Property-based testing and fuzzing

Concepts, and what they look like when the target is a protocol

Compiled 31 August 2026

What this is. A concepts-first map of generative testing: what property-based testing is, what fuzzing is, how the two relate to static analysis and formal verification, and how all of it changes when the system under test is a blockchain protocol with eleven independent implementations.

Concepts before tools. Chapter 5 is a deliberately thin reference layer. Tooling in this area churns fast enough that any specific setup dates within months, while the underlying ideas — oracles, shrinking, structure-awareness, differential testing — have been stable for decades. Knowing which category a tool belongs to, and why that category exists, is worth more than depth in any one of them.

How it was written. Factual claims were checked against a primary source on the day of writing: the Ethereum Foundation's repositories and blog, tool documentation, and the published work of the engineers named. Facts carry a bracketed source, and the sourcing caveats are in Appendix A.

Currency. Everything is current as of 31 August 2026 — Fusaka live, Glamsterdam in development. Repository details and tool status move; the techniques do not.

Contents

CHAPTER 1

Property-based testing

1.1 The definition, in one breath

The short version:

> **In one breath**
>
> An example-based test fixes one input and one expected output. A **property-based** test instead states something that must be true of *every* input in some domain, and then has the machine generate hundreds or thousands of inputs trying to violate it. When it finds a violation it **shrinks** the failing input down to the smallest one that still fails, so what lands on your desk is a minimal counterexample rather than a haystack.

The technique comes from QuickCheck — Koen Claessen and John Hughes, ICFP 2000, originally for Haskell. Effectively every modern implementation (Hypothesis in Python, proptest and quickcheck in Rust, fast-check in TypeScript, Foundry's fuzz tests in Solidity) is a descendant of that paper.

1.2 The three moving parts

Every property-based testing framework is these three things. Being able to name them separately is most of what fluency looks like.

Generator. Produces random values of a given shape — an integer in a range, a well-formed transaction, a valid block header. Generators compose: a generator for a struct is built from generators for its fields. The quality of your generators bounds the quality of your testing, because a generator that never produces the interesting case will never find the bug in it.

Property. A predicate that must hold. Usually one of a small number of recognisable shapes — see §1.3.

Shrinker. Given a failing input, finds a smaller or simpler input that still fails. This is the part people underrate and it is the part that makes the technique usable in practice: a raw random failure might be a 4 kB blob with forty fields set; the shrunk version is three bytes and tells you what the bug is.

1.2.1 Two schools of shrinking — worth knowing, cheap to say

This is the one piece of PBT trivia that is genuinely worth carrying, because it is a real design distinction and not a tool detail.

- **Type-directed / separate shrinking** (QuickCheck, and most things modelled on it). You write a generator and, separately, a shrinker. They can disagree: the shrinker can produce a value the generator never would, which violates an invariant, which gives you a "counterexample" that is not reachable in reality. That is a false positive manufactured by your own test framework.

- **Integrated shrinking** (Hedgehog, Hypothesis). The generator carries its own shrink tree, so every shrunk value is one the generator could have produced. Shrinks obey the generator's invariants *by construction*, which removes that entire class of false positive.

Why this specific fact earns its place

It is a false-positive story, and false positives are the axis the EF Protocol Security team's own published post is organised around — "a reproducer that builds some internal value by hand, one no real input could ever produce." A badly-shrunk counterexample is exactly that failure mode, produced mechanically. Connecting the two is the point: a shrinker that wanders outside its generator's invariants manufactures unreachable counterexamples for a living.

1.3 The shapes a property actually takes

In practice almost every useful property is one of five. Having this list means you can answer "how would you test X?" for basically any X.

Shape	Form	Example
Round-trip	`decode(encode(x)) == x`	Any serialisation: RLP, SSZ, JSON-RPC payloads. The cheapest real property there is, and it finds parser bugs immediately.
Invariant	Some predicate holds in every reachable state	Total supply never changes except by mint/burn; a balance is never negative; the fork-choice head is always a descendant of the finalised checkpoint.
Oracle / differential	`mine(x) == theirs(x)`	Two independent implementations must agree. *The dominant shape in Ethereum* — see Chapter 4.
Metamorphic	`f(g(x))` relates predictably to `f(x)`	You have no reference implementation, but you know a transformation that should not change the answer. Optimised and unoptimised compilation of the same program must produce the same result.
Idempotence / commutativity	`f(f(x)) == f(x)`; order-independence	Applying a state transition twice; two independent updates in either order.

