

The roc-fuzz manual

THE ROC-FUZZ CONTRIBUTORS

Version 0.4.3: Linux x86-64 with musl and Apple Silicon macOS

Table of Contents

1. Why fuzz a Roc application?	2
1.1. What a fuzz target is	2
1.2. How fuzzing complements tests	3
1.3. What “typed” and “coverage-guided” mean	4
1.4. What roc-fuzz provides	5
1.5. Choose a good first target	5
2. Fuzz your first function	7
2.1. Create the application module	7
2.2. Write the target	8
2.3. Build and find the bug	9
2.4. Fix and preserve the failure	9
2.5. Understand a normal run	10
2.6. Design the next target	11
2.7. Repository and release examples	11
3. Design effective targets	12
3.1. Start from the operation	12
3.2. Generate useful values	12
3.3. Build record inputs	13
3.4. Handle invalid input deliberately	14
3.5. Keep saved inputs meaningful	14
3.6. Arctangent builtin properties	14
4. Run and maintain fuzzing campaigns	16
4.1. Tune a run	16
4.2. Maintain a corpus	17
4.3. Diagnose an unstable failure	18
4.4. Know what counts as a failure	18
4.5. Prepare a long campaign	19
4.6. Command reference	19
5. Add fuzzing to CI	21
5.1. Keep project policy in the project	22
5.2. GitHub Actions pattern	22
5.3. Coverage limits	24
6. Release bundle provenance	25
6.1. Verify a release	26
7. Understand roc-fuzz internals and advanced features	27

7.1. Understand the in-process model	27
7.2. Understand generator dispatch	28
7.3. Assert on allocations	28
7.4. Use the libFuzzer interface	29
7.5. Further reading	30
8. Contributing to roc-fuzz	31
8.1. Toolchain	31
8.2. Repository scripting convention	32
8.3. Generate platform inputs	32
8.4. Regenerate ABI glue	34
8.5. Validation	34
8.6. Release bundle	37
8.7. Design constraints	38
9. Project information	39
9.1. Support, feedback, and releases	39
9.2. Origins and acknowledgments	39

roc-fuzz helps Roc application authors find inputs they did not think to test. You describe values to try and a rule that must always hold; roc-fuzz explores, saves any failure, and gives you commands to inspect, reproduce, and minimize it.

How to use this manual

New to fuzzing

Start with [Why fuzz a Roc application?](#) to learn what a fuzz target is, how guided search differs from random input, and when fuzzing fits.

Ready to try it

Follow [Fuzz your first function](#) to test a small Roc module and work through a saved failure.

Longer campaign

Continue with [Run and maintain fuzzing campaigns](#) for bounds, corpora, status output, and failure diagnosis.

Advanced use

Read [Understand roc-fuzz internals and advanced features](#) for the runtime model, allocation assertions, generator dispatch, and raw engine options.

Downstream maintainer

Read [Add fuzzing to CI](#) for a bounded GitHub Actions pattern and its report contract.

Release verifier

Read [Release bundle provenance](#).

Platform contributor

Read [Contributing to roc-fuzz](#) for toolchains, validation, packaging, and release behavior.

Chapter 1. Why fuzz a Roc application?

Suppose a decoder passes every example you wrote, but one unusual combination of lengths and Unicode text still makes it fail. Adding more example tests helps only when you can imagine the troublesome input in advance.

Fuzzing repeatedly runs part of your application with varied inputs. It watches which inputs reach new behavior and uses those discoveries to guide the search. You describe what valid inputs look like and what must always be true. roc-fuzz does the repetitive search and saves any input that exposes a failure.

1.1. What a fuzz target is

A **fuzz target** is a small Roc application that connects roc-fuzz to one behavior in your code. It contains three essential ideas:

- a **generator** that creates the Roc values to try;
- an **operation** that calls the application code you want to exercise; and
- a **property** that decides whether the result is acceptable.

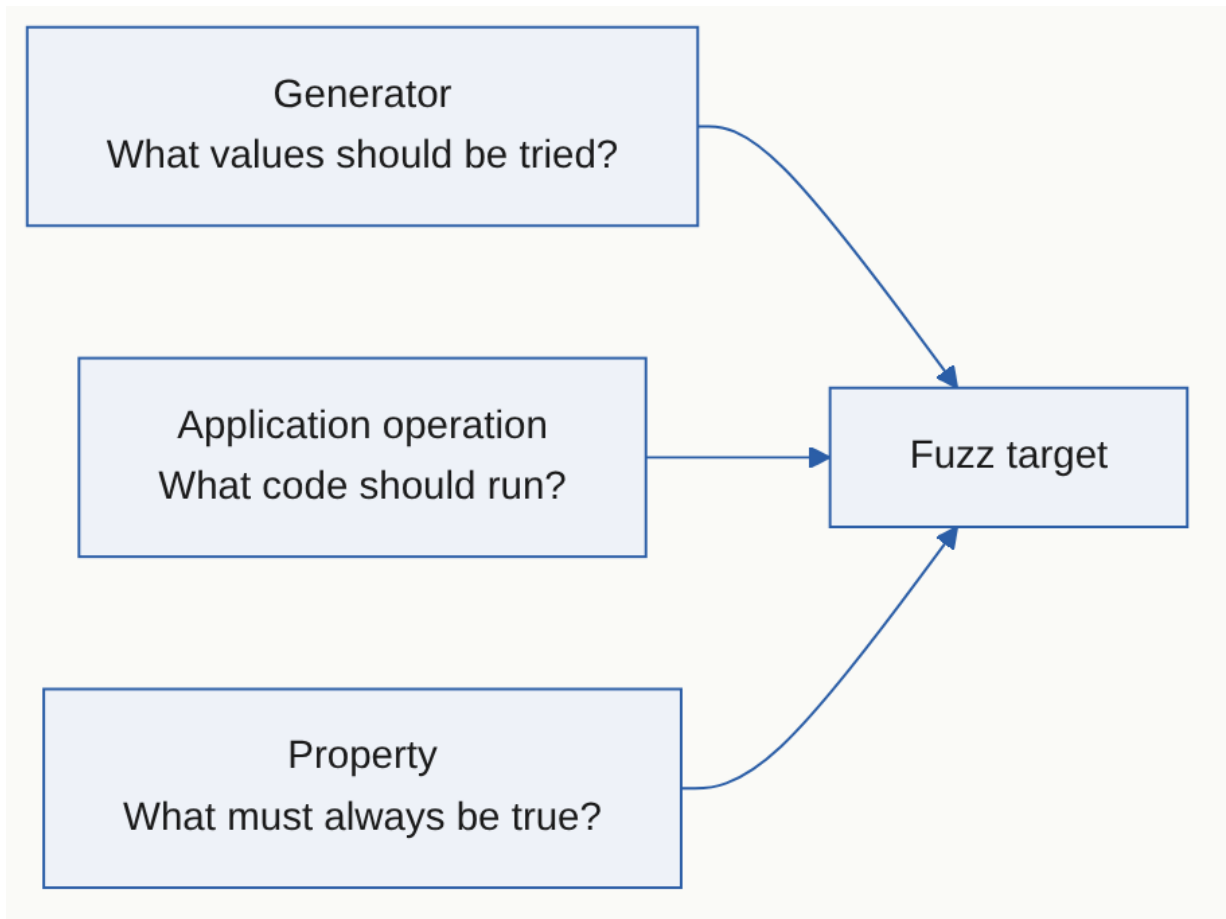


Figure 1. Target anatomy

The target is necessary because roc-fuzz cannot infer what correctness means for your application. It can explore inputs and notice crashes, hangs, leaks, and new behavior in the compiled program, but only you can state application rules such as:

- encoding and then decoding returns the original value;
- pushing an item and then popping returns that item and the original stack;
- normalizing an already normalized value changes nothing;
- two independent implementations produce the same result; or
- arbitrary input may return an ordinary error but must never crash.

A target is development and test code. It is separate from the application you ship, and it normally imports the module it exercises. Start with one operation and one property per target so failures remain easy to understand.

1.2. How fuzzing complements tests

Unit tests and fuzzing answer different questions and work best together.

Approach	What it does well	What it leaves to you
Unit tests	Check named examples and preserve known regressions quickly and deterministically.	You must choose every example.
A hand-written random loop	Tries more values than a short example list.	You must build generation, search strategy, repeatability, failure storage, and diagnostic tools.
roc-fuzz	Uses execution feedback to explore, retains useful inputs, and saves failures that can be shown, replayed, and minimized.	You must define meaningful generators and properties; a weak property can miss real bugs.

