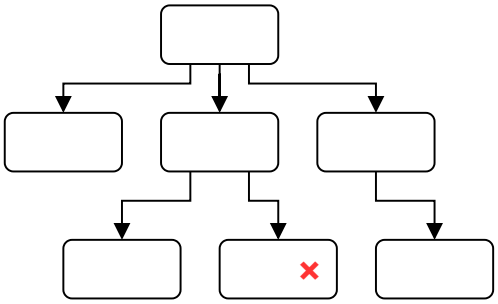


Figure 2.1: Dependencies

That is why much effort has gone into making the individual pieces of code more robust and communicating the interfaces (the arrows in fig. 2.1) more clearly. In this section, we give an overview of these techniques. Generally, they can be divided into ones that inspect the code statically (section 2.1 – section 2.2) and ones that require executing the code (section 2.3 – section 2.5).

[1] As of writing this thesis, creating a new React app automatically adds 1315 packages as dependencies.

2.1 Static Type Systems

Programming languages with static type systems describe the inputs and outputs of functions using types. Usually, you cannot execute code if types do not line up.

There are entire research fields dedicated to inventing, formalizing, and researching type systems. This chapter focuses on select works relevant to this thesis.

2.1.1 Concrete Simple Types

The simplest type systems work on concrete types, such as integers or strings. Listing 2.1 shows a concretely-typed function written in Dart[2] that sums a list of numbers.

Listing 2.1: sum in Dart

```dart
int sum(List<int> numbers) {
  var result = 0;
  for (final number in numbers) {
    result += number;
  }
  return result;
}
```

Here, the types communicate that the function only works on lists of integers. The compiler enforces that the caller and callee adhere to the type: Within the function, you can rely on `numbers` being a list of integers – and you get all the tooling benefits this brings (like better auto-completion). Outside the function, you *have* to call the function with the correct type; otherwise this will result in a compile error. Typically, your editor will show you an error directly at the call site.

2.1.2 Abstract Types

Some languages let you write functions that are generic over the types they accept. Usually, you can constrain which operations these generic types must support. These constraints are sometimes called interfaces,[3] type classes,[4] traits,[5] or abilities[6] and allow you to write reusable code that precisely communicates the requirements of types.

[2] https://dart.dev/
[3] https://docs.oracle.com/javase/specs/jls/se23/html/jls-9.html
[4] https://www.haskell.org/tutorial/classes.html
[5] https://doc.rust-lang.org/book/ch10-02-traits.html
[6] https://www.roc-lang.org/abilities

The `sum` function[7] from Haskell's standard library, shown in listing 2.2, uses two generic types, `t` and `a`. Everywhere this function is called (with concrete types), `t` must implement the `Foldable` type class[8] and `a` the `Num` type class[9]

Listing 2.2: sum in Haskell

```
sum :: (Foldable t, Num a) => t a -> a
sum = getSum #. foldMap' Sum
```

If you create a new type (such as one for complex numbers) and make it implement the `Num` type class by implementing the arithmetic operations,[10] the `sum` function still works for the new type.

Type classes are not *only* a set of operations (or methods in object-oriented terminology). Additionally, type classes have semantic meaning and might require the operations to fulfill extra invariants. For example, the `+` and `*` operations of the `Num` type class are customarily expected to be associative and commutative.

2.1.3 Conclusion

We saw different approaches to using types to document what kind of values code works on. Even though statically typed languages let you model many properties of your program in the type system, you decide which properties to type-check. For example, it would be cumbersome to prove that a variable cannot be zero every time you want to divide by it – so Haskell's division function[11] does not encode that requirement in the function signature and instead crashes. A big part of engineering software with static types is deciding which invariants to track in the type system and which to enforce during runtime.

2.2 Linters and Symbolic Execution

Apart from static type systems, which check type safety, tools can also statically analyze other properties. Just like static types, they try to understand the structure of the code rather than running it. This is usually done using a form of symbolic execution, where the tooling checks other properties of variables than types, for example, whether they are initialized. For example, linters can detect unused parameters and potentially uninitialized variables by analyzing the code's control flow. More advanced static analyses can alert you of code

[7] `https://hackage.haskell.org/package/base-4.21.0.0/docs/Prelude.html#v:sum`
[8] `https://hackage.haskell.org/package/base-4.21.0.0/docs/Prelude.html#t:Foldable`
[9] `https://hackage.haskell.org/package/base-4.21.0.0/docs/Prelude.html#t:Num`
[10] `+`, `*`, `abs`, `signum`, `fromInteger`, and `negate` or unary `-`
[11] `https://hackage.haskell.org/package/base-4.21.0.0/docs/Prelude.html#v:-47-`

that can never be reached or analyze the conditions that need to be fulfilled for some code to be reachable.

The line between compilers and linters is somewhat blurry – in many statically compiled languages, the compiler can also emit warnings that indicate how the code could be improved. Especially in dynamically typed languages like JavaScript, linters are widely used because they shorten the feedback cycle when writing the code [33] – for example, the linter ESLint can highlight unused variables, seen in fig. 2.2. [34]

```
function print(message: string) {    'print' is defined but never used.
  console.log("Hello!");
}
```

Figure 2.2: Warnings from ESLint

2.3 Testing

While statically checking code can prevent many errors, proving the code's behavior is impractical for many applications. That is why certain conditions are only checked during runtime – such as division by zero. To be confident that our code behaves correctly *when it is executed*, we can run it for some inputs: We test it.

2.3.1 Manual Testing

The most straightforward way to ensure the code does what it should is to test it: Just run your program, feed it some inputs, and see if it behaves as it should.

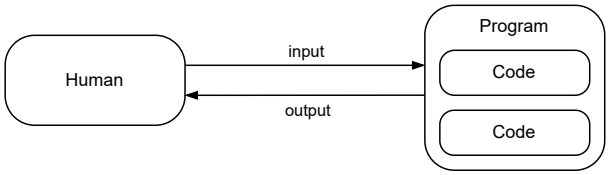


Figure 2.3: Architecture of manual testing

For big software projects, this can take so much effort that there are dedicated quality assurance (QA) teams whose only job is to use and try to break software. This is expensive, and getting feedback on a code change can last several days. Despite the high cost and slow feedback, manual tests are still worth it if you cannot exactly specify when your code is correct vs. wrong. For example, in games, many bugs are only obvious to human players. That is why games

often have dedicated QA teams, which usually make up about 10 % of the development cost [15].

Can we test with less effort?

2.3.2 Unit Tests

If you can specify the correct behavior of your code, unit tests are usually a better alternative: Rather than manually testing behavior, we specify inputs and expected outputs ("a test") and let computers test that our code produces the correct result. Because the tests are just normal code, they can test individual "units" of code (usually functions) in isolation.

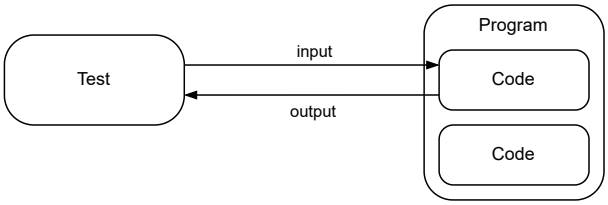


Figure 2.4: Architecture of unit testing

For most programming languages, there are testing frameworks that allow you to specify inputs and outputs. They run the tests and display mismatched results in a human-readable way. Listing 2.3 shows unit tests in Zig, where you can write top-level `test` blocks and run them from the command line.

Listing 2.3: Unit tests in Zig

```
const std = @import("std");

fn average(numbers: []i64) { ... }

test "calculate average" {
    try std.testing.expectEqual(2, average([_]i64{ 1, 2, 3 }));
}
```

Because running unit tests does not require human effort, we can automate re-running them every time the code changes. This prevents us from accidentally breaking our code later. Khorikov [13] asserts that code deterioration naturally occurs in big software projects maintained over a long timeframe, and unit tests help to slow that process down.

Metrics such as code coverage – the set of executed code – can be used to determine if a codebase is sufficiently tested. [9] Depending on the granularity, you can distinguish between different kinds of code coverage:

Statement Coverage We track which statements of the source code were executed. In practice, this is the most used form of coverage tracking. [19]

Branch Coverage We only track how execution continues at flow constructs (for example, at conditional expressions or loops). If the code does not crash, this form of coverage is more accurate than statement coverage: If we know which branches were executed, we also know which statements were executed. For empty branches, we get additional information over statement coverage.

Path Coverage In this case, we track the entire execution path (for example, how often a loop was chosen). This can be even more precise than other kinds of coverage. For example, assume two sequential `if`s are always both chosen or both not chosen. Unlike branch coverage, path coverage can distinguish this from the case where only one conditional is chosen.

Depending on how critical it is to get some part of the code right, you might have different demands on the coverage you achieve. Testing the main path may be enough for some parts of the code. For other (perhaps security-critical) parts, you may want to test every possible path the program can take.

While unit tests make testing reproducible and coverage can help you find a sweet spot on the effort-to-reward curve, testing your code thoroughly is still a lot of effort. Can we test with less effort?

2.3.3 Property-Based Testing

When writing unit tests, your workflow is essentially this: You craft inputs for your code and then compare the result of the computer's calculation with the one done in your head. Property-based testing lets the computer generate inputs for you: A framework generates random inputs. Your tests receive those inputs as arguments and can then test properties of your code that should be true for any input. [26]

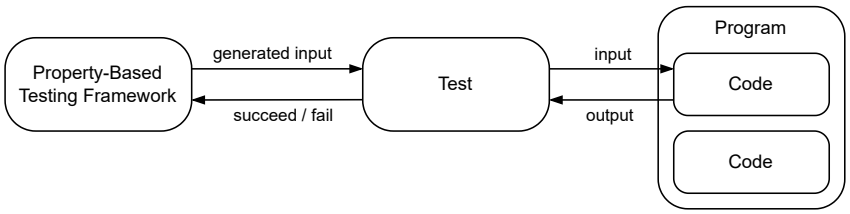


Figure 2.5: Architecture of property-based testing

In 1997, Fink et al. introduced the specification language TASPEC for defining properties [7]. Today, property-based testing frameworks usually allow you to specify properties in the same language as the implementation. There is

QuickCheck[12] for Haskell, Hypothesis[13] for Python, Glados[14] for Dart, and many more for other programming languages.

To demonstrate how property-based testing works, we will test a function that picks the maximum from a list of numbers. Listing 2.4 shows Dart code using the Glados package to test properties of a `max` function. The three tests tell the Glados framework which types of values to generate. They then accept an input, call the `max` function, and check a property about the relation between the input and output.

Listing 2.4: Property-based testing in Dart

```dart
// If the list is empty, return null, otherwise the biggest number.
int? max(List<int> list) { ... }

Glados(any.list(any.int)).test("max null -> empty", (list) {
  if (max(list) == null) {
    expect(list, isEmpty);
  }
});

Glados(any.nonEmptyList(any.int)).test("max >= items", (list) {
  final maximum = max(list);
  for (final item in list) {
    expect(maximum, greaterThanOrEqualTo(item));
  }
});

Glados(any.nonEmptyList(any.int)).test("max in list", (list) {
  expect(list, contains(max(list)));
});
```

Because all the mathematical properties that make a `max` function correct are encoded in tests, an incorrect implementation will likely fail one of these tests. For example, the implementation in listing 2.5 produces a wrong output for lists containing only negative numbers – it will return 0 instead of the largest negative number.

Listing 2.5: Wrong implementation of a max function

```dart
int? max(List<int> input) {
  if (input.isEmpty) {
    return null;
  }
  var max = 0;
  for (final item in input) {
    if (item > max) {
      max = item;
    }
  }
  return max;
}
```

If you run the tests from listing 2.4 for the function implementation from listing 2.5, the framework will produce a minimal example that violates the

[12] `https://hackage.haskell.org/package/QuickCheck`

[13] `https://hypothesis.works/`

[14] `https://pub.dev/packages/glados`, created by the author of this thesis

tested properties: a list containing -1. Listing 2.6 shows the output of running the property-based tests.

Listing 2.6: Output of property-based tests on the wrong max implementation

```
Tested 9 inputs, shrunk 2 times.
Failing for input: [-1]

  Expected: contains <0>
    Actual: [-1]
     Which: does not contain <0>
```

The framework itself does not know anything about your types. Instead, it uses generators to create new instances and to incrementally reduce an instance failing a test to create a minimal failing example. In the above example, `any.int` is a generator for integers. `any.list` is a function that accepts a generator for the list items and returns a new generator for lists.

Most property-based testing frameworks work in two phases, as shown in fig. 2.6. [26]

In the exploration phase, the framework asks the generator to create a random value. It then runs the test with that value, trying to get it to fail. The strategy of creating values differs between generators. The `any.list(any.int)` generator chooses a random size, generates that amount of integers, and puts them into a list. The `max` example from above causes these inputs to be generated:

Listing 2.7: Generated inputs from Glados

```
[5, 2, -6, -4, 7, -9, -9, 0] -> succeeds
[5, 0, -7, 10, -3, 1, 8, 11, -10] -> succeeds
[2, 8, -9] -> succeeds
[5] -> succeeds
[1, -13, -5, 10] -> succeeds
[15, -4, -8, 5, 7, -9, -10, 4, -9, -9, -12, 3] -> succeeds
[12, 7, 15, -6, 10, -15, -9, -3, 7] -> succeeds
[-4, 14, 16, 1, -14, -2, -2, -14, -4, 14] -> succeeds
[-5, -3]  -> fails
```

If the framework finds a failing input, the entire test will fail. In our example, because the test fails for `[-5, -3]`, we know that our `max` implementation or the test itself is incorrect. To ease debugging, the framework now tries to minimize the failing input before reporting it: It enters the shrinking phase.

In the shrinking phase, the framework already knows an input that causes the test to fail. It then asks the generator to make the input slightly smaller, which results in a few variations of the input. It runs the test for those values. If one of them still causes the test to fail, it uses that input as the new baseline and tries to shrink that value further. Otherwise, the minimal failing input is reported to the user.

In our case, the `any.list(any.int)` generator first tries removing items from the list and then moves the individual integers closer to zero. For example, the `[-5, -3]` list can be simplified to `[-3]`, `[-5]`, `[-4, -3]`, or `[-5, -2]`. When minimizing `[-5, -3]` for the `max` function, the framework tries the inputs

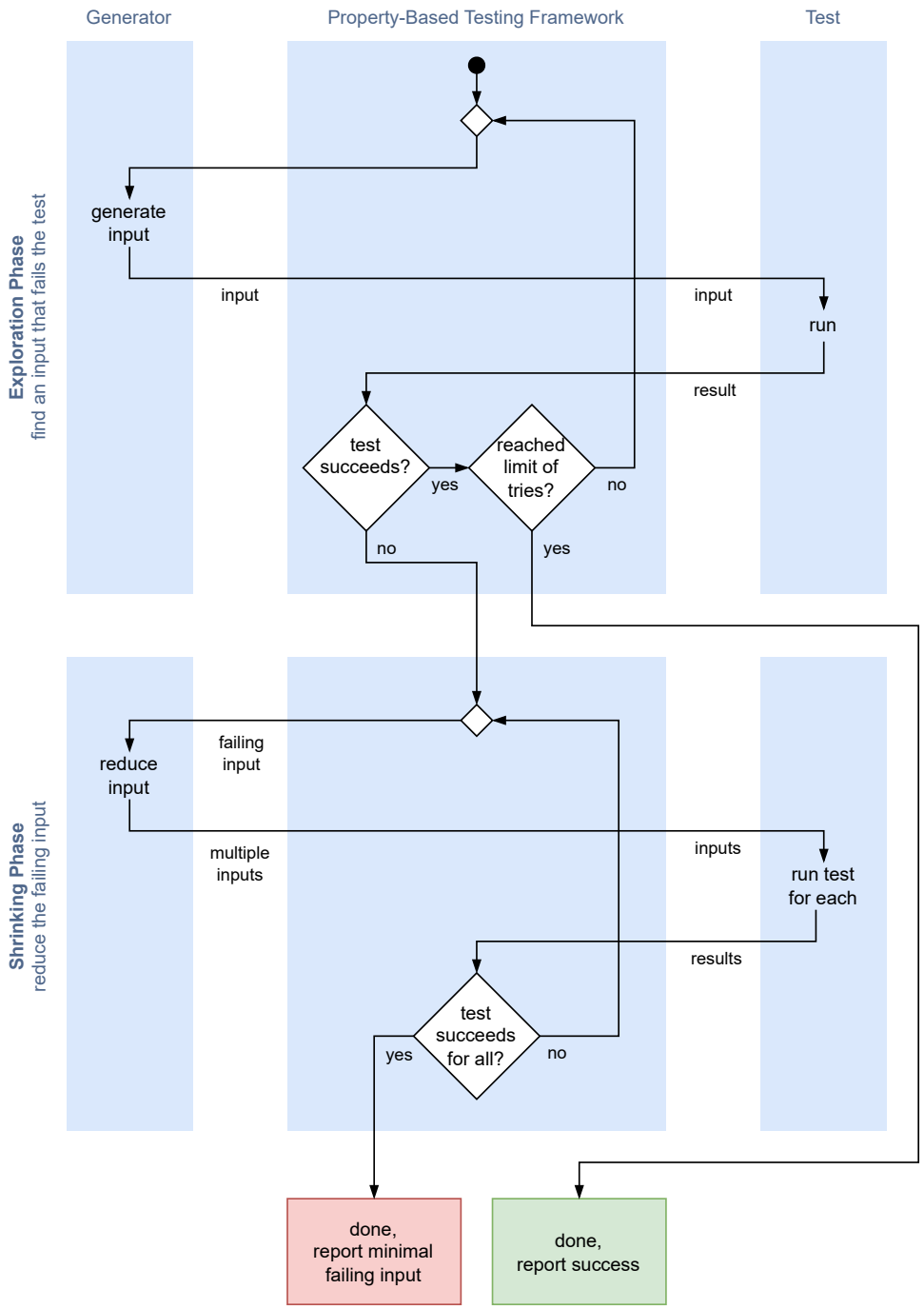


Figure 2.6: Phases of property-based testing

shown in fig. 2.7. When a simpler input fails the test, it is immediately chosen as the next baseline. The last failing input, `[-1]`, is reported.