The honest caveat to volunteer

Property-based testing is only as good as the property. If you cannot state what "correct" means, PBT gives you nothing — and the failure is silent, because a test that asserts something trivially true passes forever. That is the third false positive in the Protocol Security post, stated for formal proofs: "The statement is trivially true regardless of what the code does." Same disease, different tool.

1.4 Stateful and model-based testing

The step up, and the part that matters most for protocol work.

Stateful (model-based) testing. Instead of generating *arguments to a function*, you generate a *sequence of commands*. You run that sequence against the real system, and in parallel against a deliberately simplified **model** of what the system should do. After each step you assert the two still

agree. When they diverge, shrinking reduces the command sequence — so the counterexample is the shortest sequence of operations that breaks the system.

The vocabulary, which is standard across frameworks:

- **Command** — one operation the system offers (deposit, withdraw, process block, add peer).
- **Precondition** — when this command is legal to issue, so the generator only builds valid sequences.
- **Postcondition** — what must be true after it, given the model.
- **Model** — the simplified reference. It must be simpler than the system, or you have written the bug twice.

Why this is the important one

Individually valid steps, in an order nobody considered, is where the expensive bugs live. It is also precisely the gap the EF's own post identifies in AI agents: their weakest column is "bugs that span a sequence of valid steps," and the prescribed fix is that "the agent isn't the search tool. Its job is to suggest which sequences are worth running through a **stateful test harness**."

Which puts it plainly: *the thing agents are worst at is the thing stateful property-based testing is specifically for.*

1.4.1 The coverage question that follows

The obvious objection to generated command sequences is: how do you know the generator ever produced an interesting one? The standard answer is **distribution checking** — frameworks provide facilities (`classify`, `collect`, coverage requirements) to assert that some percentage of generated cases hit a named scenario, and to *fail the run* if they did not.

That matters because the default failure mode of generative testing is silent uselessness: ten thousand cases that all take the same trivial path. As an answer to "how do you know the testing is working," this beats a coverage percentage.

1.5 What PBT is bad at

Worth stating plainly, because the limits shape when to reach for something else.

- **Properties you cannot state.** If "correct" is only definable by reference to a full implementation, your model becomes the system and the test proves nothing.
- **Deep semantic bugs behind narrow gates.** If reaching the interesting state requires a valid signature, a specific fork, and a particular gas configuration, random generation will never arrive. You need either structure-aware generation (§2.3.3) or coverage feedback to steer.
- **Cost of the model.** Model-based testing is real engineering. Sometimes the honest answer is that a differential oracle is cheaper than a model, because someone else already wrote the reference.
- **It proves nothing.** Passing means "we did not find a counterexample in N cases." It is evidence, not a proof — which is the boundary with formal verification in Chapter 3.

CHAPTER 2

Fuzzing

2.1 The definition, in one breath

In one breath

Fuzzing is feeding a program a large volume of generated or mutated input and watching for something observably wrong. The two questions that determine whether a fuzzer is any good are: **how does it choose the next input**, and **how does it decide that something went wrong**. The second question — the **oracle** — is the one that actually matters, and it is the one people skip.

2.2 The oracle is the whole game

Oracle. The thing that decides whether an execution was a failure. A fuzzer without an oracle is just a load generator.

Oracles, from weakest to strongest. This ladder is the single most useful mental model in this document:

Oracle	Detects	Cost / limit
Crash or hang	Memory-safety failures, panics, infinite loops	Free. Misses everything that goes wrong quietly — i.e. most consensus bugs.
Sanitizer trip	Memory errors, undefined behaviour, uninitialised reads, data races	Cheap. Turns silent corruption into a loud crash. Slows execution meaningfully.
Assertion / invariant	Violations of a property you stated	You have to state it. This is property-based testing wearing a fuzzer's clothes.
Differential	Two implementations disagree	Requires a second implementation — but then "correct" is defined for free. *Dominant in Ethereum.*
Metamorphic	A transformation that should not change the output, did	No reference implementation needed. Excellent for compilers and provers.