After roc-fuzz finds a bug, fix the code and keep the minimized case as a normal regression test. The unit test prevents that exact bug from returning. Future fuzzing continues searching for cases nobody has named yet.

1.3. What “typed” and “coverage-guided” mean

Typed means the test receives ordinary Roc values rather than making every target decode an arbitrary byte array itself. A generator can produce a `U64`, a `Str`, or a record such as `{ delimiter : Str, value : Str }`. You choose the useful ranges and shapes. roc-fuzz does not automatically know your application’s domain rules.

Coverage-guided describes how the search chooses where to spend effort. roc-fuzz observes which parts of the compiled program each input reaches. When an input reaches behavior not seen before, it retains that input and creates further variations from it. This feedback makes the search more purposeful than independent random samples.

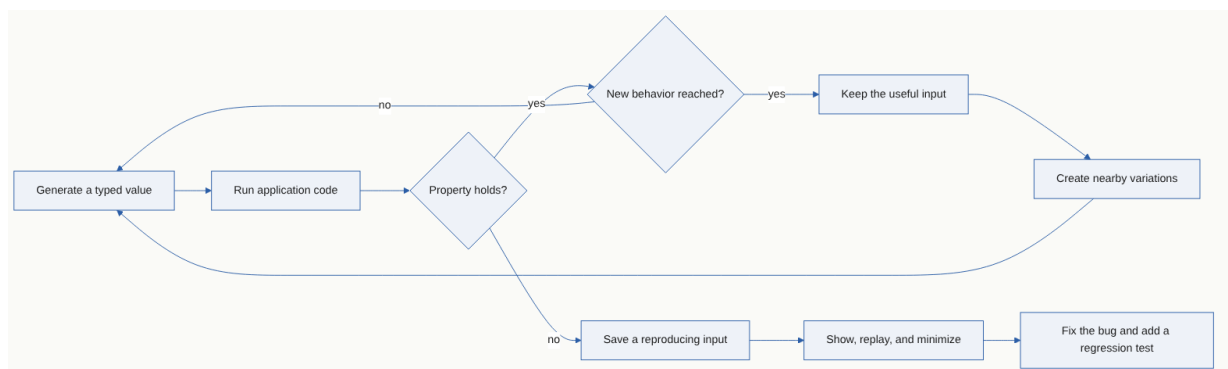


Figure 2. Coverage-guided search and failure workflow

The retained useful inputs form a **corpus**. A later run can reuse that corpus to resume from behavior already discovered. Coverage here is search feedback, not a percentage and not proof

that the program is correct.

roc-fuzz embeds a mature native fuzzing engine in the executable built by Roc. Normal users interact with roc-fuzz commands and Roc values; knowledge of that engine is needed only for the expert escape hatch described later.

1.4. What roc-fuzz provides

For a Roc application author, the platform supplies:

- typed generators for common Roc values and tools for combining them;
- coverage feedback that directs the search toward new behavior;
- a bounded local runner with sensible defaults;
- saved inputs for crashes, failed properties, hangs, memory limits, and leaks;
- commands to render a saved input as a Roc value, reproduce it, and minimize it;
- a persistent corpus that makes later runs build on earlier discoveries; and
- a supervised CI mode that records configuration, logs, hashes, and failures.

A target builds into one self-contained executable:

```
roc build --fuzz fuzz/stack/main.roc --output=stack-fuzz
./stack-fuzz run
```

The normal workflow needs no separate native project or fuzzing tool installation.

1.5. Choose a good first target

Good first targets are fast, deterministic operations over in-memory values with a rule that is easy to state.

Application behavior	Possible property	Why it fits
Parser or decoder	Every input returns a value or an ordinary error without crashing or hanging.	Large input space and clear acceptable outcomes.
Encoder and decoder	Decoding an encoded value returns the original value.	Strong round-trip property.

Application behavior	Possible property	Why it fits
Collection or state transition	An operation preserves membership, length, ordering, or an independent model.	Many combinations can exercise edge cases.
Canonicalization or normalization	Applying the operation twice gives the same result as once.	Simple idempotence property.
Two implementations of one rule	Both produce the same result.	Each implementation acts as an oracle for the other.

A target is a poor fit when one input performs slow network, filesystem, UI, or database work; depends on nondeterministic external state; can legitimately run without a practical time or memory bound; or deliberately terminates the process for ordinary invalid input. Prefer unit, integration, end-to-end, or benchmark tests for those jobs.

Fuzzing consumes CPU time and stores a corpus and failure artifacts. More time can explore more behavior, but a clean run means only that no failure was found during that search. It never proves the absence of bugs.

roc-fuzz currently supports Linux x86-64 with musl and Apple Silicon macOS 11 or newer. Continue with [Fuzz your first function](#) to build a target against a small application module and work through a failure.

Chapter 2. Fuzz your first function

This tutorial adds a fuzz target beside a small Roc module, runs it, inspects a failure, fixes the bug, and preserves the lesson as a regression test. You do not need prior fuzzing experience or knowledge of the embedded engine.

The example uses a stack because its rule is easy to state: after pushing an item, popping must return that item and the original stack. The demo keeps the application module beside its target only to make the tutorial self-contained; a real target normally imports an existing module from the application.

2.1. Create the application module

Create `fuzz/stack/Stack.roc`:

```
Stack := [].{
  push : List(a), a -> List(a)
  push = |items, item| List.append(items, item)

  pop : List(a) -> Try({ value : a, rest : List(a) }, [Empty])
  pop = |items|
    match items {
      [first, .. as rest] => Ok({ value: first, rest })
      [] => Err(Empty)
    }
}
```

This version contains a plausible implementation mistake: `push` appends the new item to the end, while `pop` removes an item from the front. A test using only an empty stack would pass, so the defect is easy to overlook.

The fuzz target is a separate development application that imports this module; the shipped application does not need to depend on `roc-fuzz`.

2.2. Write the target

Create `fuzz/stack/main.roc`:

```
app [target] { fuzz: platform "https://github.com/lukewilliamsboswell/roc-
fuzz/releases/download/0.4.1/FfSwkD2HdA63fcar5Z4EQpTBSGy4LcvEGenqzgiy9Mix.tar.zst", roc: "nightly-
2026-09-26-d6267b4" }

import fuzz.Fuzz
import Stack

test : List(U8) -> Fuzz.Outcome
test = |initial| {
  item = 0
  pushed = Stack.push(initial, item)

  match Stack.pop(pushed) {
    Ok({ value, rest }) if value == item and rest == initial => Fuzz.keep
    Ok(_) => crash "pop did not undo push"
    Err(Empty) => crash "the stack was empty after push"
  }
}

target = Fuzz.target_with({
  name: "stack-push-pop",
  generator: Fuzz.list(Fuzz.u8, 8),
  test,
  show: |initial| Str.inspect(initial),
})
```

The target connects the three ideas from the previous chapter:

Part	Role in this target
Generator	<code>Fuzz.list(Fuzz.u8, 8)</code> supplies lists containing up to eight <code>U8</code> values.
Application operation	The test calls <code>Stack.push</code> and <code>Stack.pop</code> .
Property	After pushing <code>0</code> , the popped value must be <code>0</code> and the remaining stack must equal the generated <code>initial</code> value.

`Fuzz.keep` means the generated case completed normally and satisfied the property. `crash` means roc-fuzz must save the case as a failure. `show` tells roc-fuzz how to render the generated value for a person.

The application header selects an immutable roc-fuzz release and its compatible Roc compiler. For another project, select a release from [GitHub Releases](#) and use the compiler documented with that release. Roc obtains the platform through its normal package mechanism; there is no separate roc-fuzz installer.