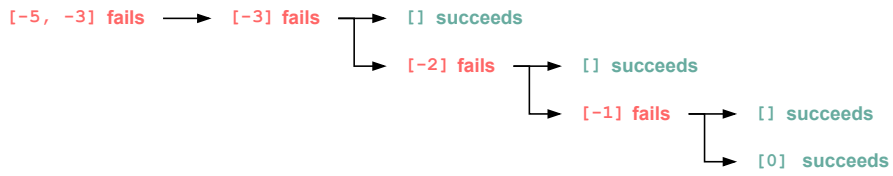


Figure 2.7: How Glados shrinks inputs that violate properties

Property-based testing was only effective in our example of the `max` function because the tests specify exactly how the function should behave. Even though most functions are not as mathematically well-defined as the `max` function, properties can still make you more confident in your code.

For example, if you have a reference implementation, you could compare your implementation to that – your property could be that both implementations result in the same output. [17] Unit testing is a form of that: You have a correct, trusted implementation in your head and want to ensure that the code behaves the same by sampling it for some inputs.

Even if you do not have a reference implementation or find properties that specify the correctness, you can still use property-based testing to check obvious properties. For example, even when you do not want to check that sorting a list behaves correctly, you can still check that the resulting list has the same length as the original one.

But still, the question remains: Can we test with less effort?

2.3.4 Fuzzing

Rather than testing individual functions, fuzzing refers to testing entire programs with random inputs to check if they crash. You can view fuzzing as a variant of property-based testing for the most general property that programs should have: That they do not crash, no matter the input.

Historically, fuzzing evolved from an outside-in view, analyzing the robustness of other people's programs. During a stormy night, Prof. Barton Miller was logged onto his workstation at the University of Wisconsin on a dial-up line from home. [24] Because of the rain, there was lots of noise on the telephone line, affecting his inputs. This caused many of the Unix utilities to crash.

In the next semester, he asked his students a research question: How many Unix utilities crash or hang if you feed them random inputs?[15] Their experiments with this technique revealed bugs in 24 % of the 90 programs that

[15] https://pages.cs.wisc.edu/~bart/fuzz/CS736-Projects-f1988.pdf

were tested. In the resulting paper, Miller notes that "while our testing strategy sounds somewhat naive, its ability to discover fatal program bugs is impressive." [24]

Today, fuzzing is a common technique to make security-critical software such as firmware, operating systems, browsers, and applications more robust. State-of-the-art fuzzing uses a similar architecture to Miller's initial implementation, shown in fig. 2.8: A component called *fuzzer* generates inputs, runs the program with those inputs, and receives feedback about the program's behavior such as whether the program crashed. Once the fuzzer finds a crash-inducing input, it gets reported.

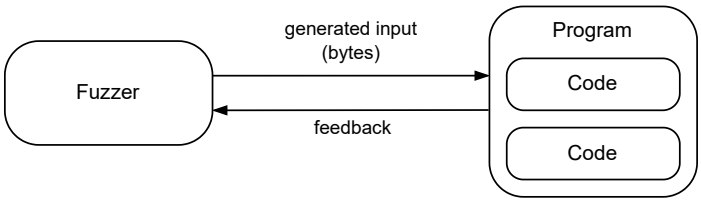


Figure 2.8: Architecture of fuzzing

While the general architecture remained, state-of-the-art fuzzers differ in the kind of feedback they receive and in how they generate the inputs. Depending on the amount of feedback, fuzzing techniques can be roughly divided into three groups: [4, 29, 39]

Blackbox fuzzing The fuzzer receives no feedback about the inner workings of the program. The only feedback is externally visible behavior such as whether the program crashed or not and the execution time. The simplest form of blackbox fuzzing is Miller's initial approach. More advanced techniques such as grammar-based fuzzing allow the user to specify a grammar for generating a variety of well-formed inputs that can fulfill even complex invariants.

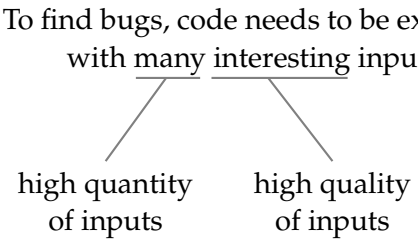
Greybox fuzzing We know a little bit more. The fuzzer receives simple coverage feedback and can use this to judge the quality of inputs. For example, inputs that only reach a few instructions at the beginning of the program are perhaps not as interesting as inputs that lead deep into the program's control flow logic. Inputs that reach new code locations are slightly mutated to explore those code paths further. Common mutations insert new bytes into the input, remove parts of the input, or replace some of the input with randomly generated new bytes. Continuing this gradual evolution of inputs over many generations, the emerging inputs usually stress most of the program's code.

Whitebox fuzzing The fuzzer can access the program code (either on source or machine code level) and analyze the program's structure. For example, it

can analyze the control flow abstractly and then use a constraint solver to generate inputs that reach an interesting location.

The most widely used open-source fuzzers are based on American Fuzzy Lop (AFL) [45], an early implementation of a Greybox fuzzer. To this date, AFL has found critical security vulnerabilities in firmware, operating system kernels, web browsers, databases, file systems, and many other projects.[16] [42, 45, 47] Because fuzzing is easy to apply to existing projects without changing the code and it is effective at finding bugs, many security-critical projects test their code using fuzzing.[17]

Fuzzing techniques differ in how they deal with the tradeoff that is inherent to thoroughly testing code:



Blackbox fuzzing solely focuses on the quantity of inputs. Generating each random input works in constant time. As a tradeoff, inputs will usually not flow deep into your program. For example, if your program starts with the guard condition `if (input[0] == 'a')`, a Blackbox fuzzer needs to generate 256 inputs on average to execute this clause just a single time. For programs that do any input validation, only a tiny fraction of inputs will reach interesting code.

Whitebox fuzzing focuses on the quality of inputs. Because it analyzes the program flow and uses sophisticated constraint solvers, there is a big constant overhead for generating inputs. For typical programs, the time spent generating inputs trumps the actual runtime.

Greybox fuzzing balances the quantity and quality of the input generation. Input generation works similarly fast to Blackbox fuzzing. At the same time, coverage feedback allows the fuzzer to identify and slightly modify inputs with high coverage, resulting in interesting inputs over time. Greybox fuzzing is typically more effective at finding bugs than the other two variants. Hence, most fuzzing research focuses on Greybox fuzzing. Table 2.1 compares the different fuzzing techniques side-by-side.

[16] Some high-profile projects where AFL found bugs: Mozilla Firefox, Apple Safari, iOS kernel, sqlite, Linux ext4, Tor, PHP, OpenSSL, OpenSSH, LibreOffice, libpng, curl, GPG, OpenCV, zstd, lz4, MySQL, . . .

[17] See also: `https://github.com/google/oss-fuzz`

	Blackbox	**Greybox**	**Whitebox**
feedback	external behavior	simple metrics	total control flow
input generation	random, dictionary	random, mutation	constraint solver
input quantity	high	high	low
input quality	low	higher over time	high

Table 2.1: Comparison of fuzzing techniques

This thesis aims to bring examples derived by fuzzing into the editor. As Greybox fuzzers generate many examples and can achieve a high quality over time, we describe how they are usually implemented. Because AFL was the first widely used open-source Greybox fuzzer, many other Greybox fuzzers followed its general architecture, which is well-documented [45] and shown in fig. 2.9.

1. The fuzzer maintains a pool of inputs. Initially, this pool is empty or filled with examples explicitly given by the developer.
2. The fuzzer either generates a new, random input or chooses an input from the input pool and mutates it slightly.
3. The fuzzer runs the program, sending the generated bytes into its standard input (stdin). The fuzzer observes the coverage and whether the program crashed. AFL implements coverage tracking by rewriting the machine code so that branch instructions update a global coverage map based on whether the branch was chosen.
4. If the input causes the program to take branches that were not taken before, the input is added to the input pool.
5. Go to step 2.

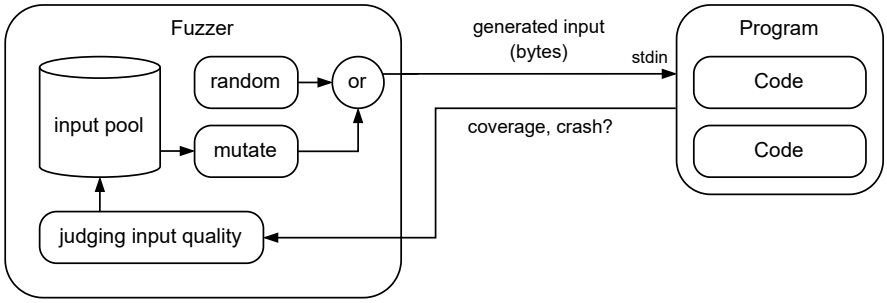


Figure 2.9: AFL's architecture

Since the inception of AFL, several papers [2, 3, 4, 8, 10, 18, 43] have proposed additions to make fuzzing even more effective. For example, Böhme et al. [3] propose favoring inputs with a unique coverage profile compared to other inputs. This compensates for over-representing the "happy path" in the input pool. Using their reweighed inputs, the fuzzer chooses uniformly among all explored code paths rather than among all inputs. AFL++[18] consolidates several such extensions to AFL into a single project, now considered state-of-the-art of open source fuzzing.

A remaining question worth looking at is what inputs programs receive. Usually, fuzzing only concerns itself with the input given over the standard input (stdin), but that is not the only factor determining what the program will do. A program may behave differently based on environment variables, the content of the file system, the result of network traffic, the scheduling of its threads, etc. To get these non-obvious inputs under control, some fuzzers add a virtualization layer between the program and the actual hardware. [29] This allows the fuzzer to generate new inputs for all these input surfaces and makes the program execution completely deterministic.

Often, a fuzzing session is followed by the minimization of crash-inducing inputs using Delta Debugging [46], a similar technique to the shrinking phase of property-based testing. Apart from that, there is no research on how to present the fuzzing results. Usually, crashing inputs are displayed in a terminal, but other insights from fuzzing (such as example inputs with a unique coverage) are not a subject of interest. We want to change that by showing examples in the editor.

2.4 Babylonian Programming

Annotating source code with concrete examples is not a new idea. This practice is called Babylonian Programming [27, 28] – 4000 years ago, the ancient Babylonians already used concrete values to understand abstract algorithms. [14] Human brains are great at recognizing patterns, so examples help us comprehend source code. Many software libraries are distributed with examples, which you can find in tests, technical documentation, and tutorials.

Rauch et al. [28] compare development environments showing concrete values right next to the source code. Figure 2.10 shows a selection of these editors.

Inventing on Principle In Bret Victor's talk "Inventing on Principle" [36], he presents the editor shown in fig. 2.10a. You can specify example values for function inputs in the right pane next to the source code. The editor

[18] https://github.com/AFLplusplus/AFLplusplus

```
function binarySearch (key, array) {

    var low = 0;
    var high = array.length - 1;

    while (1) {

        var mid = floor((low + high)/2);
        var value = array[mid];

        if (value < key) {
            low = mid + 1;
        }
        else if (value > key) {
            high = mid - 1;
        }
        else {
            return mid;
        }
    }

}
```

```
  key = 'g'
array = ['a','b','c','d','e','f']
  low = 0
 high = 5

  low =  0  |  3  |  5  |  6  |  6  |
 high =  5  |  5  |  5  |  5  |  5  |
  mid =  2  |  4  |  5  |  5  |  5  |
value = 'c' | 'e' | 'f' | 'f' | 'f' |


  low =  3  |  5  |  6  |  6  |  6  |
```

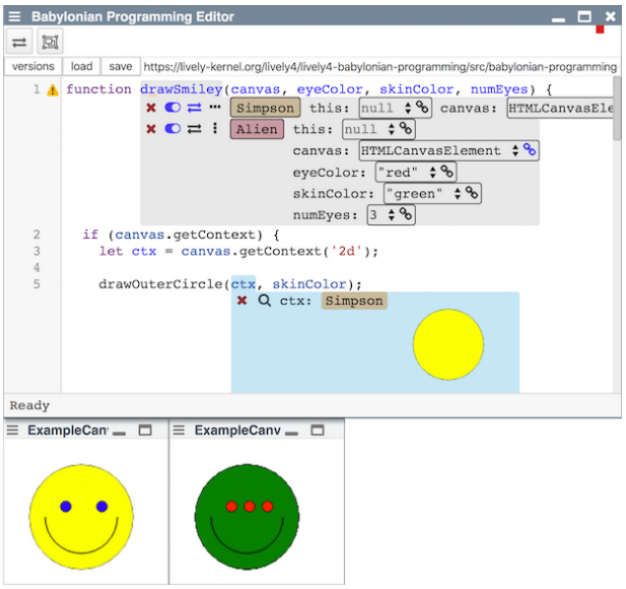
(a) Editor from Inventing on Principle

```
var low = 0;
var high = 5;

while (low <= high) {
  p(low, [0,1,2,3,4,5]);
  p(high, [5,5,5,5,5,5]);

  low++;
}
```

(b) Editor with Live Literals



(c) Editor from Babylonian Programming

Figure 2.10: Editors enabling Babylonian Programming

will then show you the concrete values of all variables and how they evolve as the function executes. This works even for code containing control flow structures. In fig. 2.10a, the editor shows the intermediate results of a binary search.

Live Literals are values in your code that the tooling automatically updates [35]. Figure 2.10b shows code containing several probes (the `p` functions). The second argument of these functions will be replaced by the literal value that the first argument evaluates to. As you type, the source code of these live literals will update to reflect changes.

Babylonian Programming Editor This editor, shown in fig. 2.10c, combines immediate feedback in the source code with a rich graphical representation of values. This representation is bidirectional – when you use sliders, the code updates to reflect that.

A major difference among these tools is how developers specify examples. Rauch et. al [28] distinguish between two methods of providing examples:

Implicit examples are provided in the source code, thus requiring code modification.

Explicit examples are provided using a special syntax or separate UI.

By using fuzzing, we add a third origin for examples: The computer itself. Unlike traditional Babylonian Programming tools, we can find interesting examples and show them to developers.

From the developer's perspective, hints based on fuzzing will feel less like a conscious debugging or exploration effort and more like passive, always-available tooling similar to information derived using static analyses.

2.5 Large Language Models

Large language models (LLMs) have become more common for assisting during development, especially for writing code [41]. LLMs are also used to generate tests automatically [31]. There is a parallel to fuzzing: Both techniques generate examples to test your code. However, the focus is different.

LLMs can generate human-readable examples deemed interesting based on the LLM's static understanding of the code. Because LLMs have to approximate the runtime behavior of code internally, the generated examples can be syntactically incorrect or not cover the entire behavior space. [16, 25, 30]

On the other hand, examples found through fuzzing are grounded in the actual semantics of the code. Some approaches [22, 40] try to integrate LLMs into the input generation and mutation parts of fuzzing, guiding the fuzzer

to sensible inputs in a more targeted way. Wang et al. [38] adapt the LLM generation to only generate tokens that result in valid data; this allows using LLMs while guaranteeing a computer-parsable result.

Because fuzzing has not been established as a basis for editor tooling, our prototype only uses a simple fuzzing approach. In the outlook, we discuss how LLMs could be adapted to this approach.

2.6 Conclusion

We have looked at several tools that help write more robust software and understand how software behaves. In general, we can draw a line between static and dynamic tools:

Static tools analyze the structure of the code. As they analyze our code while we are writing it, they give immediate feedback about the state of the code. Because static tools cannot reason about the runtime behavior, developers need to encode relevant properties for static tools – for example, by modeling properties in the type system.

Dynamic tools execute the code. Depending on the setup, they can also give immediate feedback – for example, unit tests that re-run when you edit the code. However, getting into a dynamic setting usually requires some extra effort from developers (such as writing unit tests or starting a debugging session). As dynamic tools only run the code for specific inputs, they cannot prove the correctness. [5]

Fuzzing can be a more useful dynamic tool if integrated well into the editing experience. By automatically fuzzing code as it is being written, developers could get immediate feedback about the state of their code. This has the potential to be particularly helpful for catching classes of errors such as unhandled edge cases.

3 Approach

In the previous chapter, we have seen that fuzzing does not require examples, can be completely automated without program modification, and yields useful results. We have also seen how concrete examples can help with understanding code. Now we want to combine those approaches to generate examples for functions.

To do that, we have to address the limitations of fuzzing:

Granularity Fuzzing works on entire programs, not smaller pieces of code. Fuzzers generate inputs on the byte level.

Locality Fuzzing is not integrated into the editor but is a separate tool invoked from the command line.

Performance The deeper a bug is within your code, the longer it takes for the fuzzer to reach it.

We build a fuzzer that works on the granularity of functions instead of entire programs. This enables new tooling, such as fuzzing functions in an isolated way and showing example calls next to the code. This is how we envision such a tool to behave:

Fuzz by Default Just by writing a function, it should automatically be fuzzed. The tooling should be able to generate random values for the function parameters.

Minimize distance between code and examples Fuzzing should show edge cases while the code is written, tightening the feedback loop. We want to minimize the distance between writing code and getting examples, both spatially (showing results in the editor instead of on the command line) and temporally (fuzzing automatically as you write your code).

Showcase function behavior Examples should cover the entire behavior space of a function. For example, inputs for an `average` function could show the behavior for lists with multiple items, one item, and no items (causing a crash in the implementation from fig. 3.1). Multiple examples showcasing the same behavior are not as interesting as inputs that explore the entire function's behavior.

This thesis does not attempt to advance the state-of-art of fuzzing. Instead, we want to explore how fuzzing can be deeply integrated into editor tooling. Rather than creating a generic solution that works on the lowest common denominator of programming languages and fuzzers, we limit our scope to a single programming language. This allows us to integrate fuzzing tightly into the existing language tooling as a proof-of-concept, assessing the viability of fuzzing-based tooling in the easiest way. In chapter 6, we discuss how this approach could be generalized to other languages.

3.1 Choosing a Language for Our Prototype

We need to decide on a programming language to implement a proof-of-concept implementation of function-level fuzzing. The aspect of programming languages that impacts fuzzing the most is whether the types of function parameters are known statically (at compile time) or dynamically (only at runtime).