The consequence

A fuzzer with a crash oracle finds crashes. Most of what actually threatens a blockchain is not a crash — it is two clients quietly disagreeing. So the first question to ask about any fuzzing campaign is what the oracle is, not what the mutator is.

2.2.1 Sanitizers, briefly

Compiler instrumentation that makes latent memory bugs observable: **ASan** (out-of-bounds, use-after-free), **UBSan** (undefined behaviour: signed overflow, bad shifts, misaligned access), **MSan**

(reads of uninitialised memory), **TSan** (data races). Conceptually they belong on the oracle ladder above, not in the "tools" chapter — their whole function is to strengthen the oracle.

Relevant caveat, and it is one of the false positives the EF post names: a panic that only happens in a *debug* build. Sanitizer and debug-assertion findings must be re-checked against the configuration the software actually ships in.

2.3 How the next input gets chosen

2.3.1 Black-box, greybox, white-box

Black-box. Throw input at the program, observe nothing internal. Cheap, and it plateaus almost immediately on anything with structure.

Greybox / coverage-guided. The important one. Instrument the binary to record which code edges an input reached. Keep an input in the **corpus** if it reached coverage nothing else did; mutate the corpus to make new inputs. This turns fuzzing from random into a crude search that climbs into the program. AFL/AFL++, libFuzzer, honggfuzz and Go's built-in fuzzing all work this way.

White-box / symbolic. Reason about the program's paths algebraically and solve for inputs that reach them. Precise, does not scale. See Chapter 3.

2.3.2 Mutation-based vs generation-based

Mutation-based starts from a corpus of real inputs and perturbs them: flip bits, splice two inputs, change an integer to a boundary value. **Generation-based** builds inputs from a description of what a valid input looks like — a grammar, a schema, a type.

2.3.3 Structure-aware fuzzing, and why protocols need it

The core problem

> If the input is an RLP-encoded transaction or an SSZ-encoded beacon block, random byte flips die at the parser. You get enormous throughput and zero depth: you are fuzzing the decoder's error path forever and never reaching the state transition behind it.

Structure-aware (or grammar-aware, or type-aware) fuzzing fixes this by generating inputs that are *valid by construction* and mutating them in ways that preserve validity — change a field, not a byte. It is the same idea as a property-based testing generator, arriving from the security side of the family rather than the functional-programming side.

This is not theoretical for this job: the Protocol Security team's own public repo is literally a structure-aware fuzzing framework (§4.4).

2.4 Differential fuzzing

Worth its own section because it is the technique the job is built around.

Differential fuzzing. Run the same input through two or more independent implementations of the same specification and compare the outputs. Any disagreement is a bug in at least one of them.

Why it fits blockchains so exactly:

- Ethereum deliberately has many independent client implementations, in different languages, written by different teams.

- Consensus means they must agree *exactly*. Not approximately.
- Therefore the oracle costs nothing: you do not need to know which one is right to know something is wrong.
- And the impact is maximal: a disagreement on a reachable input is a potential chain split, which is the worst class of failure the protocol has.

The variant with a designated reference is sometimes called **conformance** or **spec testing**: rather than comparing clients to each other, compare each client to an executable specification. Ethereum has both, and uses both (Chapter 4).

2.5 Corpus, triage, and the unglamorous middle

The parts nobody puts in a talk and everybody spends their time on:

- **Seed corpus.** What you start from. Real-world inputs beat random ones by a wide margin.
- **Corpus minimisation / distillation.** Reduce the corpus to the smallest set preserving coverage, so mutation effort is not wasted.
- **Crash triage and deduplication.** One bug produces thousands of crashing inputs. Bucketing them — usually by stack hash or by the shrunk reproducer — is what turns a crash pile into a finding list.
- **Reproducibility.** A finding must run for someone who did not produce it. Fixed seed, pinned revision, one documented way to build and run.
- **The known-issues ledger.** Every candidate checked against what is already known, fixed, or rejected — otherwise you rediscover the same closed bug forever.