2.3. Build and find the bug

Build a self-contained fuzzing executable:

```
roc build --fuzz fuzz/stack/main.roc --output=stack-fuzz
```

Start with the bounded default run:

```
./stack-fuzz run
```

roc-fuzz generates initial stacks and repeatedly runs the property. An empty stack passes even with the bug, while a non-empty stack can expose the mismatch between the two ends. The exact status lines and saved hash vary, but the end of the output includes commands shaped like these:

```
Next steps:
./stack-fuzz show .roc-fuzz/crash-<hash>
./stack-fuzz replay .roc-fuzz/crash-<hash>
./stack-fuzz minimize .roc-fuzz/crash-<hash> .roc-fuzz/minimized-crash
```

The saved file contains the reproducible input used by the generator. It is an implementation format for roc-fuzz, so inspect it with `show` instead of opening it as text:

```
./stack-fuzz show .roc-fuzz/crash-<hash>
```

For this target, `show` prints the generated initial stack. Reproduce the failure in a fresh process:

```
./stack-fuzz replay .roc-fuzz/crash-<hash>
```

Then ask roc-fuzz for a smaller reproducing input:

```
./stack-fuzz minimize \
  .roc-fuzz/crash-<hash> .roc-fuzz/minimized-crash
./stack-fuzz show .roc-fuzz/minimized-crash
./stack-fuzz replay .roc-fuzz/minimized-crash
```

Minimization is most useful for structured or long inputs, but using the same workflow here establishes the habit.

2.4. Fix and preserve the failure

The two stack operations disagree about which end is the top. Fix `push` in `Stack.roc`:

```
push : List(a), a -> List(a)
```

```
push = |items, item| List.prepend(items, item)
```

Rebuild the executable and replay the minimized input. It should now complete without a failed property:

```
roc build --fuzz fuzz/stack/main.roc --output=stack-fuzz
./stack-fuzz replay .roc-fuzz/minimized-crash
./stack-fuzz run
```

Append a normal regression test using the minimized non-empty stack to `Stack.roc`:

```
expect {
  initial : List(U8)
  initial = [1]

  match Stack.pop(Stack.push(initial, 0)) {
    Ok({ value, rest }) => value == 0 and rest == initial
    Err(Empty) => Bool.False
  }
}
```

Run it with the rest of the module's tests:

```
roc test fuzz/stack/Stack.roc
```

Use the actual value printed by `show` if minimization found a different one-element list. The regression test protects this known case on every test run; the fuzz target continues exploring other values and future implementation changes.

A bounded run that finds no failure is still a successful run. It means the property held for the cases explored during that run, not that the operation is proved correct.

2.5. Understand a normal run

By default, `run` stops after 60 seconds, keeps useful inputs under `.roc-fuzz/corpus`, and saves failures under `.roc-fuzz/`. The corpus is a set of inputs that reached behavior worth revisiting; later runs reuse it and continue searching from those discoveries.

During a clean run, `NEW` means the engine retained an input that reached new behavior and `DONE` means the configured bound was reached. The [campaign status glossary](#) explains the remaining fields and tuning options.

2.6. Design the next target

Choose one fast application operation and one rule that should hold for every input. Common property shapes include:

- **Round trip:** decoding an encoded value returns the original value.
- **Inverse:** an operation followed by its inverse returns the starting value.
- **Idempotence:** normalizing twice gives the same result as normalizing once.
- **Invariant:** an operation preserves ordering, length, membership, or another rule of the data structure.
- **Differential:** two independent implementations produce the same result.
- **Robustness:** arbitrary input may return success or an ordinary error, but must not crash or hang.

Generate useful values directly. If a parser accepts bytes of any shape, arbitrary bytes may be appropriate. If an operation requires a bounded port number or a non-empty list, express that rule in the generator so most search effort reaches the behavior you care about. Use `Fuzz.reject` only when a generated case lies outside the property; do not use `crash` to reject an ordinary invalid input.

Keep unrelated operations and properties in separate targets. A practical first set might include one target per parser, encoder/decoder pair, state transition, or important business invariant. There is no need to fuzz every function.

The [campaign guide](#) explains corpora, tuning, and unstable failures. The [advanced guide](#) covers generator dispatch, allocation assertions, and the low-level engine interface. Run `TARGET --help` for the command summary and browse the [generated API reference](#) for generator, combinator, and target signatures.

2.7. Repository and release examples

The repository's `examples/` applications use relative paths to the local platform; prepare its native inputs before building them. Published releases provide a `roc-fuzz-examples-VERSION.zip` download containing complete runnable applications with that release's content-addressed platform URL and publication compiler. The platform does not constrain consumers to that compiler: nightly compatibility checks exercise each release's frozen examples with newer compilers.

Chapter 3. Design effective targets

The first tutorial generated one `U8`. Application behavior usually accepts records, lists, strings, or domain types. Build a generator that produces values close to the operation’s useful input domain, then keep the property focused on one behavior.

3.1. Start from the operation

Choose a function that is fast, deterministic, and easy to call in memory. Write the property in one sentence before choosing a generator. If the sentence needs several unrelated “and” clauses, make separate targets.

Examples include:

- “Decoding an encoded message returns the original message.”
- “Sorting preserves every item and produces nondecreasing order.”
- “A parser returns `Ok` or `Err` for every string without crashing.”
- “Applying this state transition agrees with the simple reference model.”

A generator explores only the cases the property can observe. Calling an operation without checking its result finds crashes and resource failures, but cannot find an ordinary wrong answer.

3.2. Generate useful values

roc-fuzz provides generators for numeric values, strings, bytes, and collections. Use a bounded generator when the full type includes values that are irrelevant or too expensive for this target:

```
port = Fuzz.u8_in(1, 255)
name = Fuzz.str
payload = Fuzz.list(Fuzz.u8, 1024)
```

`Fuzz.list(item_generator, maximum_length)` generates a list whose elements come from `item_generator`. The maximum is a test-design bound, so choose a size that reaches meaningful behavior without making each case slow.

Use `Fuzz.map` when the domain value can be constructed from one generated value, and `Fuzz.map2` when it needs two:

```
non_empty_bytes = Fuzz.map(
  Fuzz.list(Fuzz.u8, 255),
  |rest| List.prepend(rest, 1),
)

range = Fuzz.map2(
  Fuzz.u8,
  Fuzz.u8,
  |a, b| if a <= b { start: a, end: b } else { start: b, end: a },
)
```

Constructing valid values directly gives the search more useful executions than generating a broad value and rejecting most cases.

3.3. Build record inputs

Roc's record-builder syntax can combine a generator for each field:

```
Input := { delimiter : Str, value : Str }

input_generator : Fuzz.Generator(Input)
input_generator = {
  delimiter: Fuzz.str,
  value: Fuzz.str,
}.Fuzz
```

Use the resulting generator with `Fuzz.target_with`:

```
target = Fuzz.target_with({
  name: "split-round-trip",
  generator: input_generator,
  test,
  show: |input| Str.inspect(input),
})
```

This explicit form is a good default for a generator used by one target. When a named input type owns a reusable generation policy, define its `generator_for` method and use `Fuzz.target`:

```
Input := { delimiter : Str, value : Str }.{
  generator_for = |_| {
    {
      delimiter: Fuzz.str,
      value: Fuzz.str,
    }.Fuzz
  }
}
```

```

}

target = Fuzz.target({
  name: "split-round-trip",
  test,
  show: |input| Str.inspect(input),
})

```

Both forms produce the same kind of typed target. The [advanced explanation](#) describes how Roc resolves `generator_for`; using the pattern does not require knowing its implementation.

3.4. Handle invalid input deliberately

An ordinary invalid input is not automatically a fuzzing failure. A parser may return `Err` for malformed text and still satisfy a robustness property. Return `Fuzz.keep` when the observed result is allowed.

Prefer a generator that creates valid domain values. Return `Fuzz.reject` when a precondition is awkward to encode and the generated case is outside the property you are testing. Do not use `crash` or `expect` to reject such a case; those mean roc-fuzz found a failure worth saving.

Watch the rejection rate as tooling adds support for it. If most inputs are rejected, redesign the generator so the campaign spends more time in the code that matters.

3.5. Keep saved inputs meaningful

A saved file is decoded by the target's generator. Changing that generator can make an old file produce a different typed value. Keep generators stable when a corpus or failure matters, and retain the matching executable when exact historical reproduction is important.

The [generated API reference](#) lists all generators and combinators. Continue with [Run and maintain fuzzing campaigns](#) when the target produces useful values and passes a short smoke run.

3.6. Arctangent builtin properties

The `examples/builtins/arctangentDec.roc`, `arctangentF32.roc`, and `arctangentF64.roc` targets check `atan` and `atan2` with composed numeric generators. They also assert that the numeric checks allocate no heap memory; input generation and diagnostic rendering are outside that measurement.

The float targets check NaN propagation, signed zeros, infinities, range, quadrants, and odd symmetry. F32 results are compared with F64 results rounded to F32, allowing two ULP. F64 angles are checked against normalized input coordinates using sine and cosine, with absolute tolerance `3e-15`. Quadrant checks allow two ULP at the rounded half-pi boundary, while zero and infinity cases require exact result bits.