3.1.1 Dynamically Typed Languages

Dynamically typed languages do not constrain the function inputs in any way. For example, the Smalltalk method shown in listing 3.1 calculates the sum of a `Collection`.

Listing 3.1: sum in Smalltalk

```
sum
  "Compute the sum of all the elements in the receiver"
  ↑self reduce: [:a :b| a + b]
```

This code can handle input of different types. For example, since the `Complex` type for complex numbers supports the `+` message, the `sum` function works with complex numbers, even if the original code author did not anticipate that.

On the flip side, this flexibility means that developers using a function either have to look into the function's body to understand what inputs it expects or rely on context such as the names of parameters, explanatory comments, or external documentation.

Fuzzing could substitute the documentation you otherwise gain by using type annotations. For example, a fuzzer could find out that the `sum` function crashes if you call it on a list of strings, but it does not crash if you give it a list of numbers. The fuzzer might provide a few examples of valid function calls.

However, because Smalltalk's message dispatch is dynamic, the fuzzer cannot statically decide whether a class supports a method – you can decide on a per-object basis whether calling `+` raises a `MessageNotUnderstood` exception.

This makes it difficult for the fuzzer to generate receivers and arguments for methods – each method can work with an unlimited amount of objects.

3.1.2 Statically Typed Languages

We already gave an overview of some static type systems in section 2.1. Statically typed languages that favor concrete types are straightforward to fuzz. For example, if the `sum` function takes an input of `List<Int>`, a fuzzer could generate random lists of integers, similar to how property-based fuzzing works (see section 2.3.3). Unlike in dynamically typed languages, where it is easy to find an input that causes a function to crash, a crash of a statically typed function represents an edge case – either an invariant that is not modeled in the type system or just a programming bug.

Statically typed languages with abstract types and interfaces make fuzzing difficult. For example, assume the tool should fuzz a Rust function that accepts an input implementing the `Ord` trait[1], allowing for comparing values. The fuzzer would have to implement a concrete type that fulfills the `Ord` trait. However, properties of the required `cmp` function, such as transitivity, are only specified in the documentation.

3.1.3 Conclusion

Fuzzing usually works on entire programs, where the input format is well-defined (a byte stream on the standard input). Therefore, fuzzing individual functions will work best in languages with similarly concrete types. To evaluate whether fuzzing as editor tooling is useful in principle, we implement fuzzing for Martinaise[2], a custom programming language *that is designed to be easy to fuzz*: It has a simple, static type system. It heavily favors using concrete types rather than specifying interfaces. It does not support functions as values (first-class functions). It has a small compiler and simple language tooling, allowing us to integrate fuzzing directly into the compiler pipeline.

In section 6.1, we discuss how our approach could be adapted to work with real-world languages.

3.2 Showcases

Next, we consider which kinds of interactions fuzzing might enable. As you browse the code, the editor could test visible functions using fuzzing. It could then show concrete calls next to the source code, similar to what fig. 3.1 shows.

[1] `https://doc.rust-lang.org/std/cmp/trait.Ord.html`
[2] `https://marcelgarus.dev/martinaise`

The original code author does not need to write extra code to enable these hints. Edge cases where inputs cause a function to crash can also be highlighted.

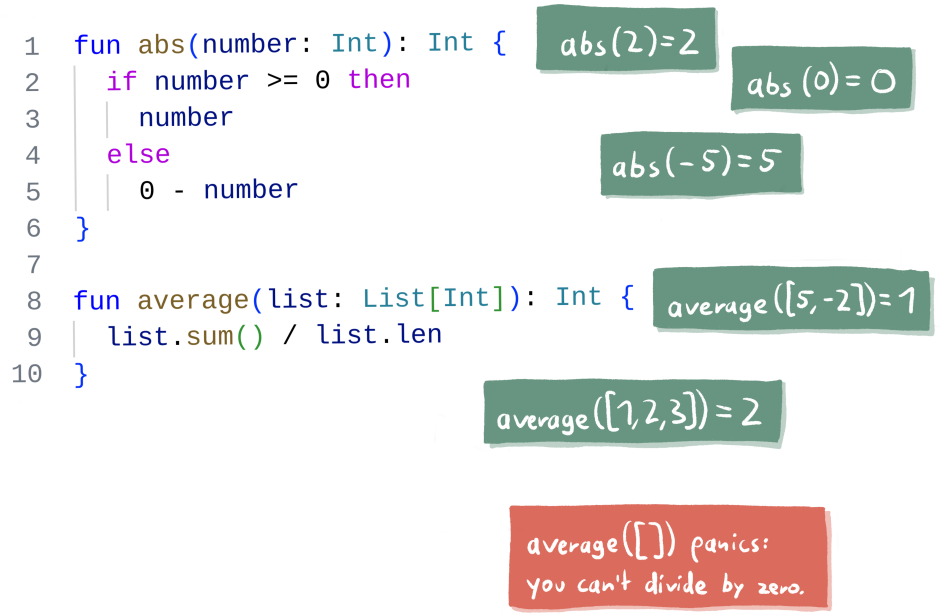


Figure 3.1: Example calls next to functions

Figure 3.2 shows how these concrete examples can be the basis for debugging sessions. Succeeding examples could be turned into unit tests. Failing examples can be debugged (if the function should not behave that way) or may be converted into asserts (if the caller is supposed to handle edge cases).

Unlike in traditional fuzzing, a crash does not necessarily represent an error in the code: Sometimes, crashing is the expected behavior. For example, the `/` operator should crash if the dividend is zero. Perhaps, the `average` function from fig. 3.1 should crash for empty lists. Hints about crashes describe the function's behavior without judgment – the developer decides whether to adjust the code.

Some data types may require invariants that are not encoded in the type system – for example, a range with a `start` and `end` might expect that `start <= end`. We want to enable fuzzing for functions that accept such custom data types. Thus, you should be able to customize how values of types are generated.

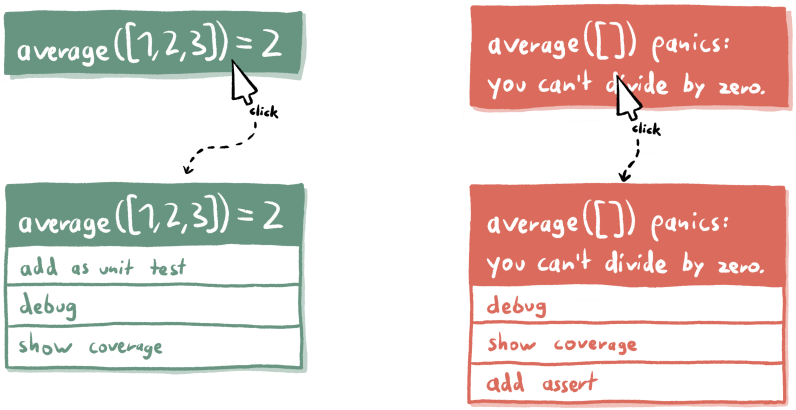


Figure 3.2: Examples as the basis for debugging sessions, tests, or asserts

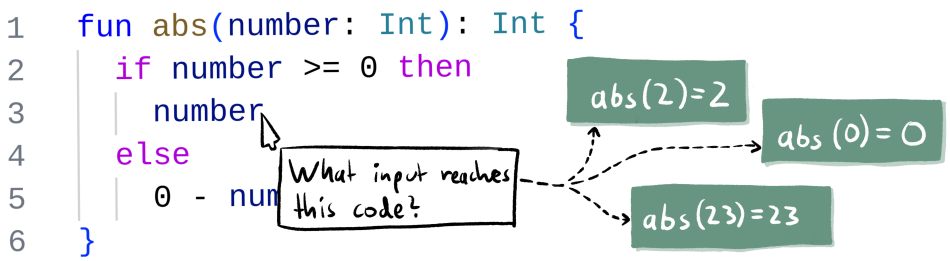


Figure 3.3: Examples that reach a piece of code

4 Implementation of Fuzzing for Editor Tooling

In the previous chapter, we described how fuzzing could improve the editing experience when integrated into the editor. We implement a proof-of-concept of this integration for a custom, small programming language, Martinaise,[1] simplifying our exploration of fuzzing-based tools.

After a language overview (section 4.1), we introduce the compiler pipeline (section 4.2) and adapt it to support fuzzing (section 4.3). Finally, we describe the existing Martinaise tooling and how language-level fuzzing is integrated into an editor, specifically Visual Studio Code (section 4.4).

4.1 Introduction to Martinaise

Listing 4.1 gives you a first impression of Martinaise. The example solves a simple leet code problem,[2] giving you a first impression of Martinaise's syntax and basic concepts.

Listing 4.1: Example program in Martinaise

```
| This is a comment.

enum Plot { empty, full }

| Checks if you can place n flowers in the flowerbed.
| To place a flower, the plots left and right to it need to be empty.
fun can_place_flowers(flowerbed: List[Plot], n: Int): Bool {
  var flowerbed = list(Plot.empty) + flowerbed + list(Plot.empty)

  for i in 1..(flowerbed.len - 1) do {
    if (flowerbed.get(i) is empty)
      and (flowerbed.get(i + 1) is empty)
      and (flowerbed.get(i - 1) is empty)
    then {
      flowerbed.&.set(i, Plot.full)
      n = n - 1
    }
    if n <= 0 then return true
  }
  false
}
fun main() {
  list(Plot.empty, Plot.full, Plot.empty).can_place_flowers(2)
}
```

[1] https://marcelgarus.dev/martinaise

[2] https://leetcode.com/problems/can-place-flowers

Most of the code will be self-explanatory to developers familiar with other programming languages. However, we want to highlight a few aspects.

4.1.1 Nominal Algebraic Data Types

Martinaise is statically typed – every value and variable has a type known at compile time (see section 2.1). If multiple branches of a conditional expression evaluate to a different type, that is a compile-time error.

Like in Rust or Haskell, you can combine existing types using structs and enums. Structs declare fields; instances contain all fields. Enums declare variants; instances have one active variant and contain the corresponding payload. Because structs and enums allow combining multiple other types using an *and* or *or* relationship (thereby forming an algebra in the mathematical sense), this combination of types is also called Algebraic Data Types. [37]

Types are nominal: They are compared by their name instead of by how they are defined. For example, if you define two structs with different names but the same fields, you cannot assign values of one type to the other type.

Martinaise does not support subtyping, inheritance, or interfaces. All types are known at compile time.

4.1.2 Function Overloading and Uniform Call Syntax

Martinaise supports function overloading: You can define multiple functions with the same name, but different parameter types. For each call, the compiler looks for a function that matches the types. This happens at compile time, so there is no dynamic dispatch.

Martinaise also has a Unified Call Syntax [32]: When you call a function using the dot syntax `left.function(arg)`, this is equivalent to calling the `function` with the `left` value as an extra parameter: `function(left, arg)`. For example, line 23 in listing 4.1 calls `list(...).can_place_flower(2)`, which matches the function defined on line 7.

You can define custom operators by creating a function where the name consists of symbols. Similar to Smalltalk, operators have no precedence; they are evaluated from left to right. Like other functions, operators support overloading. For example, the + operation in line 12 adds two integers, while the + operations in line 8 concatenate lists.

4.1.3 Generics and Monomorphization

Types and functions can take type parameters in brackets (`[]`). To use these generic parts of the code, you need to specialize them – for example, because

the `List` type takes a type parameter, you cannot use just `List` as a type, but you can use `List[Plot]`, as seen in listing 4.1 on line 5.

The compiler does not type-check generic code in isolation. Instead, the code is type-checked and compiled separately for each combination of types you use. This so-called monomorphization process[3] [12] is similar to how C++ templates work and is shown in listing 4.2 and listing 4.3.

Listing 4.2: Code before monomorphization

```
struct List[T] { ... }

fun get[T](list: List[T], index: Int): T { ... }

| more code that uses the get function for List[Int] and List[Float]
```

Listing 4.3: Code after monomorphization

```
struct List[Int] { ... }
struct List[Float] { ... }

fun get(list: List[Int], index: Int): Int { ... }
fun get(list: List[Float], index: Int): Float { ... }
```

4.1.4 Access to Low-Level Primitives

Martinaise is a low-level language that compiles to a simple byte code. If you need precise control over the performance, you can use the `asm` keyword to implement functions directly in byte code instructions. Because Martinaise specifies the memory layout of types and the calling convention of functions, `asm` functions can seamlessly interact with regular Martinaise code.

Besides `structs` and `enums`, you can also define `opaque` types, which regular Martinaise code cannot inspect. Listing 4.4 shows how opaque types and `asm` functions are used to define `Int` and its `+` operator.

Listing 4.4: A low-level function implemented directly in byte code

```
opaque Int = 8 bytes big, 8 bytes aligned

fun +(left: Int, right: Int): Int asm {
  | data on the stack:
  | - 8 bytes: address of where to write the return value
  | - 8 bytes: left
  | - 8 bytes: right
  moveib a 8  add a sp load a a | left
  moveib b 16 add b sp load b b | right
  load c sp | return value address
  add a b store c a ret
}
```

[3] mono = single, morph = shape/type. After the monomorphization, each resulting function only works on concrete types.

4.1.5 Performance

For a language to be usable for fuzzing, it should be reasonably performant. Figure 4.1 compares the compilation time and runtime performance of Martinaise to Rust and Python on three benchmarks: a recursive Fibonacci calculation, an N-Body physics simulation, and a quicksort implementation. All benchmarks were measured using the default compilers on a desktop computer with up-to-date tooling as of October 2024.[4]

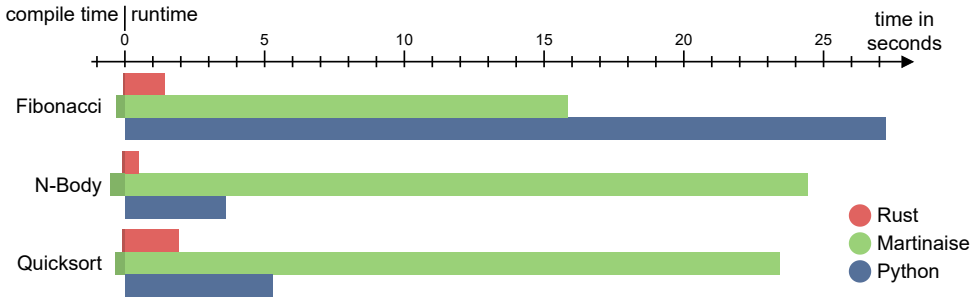


Figure 4.1: Benchmarks in Rust, Martinaise, and Python

The results make it obvious that Martinaise has no optimizing compiler. Modern compilers inline code of small functions, calculate constant operations such as `2 + 2` at compile time, etc. Martinaise does not: Calling a function results in the CPU executing a call instruction. As a result, all numeric operations like + perform function calls. Also, Martinaise's byte code instruction set is minimal – for example, it does not have a `sqrt` instruction, which modern hardware directly implements,[5] but instead the standard library implements `sqrt` entirely in software.

On the other hand, Martinaise is a low-level language. This has several performance benefits: Because the resulting byte code is on a similar abstraction level to machine code, the byte code VM can compile your code to machine code before executing. Types are known at compile time, so data is arranged in a dense memory layout. Performance-critical functions (such as `memcompare` and `memhash`, used for string comparisons and hashing) are implemented as `asm` functions. This makes the performance very predictable.

[4] The computer has an AMD Ryzen™ 7 5800X CPU running Ubuntu 24.04.1 LTS. C code code uses gcc 13.2.0. Rust code uses rustc 1.83.0-nightly. Martinaise uses Martinaise 9 for compilation and soil-zig for execution. Java uses javac 21.0.4 for compilation and OpenJDK 21.0.4 for execution. Python uses CPython 3.12.3. Each result shows the mean of 10 to 41 runs, as judged by hyperfine; the raw benchmark measurements are in the appendix.

[5] The two most common CPU architectures – x86_64 and aarch64 – support an `FSQRT` instruction to calculate square roots directly in hardware. [1, 11]

4.1.6 Usability for a Proof of Concept

As a programming language with a simple, static type system, a small compiler, and predictable performance, Martinaise lends itself well to a proof-of-concept implementation of showing editor feedback based on fuzzing.

In chapter 6, we discuss how our approach could be adapted to different programming languages, such as dynamically typed languages, and which insights apply in those cases.

4.2 Compiler Pipeline

Typical Greybox fuzzers have only minimal information about the code – they only care about whether a program crashed and its coverage. When fuzzing individual functions rather than entire programs, the fuzzer needs to understand which functions exist, which types they accept, and how those types are defined.

The compiler already knows the entities in the language, so we integrate fuzzing into the existing compiler instead of building a separate tool. This addresses the typical limitations of fuzzing, discussed in chapter 3:

Granularity The compiler knows all functions and type definitions. It can use that knowledge to generate random values of arbitrary types. By generating values for function arguments, the fuzzer can test individual functions.

Locality Fuzzing can run alongside the existing analysis, using all of the compiler's existing knowledge. Insights found through fuzzing can be displayed directly in the editor, using the same infrastructure as compiler errors.

Performance Traditional fuzzers start from the program entry point. To test a specific function, they must first find a combination of inputs that reaches that function. A function-based fuzzer can test every function directly, greatly reducing the required effort. Also, useful examples do not necessarily need to break the code: A hint that shows how an input flows through the program can already help.

Because we integrate fuzzing into the compiler, we give a quick overview of the compiler architecture. Martinaise has a small compiler (about 4500 lines). Like most compilers, it represents the program using a series of intermediate data structures ("stages"). Figure 4.2 shows the journey of the source code when it is compiled.

Source code We start with textual source code, potentially split across multiple files.

Abstract Syntax Tree The compiler parses source code into a tree representing the program's syntax. Each tree node corresponds to an action in the language such as calling a function, accessing a struct field, or defining a variable.

Some language constructs are already resolved at this stage. For example, `if` and `for` no longer exist in the abstract syntax tree. Instead, they are represented by the more general `switch` and `loop` constructs. Imported files are parsed as well. The final syntax tree contains functions and variable definitions from all imported files.

Monomorphized Intermediate Representation Starting from `main`, the compiler typechecks and compiles all called functions. This is where generics disappear: The `main` function only has concrete types. The compiler chooses the correct function for each call based on the parameter types and then compiles that function for concrete types.