Why this middle section is the job

The EF post's whole argument is that this middle section — not the generation — is the job: "Signal-to-noise is most of the work," and the bottleneck "moved from finding bugs to trusting the results." If you are deciding what to build first on a new campaign, it is almost always something in this list rather than a better mutator.

2.6 The limits, stated honestly

- **Fuzzing shows presence, never absence.** It is sampling. Passing means you did not find it.
- **Coverage plateaus.** Campaigns saturate; new coverage per CPU-hour falls off a cliff. Knowing when to stop and change the harness rather than buying more cores is a real skill.
- **Coverage is a proxy, not the goal.** High coverage with a weak oracle finds nothing. Reaching the line is not checking the line.
- **Finding nothing is a result.** From the EF post: "Run this against mature, heavily audited code and almost nothing survives, which is still worth knowing. 'We looked hard and found nothing' is a real result."

2.7 So what is the difference between fuzzing and PBT?

It is a common question, and the weak answer is a taxonomy. The better answer is this:

They converged

Historically they are two traditions that converged on the same idea. Property-based testing

> came out of functional programming and cared about **stating a property** and **shrinking** the counterexample. Fuzzing came out of security and cared about **throughput**, **coverage feedback** and **crash oracles**. They have met in the middle: a modern fuzz harness that decodes structured input and asserts an invariant *is* a property-based test, and a property-based framework with coverage feedback *is* a fuzzer.
>
> So the label is not very useful. The two questions that are: what is the oracle, and does a failure shrink to something a human can read? Those determine whether a campaign produces findings or noise.

If you want one crisp distinction to offer alongside it: PBT usually knows what *valid* input looks like and asks whether the system is *correct*; classical fuzzing usually does not care about validity and asks whether the system *survives*. Structure-aware fuzzing is the point where that distinction dissolves.

CHAPTER 3

The rest of the family

Fuzzing and property testing are two members of a larger set. You do not need depth in the others — you need to be able to place any of them correctly when someone drops the name, and to say why you would reach for one over another.

3.1 The one axis that organises everything

The trade

Every technique here sits somewhere on a single trade-off: **how much it proves** against **how big a system it can handle**. Fuzzing scales to a whole client and proves nothing. Formal verification proves a theorem about a few hundred lines. Everything else is in between, and the engineering judgement is picking the weakest technique that still answers your question.

Technique	What it does	Where it breaks
Static analysis	Examines code without running it — AST, control-flow graph, data-flow. Finds known-shape defects fast across a whole codebase.	No execution means no ground truth. High false-positive rate; cannot reason about values it cannot infer.
Fuzzing	Runs the real system on many inputs, checks an oracle. Finds real, reproducible failures.	Sampling. Proves nothing. Blind to states it cannot reach.
Property-based testing	Runs the real system on generated inputs against a *stated* property, shrinks failures.	Only as good as the property and the generator. Still sampling.
Symbolic / concolic execution	Runs the program with symbolic instead of concrete inputs; uses an SMT solver to find inputs that reach a path or violate an assertion.	Path explosion. Loops and hashing are murder. Usually bounded to a depth, so "verified" means "verified up to N."
Bounded model checking	Proves a property holds for all executions up to a bound.	The bound. Says nothing beyond it.
Formal verification / proof	Machine-checked proof that a property holds, unconditionally, of a model.	Cost, and the model gap: the proof is about the model, not the deployed artifact.
Runtime verification	Checks invariants in production; assertions, monitors, chain watchers.	Detects after the fact. But it is the only one that sees real traffic.

3.2 The model gap — the sophisticated point to make

The most valuable thing to be able to say about formal verification, and the one that shows you are not just impressed by it:

The model gap

A proof is about a model. Deployment is about an artifact. The proof holds absolutely *of the thing that was proved* — and then there is an assumed correspondence between that thing and the

bytecode or binary that actually runs. That assumption is usually where the residual risk lives, and it is exactly why formal verification and fuzzing are complements rather than alternatives: the proof covers the model, and differential fuzzing checks that the deployed implementation matches it.