The Dec generator composes two U64 values into a signed 128-bit attosecond payload and varies its magnitude with a shift. It checks axes, range, symmetry, and agreement between `atan(y)` and `atan2({ x: 1, y })`, allowing 128 attoseconds. Comparison with the separate F64 implementation allows $4e-15$ for conversion and floating-point error. `Dec.lowest` is tested but excluded from negation checks because its negation overflows.

These checks supplement the compiler's high-precision unit tests; shared algorithms and finite tolerances mean they do not prove correct rounding. To run just these inventory entries with a compiler that includes the `atan2` builtins:

```
export ROC_NIGHTLY=/path/to/roc
scripts/test_targets.roc -- --operation all --max-total-time 60 \
  --target arctangentDec --target arctangentF32 --target arctangentF64
```

Use the normal runner's `show`, `replay`, and `minimize` commands to investigate saved failures, as described in the campaign chapter.

Chapter 4. Run and maintain fuzzing campaigns

After the first target works, use these goal-oriented guides to choose bounds, interpret status output, retain useful inputs, investigate unstable failures, and prepare longer searches. The [advanced guide](#) covers the embedded engine and target implementation details.

4.1. Tune a run

Give every new target a short, bounded smoke run before a long campaign:

```
TARGET run \
  --time=60 \
  --max-input-size=4096 \
  --timeout=5 \
  --memory-limit=2048 \
  --print-final-stats
```

The most useful limits are:

Option	Meaning	Starting point
<code>--time=N</code>	Stop after N seconds; 0 is unbounded.	Defaults to 60.
<code>--runs=N</code>	Stop after N generated inputs.	Useful for repeatable CI checks.
<code>--max-input-size=N</code>	Limit raw entropy bytes supplied to the generator.	Start small and raise only when needed.
<code>--timeout=N</code>	Treat one input taking longer than N seconds as a failure.	A few seconds for a normally fast target.
<code>--memory-limit=N</code>	Stop when the process exceeds this many MB.	Set from the expected working set.

Option	Meaning	Starting point
<code>--dictionary=FILE</code>	Add important byte tokens to mutations.	Useful for text or binary formats.
<code>--seed=N</code>	Select the engine's random seed.	Useful for repeating a campaign with the same runner build and configuration.
<code>--print-final-stats</code>	Print final execution and resource counters.	Useful in CI and smoke tests.

`--max-input-size` limits the generator's raw input, not necessarily every typed list or string it produces. Put bounds in the generator as well when a typed value could become expensive.

Do not confuse the campaign seed with a saved input. Different engine builds, coverage layouts, or configurations may generate different mutation sequences from `--seed`. Reproduce a failure with its saved artifact and `replay`, not with the campaign seed alone.

The runner reports compact status fields:

- `NEW`: an input reached behavior not represented by the corpus and was saved.
- `REDUCE`: a smaller input preserved previously found behavior.
- `cov`: covered edges or blocks.
- `corp`: number and total byte size of in-memory corpus entries.
- `exec/s`: target executions per second.
- `rss`: current process memory use.
- `DONE`: the run reached its time or iteration limit.

Coverage growth shows that the fuzzer is exploring. A plateau is a reason to inspect the target, generator, seeds, and dictionary, not proof that the code has no bugs. Very low `exec/s`, steadily growing `rss`, or repeated timeouts usually calls for a narrower target or tighter bounds.

4.2. Maintain a corpus

A corpus is a collection of small raw inputs that collectively reach useful code. It is not a directory of every attempted input. Use a stable corpus directory across campaigns:

```
TARGET run quality/corpus/my-target --time=3600 --max-input-size=4096
```

Seed a target with a few small examples that reach meaningfully different behavior, then let the

fuzzer extend them. Because roc-fuzz decodes raw entropy into typed values, a seed file's bytes may not resemble the value printed by `show`. Check manually created seeds with the target executable.

Reduce a large corpus into a new, initially empty directory while preserving coverage:

```
TARGET reduce-corpus quality/corpus/my-target quality/corpus/my-target-small
```

Keep valuable seeds and fixed crash inputs with the target in version control. Use a deliberate directory such as `fuzz/corpus/<target>/`, document why each seed matters, and replay it in tests. Generated `.roc-fuzz/` directories are ignored at every depth; keep evolving campaign corpora in local storage or CI caches, and preserve findings as CI artifacts until reduced to regressions. Run them after code changes so the target does not silently stop compiling or exercising the intended behavior.

Dictionaries are most effective when recognizable bytes flow fairly directly into a parser. They may help less when a structural generator heavily transforms the raw input.

Changing a generator can make old corpus and crash files decode to different typed values. The generator is part of the target's input format. Keep it stable when practical and use `show` to check saved inputs after a change.

4.3. Diagnose an unstable failure

Always replay a saved failure in a fresh process. If it does not reproduce, investigate:

- a generator or target change that altered the typed value or property;
- a different or truncated raw input file;
- an executable built from different target code; or
- a compiler or runtime defect.

Treat the failure as actionable once its conditions are understood and, ideally, it reproduces from a clean process.

4.4. Know what counts as a failure

Coverage feedback tells the engine which inputs explore new behavior; it does not tell the runner whether a result is correct. The property in the target is the correctness oracle.

roc-fuzz saves explicit Roc `crash` and failed `expect` calls, per-input timeouts, and process memory-limit failures. A wrong answer that does not break the target's property is invisible to the fuzzer,

which is why choosing a strong property matters more than simply calling the function under test.

4.5. Prepare a long campaign

Before committing substantial CPU time, check that:

- the target tests one named behavior with a meaningful property;
- invalid input returns normally unless invalid input crashing is the bug;
- the generator creates useful values without excessive rejection;
- input sizes and allocations are bounded;
- every generated input terminates or is caught by the timeout;
- a one-minute run gains coverage without immediate hangs or memory growth;
- a deliberate temporary `crash` is saved, shown, replayed, and minimized as expected; and
- the corpus and fixed failures have a clear place in the repository and CI.

4.6. Command reference

Run `TARGET --help` for the executable's own summary.

Command	Purpose
<code>TARGET run [CORPUS] [OPTION...]</code>	Run a local campaign, using the default corpus or the supplied directory.
<code>TARGET show INPUT</code>	Render the typed Roc value decoded from one saved input.
<code>TARGET replay INPUT</code>	Run one saved input in a fresh process.
<code>TARGET minimize INPUT OUTPUT</code>	Find a smaller input that reproduces the same failure and write it to <code>OUTPUT</code> .
<code>TARGET reduce-corpus INPUT OUTPUT</code>	Copy a smaller coverage-preserving subset of <code>INPUT</code> into an empty <code>OUTPUT</code> directory.
<code>TARGET ci REPORT_DIR [CORPUS] [OPTION...]</code>	Run a supervised campaign and write its report, summary, log, hashes, and failure files to a new or empty report directory.

The stable `run` and `ci` options are:

Option	Meaning
<code>--time=N</code>	Stop after <code>N</code> seconds. The default is <code>60</code> ; <code>0</code> is intentionally unbounded.
<code>--runs=N</code>	Stop after <code>N</code> generated inputs.
<code>--max-input-size=N</code>	Limit the raw entropy bytes supplied to the typed generator.
<code>--memory-limit=N</code>	Stop when the process exceeds <code>N</code> MB.
<code>--timeout=N</code>	Treat one input taking longer than <code>N</code> seconds as a failure.
<code>--dictionary=FILE</code>	Add format-specific byte tokens to the mutation strategy.
<code>--seed=N</code>	Select the campaign's random seed. Preserve a failure with its saved input, not this seed alone.
<code>--artifact-dir=DIR</code>	Write failure inputs under <code>DIR</code> .
<code>--no-detect-leaks</code>	Disable the per-input leak check for a documented exceptional campaign.
<code>--print-final-stats</code>	Print execution and resource counters when the run finishes.

The `ci` command also accepts provenance fields including `--source-revision`, `--roc-version`, `--platform-release`, and `--platform-sha256`. See [Add fuzzing to CI](#) for their use and the report layout. The [raw engine command](#) is an advanced escape hatch rather than part of the normal interface.

Chapter 5. Add fuzzing to CI

A short fuzz run on every pull request can catch easy failures early, while a longer scheduled run explores more cases without delaying ordinary development. Keep normal unit and regression tests in both paths: fuzzing searches for new failures but does not replace known examples.