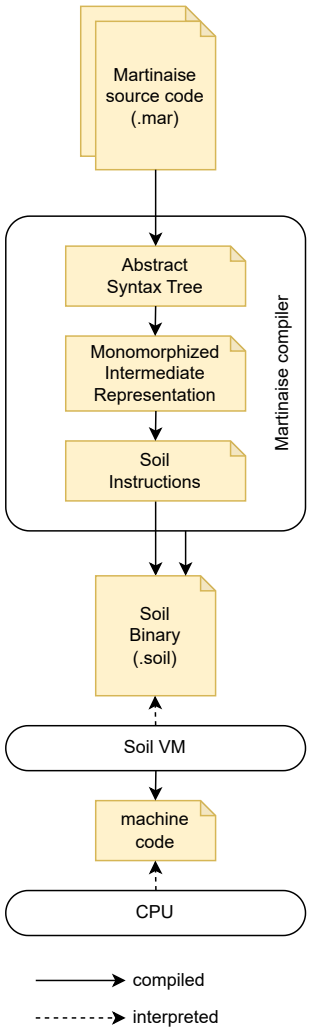


Figure 4.2: The Martinaise compiler pipeline

Soil Instructions Next, the compiler calculates efficient memory layouts for types and converts the content of monomorphized functions into instructions. Control flow constructs (`loop`s and `switch`es) become jumps. Local variables are explicitly loaded from and saved to stack memory. In the end, functions only contain a flat sequence of instructions. There is no longer a distinction between `asm` functions and normal Martinaise code.

The instruction set is called Soil.[6] The instructions are low-level[7] and register-based,[8] similar to instructions understood by CPUs. Listing 4.5 shows an example of Soil instructions.

[6] `https://marcelgarus.dev/soil`
[7] For a full list of the instructions and explanations, see the instruction table in section 6.2.
[8] The registers are: `sp` (stack pointer), `st` (status register), `a` – `f` (general purpose)

Listing 4.5: Exiting with 42

```
moveib a 21
moveib b 2
mul a b
syscall 0
```

The `moveib` instructions ("move immediate byte") moves concrete values into the `a` and `b` registers. Then, the `mul` instruction multiplies these registers, saving the result in the first register (`a = 42`).

Syscalls allow a Soil program to access functionality outside the VM. Syscalls have a defined number and calling convention. For example, `syscall 0` as seen in the example causes the VM to exit using the content of the `a` register as the status code. However, there are also syscalls to e.g. print to the standard output and perform file operations.

Soil Binary Finally, the compiler serializes instructions into byte code, a dense encoding of the instructions. It saves this byte code and string constants into a `.soil` file. This file contains everything that a Soil VM needs to initialize the VM's memory and start executing code.

Executing Soil Binaries Because the Soil instruction set is small, writing a Soil interpreter is not a lot of effort. Interpreters for Soil were written in JavaScript, Dart, Rust, C, Zig, and x86_64 assembly. Some of these even compile the byte code into machine code for better performance – this is manageable because machine code and Soil byte code are on a similar abstraction level.

4.3 Integrating Fuzzing Into the Compiler

We want to extend Martinaise to support the fuzzing of individual functions. Fuzzing should work by default, without extra effort. In particular, you should not have to tell the fuzzer how to generate inputs for your code.

Throughout this section, we will use a simple calculator program as an example to fuzz. We define terms like this:

Listing 4.6: Definition of mathematical terms

```
enum Term {
  number: Int,
  add: Operands,
  subtract: Operands,
  multiply: Operands,
  divide: Operands,
}
struct Operands { left: &Term, right: &Term }
```

Each `Term` is a number, an addition, subtraction, multiplication, or division. For numbers, the `Term` directly stores the `Int`. For arithmetic operations, the

term stores operands, which contain a left and right term. Essentially, `Terms` are binary trees. Figure 4.3 shows some examples.

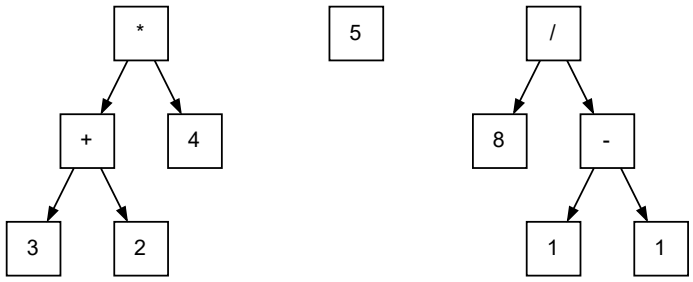


Figure 4.3: Examples of terms

To evaluate terms to a single number, we switch on `Terms`, shown in fig. 4.4. Numbers evaluate to themselves. The other terms evaluate the two child expressions and perform the corresponding operation. We want to show example calls and their result in the editor automatically. Not only that, we also want to highlight terms causing unusual behavior (such as a division by zero).

```
fun eval(term: Term): Int {  6 - 3 / 0 panics  5 * 0 -> 0  2 / 4 -> 0  0 + 3 - -
  switch term
  case number(num) num
  case add(op) op.left.eval() + op.right.eval()
  case subtract(op) op.left.eval() - op.right.eval()
  case multiply(op) op.left.eval() * op.right.eval()
  case divide(op) op.left.eval() / op.right.eval()
}
```

Figure 4.4: Hints for the eval function

To do that, our tool uses several techniques from property-based testing and fuzzing, seen in fig. 4.5: It generates random `Terms`, runs the `eval` function, observes the coverage, and uses that to judge how deeply inputs reach into the function code. It mutates promising inputs to explore the entire code of the function. The editor shows inputs representing the function's behavior.

To implement this, we need to solve several subtasks:

- Generate random values
- Mutate values
- Get coverage feedback when running a function
- Judge the quality of inputs for exploring the function

Some of these tasks have an obvious default: To generate a `Term`, the compiler could choose a random variant (for example, addition) and generate random operands (for example, two numbers). However, for some types, there are some

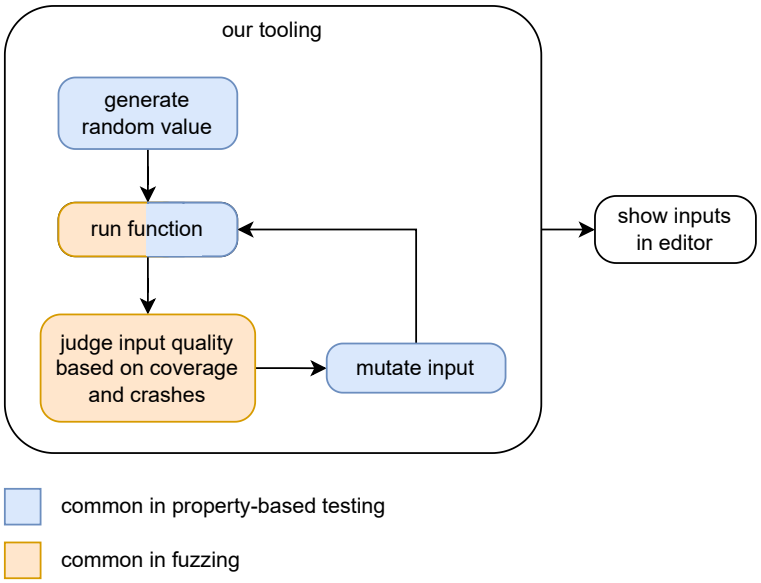


Figure 4.5: High-level overview of how the tool tests functions

implicit constraints the compiler does not know about. Take the definition of a range, shown in listing 4.7.

Listing 4.7: The Range type

```
struct Range[T] {
  start: T,   | inclusive
  end: T,     | exclusive; >= start
}
```

Usually, you create ranges using the `..` operator (as in, `1..3`). However, the property that `start <= end` is not encoded in the type system. The compiler cannot know that e.g. the range `5..2` is invalid.

We still want to be able to fuzz functions that accept types like `Range`, which have properties not encoded in the type system. However, for most types, the compiler should automatically implement code to generate and mutate values.

4.3.1 Generating Values

The fuzzer needs to be able to generate random values. As a simple, statically typed language, Martinaise code cannot reflect on types. Because we still want to be able to fuzz any function, the compiler inspects types and implements functions for generating values. We achieve this by defining a `generate` function in the standard library that works for any type, shown in listing 4.8.

Listing 4.8: The generate function

```
fallback fun generate[T](
  static: Static[T], random: &Random, complexity: Int
): T { ... }
```

The function is generic over a type `T`. The `...` as the function body is part of the source code, indicating that the compiler implements this function. The `fallback` keyword means this function is only chosen if no other function matches a call. This allows developers to customize the behavior of generating values for types.

During the monomorphization stage described in section 4.2, the compiler specializes functions for concrete types. This allows the compiler to implement the `generate` function differently depending on the concrete type used as `T`. For example, if the generate function is called for our `Term` type, the compiler can implement a version specific to `Term`s. It knows the definition of the `Term` type and can create straightforward code. After the monomorphization, only concrete, specialized functions exist, so the `generate` function is treated like all other functions in the rest of the compiler pipeline.

Depending on which type needs to be generated, the compiler has a different strategy:

Structs The compiler generates a random value for each field and creates a struct with those fields.
For example, if we want to create `Operands` (a struct defined in listing 4.6), we need a left and a right term. The fuzzer will generate two terms and then construct an `Operands` struct using them.

Enums First, the fuzzer randomly chooses a variant. Then, it generates a random payload for the variant. Finally, it constructs the enum variant with the payload.
For example, to generate a `Term` (an enum defined in listing 4.6), the fuzzer chooses which variant it should be – `number`, `add`, `subtract`, `multiply`, or `divide`. Then, it generates the corresponding payload (an `Int` for the `number` variant, `Operands` for all other variants) and creates the enum value.

Opaque types The compiler cannot generate opaque types. If the built-in `generate` function is used for an opaque type, that results in a compiler error. The standard library defines `generate` functions for some opaque types (such as `Int` or `Float`). Other opaque types can intentionally not be generated (such as `File` or `Address`). Consequently, you cannot fuzz functions that accept a `File` parameter – the editor shows no hints next to these functions.

However, this simple approach to generating values may not terminate for recursive types. Consider the `Term` type: There is a $\frac{1}{5}$ chance that the fuzzer will pick the `number` variant and terminate immediately. For the other variants, the fuzzer will try to generate two new terms; those have to terminate so that the entire generation terminates. Thus, the total probability of the `Term` generation terminating is

$$\begin{aligned} p_{\text{Term}} &= \frac{1}{5} \cdot p_{\text{Int}} + \frac{4}{5} \cdot p_{\text{Operands}} \\ &= \frac{1}{5} \cdot 1 + \frac{4}{5} \cdot p_{\text{Term}} \cdot p_{\text{Term}}. \end{aligned}$$

Solving this results in $p_{\text{Term}} = \frac{1}{4}$. In other words: Generating a `Term` only terminates in 25 % of cases. To fix that, the fuzzer needs to have more control over the shape of the terms that are being generated. We pass a desired complexity to the `generate` function, indicating how big the generated value should be. If the requested complexity is below a threshold, the `generate` function only chooses among non-recursive enum variants (in the case of `Terms`, it will always generate the `number` variant). As a result, generators will always terminate, even for recursive types like our `Term`.

Having a desired complexity even benefits non-recursive types: Assume a tested function accepts a `List[Int]`. Testing small lists is practical because they can quickly be generated and running functions with small inputs is usually faster. But some errors might only happen for big lists (or a big list is more likely to induce errors). The complexity allows the fuzzer to increase the input size over time.

We could add more advanced indicators than complexity to allow the fuzzer to control the shape of the generated values. For example, fig. 4.6 shows two terms with the same number of nodes, but different depths. If the fuzzer had a way of explicitly generating deep or shallow values, it could more easily test function behavior for these shapes. However, all this increases the implementation burden for `generate` functions. Having a target complexity is enough to guard against recursive cases. A developer who desires more sophisticated value generation policies can always implement a custom `generate` function for a type.

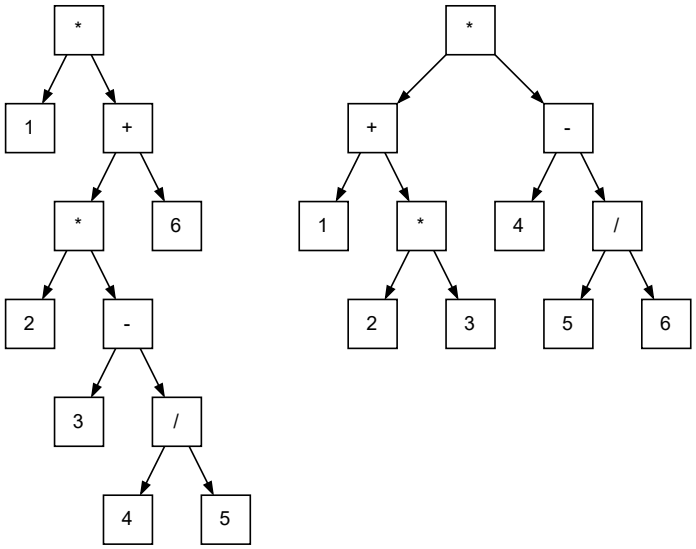


Figure 4.6: Deep and shallow terms

Finally, these are the parameters of the `generate` function:

RANDOM First, because we want the value generation to be reproducible, the `generate` function accepts a random number generator (called `Random` in the standard library). Being able to generate values deterministically simplifies testing of the fuzzer itself.

COMPLEXITY Second, the `generate` function accepts a `complexity: Float`. As discussed above, this allows the fuzzer to increase the input size over time and guards against infinite recursion for recursive types. There is no concrete requirement of what the complexity represents, it should just be a rough estimate of the input's size. The standard data structure's `generate` functions try to scale the size of the generated value proportional to the `complexity`. For example, the `List`'s `generate` function chooses a length and then divides the complexity among the items to be generated.

STATIC Third, the fuzzer needs to be able to call the correct `generate` function for a given type `T`. Because function overloading in Martinaise is based on the types of parameters, the function has to contain `T` somewhere in its type signature. We achieve that by defining an empty `struct Static[T] {}`. All `generate` functions accept a `Static[T]` as the first parameter, so they contain the type `T` in their call signature without requiring an instance of `T`. For example, to generate a `Term`, the fuzzer can create a `Static[Term]`, use that as the first argument, and the correct `generate` function will be chosen.

Listing 4.9 shows the `generate` function for the `Operands` struct[9] Line 4 splits the complexity into complexities for the two fields – for example, a complexity of `10` might be split into `[2, 8]`. Lines 5 and 6 then generate two `Terms` using the `generate` function for terms. Finally, line 7 creates an `Operands` struct.

Listing 4.9: Generating operands

```
1  fun generate(
2    static: Static[Operands], random: &Random, complexity: Float
3  ): Operands {
4    var complexities = random.split(complexity, 2)
5    var left  = static[Term]().generate(random, complexities.get(0))
6    var right = static[Term]().generate(random, complexities.get(1))
7    Operands { left, right }
8  }
```

Listing 4.10 shows the `generate` function for terms. Lines 4 – 13 choose a variant. The `chosen_variant` variable will contain the index of the variant to generate – 0 corresponds to a number, 1 to an addition, etc. For small complexities below the arbitrary threshold of 10, line 6 always creates a `number` term as this is the only non-recursive variant. For bigger complexities, line 8 chooses any variant. Finally, lines 10 – 21 map the chosen variant index to an actual instance of the enum variant by generating the corresponding payload and constructing the enum value.

Listing 4.10: Generating terms

```
1  fun generate(
2    static: Static[Term], random: &Random, complexity: Int
3  ): Term {
4    var chosen_variant =
5      if complexity < 10 then {
6        random.choose(list(0))
7      } else {
8        random.next_int(0..5)
9      }
10   if chosen_variant == 0 then
11     Term.number(static[Int]().generate(random, complexity))
12   else if chosen_variant == 1 then
13     Term.add(static[&Operands]().generate(random, complexity))
14   else if chosen_variant == 2 then
15     Term.subtract(static[&Operands]().generate(random, complexity))
16   else if chosen_variant == 3 then
17     Term.multiply(static[&Operands]().generate(random, complexity))
18   else if chosen_variant == 4 then
19     Term.divide(static[&Operands]().generate(random, complexity))
20   else
21     unreachable()
22 }
```

Table 4.2 shows the terms generated for different complexities.

[9]The compiler implements the functions in a lower-level intermediate representation. The version shown here is manually translated into code.

complexity	term
0	2
10	$-2 \cdot 9$
20	$3 \cdot 2 - 5$
30	$-10 \cdot -5/(7 + 8)$
40	$1 - (-8 + 4 + -5)$
50	$-2 \cdot 4 \cdot -9 \cdot -5/(-6 \cdot -2 \cdot -9/-5)$
60	6
70	301
80	20268
90	$7 \cdot -7 \cdot -6 \cdot 8 \cdot -4 \cdot 3 \cdot -6 + (0 + -1)/(6/-1) + (-9 - 8)/(-7 \cdot 2)$
100	$(-10 + 2) \cdot (0 + 3) + 10/0 - (-10 - 4) - (-2 - 4 + 0/-5)/((-9 - -7) \cdot (-10 - 1))$
200	814416606646585705
300	$16119763526034 - (-3 + -9 + 8 - 3 + 10/7 \cdot 0/2 - ((-8 + -7) \cdot -1 \cdot 7 + -2 - 4 + -8/8))$
400	$(33412542 + -10 - (3 - -5)/4 + 10 \cdot (-8 - 0) \cdot (3 + 6))/-1433751274838839 56$
500	$51052245512/(6 + 7 + -36) \cdot 54/28/(1/-4/1/(10 \cdot (7 + 9)) \cdot 47)$

Table 4.2: Generated terms of different complexities

4.3.2 Mutating Values

Only being able to generate values is not enough: What makes Greybox fuzzers so effective is the evolutionary approach – based on coverage feedback, they change inputs and test the code for similar inputs. Our fuzzer mutates inputs by calling a `mutate` function on the value. The compiler implements a fallback function, similar to how it treats the `generate` function. Listing 4.11 shows the definition of the `mutate` function in the standard library.