This is not theoretical — it is the EF Protocol Security team's stated experience, in their own words: run agents "against formally verified code, where a machine-checked proof covers a model and the deployed bytecode is only assumed to match it, and more gets through." [blog.ethereum.org, 9 Jul 2026] They found *more* bugs in formally verified code, for precisely this reason.

And the matching false positive, from the same post: a proof that "goes through but doesn't mean what you wanted. The statement is trivially true regardless of what the code does, or it's weaker than the property you meant to capture. The verifier is satisfied, but the theorem doesn't constrain the behavior you actually cared about."

3.3 How to choose — the answer to "how would you test X?"

A defensible order of operations, which doubles as an answer to almost any open-ended testing question:

1. **Ask what "wrong" means here.** That is the oracle, and it determines everything else. If you cannot answer it, no tool helps.
2. **Is there a second implementation, or a spec you can execute?** If yes, differential testing is the cheapest strong oracle available and you should start there.
3. **If not, can you state an invariant?** Then property-based testing, with a structure-aware generator.
4. **Is the bug likely to need a *sequence*?** Then stateful / model-based testing, not single-shot generation.
5. **Is the surface a parser or a decoder?** Coverage-guided fuzzing with sanitizers, seeded from real inputs.
6. **Is the component small, critical and stable — crypto, a state transition, a precompile?** That is where symbolic execution or a proof pays for itself.
7. **Whatever you find, can someone else re-run it?** If not, it is not a finding.

The trap in this question

The wrong answer is to name a tool. What matters is reasoning from the oracle outward; opening with "use *[tool]*" skips the only interesting step.

3.4 Coordinated disclosure, in one page

Testing-adjacent, and the practice that decides what happens after a finding is real.

The basic shape. Reporter finds a vulnerability, reports privately to those who can fix it, a fix is developed and distributed, and only then is the issue published. The point is to keep the window in which attackers know and defenders cannot act as small as possible.

Why it is harder here. There is no single vendor to patch. A protocol bug may affect several independent client teams who must all ship, and the fix may be visible in a public repository before most nodes have upgraded — so the patch itself can leak the vulnerability. Coordinating *silent* patches and staged releases across teams is a real part of the job, and it is why "crisis coordination" appears in the EF Mandate's list of work only the Foundation can do.

The rules you can cite. The EF bug bounty requires a proof of concept and states that public disclosure without prior agreement forfeits the reward — disclosure discipline enforced economically. [ethereum.org/bug-bounty]

The judgement call. Who to tell, in what order, and when to go public is not a procedure, it is a decision. The Mandate's *Discipline* principle covers it: "We choose relevant timing over either going fast or going slow, which may include not acting at all."

One more from their own post

> "A person makes the final call: agents suggest. They don't decide what's real, what's a duplicate of a known issue, or **what gets disclosed and when**." Disclosure is explicitly carved out as a human decision.

CHAPTER 4

When the target is Ethereum

4.1 Why differential testing dominates here

Everything about how Ethereum is tested follows from one design decision: **client diversity**. There is no reference binary. There are many independent implementations of the same specification, written by different teams in different languages — and consensus requires them to agree exactly.

Execution layer	Geth (Go), Nethermind (C#), Besu (Java), Erigon (Go), Reth (Rust)
Consensus layer	Prysm (Go), Lighthouse (Rust), Teku (Java), Nimbus (Nim), Lodestar (TS), Grandine (Ru

[the in-scope list of the EF bug bounty]

The consequence, stated cleanly

Client diversity is a security property: it means one implementation bug is not a network failure. But it also hands you a free oracle. You never need to know which client is right — a disagreement on a reachable input is already a finding, and potentially a chain split. That is why differential testing is not one technique among many at the EF; it is the default.

4.2 The executable specification

The second structural fact: for both layers, the specification is not prose — it is runnable Python.