The `ci` command runs one compiled target with explicit bounds and retains the information needed to investigate a failure. The complete workflow later in this chapter shows how a downstream project can run several targets and upload their results even when one fails.

Every executable built with `roc-fuzz` contains its own supervised CI runner. It can preserve a bounded campaign's configuration, provenance, log, corpus manifest, and reproducing failures without installing a separate helper:

```
my-target ci .roc-fuzz/evidence/my-target .roc-fuzz/corpus/my-target \
  --time=60 \
  --max-input-size=4096 \
  --timeout=5 \
  --memory-limit=2048 \
  --source-revision="$SOURCE_REVISION" \
  --roc-version="$ROC_VERSION" \
  --platform-release="$ROC_FUZZ_RELEASE" \
  --platform-sha256="$ROC_FUZZ_SHA256"
```

The command launches the same executable as a child. This lets the supervising process write evidence even when a Roc `crash`, failed `expect`, detected leak, timeout, memory limit, or native signal terminates the fuzzing child. A machine or supervisor process kill can still prevent report generation.

The report directory must be absent or empty and must not overlap the corpus. It receives:

- `report.json`, following `roc-fuzz-ci/v1`;
- `summary.md`, suitable for a CI job summary;
- `run.log`, containing combined stdout and stderr; and
- `failures/`, containing the runner's reproducing inputs.

The JSON manifest gives every corpus and failure file a size and SHA-256 digest. The target executable and log are also hashed. Provenance fields are optional for local work, but supply all four when publishing CI evidence. Arbitrary environment variables are deliberately not copied into

the report.

5.1. Keep project policy in the project

The host operates one compiled target. A downstream repository remains responsible for:

- deciding which target source files constitute its complete target inventory;
- pinning and installing Roc and the roc-fuzz platform release;
- generating domain-specific seeds and dictionaries;
- selecting per-target input, timeout, and memory bounds; and
- turning understood findings into deterministic regression tests.

This is important for targets such as Unicode conformance fuzzers, whose seed corpora are derived from versioned data files. A generic host cannot infer that application policy. A checked-in script or CI matrix should enumerate targets and invoke each resulting binary's `ci` command.

Leak checks remain enabled by default. `--no-detect-leaks` is available for a documented exceptional campaign, and the report records that it was used. Projects testing allocation-sensitive operations should also place `Fuzz.expect_allocs_at_most!` or `Fuzz.expect_allocs_at_least!` around the exact operation whose cost is an invariant.

5.2. GitHub Actions pattern

The following complete pattern assumes one target file per `fuzz/TARGET.roc` and optional checked-in seeds under `fuzz/corpus/TARGET/`. Repositories with generated seeds should replace only the seed-preparation step with their own script.

```
name: Fuzz campaigns

on:
  pull_request:
  schedule:
    - cron: "17 11 * * *"
  workflow_dispatch:

permissions:
  contents: read

jobs:
  fuzz:
    runs-on: ubuntu-24.04
    strategy:
      fail-fast: false
    matrix:
      target: [parser, decoder]
```

```

env:
  ROC_FUZZ_RELEASE: "REPLACE_WITH_THE_RELEASE_USED_BY_FUZZ_SOURCES"
  ROC_FUZZ_SHA256: "REPLACE_WITH_THE_RELEASE_BUNDLE_SHA256"
steps:
  - uses: actions/checkout@3d3c42e5aac5ba805825da76410c181273ba90b1 # v7

  - name: Read the pinned Roc version
    run: |
      pin="$(sed -n 's/.*roc: "\([^"]*\)".*/\1/p' fuzz/parser.roc)"
      test -n "$pin"
      printf 'ROC_NIGHTLY_TAG=%s\n' "$pin" >> "$GITHUB_ENV"

  - uses: lukewilliamsboswell/setup-roc@89a8a2945bd107f27b160096b3af7f52d2d78369 # stable-and-
nightly fork
    with:
      version: nightly-new-compiler
      nightly-tag: ${ env.ROC_NIGHTLY_TAG }

  - name: Restore this target's corpus
    uses: actions/cache@caa296126883cff596d87d8935842f9db880ef25 # v5
    with:
      path: .roc-fuzz/corpus/${ matrix.target }
      key: fuzz-${ runner.os }-${ matrix.target }-${ env.ROC_NIGHTLY_TAG }-${
env.ROC_FUZZ_RELEASE }
      restore-keys: fuzz-${ runner.os }-${ matrix.target }-

  - name: Build and seed the target
    env:
      TARGET: ${ matrix.target }
    run: |
      mkdir -p ".roc-fuzz/bin" ".roc-fuzz/corpus/$TARGET"
      roc build --fuzz "fuzz/$TARGET.roc" --output=".roc-fuzz/bin/$TARGET"
      if test -d "fuzz/corpus/$TARGET"; then
        cp -R "fuzz/corpus/$TARGET/." ".roc-fuzz/corpus/$TARGET/"
      fi

  - name: Run the supervised campaign
    env:
      TARGET: ${ matrix.target }
      CAMPAIGN_SECONDS: ${ github.event_name == 'schedule' && '900' || '60' }
    run: |
      roc_version="$(roc version | tr '\n' ' ')"
      ".roc-fuzz/bin/$TARGET" ci \
        ".roc-fuzz/evidence/$TARGET" \
        ".roc-fuzz/corpus/$TARGET" \
        --time="$CAMPAIGN_SECONDS" \
        --max-input-size=4096 \
        --timeout=5 \
        --memory-limit=2048 \
        --source-revision="${ github.sha }" \
        --roc-version="$roc_version" \
        --platform-release="$ROC_FUZZ_RELEASE" \
        --platform-sha256="$ROC_FUZZ_SHA256"

  - name: Publish the Markdown summary
    if: always()
    env:
      TARGET: ${ matrix.target }

```

```

run: |
  summary=".roc-fuzz/evidence/$TARGET/summary.md"
  if test -f "$summary"; then cat "$summary" >> "$GITHUB_STEP_SUMMARY"; fi

- name: Upload evidence and reproducing failures
  if: always()
  uses: actions/upload-artifact@043fb46d1a93c77aae656e7c1c64a875d1fc6a0a # v7
  with:
    name: fuzz-evidence-${{ matrix.target }}
    path: .roc-fuzz/evidence/${{ matrix.target }}
    if-no-files-found: error

```

Pull requests therefore get a one-minute bounded campaign per target, while the scheduled workflow gives each target fifteen minutes. Keep normal unit and regression tests on every change as well; a fuzz corpus is not a correctness oracle by itself.

5.3. Coverage limits

These artifacts provide reviewable evidence that dynamic analysis ran with explicit bounds and assertions, and that findings were retained. They do not measure source coverage.

The runner's `cov` and `ft` values are search-feedback counts from LLVM SanitizerCoverage. They have no source denominator and are not statement or branch percentages. Do not report them as either kind of percentage.

True Roc source coverage requires the Roc compiler to emit LLVM profile instrumentation and coverage mappings, link the matching profile runtime, and preserve stable source paths. Until that compiler work ships, roc-fuzz can report search progress and retained evidence but cannot provide a Roc source-coverage percentage.

Chapter 6. Release bundle provenance

roc-fuzz does not store native archives or objects in Git. The independent, workflow-dispatch-only `native-libraries.yml` producer builds x64-musl and Apple Silicon inputs on native GitHub-hosted runners, packages them without `libhost.a`, and tests the exact archives with fresh hosts and final bundles. It attests a canonical manifest and each archive but cannot publish. The trusted default-branch publisher verifies that evidence for the exact same-repository pull-request ref and SHA before creating an immutable release.

Platform builds restore the reviewed `native-libraries.lock.json` manifest and archives, verify sizes and SHA-256 digests on downloads and cache hits, validate their locked release, source, fingerprint and inventory, and build fresh hosts. The reviewed content hash is the routine trust anchor, so a valid cache hit uses zero network and does not contact GitHub's attestation service. A corrupt cache entry is discarded and only the exact locked asset may be downloaded again. The producer attestations are admission evidence checked once by the trusted publisher. See [linker-input setup](#).

After those consumer tests pass, GitHub's attestation service creates signed SLSA build provenance and an SBOM attestation for the exact `.tar.zst` bundle. GitHub binds the attestation to the release workflow and source revision with its OIDC identity, signs it through Sigstore, and records it in the repository attestations API. No long-lived signing key is stored in the repository.