Listing 4.11: The mutate function

```
fallback fun mutate[T](
  value: T, random: &Random, temperature: Int
): T { ... }
```

The "value" parameter is the original value to be mutated. The "random" parameter is the source of randomness, just like in the `generate` function. The "temperature" parameter indicates how different the returned value should be.

Similar to the `generate` function, the compiler implements this function for concrete types `T` depending on what type it is:

STRUCT For structs, we choose a random field and mutate it.

ENUM For enums, we either generate a completely new value (possibly switching the enum variant) or mutate the payload.

OPAQUE Opaque types cannot be mutated by default. You must manually implement a `mutate` function to fuzz functions accepting these types.

Applying these strategies to our calculator example, Listing 4.12 shows the compiler-generated function for mutating `Operands`. Line 4 chooses a random field (the left or right operand). Lines 5 – 16 create a new struct, mutating the chosen field.

Listing 4.12: Mutating operands

```
1  fun mutate(
2    operands: Operands, random: &Random, temperature: Int
3  ): Term {
4    var field_to_change = random.next_int(0..2)
5    Operands {
6      left =
7        if field_to_change == 0 then
8          operands.left.mutate(random, temperature)
9        else
10         operands.left,
11     right =
12       if field_to_change == 1 then
13         operands.right.mutate(random, temperature)
14       else
15         operands.right,
16   }
17 }
```

Listing 4.13 shows the fuzzer mutates terms. Lines 2 and 3 choose a new enum variant in 50 % of cases. Otherwise, lines 5 – 10 keep the enum variant but mutate the payload.

Listing 4.13: Mutating terms

```
1  fun mutate(term: Term, random: &Random, temperature: Int): Term {
2    if random.next_bool() then
3      return static[Term]().generate(random, temperature)
4
5    switch term
6    case number(number) Term.number(number.mutate(random, temperature))
7    case add(ops)       Term.add(ops.mutate(random, temperature))
8    case subtract(ops)  Term.subtract(ops.mutate(random, temperature))
9    case multiply(ops)  Term.multiply(ops.mutate(random, temperature))
10   case divide(ops)    Term.divide(ops.mutate(random, temperature))
11 }
```

Table 4.4 shows how the term $-2 \cdot 4 \cdot -9 \cdot -5/(-6 \cdot -2 \cdot -9/-5)$ is mutated for different temperatures. The higher the temperature, the more the mutated term differs from the original one. This allows the fuzzer to widen or narrow the search space of new inputs.

temperature	term
0	$-2 \cdot 4 \cdot -9 \cdot -5/(9 \cdot -4)$
10	6
20	$-2 \cdot 4 \cdot -9 \cdot -5/(-10 \cdot (5-7))$
30	$(1-5) \cdot 5$
40	$-5 \cdot -3 \cdot -3/4$
50	$-5 - -6 \cdot -2 \cdot -9/-5$
60	$-2/-9$
70	$((4--6)/8 + -4 \cdot 3 \cdot -6)/(-6 \cdot -2 \cdot -9/-5)$
80	$(8 \cdot (-3+7) + -10/-1 - 3/1)/0$
90	$-2 \cdot 4 \cdot -9 \cdot -5/((0+3-(-1-6)+(-10-4)/(-7/-2)) \cdot (0/-5 + -10 \cdot -5) \cdot -3)$
100	$-2 \cdot 4 \cdot -3947964/(-6 \cdot -2 \cdot -9/-5)$
200	$-9/5 - (3-0) + (7-9)/(2-7) + -8 + -7 + (1 \cdot 8 - 4/2 - 7) \cdot (9 + -8)/-10 \cdot (3 - -5)/4$
300	-682685205777880502
400	$((4/-7 - 10/-5 + 6 + 1) \cdot 13425180 - 0/1/(1/-4/1/(10 \cdot (7 + 9)) \cdot -1))/(-6 \cdot -2 \cdot -9/-5)$
500	$-2 \cdot 4 \cdot -9 \cdot -5/(111987856453730220 + -63156741107 \cdot -6/8 \cdot -1/9 \cdot -10/((0 + 0 - (-5-9))/((-6+8) \cdot -3)))$

Table 4.4: Mutations of $-2 \cdot 4 \cdot -9 \cdot -5/(-6 \cdot -2 \cdot -9/-5)$ with different temperatures

4.3.3 Tracking the Coverage

Greybox fuzzers are so effective because they receive feedback about the coverage that an input achieves. So, we need a way to get the coverage when running a Martinaise function.

Existing fuzzers like AFL *instrument* binaries to track the code coverage: They change the instructions so that each conditional jump also updates a global bitset. After running some code, this bitset automatically indicates which branches were chosen. State-of-the-art fuzzers collect not just which branches were chosen, but also the number of iterations for each loop.

As the compiler pipeline from section 4.2 shows, Martinaise code is translated into Soil byte code, which is then executed. Similar to AFL, we instrument the byte code to track the coverage. As a proof-of-concept, our implementation only tracks the branch coverage of the fuzzed function.

The only way that data can influence the control flow is with `cjump` instructions and `panic`s. To track the code coverage, we remember which `cjump` instructions were taken. To do so, the program initially reserves a bitset storing two flags (taken and not taken) for each conditional jump, as seen in fig. 4.7.

These two flags are represented as two bytes, either zero or one. This results in four possible constellations:

00 The conditional jump was not reached.
01 The conditional jump was reached and not taken.
10 The conditional jump was reached and taken.
11 The conditional jump was reached multiple times. At one point, it was taken, at another not.

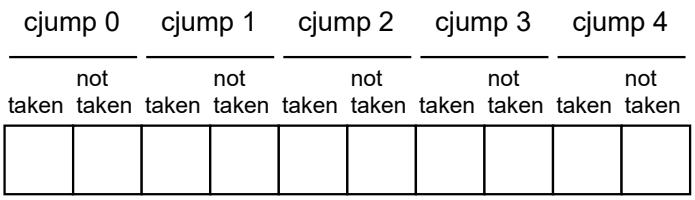


Figure 4.7: The bitset used for coverage tracking

Before each conditional jump, we insert additional instructions that update this bitset. Usually, you want to jump as the result of comparing values (such as two numbers or enum variants). This is done by executing the comparison instruction `cmp`, which sets the status register to the difference of two values. Then, a test instruction like `isequal`, `isgreater`, etc. reads that status register value and sets it to 0 or 1. Finally, the `cjump` instruction jumps if the status register is not zero.

Figure 4.8 shows the byte code instructions inserted before conditional jumps: Because the inserted instructions need registers for temporary values, they push the values of two registers to the stack (lines 3 and 4) and pop them

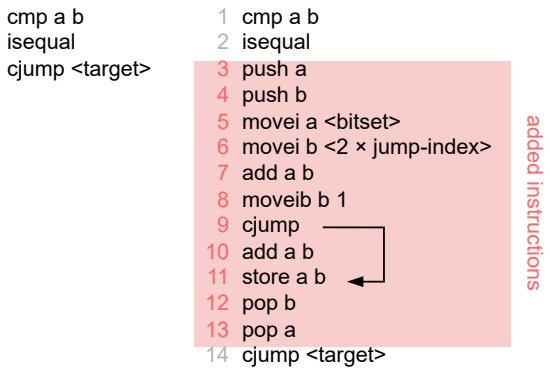


Figure 4.8: Instrumentation of Soil byte code

back into the registers at the end (lines 12 and 13). The instructions in between (lines 5 through 11) calculate and set a byte of the coverage bitset.

If the original `cjump` triggers, then the `cjump` in line 9 also jumps because it depends on the same status register. Line 10 is skipped and the `store` instruction on line 11 sets the flag at address bitset + 2 · jump-index to 1.

If the original `cjump` does not trigger, then the `cjump` in line 9 does not jump. Line 10 executes and line 11 sets the flag at address bitset + 2 · jump-index + 1.

After running a function with an input, the coverage map automatically represents the input's branch coverage.

Syscalls with side effects also get replaced during instrumentation. Syscalls that output data (such as printing, or writing a file) just get omitted. If a function performs a syscall that inputs data (such as reading a line from the standard input or a file), the function is not fuzzable.

4.3.4 Assessing the Complexity of Inputs

Coverage information allows us to gradually evolve the input and explore the code of a function. Unlike with traditional fuzzing, our goal is not just to find crashes, but also to highlight useful examples. This makes it more important to find helpful, concise inputs as visual space in the editor is limited.

If we want to display examples highlighting the entire breadth of a function's behavior, we need to select only a handful of inputs to show. In particular, when the fuzzer finds two inputs that achieve the same coverage, it needs to decide which input is more helpful.

We assume that showing small inputs is a good idea. We can display them more comfortably in the editor. Mentally calculating the correct result is faster for smaller inputs, so showing small inputs allows you to validate a function's behavior more easily. Small inputs also highlight edge cases more obviously – there is less noise around the part of the input that induces the special behavior.

Traditional fuzzers like AFL have a separate minimization step that reduces the input (bytes) to a minimal version. Property-based testing frameworks also have a shrinking phase as described in section 2.3.3. We will do something similar: The fuzzer uses a `fuzzing_complexity` function to assess the complexity of inputs, shown in listing 4.14.

Listing 4.14: The complexity evaluation function

```
fallback fun fuzzing_complexity[T](value: T): Float { ... }
```

Similar to the other compiler-implemented functions described in this chapter, you can implement it for a type to customize the behavior. For example, `Floats` report a lower complexity if they do not have any decimal digits – meaning 2.0 is considered less complex than 1.89. Listing 4.15 shows the code implementing this behavior.

Listing 4.15: Judging the complexity of floats

```
fun fuzzing_complexity(float: Float): Float {
  var extra =
    if float.to_int().to_float() == float then 0.0 else 10.0
  float.abs().to_int().log_2().to_float() + extra
}
```

Complexity is not a precisely defined measure (like the memory used), but a measure of how "beautiful" or "easy to think about" an input is.

4.3.5 Fuzzing

Given the functionality for generating inputs, judging their complexity, tracking coverage, and mutating them, we can now implement Greybox fuzzing. We have demands that are somewhat different from those of existing Greybox fuzzers: We want to find crashes, but also interesting inputs. The editor should show representative examples as soon as possible.

Our process is very similar to how AFL works: [45] The fuzzer maintains a list of inputs and a cursor through that list. The item at the cursor position is mutated repeatedly. Mutated versions that achieve new coverage are appended to the list. This process explores the function behavior over time. Because we immediately want to show hints in the editor, we interleave the process with minimizing inputs. In each step, our fuzzer does the following:

- **If the cursor is at the end of the list,** the fuzzer generates a new input and tests it.
  - **If it achieves new coverage**, the fuzzer appends it to the list.
  - **Otherwise,** the fuzzer discards it.
- **If the cursor is not at the end of the list,** the fuzzer mutates the value at the cursor position and tests it.
  - **If the original and mutated values achieve the same coverage,** the fuzzer compares their complexity.
    - **If the mutated value is less complex,** it replaces the item in the list.
    - **Otherwise,** it is discarded.
  - **If the original and the mutated value achieve different coverage,** the fuzzer checks if the mutated one achieves new overall coverage.
    - **If it achieves new coverage,** the fuzzer appends it to the list.
    - **Otherwise,** the fuzzer discards it.

If item mutations do not result in new coverage or smaller values for a fixed number of steps, the cursor advances to the next item.

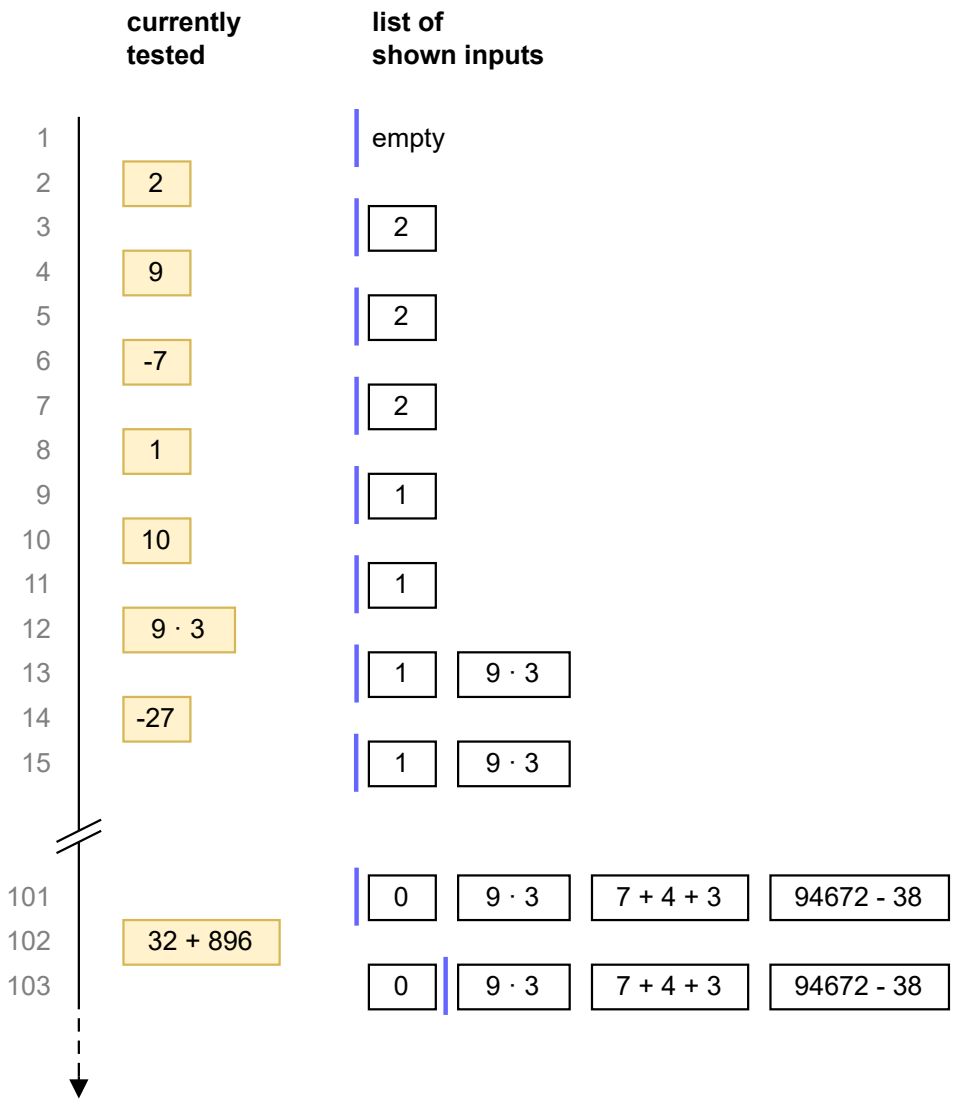


Figure 4.9: The process of fuzzing

Figure 4.9 shows the process of fuzzing the `eval` function, which is tested with the yellow values on the left. The right side shows the list of examples displayed in the editor and a cursor that steps through this list (blue). These examples are also displayed in the editor.

Initially, the list is empty (line 1). Because the cursor is at the end, the fuzzer generates a new term, 2 (line 2). As it achieves new coverage, it is immediately shown in the editor (line 3). Next, the term 2 is minimized (line 4 following). The fuzzer tests mutations like 9 (line 4), −7 (line 6), etc. The term 1 (line 8) achieves the same coverage as 2 while being less complex, hence it replaces

the 2 (line 9). Mutated terms that achieve new coverage (line 12) are appended to the list (line 13).

After the break (line 101), the first term has been minimized to 0. Because the fuzzer did not achieve new coverage for some time, it advances the cursor to the next item (line 103). Next, it will minimize the term $9 \cdot 3$ and then the rest of the terms in the list, which is not shown in the figure.

4.4 Editor Integration

We want to show hints based on fuzzing. As a proof of concept, we will implement that functionality for Visual Studio Code[10] (VS Code), a popular code editor developed by Microsoft, shown in fig. 4.10.

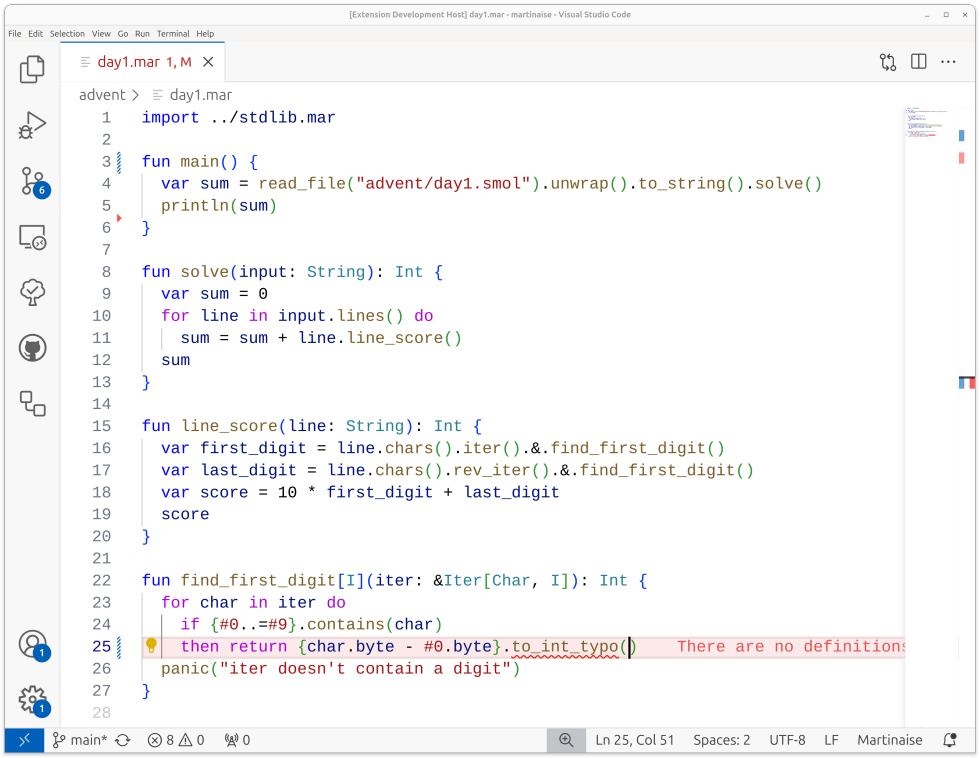


Figure 4.10: Martinaise code in VS Code

Since 2015, Microsoft has developed the Language Server Protocol (LSP) [23], a standard for communication between code editors and language-specific tooling for features like auto-completion or go-to-definition. The LSP lowers the bar for new code editors to re-use existing language tooling and for new

[10] https://code.visualstudio.com/

programming languages to offer tooling in many editors. Figure 4.11 shows how the LSP mediates between editors and language tooling.