`ethereum/execution-specs` **(EELS).** The execution layer specification as a Python reference implementation. Because it executes, it can serve as the reference side of a differential comparison.

`ethereum/consensus-specs`**.** The same for the consensus layer — the beacon chain state transition, fork choice, and the rest, in Python.

`ethereum/execution-spec-tests` **(EEST).** The framework and test suite that generates the test vectors every execution client must pass. In the EF's own description, it "is responsible for the Ethereum protocol reference tests, used by all clients to detect consensus issues."

`ethereum/consensus-spec-tests`**.** The generated test vectors for the consensus layer.

`ethereum/hive`**.** The end-to-end harness that runs real client binaries in containers against each other and against test suites — block import, Engine API, sync. This is where multi-client integration is actually exercised.

The pipeline, in one sentence

A spec change lands in the executable spec, test vectors are generated from it, every client must pass them, and Hive exercises the real binaries together end-to-end — so "did this fork break anything" has a mechanical answer before mainnet.

The rule for fork work is worth knowing: a protocol change needs tests that *fail before the change and pass after*. That is the conformance equivalent of a regression test, and it is what turns reviewing an upcoming hardfork into a tractable activity rather than a reading exercise.

4.3 What protocol security fuzzing actually looks like day to day

The most useful evidence available, because it is a public record of the exact job by someone already doing it. **Bhargava Shastry** (`bshastry`) is a Security Engineer at the EF and one of the public members of the `protocol-security` organisation. His published account of his own work: [bshastry.github.io, retrieved 31 Aug 2026]

- **Ethereum protocol security, 2022–present.** Differential fuzzing across execution clients — Geth, Besu, Nethermind, Erigon, revm — and hard-fork readiness testing validated against the executable specifications. Representative findings: a genesis-block `BLOCKHASH` divergence in revm, a memory-size overflow crash in Nethermind, coinbase state handling issues.
- **Solidity compiler, 2018–2022.** A differential fuzzer for the compiler (SolSmith); 25 patched miscompilation bugs; seven entries in Solidity's security-relevant bug ledger.
- **Post-quantum cryptography, 2026–present.** Differential testing of ML-KEM (FIPS 203) and ML-DSA (FIPS 204) implementations — i.e. the same technique pointed at the primitives the Strawmap's Post-Quantum L1 goal depends on.
- **Side channels, 2026–present.** Timing vulnerabilities in Mbed TLS and TF-PSA-Crypto, including variable-time division in RSA prime generation.

Notice the through-line

Clients, then a compiler, now cryptographic primitives — and it is the *same method* every time: two independent implementations of one specification, run against each other, disagreement is the bug. Differential testing is the clearest case in security work of a method that generalises far beyond the target it was learned on.

4.4 The team's own fuzzing repo

`protocol-security/fuzztools` — "Struct-aware fuzzing framework + some fuzzers," Rust, public, created October 2025, 31 stars. [gh api repos/protocol-security/fuzztools; README]

It is worth a look as a worked example, because it is a direct instance of §2.3.3:

- A `Mutable` derive macro that automatically implements mutation for arbitrary Rust structs — mutate *fields*, not bytes.
- A `builders` crate whose stated job is "creating **VALID** instances of *to-be-fuzzed* types." Validity by construction, which is the property-based testing generator idea imported into a fuzzer.
- Crates for consensus and execution spec types, transactions, and a batched JSON-RPC client.
- **Rakoon** — a transaction fuzzer for the protocol, crediting Marius van der Wijden's `tx-fuzz` as reference. Its README lists found crashes in anvil, go-ethereum and reth.
- **Noiruzz** — a *metamorphic* fuzzer for the Noir compiler: build an AST for a circuit, apply transformations that by definition should not change the output, and check the output did not change. Several Noir compiler bugs listed.

Why it is worth reading

Both fuzzers on display are textbook instances of concepts from Chapters 1–2: structure-aware generation, and a metamorphic oracle used precisely because there is no reference compiler to diff against. It is a short repository, and it is one of the few places where a working protocol-security fuzzing setup is visible in public.