Each release also publishes `<bundle>.intoto.jsonl` for offline attestation verification and `<bundle>.spdx.json` as an SPDX 2.3 software bill of materials. The SBOM records the bundle digest and the libFuzzer, Zig runtime, musl, and LLVM runtime components contained in the supported native inputs. When using released libraries, the bundle includes per-target `NATIVE_LIBRARIES.json` records and its SBOM identifies the library release archive URLs and digests. Source builds clear these records to avoid claiming provenance from previously restored archives.

The build pins Zig 0.16.0 and the SHA-256 of the `libfuzzer-sys 0.4.5` source archive. The macOS build necessarily uses the SDK supplied by the selected GitHub-hosted macOS runner. Native-library archives and the final platform bundle are separate published and attested subjects, with independent versions.

6.1. Verify a release

Download a bundle from the GitHub release, then run:

```
gh attestation verify roc-fuzz-<version>.tar.zst \  
--repo lukewilliamboswell/roc-fuzz
```

Verification checks the bundle digest, Sigstore signature, certificate and repository identity, and transparency information. This project does not claim a particular SLSA level; the attestation establishes the workflow and revision that produced the released bundle.

For offline or archived verification, download the matching `<bundle>.intoto.jsonl` asset and pass it to GitHub CLI:

```
gh attestation verify roc-fuzz-<version>.tar.zst \  
--bundle roc-fuzz-<version>.tar.zst.intoto.jsonl \  
--repo lukewilliamboswell/roc-fuzz
```

The adjacent `.spdx.json` file is human- and machine-readable release inventory. Its root package checksum must equal the SHA-256 of the downloaded bundle.

Chapter 7. Understand roc-fuzz internals and advanced features

This chapter explains the runtime model, reusable generator dispatch, allocation assertions, and the low-level libFuzzer escape hatch. Application authors do not need this material for the normal target workflow.

7.1. Understand the in-process model

The embedded libFuzzer calls a target thousands of times in the same process. It does not start a clean executable for every generated input. This is what makes it fast. Roc's purity means target calls cannot leak application state or background work into later calls, and determinism makes saved inputs reliable with the same target and generator.

A reliable target still needs all of the following properties:

- Invalid or unusual inputs return ordinary values such as `Err`, rather than triggering an intentional assertion or process exit.
- One input normally finishes in less than 10 milliseconds.
- Inputs and allocations have sensible upper bounds.
- Every input terminates.

The runner is a poor fit when one test takes a significant fraction of a second, can grow without a practical memory bound, or is expected to terminate the process for invalid input. If a crash, timeout, or memory blow-up would be a bug, it remains a useful fuzzing failure.

A saved raw input produces the same typed value and result when the target and generator are unchanged. Keep the executable alongside important failure artifacts when exact historical reproduction matters.

7.2. Understand generator dispatch

`generator_for` uses Roc's ordinary static dispatch. It is a method associated with the target's input type, not a runtime interface, registry, or callback table. Package APIs can require their own methods with `where` clauses, so the method does not need to be a compiler-defined name.

The pattern deliberately mirrors `Json.parser_for`:

- `Json.parse` requires its result type to have a compatible `parser_for` method and asks that type for a JSON parser.
- `Fuzz.target` requires its input type to have a compatible `generator_for` method and asks that type for a fuzz generator.
- The checker resolves both to a specific method implementation at compile time. The compiled call has no dynamic-dispatch overhead.

Conceptually, `Fuzz.target` has this requirement:

```
target : { name : Str, test : a -> Outcome, show : a -> Str } -> Target
  where [a.generator_for : FuzzEncoding -> Generator(a)]
```

`FuzzEncoding` currently contains the `Default` policy. Passing a policy value follows the same shape as `JsonEncoding` and leaves room for additional generation policies without changing the method pattern.

Use `Fuzz.target` when generation is part of a reusable named input type. Use `Fuzz.target_with` with an explicit `Generator(a)` for a local structural input or when choosing between generators for the same type in different targets.

7.3. Assert on allocations

A property can check what a builtin computes but not what it costs. A builtin that stops mutating a uniquely owned value in place and starts copying it still produces correct answers, so no content-based property will notice the regression. `Fuzz` exposes the platform's allocation counters so a target can assert on cost as well as correctness:

```
Fuzz.alloc_count! : () => U64          # cumulative allocations served this process
Fuzz.live_alloc_count! : () => U64      # allocations not yet freed
Fuzz.measure_allocs! : ({ } => a) => { value : a, allocations : U64 }
Fuzz.expect_allocs_at_most! : U64, ({ } => a) => a
Fuzz.expect_allocs_at_least! : U64, ({ } => a) => a
Fuzz.expect_no_leaks! : ({ } => a) => { }
```

`alloc_count!` and `live_alloc_count!` are process-wide and monotonic, and `libFuzzer` reuses one process across millions of inputs, so never assert on a raw value. Always read the counter before

and after the region you care about and assert on the difference; `measure_allocs!` and `expect_allocs_at_most!` and `expect_allocs_at_least!` do this for you.

Only assert zero allocations for a value that is genuinely uniquely owned and pre-sized. If the value is aliased anywhere, copy-on-write allocation is correct behavior, not a regression. Measure only the region you mean to check: a constructor call such as `Dict.with_capacity` allocates the entries and bucket lists, so keep it outside the measured window.

```
# Build the dict OUTSIDE the measured region: Dict.with_capacity itself
# allocates the entries and bucket lists.
d = Dict.with_capacity(n)

filled = Fuzz.expect_allocs_at_most!(
  0,
  |{}| {
    var $result = d
    var $i = 0
    while $i < n {
      $result = Dict.insert($result, $i, $i * 2)
      $i = $i + 1
    }
    $result
  },
)
```

Allocation assertions need an effectful test, so build the target with `Fuzz.target_with!` or `Fuzz.from_bytes!` instead of their pure counterparts. `target_with!` takes `test! : a ⇒ Outcome`; `from_bytes!` takes `test! : List(U8) ⇒ U8`.

These constructors exist because the runner calls a target through the widened `run! : List(U8) ⇒ U8`, not the older pure `run : List(U8) → U8`. This is a backward-compatible change: a pure function body already satisfies an effectful signature, so every existing target keeps compiling and running unchanged. Keep targets pure unless the property itself needs one of the allocation combinators above — a pure target is deterministic, and libFuzzer’s crash replay and minimization rely on that determinism to reproduce and shrink a saved failure.

7.4. Use the libFuzzer interface

`TARGET raw [LIBFUZZER_ARG...]` passes arguments directly to the embedded libFuzzer CLI. Run this for its complete option list:

```
TARGET raw -help=1
```

Prefer `run`, `show`, `replay`, `minimize`, and `reduce-corpus` for the normal workflow. Low-level flags such as `-max_total_time` and `-max_len` belong behind `raw`; `run` accepts the stable friendly options listed above. `raw` is an escape hatch for libFuzzer features the friendly command layer does not yet

expose directly.

7.5. Further reading

- [LLVM libFuzzer documentation](#) covers the in-process execution model, options, corpus behavior, output, and FAQ.

Chapter 8. Contributing to roc-fuzz

The platform has four layers:

- `platform/Fuzz.roc` defines typed generators, record-builder composition, and statically dispatched target construction.
- `platform/Target.roc` erases the app's input type behind closures.
- `platform/main.roc` exposes the byte-oriented native boundary and declares the x64-musl and Apple Silicon macOS targets.
- `src/main.zig` adapts generated Roc ABI calls to libFuzzer and translates the friendly runner commands.

`src/roc_platform_abi.zig` is generated from Roc's `Zig6lue.roc`.

8.1. Toolchain

Normal documentation, Roc source validation, target builds, and packaging require only:

- stable Roc for the repository's typed tooling; and
- the last validated nightly selected by `roc_nightly` in `scripts/workspace-deps.json`.

Linker inputs have an independent, content-addressed release cycle; `libhost.a` is deliberately outside that cycle and is built with each platform update. This separation means a host or Roc source change does not rebuild large, stable runtime inputs. Source development and the explicitly dispatched producer require Zig 0.16.0. The builder uses `zig ar`.

The setup action installs separate `roc-stable` and `roc-nightly` executables. The local `setup-workspace-roc` action reads their exact selections from workspace metadata and checks the installed versions. Tooling has its own fixed `roc_stable` selection; it does not advance with routine project nightly updates. `ROC_STABLE` and `ROC_NIGHTLY` allow explicit local overrides.