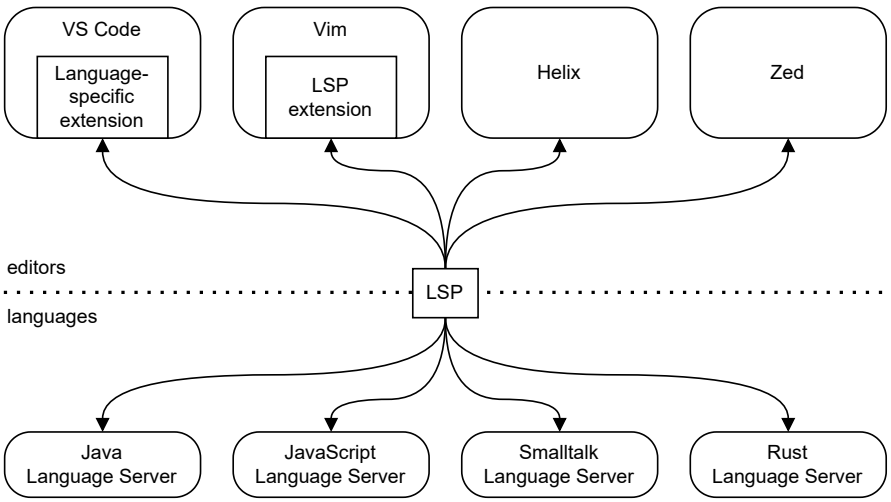


Figure 4.11: The Language Server Protocol

VS Code does not support the LSP directly (unlike some newer editors like Helix[11] or Zed[12]). Instead, extensions for VS Code are developed in JavaScript[13] and use the JavaScript VS Code API[14] to provide functionality. Microsoft does offer a JavaScript library that translates LSP requests into VS Code API calls. This separation allows Microsoft to offer VS-Code-specific APIs while still supporting the LSP.

To understand how we could show insights gained from fuzzing in the editor, we will briefly look at some LSP features[15] that could be used to show hints directly in the editor: inline values, inlay hints, and diagnostics. The screenshot in fig. 4.12 shows all three of them in VS Code. Other editors might display information they get through the LSP differently, but the information is the same.

```
fun test() {
  var foo: Int = 5 + 3   foo = 8
  var bar = hello()    There are no defintions named "hello".⏎
}
```

Figure 4.12: Inline values, inlay hints, and diagnostics in VS Code.

[11] https://helix-editor.com/
[12] https://www.zed.dev
[13] https://code.visualstudio.com/api/get-started/extension-anatomy
[14] https://code.visualstudio.com/api/references/vscode-api
[15] All my mentions of the LSP refer to the current version as of writing, LSP 3.17.

Inline Values On the right, there is a yellow inline value. These kinds of hints are only shown when a debugging session is active, so this feature does not quite match our use case – we want to provide such hints without extra effort from the developer. Also, we want to show a variety of inputs for function signatures rather than a single value for an expression.

Inlay Hints The faded-out type annotation `Int` is not part of the source code but appears automatically. When typing or moving the cursor, you skip over the annotation. We can potentially show inlay hints containing examples after function signatures. However, inlay hints are typically designed to integrate visually with the rest of the code. Our examples should not look like they are part of the source code.

Diagnostics Compilers and linters can report diagnostics. There are different severity levels: errors, warnings, information, and hints. The first three represent problems and VS Code offers a separate "Diagnostics" tab that aggregates these reports. Hints only appear as three dots in the editor, requiring hovering to show additional information (see fig. 4.13).

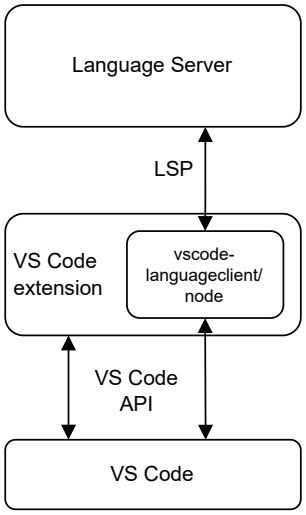
Figure 4.13: Diagnostic hints in VS Code

None of these existing LSP features quite match our use case. We want to show hints by default, without requiring a specific editor state (like an active debugging session). We want to show hints prominently next to the source code, without requiring hovering or making it look like examples are part of the code. We want to show hints without adding items to the diagnostics tab or requiring hovering.

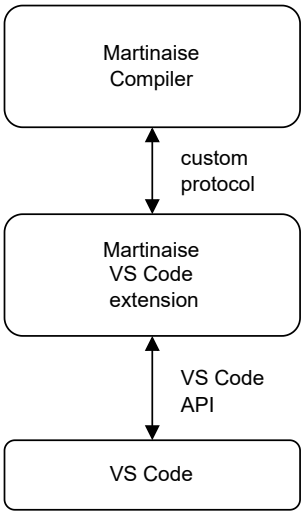
Fundamentally, the LSP is designed for common tooling needs. Its goal is to be presentation-agnostic, so all messages work on a semantic level – for example, it communicates errors and warnings rather than text decorations. Because fuzzing-driven examples are a new use case, the LSP does not explicitly support them.

As shown in fig. 4.14, VS Code provides a more general extension API with more features than the ones available via the LSP. Usually, VS Code language extensions spawn a language server process and then delegate to the `vscode-`

`languageclient/node` package[16] to communicate with the language server and translate messages into VS-Code-specific actions (see fig. 4.14a). We use the VS Code API directly for our proof-of-concept (see fig. 4.14b). This ties our implementation to VS Code but gives us more customization options for displaying examples.



(a) Typical VS Code language extension



(b) Martinaise VS Code extension

Figure 4.14: Extension architectures

The existing VS Code extension for Martinaise only supports syntax highlighting and reporting errors. It starts a process that provides the language tooling and communicates with it over standard input and output. However, it does not use the LSP, but a simpler, custom protocol shown in listing 4.16. Like the LSP, the protocol uses JSON. However, it only supports a few message types: reading files and reporting errors.

When the compiler wants to read a file, it asks for the content by sending a message on the standard output. The extension answers with the file content on standard input. This allows the compiler to analyze unsaved file contents. Finally, the compiler answers with errors that are shown in the editor.

Listing 4.16: Communication between extension and compiler

```
<- {"type": "read_file", "path": "path/to/example.mar"}
-> {"type": "read_file", "success": true, "content": "<file-content>"}
<- {"type": "read_file", "path": "path/to/stdlib/stdlib.mar"}
-> {"type": "read_file", "success": true, "content": "<file-content>"}
...
<- {
     "type": "error",
     "title": "There are no definitions named \"sum_typo\".",
```

[16] https://www.npmjs.com/package/vscode-languageclient

```
    "description": "",
    "src": {
      "start": {"line": 13, "column": 7},
      "end": {"line": 13, "column": 7},
      "file": "path/to/example.mar"
    },
    "context": []
  }
```

Now, we extend the extension usage to report fuzzing results. First, we adapt the compiler analysis to report function definitions. Listing 4.17 shows the messages for functions.

Listing 4.17: Functions reported by the compiler

```
<- {
    "type": "function",
    "signature": "average(List[Int])",
    "src": {
      "start": {"column": 4, "line": 5},
      "end": {"column": 4, "line": 5},
      "file": "path/to/example.mar"
    },
    "fuzzable": true
  }
<- {"type": "function", "signature": "foo(Int)", "src": ..., "fuzzable": true}
<- {"type": "function", "signature": "bar(Int)", "src": ..., "fuzzable": true}
...
```

Using the `vs.window.onDidChangeTextEditorVisibleRanges` API,[17] the extension registers a callback when it starts up. Every time the visible ranges of files change, the extension is notified, checks which functions are visible, and spawns fuzzers for these functions.

These fuzzers report their results as example messages, as shown in listing 4.18. As the fuzzing progresses, every set of example calls replaces the previous example calls for the function. This way, the fuzzer continually provides progress reports.

Listing 4.18: Fuzzer reporting example calls

```
<- {
    "type": "example_calls",
    "fun_start_line": 5,
    "fun_signature": "average(List[Int])",
    "fun_name": "average",
    "calls": [
      {
        "inputs": ["[0]"],
        "result": {"status": "returned", "value": "0"}
      },
      {
        "inputs": ["[]"],
        "result": {"status": "panicked", "message": "Can't divide by zero."}
      }
    ]
  }
...
```

[17] https://vscode-api.js.org/modules/vscode.window.html#onDidChangeTextEditorVisibleRanges

VS Code maps LSP results such as inline values, inlay hints, and diagnostics to `TextEditorDecorations`. These decorations allow you to style parts of the editor's text, including custom colors, fonts, borders, margins, and text before and after the decoration. All decorations have a type that determines their style. For our fuzzing examples, we create a new type when the extension starts up, shown in listing 4.19.

Listing 4.19: Creating a new text editor decoration type

```
const exampleDecoration = vs.createTextEditorDecorationType({
  after: {
    color: new vs.ThemeColor("martinaise.example.foreground"),
    backgroundColor: new vs.ThemeColor("martinaise.example.background"),
    margin: "0 0 0 16px",
  },
  rangeBehavior: vs.DecorationRangeBehavior.ClosedOpen,
});
```

The `martinaise.example.*` strings are references to colors that can be customized using themes or settings. The `margin` adds space on the left of the fuzzing examples. The `rangeBehavior` ensures that the cursor can never appear on the right side of the example. When a new set of fuzzing examples is reported, the extension maps them to decorations and updates them in the editor, shown in fig. 4.15.

```
fun average(list: List[Int]): Int {  [] panics  [0] -> 0
  list.sum() / list.len
}
```

Figure 4.15: Examples for the average function

4.5 Conclusion

We described how we extended the Martinaise compiler to support fuzzing. This impacted many parts of the compiler pipeline and tooling, including implementing type-specific code for generating and mutating values, judging their complexity, instrumenting byte code, and adapting the editor tooling. In total, the compiler changed from 4700 lines to 6500 lines.

In the standard library, only fundamental types like `Int` and `Float` and container types like `List` and `Map` had to be adapted to enable type-specific generation and mutation.

5 Evaluation

To evaluate our tool, we first give an overview of its behavior. We evaluate the generated examples' quality, for which code fuzzing works well, the performance, and how examples are displayed. Finally, we compare our tool to existing development tools.

5.1 Overview

In the introduction, we described a function that calculates the average. We used this function as a motivating example of how fuzzing could give immediate feedback while writing code. Figure 5.1 shows the feedback our tool provides for such a function implementation. The red and green hints at the end of the first line automatically appear when our extension is activated. The red hint informs us that the `average` function crashes if you pass in an empty list. The green hint tells us that the `average` of a list containing a zero is zero.

```
fun average(list: List[Int]): Int {  [] panics  [0] -> 0
  list.sum() / list.len
}
```

Figure 5.1: Examples for the average function

Our tool tries to find example inputs that cover the entire function. Figure 5.2 shows an evaluation function for mathematical terms, first introduced in section 4.3. Our tool implements code for generating and mutating random terms and produces examples displayed after the function definition. As not all examples fit on the screen, fig. 5.3 presents all the examples you can see when scrolling to the right: Five examples for the five branches inside the function and an additional example that crashes the function.

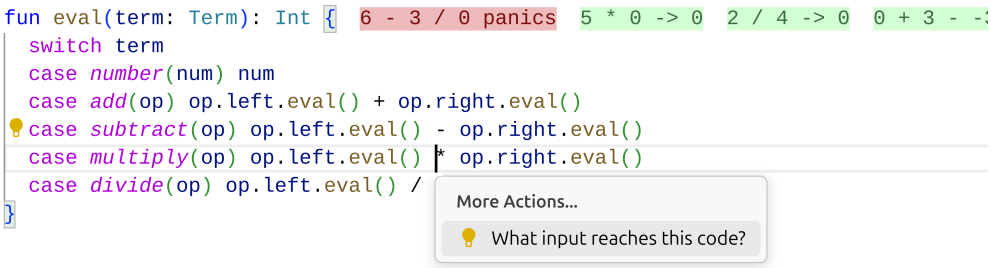
```
fun eval(term: Term): Int {  6 - 3 / 0 panics  5 * 0 -> 0  2 / 4 -> 0  0 + 3 - -3
  switch term
  case number(num) num
  case add(op) op.left.eval() + op.right.eval()
  case subtract(op) op.left.eval() - op.right.eval()
  case multiply(op) op.left.eval() * op.right.eval()
  case divide(op) op.left.eval() / op.right.eval()
}
```

Figure 5.2: Examples for the term evaluator

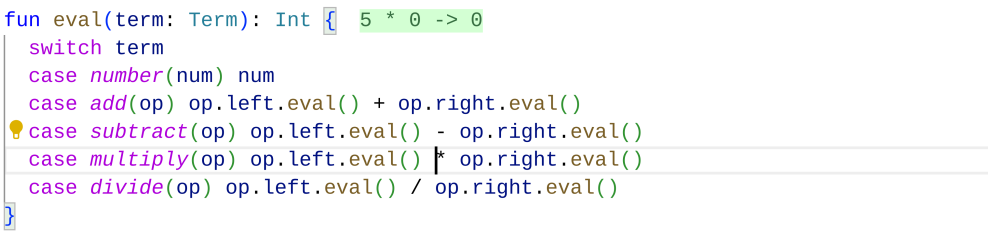
```
6 - 3 / 0 panics   5 * 0 -> 0   2 / 4 -> 0   0 + 3 - -3 -> 6   0 + 0 -> 0   0 -> 0
```

Figure 5.3: All examples for the term evaluator

In VS Code, you can use code actions to perform refactorings. Figure 5.4 shows how to use a code action to filter the examples to ones reaching a particular location. This does not change the fuzzing behavior such as the directed fuzzing described by Böhme et al. [2], it only changes which inputs are displayed.



(a) Code action asking for inputs



(b) Filtered fuzzing examples

Figure 5.4: Asking for examples that reach a given code location

5.2 Quality of Examples

Like other Greybox fuzzers, our tool does not attempt to understand Turing-complete functions. Instead, it only understands when a new control flow is discovered. This is enough to explore functions with straightforward control flow. Figure 5.5 shows a function where the fuzzer successfully gives one example for each way that the function can return (fig. 5.6).

However, achieving full coverage on *any* function is impossible. Figure 5.7 shows a function that only returns an integer if the SHA-256 hash of the input matches an expected value. While in theory, the fuzzer could generate and show an example input that reaches the conditional branch, doing so in a reasonable time frame requires breaking SHA-256.

```
fun is_valid_email(string: String): Bool {  "@" -> false  "" -> false  "bc]abaab(
  if not(string.contains("@")) then
    return false
  var parts = string.split("@")
  var name = parts.get(0)
  var host = parts.get(1)

  if name.len < 8 then
    return false
  if not(name.is_alphanumeric()) then
    return false

  true
}
```

Figure 5.5: An email address checker with more complicated control flow

```
"@" -> false  "" -> false  "bc]abaab@" -> false  "abaacaca@" -> true
```

Figure 5.6: All hints for the email address checker

```
var hash = "5E884898DA28047151D0E56F8DC6292773603D0D6AABBDD62A11EF721D1542D8"

fun secret_number(password: String): Maybe[Int] {  "" -> none
  if sha_256(password) == hash then {
    some(42)
  } else {
    none[Int]()
  }
}
```

Figure 5.7: Code that is impossible to reach efficiently by fuzzing

In practice, our fuzzer already struggles with simpler code. For example, our fuzzer does not reach the inner code in listing 5.1 because it only tracks the branch coverage, not the number of loop iterations. State-of-the-art fuzzers can fuzz this code because inputs where the prefix is a different prefix of `"https://"` (such as `"hello"` and `"httttttt"`) report different coverage.

Listing 5.1: Checking a URL prefix

```
if url.starts_with("https://") then {
  ...
}
```

Apart from finding inputs that crash functions, the examples discovered by our tool can also be used to understand how a function behaves without looking at its implementation. However, while code coverage and input complexity generally correlate with the helpfulness of examples, we recognize that it is not a perfect proxy. In everyday use, we identify two major limitations when using our tool to understand existing code.

First, minimizing examples can be counterproductive. In fig. 5.1, our tool shows that `average([0]) = 0`. While a list containing a single number is enough to exercise all code of the function (achieving full code coverage), it fails to give the developer confidence that the implementation is correct. `[1, 2, 3]` would be a better example.

Second, the examples our tool shows are often unnatural and inhuman. In fig. 5.8, our tool greets someone with an empty name. While this is the expected behavior of fuzzing, in our experience, such unnatural examples introduce some mental overhead compared to more natural examples. This problem is difficult to address in a general way because the perceived complexity of values is context-dependent: Even if `0` is perceived as less complex than `3` in isolation, most humans perceive `[1, 2, 3, 4]` as less complex than `[1, 2, 0, 4]`.

```
fun greet(name: String): String {  "" -> "Hello, !"
  "Hello, {name}!"
}
```

Figure 5.8: Greeting an empty string

5.3 Adapting Code for Fuzzing

The previous sections showed how fuzzing provides examples without any extra effort from the developer. However, sometimes there are better results if you customize how examples are validated, generated, and mutated.

Consider a tile-based game that stores a level and the player's position, shown in fig. 5.9. The player's position should always be inside the bounds of the level, but the fuzzer does not know about that. Thus, it produces invalid games as examples for the `move_right` function in fig. 5.9.

Figure 5.10 visualizes these examples. The first example has a level size of 0×0, so the player is out of bounds. The second example is a valid game where the player can move right. In the third example, the movement is blocked by a wall, but the player is also out of bounds.