CHAPTER 5

The tool landscape

A placement layer, not study material

The point of this chapter is to let you place a name when one comes up — *that's the Solidity fuzzer*, *that's a static analyser*, *that's symbolic execution* — and then reason about the underlying technique, which is Chapters 1–3. Tools churn; the techniques do not. None of these is worth memorising, and depth in any particular one matters far less than knowing which category it belongs to and why that category exists.

5.1 General-purpose fuzzers

Name	Who	One line
AFL / AFL++	community	The canonical coverage-guided mutational fuzzer for native code. AFL++ is the maintained fork with modern mutators and instrumentation.
libFuzzer	LLVM	In-process, coverage-guided; you write a `LLVMFuzzerTestOneInput` entry point. Pairs naturally with sanitizers.
honggfuzz	Google	Coverage-guided, feedback-driven; hardware-counter based coverage among others.
Go native fuzzing	Go team	Built into the toolchain since Go 1.18 — `func FuzzXxx(f *testing.F)`. Relevant because much of Ethereum is Go.
Hypothesis	Python	Property-based testing for Python, with integrated shrinking. Relevant because the executable specs are Python.
Hedgehog / QuickCheck / FsCheck / proptest	various	The property-based testing family. See Chapter 1 for the distinction that matters.

5.2 Ethereum protocol and client fuzzers

Name	Who	One line
fuzztools	EF Protocol Security	Struct-aware framework, plus Rakoon (transaction fuzzer) and Noiruzz (metamorphic fuzzer for the Noir compiler). §4.4.
tx-fuzz	Marius van der Wijden	Sends interesting transactions at a network; used for load and for shaking out client differences.
FuzzyVM	Marius van der Wijden	Differential fuzzer for EVM implementations — generates state tests and compares clients.
beacon-fuzz	Sigma Prime, for the EF	Differential fuzzer for eth2 consensus implementations. **Archived** — historical context, not current tooling.

5.3 Smart-contract tooling

Adjacent to the role rather than central to it — protocol security is client and specification work, not Solidity auditing. Know the names, know which category each falls in, stop there.

Name	Category	One line
Echidna	fuzzer	The well-known Solidity/EVM property-based fuzzer. You write Boolean invariant functions in the contract; it generates transaction sequences trying to falsify them. Trail of Bits.
Medusa	fuzzer	Trail of Bits' newer parallelised, coverage-guided contract fuzzer, built on go-ethereum. Same idea as Echidna, different engine.
Slither	static analysis	*Not* a fuzzer — a static analyser for Solidity and Vyper. Also Trail of Bits. See the note below.
Foundry	test framework	`forge` has both *fuzz tests* (property-based, per-function) and *invariant tests* (stateful — random call sequences against invariants). Its `anvil` node appears in fuzztools' findings list.
Halmos	symbolic	Symbolic testing for EVM contracts — same test style as Foundry, but explores paths with a solver instead of sampling. a16z.
Kontrol / KEVM	formal	Runtime Verification's stack: the K semantics of the EVM, used to prove properties of contracts.
Certora	formal	Commercial formal verification; you write rules in a specification language and it proves or refutes them.
ItyFuzz	fuzzer	Bytecode-level hybrid (fuzzing + symbolic) contract fuzzer.

Two things people get wrong about Slither

The name. It is *Slither* — the Trail of Bits naming runs Slither / Echidna / Manticore / Medusa, snakes and monsters — and it is routinely misremembered as "Slytherin."

The category. Slither is a *static analyser*, not a fuzzer. Grouping it with Echidna mixes the two halves of Chapter 3, and the distinction — reasons about the code versus executes the code — is the whole point. It is also why their outputs need reading differently: a static analyser reports suspicions, a fuzzer reports reproductions.

5.4 Verification and specification, non-EVM

Two worth a sentence, because they are current EF work rather than ecosystem background:

- **Lean** — the proof assistant. The EF's Formal Verification team runs `better.codes`, an "autoresearch challenge" where participants use any AI models and tooling they like against a Lean-checked benchmark, raising proven soundness bounds for hash-based SNARKs toward 128-bit security. The design point is that *the verification is open even when the tooling is not.* [blog.ethereum.org, 20 Aug 2026]
- **Noir** — a ZK circuit language whose compiler the team fuzzes metamorphically (§4.4). Useful as evidence that "protocol security" at the EF now includes proving-stack correctness, not just clients.