The shared nightly updater resolves the latest published compiler once, tests that exact candidate on both hosts, then advances only `roc_nightly` after all required checks pass. The platform and local examples contain no compiler pins. This metadata records a tested compiler, not an exclusive compatibility claim for a released platform.

Repository examples use relative paths to the working-tree platform. Each new release provides `roc-fuzz-examples-VERSION.zip` with complete applications, supporting modules, seeds/test inventory, an exact platform release URL, and the publication compiler in application headers. Newer compilers are checked against that frozen suite, independently of APIs added to the current repository examples. Only temporary application compiler pins are updated for these compatibility runs; the released platform bytes and URLs remain unchanged.

8.2. Repository scripting convention

Repository-owned automation is written in Roc. Put executable entry points in `scripts/.roc` and reusable logic in `scripts/src/.roc`. Use the existing shared modules for commands, paths, hashing, source edits, archives, and release metadata. Do not add Python scripts, shell launchers, or build-before-run wrappers to work around compiler behavior. Keep workflow YAML limited to job orchestration and small environment/output plumbing; substantive automation belongs in Roc.

Run scripts directly with `roc-stable scripts/NAME.roc --ARGUMENTS`, or their `#!/usr/bin/env roc-stable` shebang. Nested tooling calls use `ROC_STABLE` through `Script.roc_stable!`; project validation uses `ROC_NIGHTLY` separately. The fixed `roc_stable` selection must execute the tooling and its dependencies without warning-exit workarounds. Upgrade it deliberately when a compatible compiler and dependencies are available. A successful script must return zero, and a failed script must remain a failure.

Roc scripts may invoke standard tools such as Git, curl, tar, zip, and unzip via argument lists, without shell evaluation. Roc owns selection, validation, and control flow. Pinned external tools and shared actions remain external dependencies; for example, the shared documentation builder currently uses Python internally. That does not justify adding repository-owned Python orchestration.

Pure regression checks belong in shared-module expect's imported by `scripts/tooling_tests.roc`. Filesystem and process checks belong in `scripts/tooling_integration_tests.roc`; both run from `scripts/check_all.roc`. The archive tools require Git, curl, tar, zip, and unzip on local and CI hosts.

8.3. Generate platform inputs

Generate the current host's ignored inputs with:

```
scripts/build_platform.roc
```

Pass `--target x64musl` or `--target arm64mac` explicitly in automation. The default restores the archives selected by `native-libraries.lock.json`, rehashes their manifest and target archive even on a cache

hit, validates the complete archive inventory, builds the current host, and writes local `SHA256SUMS`. The archive digest is the reviewed authorization for routine use; the cache key only locates likely bytes. This is why cache hits are still hashed and why a cache miss downloads the exact locked release rather than rebuilding. It avoids repeat network traffic without allowing a poisoned or truncated cache entry to become trusted. Routine restoration does not query GitHub's attestation service. Never commit generated archives, objects, or manifests.

The workflow-dispatch-only `Linker inputs` producer packages both targets without `libhost.a`, tests each candidate through a fresh host and final bundle, emits a canonical aggregate manifest, and attests that manifest and both archives. It has no publication authority. Release new inputs for changes to their sources, runtimes/toolchains, flags, targets, security fixes, or the macOS adapter compiled into `libfuzzer.a`. The macOS coverage shim stays with `libhost.a`.

`Publish linker inputs` runs trusted controller code pinned to a reviewed full `roc-automation` commit. It dispatches the producer at the exact same-repository PR head, verifies the workflow/ref/SHA, hashes and attestations, publishes an immutable release named from the aggregate manifest hash, and adds only the lock in a lease-guarded GitHub-signed commit. Candidate code cannot publish, while the privileged job never executes candidate code. The PR-built release remains the permanent dependency after merge; rebuilding it would break the evidence-to-byte identity that the review admitted.

GitHub immutable releases must be enabled before the first publication. The controller checks the resulting release state and refuses to adopt a lock when the hosting policy does not enforce immutability; a content-derived name alone cannot prevent an administrator from replacing assets.

Before adopting a release, review its producer changes, exact source revision, input fingerprint, manifest and archive digests, sizes, target coverage, native link tests, and workflow attestations. Merge with a merge commit so the attested producer SHA and signed lock commit remain reachable. Ordinary PRs and nightly updates then know only the reviewed lock, cache, verifier, and target files; they never dispatch or reproduce linker inputs.

When the linker-input fingerprint changes, use this sequence:

1. Stage the source and tool-pin changes in a same-repository PR. Its regular CI may fail with `linker-input lock is stale` until the new lock is adopted.
2. To inspect a nonpublishing candidate, dispatch `Linker inputs` on the PR branch with `release_candidate=true` and `expected_sha` set to its exact current head. For example, run `gh workflow run native-libraries.yml --ref BRANCH -f release_candidate=true -f expected_sha=SHA`, replacing `BRANCH` and `SHA`. Review both target tests, the manifest, archive hashes, and attestations. This run does not create a GitHub release or edit the PR.
3. From `trunk`, dispatch `Publish linker inputs` with `pull-request` set to that PR number: `gh workflow run publish-linker-inputs.yml --ref trunk -f pull-request=NUMBER`. The trusted publisher reruns

the producer at the exact PR head, publishes the immutable linker-input release, and appends a GitHub-signed `native-libraries.lock.json` -only commit to the PR. Do not dispatch it from the PR branch or rebuild the inputs after merging.

4. Review that lock commit and revalidate the updated PR. After its required checks pass, merge the PR with a merge commit. Publish a new Roc platform bundle later through the separate `Release` workflow, using the adopted lock.

The optional preview in step 2 does not replace the producer run performed by the trusted publisher in step 3. The publisher is the release operation; stop before that dispatch when only candidate validation is authorized.

The fully static build excludes `FuzzerInterceptors.cpp`. Its wrappers locate libc functions through `dlsym`, which is not usable in the static musl executable. The standard libFuzzer scheduler, mutators, corpus engine, crash handling, minimizer, and merge engine remain unchanged. Roc's compare and switch instrumentation still supplies value feedback.

The macOS target includes `FuzzerInterceptors.cpp`: its `dlsym` lookups work with macOS's dynamic `LibSystem`. Its deployment target is macOS 11.0, and its only allowed dynamic dependencies are system libraries.

`src/macos_sancov.c` supplies the public TLS slot required by Roc's macOS stack-depth coverage instrumentation. `src/macos_fuzzer_ext_functions.cpp` binds the standalone runner hooks directly because upstream libFuzzer's Darwin `dlsym` lookup requires `-export_dynamic`, which Roc does not pass. Both files are repository-owned platform glue covered by the project license; the native build, `show/replay`, `fuzz`, `failure-artifact`, and `packaged-bundle` smoke tests exercise them on Apple Silicon.

8.4. Regenerate ABI glue

Regenerate glue only when the Roc natural ABI changes:

```
export ROC_NIGHTLY="$HOME/roc_nightly-linux_x86_64-2026-09-26-d6267b4/roc"
scripts/build_platform.roc -- --regenerate-glue
```

Review the generated Zig diff and rebuild every example after regeneration.

8.5. Validation

Every bug fix and significant new behavior should include an automated test at the narrowest useful layer. Changes to public behavior also update the relevant examples, interface documentation, or report schema. A pull request should explain the behavior changed and the

validation performed.

Use the compiler under development through `ROC_NIGHTLY`:

```
export ROC_NIGHTLY=/path/to/roc
scripts/test_targets.roc -- --operation check --verbose
scripts/test_targets.roc -- --operation build --verbose
scripts/test_targets.roc -- --operation seed
scripts/test_targets.roc -- --operation fuzz --max-total-time 2
```

After native inputs are prepared, `check_all.roc` validates the working-tree platform and every example. `test_targets.roc` provides the focused operations shown above. Checked-in examples use the local platform. The published compatibility lane uses the selected release's own frozen examples and immutable bundle. Validation checks the exact example inventory, Roc formatting and types. After generating the current host inputs, building creates every self-contained executable and verifies a static x86-64 ELF on Linux or an arm64 Mach-O with system-only dynamic dependencies on macOS. Seed validation renders and replays each deterministic input. The fuzz operation runs short campaigns and verifies that an intentional Roc failure is saved byte-for-byte with follow-up commands.

Build the HTML manual, PDF, and generated API reference with:

```
scripts/build_docs.roc
```

The command fetches the immutable shared renderer from `roc-automation`, uses Docker for the Asciidoctor toolchain, and writes the complete publication under `.docs-out/`. Set `ROC_AUTOMATION_ROOT` to a local checkout while developing the shared action. The final API audit fails if an exposed module or public entry has no rendered documentation. The documentation and release workflows run the same build and audit.