You can get better examples by customizing how examples are generated and mutated. The appendix contains functions for `Game` that ensure that the level size is at least 1×1 and the player is spawned within its bounds. If you customize the functionality, the fuzzer provides the examples seen in fig. 5.11. These examples execute the same code paths as the examples from fig. 5.10, but all games are valid.

In practice, fuzzing works better for some functions than others. We observe that examples are most helpful for mathematical functions that map an input to an output. Our tool is less useful for functions that mutate the input or mainly produce side effects.

```
struct Game {
  level: Matrix[Tile],
  player: Player,
}

enum Tile { wall, free }

struct Player {
  position: Point,
}

fun move_right(game: &Game) {  &Game { level = [], player = Player { position =
  var new_position = game.player.position + (1 @ 0)
  if game.level.get(new_position) is free then
    game.player.position = new_position
}
```

Figure 5.9: A game

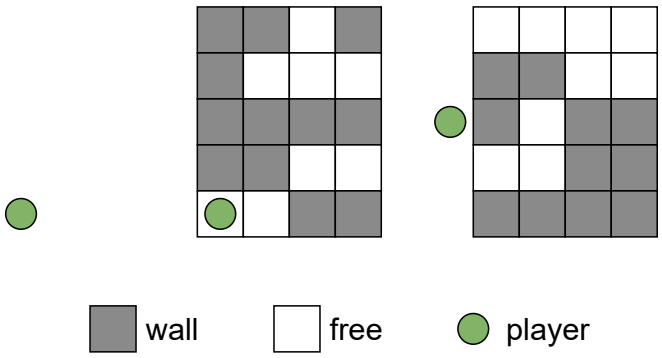


Figure 5.10: Example games for moving right

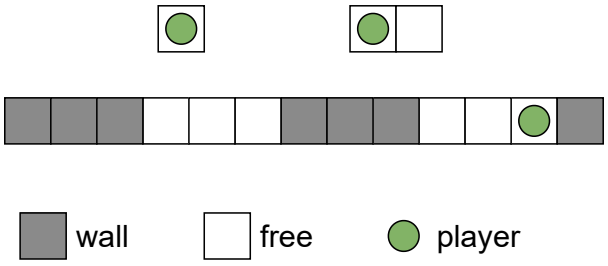


Figure 5.11: Better example games for moving right

Some functions have implicit assumptions not inherent to types. For example, "internal" helper functions often require that the input is in a particular state, leading to many "false positive" crashes that never occur for the actual usage of these functions. In contrast, "exposed" functions from libraries that try to handle every input gracefully are great candidates for fuzzing.

5.4 Performance

Fuzzing takes time. Compared to traditional fuzzing, an advantage of fuzzing on the granularity of functions as opposed to entire programs is that the fuzzer does not have to reach potentially crashing code indirectly through the `main` function. For example, if the entry part of the code is guarded with code that is difficult to pass such as the SHA-256 code from section 5.2, our tool can still fuzz internally used functions in isolation.

A unique property of fuzzing compared to static tools like type checkers or linters is that it provides better results over time. Table 5.1 shows a timeline of how examples for the email address checking function from fig. 5.5 change over time. Every line shows a snapshot; it shows when the examples appear, the number of function tests that have happened up to that point, and the examples themselves.

time (ms)	runs	hints in the editor
0	0	
989	1	`"" -> false`
991	28	`"" -> false` `"&EC]vFI>mWmFhh}@qwt~8&$~%kA" -> false`
1002	82	`"" -> false` `"&EC]vFI>mWmFhh}@qwt~8&$~%kA" -> false` `"IFj@(`~B]G^6l[k`
2155	1140	`"" -> false` `"lC]vFbmh@qta8Myme" -> false` `"IFj@(`~B]G^6l[kH[]H^aS:2?`
2177	1334	`"" -> false` `"lCKvcafh@h" -> true` `"lC]vcafh@h" -> false` `"IFj@(`~B]G`
2592	2409	`"" -> false` `"lCKvcafh@h" -> true` `"bd]ccaab@" -> false` `"IFj@(`~B]G^`
3450	3794	`"" -> false` `"bd]abaab@" -> false` `"lCKvcafh@h" -> true` `"IFj@(`~B]G^`
3664	4236	`"" -> false` `"lCKvcafh@h" -> true` `"bc]abaab@" -> false` `"IFj@(`~B]G^`
4906	5272	`"@" -> false` `"" -> false` `"lCKvcafh@h" -> true` `"bc]abaab@" -> false`
7183	9367	`"@" -> false` `"" -> false` `"bc]abaab@" -> false` `"abaacaca@" -> true`

Table 5.1: Better examples over time for an email address checking function

The first row of the table indicates that no examples are shown while the function compiles. After 989 ms, an empty input appears as an example. Over the next 13 ms, our tool discovers random-looking examples that cover all code. Then, it shrinks them – over time, the examples get shorter and less complex. Because of this continuous refinement, the feedback feels reasonably fast in practice, even if it takes some time for the fuzzer to explore the entire code.

The table shows the average time of ten runs on a desktop computer.[1] Because the fuzzing was performed on the same randomness seed, the number

[1] The computer has an AMD Ryzen™ 7 5800X × 16 and 32 GiB of RAM. It runs Ubuntu 24.04.1 LTS.

of runs and the fuzzing hints are deterministic. The table also only shows a subset of all updates – in total, the fuzzer produces many more intermediate versions, most of which differ only by a single character from the previous version.

Figure 5.12 visualizes these updates in a different format: Each dot on the timeline corresponds to a new set of examples in the editor. The fuzzer quickly finds minimal examples for the `average` function. For the `eval` function, it finds inputs, but minimizing them takes time. Each dot in the `eval` timeline is really a tight clustering of dots: When the fuzzer moves on to the next example, it very quickly minimizes that term and spends the rest of the time trying out other mutations before moving on to the next item. The `is_valid_email` function is the most visually interesting: Because the function is more complex than the others, the fuzzer produces many more intermediary inputs. Generally, a dense burst of new examples after a longer period with none indicates that the fuzzer moved on to the next example.

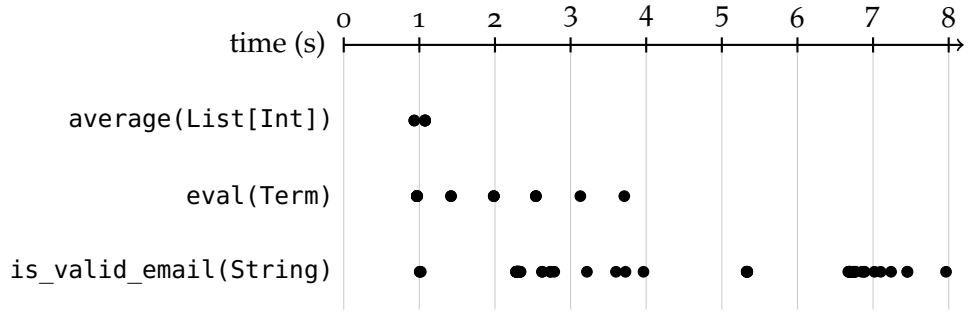


Figure 5.12: Fuzzing updates over time

5.5 Displaying Examples

Apart from the quality of examples, we also want to look at how and where examples are displayed. Currently, our tool formats examples as text and shows them next to the function signature.

Examples with unique coverage profiles would be more helpful next to the covered code. In the term evaluator from fig. 5.2, showing examples next to the branch they execute would make the relationship between the example and covered code obvious.

For many values, there are more appropriate representations than formatted text. For example, games like the one from fig. 5.9 are most naturally rendered visually. It may make sense to expand our tool to other editors that do not have VS Code's constraints, such as the ones discussed by Rauch et al. [28]

While our tool produces examples in stable order, VS Code guarantees no order of hints applying to the same code position. As seen in table 5.1, the hints sometimes switch places, which can be visually jarring.

Functions where the main action is a mutation of the input or a side effect such as printing return `Nothing`, a type with no information. The examples in the editor do not show you the function's effect. In Figure 5.13, our tool does not tell you the printed message.

```
fun greet(user: User) {  User { first_name = "", last_name = "" } -> nothing
  println("Hello, {user.first_name} {user.last_name}!")
}
```

Figure 5.13: Greeting produces no useful value

5.6 Comparison With Existing Tools

In chapter 2, we described existing tools for making code more robust. Here, we compare our prototype with these tools:

Static type systems, Linters & Symbol Execution analyze the code statically. Our tool instead explores the code's dynamic behavior. All reported examples and crashes show concrete values rather than generic types, making it easier to start a debugging session.

Unit Testing only tests manually created examples. Our tool runs without extra effort from the developer. Rather than verifying that the code behaves as expected for a few values, our tool actively explores and finds edge cases in the code.

Property-Based Testing requires writing tests and thinking of properties. They are usually run manually and only focus on finding inputs that violate properties, not generally useful examples.

Fuzzing works on entire programs or requires writing test harnesses. Our tool tests code with structured data rather than unstructured bytes. The behavior of our tool can be customized directly in the code by writing custom `generate` functions.

Babylonian Programming focuses on exploring the code's behavior for a single example or a small number of examples. Our tool focuses on the step before that – coming up with exemplary input.

Large Language Models can generate usage examples [20] for code and find security risks. However, their analysis is not grounded in the execution of the code, but in an approximation of the code semantics within the model.

5.7 Conclusion

We achieved the goals set when describing our approach in chapter 3:

Fuzz by Default Our tool can fuzz functions in isolation without requiring writing custom test harnesses or generators for types.

Minimize distance between code and examples Our tool displays examples directly next to the function signature. It shows examples as quickly as possible and refines them over time.

Showcase function behavior Our tool tries to build up a set of examples that achieve full code coverage. It highlights crashes.

In the introduction, we also asked two research questions: Can fuzzing highlight edge cases while writing code? Can fuzzing help understand code with example inputs? We can confirm that our tool highlights edge cases that are crashing. However, the examples it produces are often too minimal to help understand the code. Being able to request edge cases reaching a particular location is helpful, but unnatural examples introduce more mental overhead than manually curated ones.

In our personal experience, our tool enables interactions with code that are impossible to achieve with other tools in a similarly effortless way. The fuzzing quality and performance are acceptable for daily use – the main limitation is that examples are only displayed as formatted text. Interactive examples or using examples as the foundation for Babylonian Programming techniques or debugging sessions would be the most straightforward way to make this tool more useful.

6 Conclusion and Future Work

We showed that automatically fuzzing functions can be a new source of information that the editor can show developers. Now we want to highlight areas for improvements that would make our tool more useful:

Better performance Implementing fuzzing for a real-world language with an optimizing compiler will improve the performance. As the provided examples update over time, throughput is not the main concern of the fuzzer. Instead, the most noticeable performance limit is the initial delay during the compilation of the fuzzed function. Especially just-in-time-compiled languages or languages with an incremental compiler could minimize this duration. The same incremental caching could be used to only fuzz functions if they change.

Better fuzzing Our tool only tracks the branch coverage in the fuzzed function and sometimes cannot reach code deep inside a function. State-of-the-art fuzzers collect coverage for the entire codebase and track additional information such as the number of iterations in a loop; this makes it possible to explore more code. Various other aspects of fuzzing research could also be adapted. For example, directed fuzzing [2] prioritizes inputs that come close to a target location; this would be useful when asking our tool for inputs that reach a particular location.

Fuzz with more context Currently, functions are fuzzed in complete isolation. If a function calls another function, the arguments could be saved as an example for the called function. Rather than fuzzing functions individually or indirectly through the `main` function, we could only record examples that passed through a fixed level of other functions. These indirectly found examples are potentially more natural because the code in outer functions restricts which values reach inner functions.

More human examples We discovered that the usual objective of fuzzing – finding a small input that breaks the code – is insufficient for finding helpful, representative inputs for a piece of code. Recent developments in LLMs make it practical to generate natural-looking examples automatically, as shown by Mattis et al. [20] Combining LLMs and fuzzing might lead to helpful and

correct examples grounded in actual execution. Generally, we recommend conducting user studies to understand which inputs are perceived as helpful.

Relate examples to coverage The link between examples and achieved coverage could be clearer. This could be achieved by moving examples to unique code locations that they reach. Interactive solutions, such as greying out unreached code when clicking on an example, would clear up the relationship between an example and the reached code.

Prioritize examples Currently, the order of examples is chosen ad-hoc by VS Code. An intentional ordering of examples might first show an example covering a happy path and examples covering crashes, and only then display examples that reach niche code locations. Alternatively, examples reaching the cursor position could be considered more relevant, leading to a reordering of examples as you move the cursor through the code.

Interactive examples Currently, examples are displayed as text that you cannot interact with. Examples should be the starting point for visualizations, debugging sessions, creating unit tests, and other tools.

6.1 Generalizing to Other Languages

We chose Martinaise because it is the best-case scenario for fuzzing: It has statically known types, most of them concrete. There are no interfaces. There are no first-class functions.

This allowed us to investigate using fuzzing as an always-on editor tool without focusing on more advanced challenges. Now we want to share thoughts on how our prototype could be adapted when building a similar tool for real-world, production-ready languages.

6.1.1 Dynamically Typed Languages

In dynamically typed languages, the fuzzer gets no type information. As a consequence, generating valid inputs is a lot more difficult. Because most of the generated inputs will likely cause the fuzzed function to crash, the fuzzer has to rely entirely on coverage feedback to guide it toward interesting inputs.

We should note that the potential usefulness of fuzzing in these languages is pretty high. Because functions have no type annotations, having a few examples of valid (read: non-crashing) calls is extremely valuable when trying

to use code correctly without reading the implementation. Take these function signatures in JavaScript:

Listing 6.1: Function signatures in JavaScript

```
function average(numbers)
function average(grades)
```

Based on the parameter name, you might reasonably expect the first function to accept a list of numbers. The second one seems to do something more complicated such as calculating the average of a grading of an exam. Seeing a concrete example call, like `average({15: 5, 14: 6, 13: 4, 11: 6, 10: 8, 9: 4})` is very helpful.

On the other hand, crashes in dynamically typed languages are not as meaningful anymore. Take this average function, written in JavaScript:

Listing 6.2: An average function in JavaScript

```
function average(list) {
  return sum(list) / list.length;
}
```

This function does not handle empty lists – those crash the function. This might be a bug, but `average("foo")` also crashes. From the point of view of the fuzzer, none of these crashes is more special than the other.

In dynamically typed languages, fuzzing can only be used as a descriptive tool that automatically finds inputs rather than as a correctness checker that finds bugs.

6.1.2 Languages With Pure Functions

Mathematical functions cannot modify their inputs or have side effects (like printing something). Instead, they are only a mapping between one set of values to another set. Some programming languages model their code similarly: They favor immutable data structures and functions without side effects (“pure functions”).

In these languages, fuzzing would not need to guard against mutations of inputs, or side effects. Communicating the insights from fuzzing would be much easier for these languages – for example, our prototype in Martinaise currently does not check or communicate how a function modifies an input.

6.1.3 Languages With Functions as Values

A common language feature missing from Martinaise is passing functions around as values (“first-class functions”). Adding first-class functions would enable writing higher-order functions. For example, here is pseudo-code that

defines a function mapping a list from one type to another using a provided function:

Listing 6.3: How higher-order functions could look like in Martinaise

```
fun map[A, B](from: List[A], mapper: A -> B): List[B] {
  var to = list[B]()
  for item in from do
    to.&.push(mapper(item))
  to
}
```

To fuzz this higher-order function, the fuzzer needs to generate a function `A -> B`. The representation the fuzzer chooses must fulfill some goals: First, the fuzzer must be able to slightly alter the function's behavior as that is what makes the evolutionary fuzzing process so effective. Second, there should be a way to display functions in an understandable way to the developer. Third, most functions are deterministic – they produce the same return value for the same input. Depending on whether the language supports them, the fuzzer should be able to generate deterministic and non-deterministic functions.

Generate code The fuzzer could generate code for the function. This can be done reliably by generating code in a compiler-internal intermediary representation and then converting that into syntax.

However, as code is Turing-complete, the relation between input and output can be difficult to reason about – a small change to the code can result in big changes in its behavior and the code may run infinitely. On the other hand, this approach makes it trivial to display functions. The generated code inherits the determinism properties of the underlying programming language.

Generate a random seed Rather than generating full-blown code, the fuzzer could generate a random seed. When the function is called, it can use the entropy from the input and the seed to generate a new output on the fly.

Similar to generating code, the function is unpredictable for the fuzzer. Displaying these functions is a challenge. These functions are deterministic.

Generate a list of outputs The fuzzer could generate a function that stores a list of outputs. Each time it is called, it returns the next output. Once all outputs are used up, it could crash or start over with the first output.

Here, the fuzzer can locally reason about the function's behavior – by changing one of the return values, it only slightly changes the function's behavior. Displaying these functions is unusual, but possible. Functions are not deterministic.

Generate a map from inputs to outputs Representing a function simply as a map from inputs to outputs mirrors the mathematical definition of functions as relations between two sets. [21]

The fuzzer can locally reason about the behavior. Displaying such a function is as easy as showing a map and probably aligns with how developers think about most functions. Finally, the function is deterministic.

Technique	Small changes	Intuitive display	Deterministic
Code	no	yes	depends on language
Random seed	no	no	yes
List of outputs	yes	partial	no
Map	yes	yes	yes

Table 6.1: Comparison of closure representations for fuzzing

Table 6.1 shows the tradeoff between these approaches. For languages where deterministic functions are sufficient, we recommend modeling fuzzer-generated functions with a map from inputs to outputs.

6.1.4 Languages Designed for Fuzzing

Many languages are designed with tooling in mind. You could imagine languages specifically designed so that fuzzing can give accurate hints.