APPENDIX A

Sources

All checked 31 August 2026.

A.1 Concepts

- QuickCheck — Koen Claessen, John Hughes, *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*, ICFP 2000. The origin of property-based testing.
- Hypothesis (Python, integrated shrinking) — https://hypothesis.readthedocs.io
- Hedgehog (integrated shrinking, state-machine testing) — https://github.com/hedgehogqa/haskell-hedgehog and https://github.com/hedgehogqa/fsharp-hedgehog
- libFuzzer — https://llvm.org/docs/LibFuzzer.html
- AFL++ — https://github.com/AFLplusplus/AFLplusplus
- honggfuzz — https://github.com/google/honggfuzz
- Go native fuzzing — https://go.dev/doc/security/fuzz/
- Foundry invariant (stateful) testing — https://book.getfoundry.sh/forge/invariant-testing
- Building Secure Contracts (Trail of Bits, property/invariant guidance) — https://secure-contracts.com and https://github.com/crytic/properties

A.2 Ethereum test infrastructure

Repository	URL
execution-specs (EELS)	https://github.com/ethereum/execution-specs
execution-spec-tests (EEST)	https://github.com/ethereum/execution-spec-tests
consensus-specs	https://github.com/ethereum/consensus-specs
consensus-spec-tests	https://github.com/ethereum/consensus-spec-tests
hive	https://github.com/ethereum/hive
Bug Bounty (scope = the client list)	https://ethereum.org/en/bug-bounty/

A.3 Protocol Security's own work

Item	URL
fuzztools (struct-aware framework; Rakoon; Noiruzz)	https://github.com/protocol-security/fuzztools
The triage is the product (Nikos Baxevanis, 9 Jul 2026)	https://blog.ethereum.org/2026/07/09/triage-is-the-product
better.codes (Formal Verification team, 20 Aug 2026)	https://blog.ethereum.org/2026/08/20/better-codes-challenge
Bhargava Shastry — differential testing work	https://bshastry.github.io

Item	URL
CVE-2025-30147 (Besu subgroup check)	https://blog.ethereum.org/2025/05/07/the-curious-case

A.4 Ecosystem fuzzers referenced

Tool	URL
tx-fuzz	https://github.com/MariusVanDerWijden/tx-fuzz
FuzzyVM (differential EVM fuzzer)	https://github.com/MariusVanDerWijden/FuzzyVM
beacon-fuzz *(archived)*	https://github.com/sigp/beacon-fuzz
Echidna	https://github.com/crytic/echidna
Medusa	https://github.com/crytic/medusa
Slither *(static analysis)*	https://github.com/crytic/slither
Halmos	https://github.com/a16z/halmos
Kontrol	https://github.com/runtimeverification/kontrol
ItyFuzz	https://github.com/fuzzland/ityfuzz

A.5 Attribution

The two EF engineers whose published work is cited above, both writing under their own names: **Nikos Baxevanis** (`moodmosaic`), author of the July 2026 post on agent-driven auditing, and **Bhargava Shastry** (`bshastry`), whose differential-testing work is summarised in §4. `protocol-security/fuzztools` was created in October 2025 and last pushed in April 2026.

A.6 Caveats

- The Bhargava Shastry findings and publication list come from his own site, which is self-reported (though the underlying Solidity and client fixes are public). Treat the specific bug descriptions as indicative.
- The Hedgehog-versus-QuickCheck shrinking comparison in Chapter 1 is presented as a design distinction, which is well established. Quantitative comparisons between the two exist but are contested; do not quote figures.
- `beacon-fuzz` is archived. Mention it as history if at all, not as live tooling.
- Guido Vranken's early eth2 differential fuzzing work is referenced in `beacon-fuzz`'s lineage; his original `cryptofuzz` repository no longer resolves at the commonly cited URL, so no link is given.