Every example runs the `check`, `test`, `build`, `seed`, and `fuzz` stages by default. A temporary exception must use a `skip` entry in `tests/targets.json` with both a concrete reason and a full GitHub issue URL. The driver rejects unexplained skips; skipping a prerequisite also requires skipping its dependent stages.

CI separates published compatibility, working-tree execution and packaged-platform execution. Published checks use the selected release's frozen suite on fresh runners; development checks use local examples and current source. Both run on every PR and manual/nightly dispatch; there is no change classifier or migration exception. The release workflow additionally tests the exact candidate archives. `nightly_validation: true` never publishes or deploys.

Linux CI also rebuilds the checksum-pinned libFuzzer sources with AddressSanitizer and runs a short native smoke campaign. This detects heap, stack, use-after-free, and leak errors in the

embedded C++ fuzzing engine on every pull request, every trunk update, and before release publication. Run the same check locally with `roc-stable scripts/test_native_sanitizers.roc`; it requires Clang, `curl`, and `tar`. The sanitizer executable is CI-only and does not change the static release artifacts.

The Roc driver runs on Linux and Apple Silicon without platform skips. Scripts execute directly with the fixed tooling compiler; compilation errors, warnings, and application failures retain their normal Roc exit status.

Use `--suite-root PATH` to run a frozen examples tree containing `examples/` and `tests/targets.json`. Each invocation has fresh temporary executables and corpora. Fuzz smoke tests pass corpus directories to libFuzzer; passing an individual seed file would only replay it. Repository formatting is checked separately, not against an older release's frozen source.

Published compatibility selects the highest stable semantic-version platform release, ignoring native-library releases and RCs. Both hosts consume the same selection and verify asset hashes. Releases through 0.4.1 use examples from the exact tag commit, rebound to that release's own bundle (the old manual URL follow-up could occur after tagging). Newer releases must include the examples archive. The compatibility artifact records the release, source commit, compiler and asset identities used; a missing archive or failed download cannot become a passing test.

Every target asserts on allocation behaviour as well as on results. A property can check what an operation computes but not what it costs, so a builtin that stops mutating a uniquely owned value in place and starts copying it still returns the right answer and no content property notices. Pin the cost with `Fuzz.expect_allocs_at_most!` or `Fuzz.expect_allocs_at_least!`, and build the target with `Fuzz.target_with!` or `Fuzz.from_bytes!` so the property may perform effects. The `check` stage requires one of those assertion helpers; raw counter reads, measurements, and leak-only checks do not satisfy the policy. A target that genuinely cannot assert on cost belongs in the allocation-assertion exemptions in `scripts/test_targets.roc` with a concrete reason.

Assert on a difference between two counter reads, never on a raw value: the counter is process-wide and libFuzzer reuses one process across millions of inputs. Only assert zero for a value that is uniquely owned and pre-sized, because copy-on-write allocation is correct when a value is aliased.

Separately, the runner checks after every input that the target freed everything it allocated, and fails the input otherwise. Pass `--no-detect-leaks` to a run to turn that off.

The former byte-oriented builtin targets live under `examples/builtins/` and use `Fuzz.from_bytes`. This keeps their existing properties in the regression matrix. Top-level examples are an end-user gallery and should prefer `Fuzz.target`, typed generators, and the `.Fuzz` record builder. A multi-file example must live in its own directory with `main.roc` as its app root; the test driver enforces this so editor tooling can discover the project naturally.

8.6. Release bundle

Create a Roc platform bundle after generating both target input sets with:

```
scripts/build_bundle.roc -- --output-dir dist
```

`build_bundle.roc` verifies every generated input checksum, then includes the archives and license notices in `roc bundle`. Production release jobs build both hosts on native hosted runners before bundling. Test the resulting bundle from an external target with:

```
scripts/test_bundle.roc -- dist/<bundle>.tar.zst
```

Production releases use the `Release` workflow. From the repository's **Actions** tab, run it on the default branch with an `X.Y.Z` version (or an `X.Y.Z-rcN` release candidate). Pull requests run the same workflow in dry-run mode. Before dispatching any publication, add a non-empty `docs/releases/X.Y.Z.adoc` or `docs/releases/X.Y.Z-rcN.adoc`. The first heading must be `= roc-fuzz VERSION`. Release notes are reviewed as documentation and should explain the release to users: lead with the theme, show notable features and compatibility changes, name the supported hosts and Roc compiler, link to verification material, and give upgrade or feedback guidance.

Use `{platform-url}` where the copyable app header needs the content-addressed bundle URL and `{manual-url}` for the release's PDF. The workflow resolves both after packaging, translates the note's headings and source blocks to GitHub Markdown with `scripts/render_release_notes.roc`, and uses that result as the description for stable releases and release candidates. A missing or mismatched note stops publication. A real release:

1. validates all sources, tests, and short fuzz campaigns;
2. builds the platform bundle and tests that packaged bundle as an external consumer;
3. generates and validates a version-stamped PDF manual and HTML site containing the API reference; and
4. renders the matching narrative release note and publishes it with the bundle, PDF, zipped documentation site, SBOM, and provenance in a GitHub release.

Pages is independent of a release and always tracks `trunk`. Pull requests build the same site without deploying it.

The bump check requires a version increment from the previous platform release. The release policy explicitly permits exact-nightly bootstrap on `trunk`; no stable compiler compatibility or maintenance branch is implied.

To unblock a source PR before stable bootstrap, dispatch `Release` on that branch with `release_candidate=true`, a new `X.Y.Z-rcN` version, and the full source SHA in `expected_sha`. Both target bundles must pass before the workflow attests and publishes the RC. Only this explicit RC path

permits source-built libraries. It preserves the latest stable release and the trunk-tracking Pages site, and does not create a default-branch URL follow-up. Verify the published archive and its frozen examples before use. Stable releases still require the independent library lock.

Release preparation generates and tests the examples archive against the exact candidate bundle, then publishes and attests those same bytes. It does not rewrite repository example URLs. Download the examples asset when using a published release; cloning the repository gives examples for local platform development.

A routine nightly update never fixes or publishes a platform. If a new compiler breaks the latest release, the updater reports the failure on its pull request and leaves the accepted compiler unchanged. GitHub check notifications (according to each maintainer's notification settings) and the PR report flag the failure. Maintainers manually investigate, fix, and release the platform. After that manual intervention, retry the nightly updater against the newly released suite. The updater records the new compiler only after validation passes; there is no automated platform-repair or release-recovery workflow.

A tooling/compiler-recipe migration can require a one-time linker-input publication through the existing reviewed PR publisher. The linker fingerprint includes the fixed tooling compiler and recipe, but excludes `roc_nightly`; subsequent project compiler-only updates reuse the admitted native libraries.

Inspect partial publication before recovery; never replace existing tags or assets or rebuild an already-published release from a moving branch. Preserve the exact tested artifacts, tag SHA and uploaded digests. No changes to historical releases are required to adopt this workflow.

8.7. Design constraints

Static dispatch happens while `Fuzz.target` is specialized for the app's input type. `Target` then erases that type, preventing the native ABI from depending on every app record or tag-union layout.

The platform supports x64-musl and Apple Silicon macOS. Keep native calls behind generated glue so compiler ABI changes are caught by regeneration and compilation.

Chapter 9. Project information

9.1. Support, feedback, and releases

Use [GitHub issues](#) for reproducible bugs and enhancement requests. Include the roc-fuzz release, Roc compiler version, operating system and architecture, target source or a minimal reproduction, and the relevant command output. Proposed changes use pull requests and follow the [contributor guide](#).

Suspected vulnerabilities follow the [private security reporting process](#) and must not be disclosed in a public issue. Published bundles, PDF manuals, documentation archives, checksums, provenance, and human-readable release notes are available from [GitHub Releases](#).

9.2. Origins and acknowledgments

The foundational and exploratory work for roc-fuzz was created by [Brendan Hansknecht](#). Brendan established the original Roc fuzzing integration, arbitrary input machinery, quality-target suite, and the early [trophy case](#) demonstrating the bugs this approach could find.

The platform in this repository extends Brendan's experiment into a modern Roc workflow with typed generators, reusable target construction, and a self-contained executable built directly with `roc build --fuzz`. This work would not exist without the foundation he developed.