We mentioned that adding assertions throughout the code can improve the quality of fuzzing. We implemented a prototype of a dynamically typed language called Candy[1] where you *have* to put all assumptions in the code. A `needs` keyword states something that should be true at that point in the code. Figure 6.1 shows how the `divide` function is implemented in Candy: It asserts that the inputs are integers and the divisor is not zero.

```
intDivideTruncating dividend divisor :=
  needs (dividend | typeIs Int)
  needs (divisor | typeIs Int)
  needs (divisor | equals 0 | not) "You can't divide by zero."
  ✨.intDivideTruncating dividend divisor
```

Figure 6.1: divide in Candy

Unlike assertions in other languages, a failing `needs` has a more precise meaning than "something is wrong": The caller of the lexically surrounding function is at fault; if you call a function, you are responsible for ensuring its

[1] https://github.com/candy-lang/candy

needs are met. Not fulfilling `needs` is the only way to crash, so every crash can be attributed to a responsible call in the code.

Because of these precise semantics of `needs`, the fuzzer can distinguish between crashes caused by invalid inputs (the fuzzer is at fault) and crashes that are the fault of the function itself. These crashes indicate faulty code and result in an error in the editor.

Figure 6.2 shows the tooling that this enables: All functions are fuzzed. Green hints indicate a successful call. Yellow hints indicate a crashing call, in which the fuzzer was responsible. Red hints indicate a crashing call that is the function's fault. In the example, the average function has `needs` for checking that the input is a list of numbers (lines 5 and 6). The editor highlights that for an empty list, the divide operation fails.

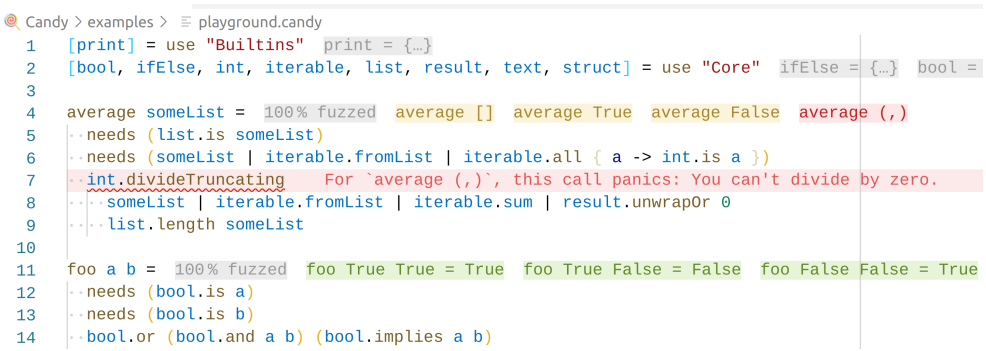


Figure 6.2: average in Candy

However, because Candy has no statically known types, the baseline performance is not great. This problem is compounded by the many `needs`, even in the simple example shown above: The `average` function shown in fig. 6.2 iterates the entire list to ensure it only contains numbers. The `sum` function iterates the same list *again*. Even with aggressive inlining and constant folding, we did not achieve acceptable performance – on common tasks, Candy is about 780 times slower than Python. Our optimizations would optimize out some of the `needs`, but re-checking big data structures at the beginning of functions generally remains an unsolved problem that proved to be prohibitively expensive in practice.

6.2 Conclusion

Testing code with random inputs and observing the coverage is enough to construct interesting inputs. While fuzzing is a well-researched, established technique for finding bugs in security-critical software, using fuzzing as the basis for other development tooling has not been researched. Unlike other tooling such as static type systems or large language models, fuzzing can provide concrete examples grounded in the dynamic behavior of code. This makes it an intriguing basis for Babylonian Programming tools.

Our prototype shows that it is possible to build editor tooling based on fuzzing that can highlight edge cases and give example inputs without extra effort for the developer. While our prototype communicates examples only as text, these examples have the potential to be the basis for visualizations, debugging sessions, unit tests, and other tools that benefit from concrete examples.

Bibliography

[1] Arm. *A-profile A64 Instruction Set Architecture*. Sept. 2024.

[2] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. "Directed Greybox Fuzzing". en. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. Dallas Texas USA: ACM, Oct. 2017, pages 2329–2344. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134020.

[3] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-based Greybox Fuzzing as Markov Chain". en. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. Vienna Austria: ACM, Oct. 2016, pages 1032–1043. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978428.

[4] Peng Chen and Hao Chen. "Angora: Efficient Fuzzing by Principled Search". In: *2018 IEEE Symposium on Security and Privacy (SP)*. ISSN: 2375-1207. May 2018, pages 711–725. DOI: 10.1109/SP.2018.00046.

[5] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured programming*. GBR: Academic Press Ltd., 1972. ISBN: 978-0-12-200550-3.

[6] Arthur Stanley Eddington. *The nature of the physical world*. eng. New York, The Macmillan Company; Cambridge, Eng., The University Press, 1928.

[7] George Fink and Matt Bishop. "Property-based testing: a new approach to testing for assurance". In: *SIGSOFT Softw. Eng. Notes* 22.4 (July 1997), pages 74–80. ISSN: 0163-5948. DOI: 10.1145/263244.263267.

[8] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. "GREYONE: Data Flow Sensitive Fuzzing". en. In: 2020, pages 2577–2594. ISBN: 978-1-939133-17-5.

[9] Hadi Hemmati. "How Effective Are Code Coverage Criteria?" In: *2015 IEEE International Conference on Software Quality, Reliability and Security*. Aug. 2015, pages 151–156. DOI: 10.1109/QRS.2015.30.

[10] Christian Holler, Kim Herzig, and Andreas Zeller. "Fuzzing with Code Fragments". en. In: 2012, pages 445–458.

[11] Intel. *Intel® 64 and IA-32 Architectures Software Developer Manuals*. en.

[12] Mark P. Jones. *Partial evaluation for dictionary-free overloading*. Yale University. Department of Computer Science, 1993.

[13] Vladimir Khorikov. *Unit Testing Principles, Practices, and Patterns*. en. Google-Books-ID: rDszEAAAQBAJ. Simon and Schuster, Jan. 2020. ISBN: 978-1-63835-029-3.

[14] Donald E. Knuth. "Ancient Babylonian algorithms". en. In: *Communications of the ACM* 15.7 (July 1972), pages 671–677. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/361454.361514.

[15] Mathieu Lachance. *How much people, time and money should QA take? Part1*. en. Jan. 2016.

[16] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. "Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation". en. In: *Advances in Neural Information Processing Systems* 36 (Dec. 2023), pages 21558–21572.

[17] Lui Sha. "Using simplicity to control complexity". en. In: *IEEE Software* 18.4 (July 2001), pages 20–28. ISSN: 0740-7459. DOI: 10.1109/MS.2001.936213.

[18] Dominik Maier, Lukas Seidel, and Shinjo Park. "BaseSAFE: Baseband SAnitized Fuzzing through Emulation". en. In: *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. arXiv:2005.07797 [cs]. July 2020, pages 122–132. DOI: 10.1145/3395351.3399360.

[19] Aditya P. Mathur. *Foundations of Software Testing, 2/e*. en. Google-Books-ID: OUA8BAAAQBAJ. Pearson Education India, 2013. ISBN: 978-93-325-1765-3.

[20] Toni Mattis, Eva Krebs, Martin C. Rinard, and Robert Hirschfeld. "Examples out of Thin Air: AI-Generated Dynamic Context to Assist Program Comprehension by Example". In: *Companion Proceedings of the 8th International Conference on the Art, Science, and Engineering of Programming*. Programming '24. New York, NY, USA: Association for Computing Machinery, July 2024, pages 99–107. DOI: 10.1145/3660829.3660845.

[21] Christoph Meinel and Martin Mundhenk. *Mathematische Grundlagen der Informatik: Mathematisches Denken und Beweisen Eine Einführung*. de. Wiesbaden: Springer Fachmedien Wiesbaden, 2015. DOI: 10.1007/978-3-658-09886-5.

[22] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. "Large Language Model guided Protocol Fuzzing". en. In: *Proceedings 2024 Network and Distributed System Security Symposium*. San Diego, CA, USA: Internet Society, 2024. ISBN: 978-1-891562-93-8. DOI: 10.14722/ndss.2024.24556.

[23] Microsoft. *Language Server Protocol*. 2024.

[24] Barton P. Miller, Lars Fredriksen, and Bryan So. "An empirical study of the reliability of UNIX utilities". en. In: *Communications of the ACM* 33.12 (Dec. 1990), pages 32–44. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/96267.96279.

[25] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Zhen Ming (Jack) Jiang. "GitHub Copilot AI pair programmer: Asset or Liability?" In: *Journal of Systems and Software* 203 (Sept. 2023), page 111734. ISSN: 0164-1212. DOI: 10.1016/j.jss.2023.111734.

[26] Joe Nelson. *The Design and Use of QuickCheck*. en. Jan. 2017.

[27] Fabio Niephaus, Patrick Rein, Jakob Edding, Jonas Hering, Bastian König, Kolya Opahle, Nico Scordialo, and Robert Hirschfeld. "Example-based live programming for everyone: building language-agnostic tools for live programming with LSP and GraalVM". en. In: *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Virtual USA: ACM, Nov. 2020, pages 1–17. ISBN: 978-1-4503-8178-9. DOI: 10.1145/3426428.3426919.

[28] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. "Babylonian-style Programming: Design and Implementation of an Integration of Live Examples into General-purpose Source Code". In: *The Art, Science, and Engineering of Programming* 3.3 (Feb. 2019). arXiv:1902.00549 [cs], page 9. ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2019/3/9.

[29] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. "Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types". en. In: 2021, pages 2597–2614. ISBN: 978-1-939133-24-3.

[30] Bundesamt für Sicherheit in der Informationstechnik. *AI Coding Assistants*. en. Sept. 2024.

[31] Mohammed Latif Siddiq, Joanna Cecilia Da Silva Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinícius Carvalho Lopes. "Using Large Language Models to Generate JUnit Tests: An Empirical Study". en. In: *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. Salerno Italy: ACM, June 2024, pages 313–322. DOI: 10.1145/3661167.3661216.

[32] Bjarne Stroustrup and Herb Sutter. *Unified Call Syntax: x.f(y) and f(x,y)*. en. 2015.

[33] Kristín Fjóla Tómasdóttir, Mauricio Aniche, and Arie van Deursen. "Why and how JavaScript developers use linters". In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Oct. 2017, pages 578–589. DOI: 10.1109/ASE.2017.8115668.

[34] Kristín Fjóla Tómasdóttir, Maurício Aniche, and Arie Van Deursen. "The Adoption of JavaScript Linters in Practice: A Case Study on ESLint". In: *IEEE Transactions on Software Engineering* 46.8 (Aug. 2020). Conference Name: IEEE Transactions on Software Engineering, pages 863–891. ISSN: 1939-3520. DOI: 10.1109/TSE.2018.2871058.

[35] Tijs Van der Storm and Felienne Hermans. *Live Literals*. Presented at the Work- shop on Live Programming (LIVE). 2016.

[36] Bret Victor. *Inventing on Principle*. Feb. 2012.

[37] Eric G. Wagner. "Algebraic Theories, Data Types, and Control Constructs". In: *Fundamenta Informaticae* 9.3 (July 1986), pages 343–370. ISSN: 18758681, 01692968. DOI: 10.3233/FI-1986-9305.

[38] Jiaye Wang. *Guiding Large Language Models to Generate Computer-Parsable Content*. arXiv:2404.05499 [cs]. Apr. 2024. DOI: 10.48550/arXiv.2404.05499.

[39] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. "Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization". en. In: *Proceedings 2020 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2020. ISBN: 978-1-891562-61-7. DOI: 10.14722/ndss.2020.24422.

[40] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. "Fuzz4All: Universal Fuzzing with Large Language Models". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE '24. New York, NY, USA: Association for Computing Machinery, Apr. 2024, pages 1–13. DOI: 10.1145/3597503.3639121.

[41] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. "A systematic evaluation of large language models of code". In: *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. MAPS 2022. New York, NY, USA: Association for Computing Machinery, June 2022, pages 1–10. ISBN: 978-1-4503-9273-0. DOI: 10.1145/3520312.3534862.

[42] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. *Krace: Data Race Fuzzing for Kernel File Systems*. 2019.

[43] Michal Zalewski. *Binary fuzzing strategies: what works, what doesn't*. Aug. 2014.

[44] Michal Zalewski. *Pulling JPEGs out of thin air*. Nov. 2014.

[45] Michal Zalewski. *Technical "whitepaper" for afl-fuzz*. en. Jan. 2015.

[46] A. Zeller and R. Hildebrandt. "Simplifying and isolating failure-inducing input". In: *IEEE Transactions on Software Engineering* 28.2 (Feb. 2002). Conference Name: IEEE Transactions on Software Engineering, pages 183–200. ISSN: 1939-3520. DOI: 10.1109/32.988498.

[47] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. "High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation". en. In: 2019, pages 1099–1114. ISBN: 978-1-939133-06-9.

[48] Hong Zhu, Patrick A. V. Hall, and John H. R. May. "Software unit test coverage and adequacy". In: *ACM Comput. Surv.* 29.4 (Dec. 1997), pages 366–427. ISSN: 0360-0300. DOI: 10.1145/267580.267590.

Appendix

Benchmark Measurements

The time it takes to compile and run a straightforward recursive Fibonacci implementation and an N-Body simulation:

Benchmark	Language	Compile Time (mean ± σ)	Runtime (mean ± σ)
Fibonacci	Rust	71.9 ms ± 2.2 ms	1.430 s ± 0.039 s
	Martinaise	309.5 ms ± 5.0 ms	15.830 s ± 0.106 s
	Java	322.4 ms ± 12.3 ms	744.8 ms ± 13.9 ms
	Python	-	27.215 s ± 0.376 s
N-Bodies	Rust	92.8 ms ± 2.7 ms	488.3 ms ± 6.3 ms
	Martinaise	509.7 ms ± 6.4 ms	24.432 s ± 0.410 s
	Java	352.5 ms ± 11.3 ms	136.9 ms ± 7.0 ms
	Python	-	3.595 s ± 0.076 s
Quicksort	Rust	85.5 ms ± 3.8 ms	1.920 s ± 0.020 s
	Martinaise	327.1 ms ± 6.3 ms	23.422 s ± 1.239 s
	Java	299.8 ms ± 8.3 ms	138.4 ms ± 0.3 ms
	Python	-	5.268 s ± 0.052 s

Soil Instructions

Soil is register-based: There is a stack pointer register `sp`, a status register `st`, and general-purpose registers `a`, `b`, `c`, `d`, `e`, and `f`.

Instruction	Description
nop	Does nothing.
panic	Panics.
trystart *catch*	Starts a panic scope: If a panic occurs, catches it, resets `sp`, and jumps to the *catch* address.
tryend	Ends a scope started by "trystart".
move *to from*	Sets the *to* register to the *from* register.
movei *to value*	Sets the `to` register to the word `value`.
moveib *to value*	Sets the `to` register to the byte `value`, zeroing the upper bits.

Instruction	**Description**
load *to from*	Interprets the *from* register as an address and sets *to* to the 64 bits at that address in memory.
loadb *to from*	Interprets *from* as an address and sets *to* to the 8 bits at that address in memory.
store *to from*	Interprets the *to* register as an address and sets the 64 bits at that address in memory to *from*.
storeb *to from*	Interprets *to* as an address and sets the 8 bits at that address in memory to *from*.
push *reg*	Decreases `sp` by 8, then runs "store `sp` *reg*".
pop *reg*	Runs "load *reg* `sp`", then increases `sp` by 8.
jump *to*	Continues executing at the *to*th byte.
cjump *to*	Runs "jump *to*" if `st` is not 0.
call *target*	Runs "jump *target*". Saves the formerly next instruction on an internal stack so that "ret" returns.
ret	Returns to the instruction after the matching "call".
syscall *number*	Performs the syscall with the given number. Behavior depends on the syscall. The syscall can access and change all registers and memory.
cmp *left right*	Saves *left* - *right* in `st`.
isequal	If `st` is 0, sets `st` to 1, otherwise to 0.
isless	If `st` is less than 0, sets `st` to 1, otherwise to 0.
isgreater	If `st` is greater than 0, sets `st` to 1, otherwise to 0.
islessequal	If `st` is 0 or less, sets `st` to 1, otherwise to 0.
isgreaterequal	If `st` is 0 or greater, sets `st` to 1, otherwise to 0.
isnotequal	If `st` is 0, sets `st` to 0, otherwise to 1.
fcmp *left right*	Saves *left* - *right* in `st`.
fisequal	If `st` is 0, sets `st` to 1, otherwise to 0.
fisless	If `st` is less than 0, sets `st` to 1, otherwise to 0.
fisgreater	If `st` is greater than 0, sets `st` to 1, otherwise to 0.
fislessequal	If `st` is 0 or less, sets `st` to 1, otherwise to 0.
fisgreaterequal	If `st` is 0 or greater, sets `st` to 1, otherwise to 0.
fisnotequal	If `st` is 0, sets `st` to 0, otherwise to 1.
inttofloat *reg*	Interprets *reg* as an int and sets it to a float of about the same value.
floattoint *reg*	Interprets *reg* as a float and sets it to its int, rounded down.
add *a b*	Adds *b* to *a*. Saves the result in *a*.
sub *a b*	Subtracts *b* from *a*. Saves the result in *a*.
mul *a b*	Multiplies *b* and *a*. Saves the result in *a*.
div *a b*	Divides *a* by *b*. Saves the quotient in *a*.
rem *a b*	Divides *a* by *b*. Saves the remainder in *a*.

Instruction	Description
fadd *a b*	Adds *b* to *a*, interpreted as floats.
fsub *a b*	Subtracts *b* from *a*, interpreted as floats.
fmul *a b*	Multiplies *a* and *b*, interpreted as floats. Saves the result in *a*.
fdiv *a b*	Divides *a* by *b*, interpreted as floats. Saves the quotient in *a*.
and *a b*	Binary-ands *a* and *b*. Saves the result in *a*.
or *a b*	Binary-ors *a* and *b*. Saves the result in *a*.
xor *a b*	Binary-xors *a* and *b*. Saves the result in *a*.
not *a*	Inverts the bits of *a*.

Game Generation

Listing 4: A custom generator for a game

```
fun generate(static: Static[Game], random: &Random, complexity: Int): Game {
  var level = static[Matrix[Tile]]().generate(random, complexity)
  Game {
    level,
    player = Player {
      position = Point {
        x = random.next_int(0..level.width),
        y = random.next_int(0..level.height),
      },
    }
  }
}
```