Property-Based Testing and Fuzzing for Cryptographic Software

Zach Kelling

Lux Industries

2025

Abstract

Cryptographic code has a unique testing challenge: correctness is not about producing the right output for a few test vectors—it is about producing the right output for all inputs, and never leaking information through side channels, timing, or error paths. Property-based testing and fuzzing address this by generating thousands to millions of random inputs and checking that mathematical properties hold. This paper teaches both techniques from first principles and applies them to Lux Network's cryptographic implementations: Go's native fuzzing for NTT and polynomial arithmetic, Foundry fuzz for smart contract invariants, and Halmos symbolic testing for FHE noise budget properties. We cover property selection, coverage-guided mutation, corpus management, shrinking failing inputs, and interpreting coverage metrics. The result is a methodology that has found 23 bugs in production cryptographic code—bugs that unit tests and code review missed.

Contents

1 Why Fuzz Cryptographic Code

1.1 The Problem with Test Vectors

Cryptographic implementations are typically tested against known-answer tests (KATs): standardized input/output pairs from the specification. KATs verify that the code produces correct output for specific inputs. They do not verify:

- Behavior at boundary values (zero, one, $q-1$, q, $q+1$).
- Behavior with malformed inputs (wrong length, invalid encoding).
- Absence of panics or undefined behavior.
- Consistency: $\text{decrypt}(\text{encrypt}(m)) = m$ for all m.
- Algebraic properties: associativity, commutativity, distributivity.

1.2 Property-Based Testing

Property-based testing (PBT) inverts the test structure. Instead of specifying inputs and expected outputs, you specify *properties* that must hold for all inputs. The framework generates random inputs and checks the properties.

Definition 1 (Property)**.** *A property is a universally quantified assertion: $\forall x \in D.\ P(x)$, where D is the input domain and P is a predicate.*

Examples of cryptographic properties:

- **Round-trip**: $\text{decrypt}(k, \text{encrypt}(k, m)) = m$
- **Determinism**: $\text{hash}(x) = \text{hash}(x)$
- **Length preservation**: $|\text{NTT}(a)| = |a|$
- **Commutativity**: $\text{NTT}(a \cdot b) = \text{NTT}(a) \otimes \text{NTT}(b)$
- **Idempotence**: $\text{NTT}(\text{INTT}(\text{NTT}(a))) = \text{NTT}(a)$

1.3 Fuzzing vs. Property-Based Testing

	Property-Based Testing	**Fuzzing**
Input generation	Type-aware random	Mutational, coverage-guided
Oracle	Explicit property	Crash / sanitizer
Shrinking	Yes (minimal failing input)	Corpus minimization
Coverage feedback	Usually no	Yes
Typical runs	1,000–100,000	Millions–billions

Table 1: PBT vs. fuzzing. Modern tools blend both approaches.

Go's `testing.F` provides coverage-guided fuzzing with type-aware seed corpora. Foundry provides both random (fuzz) and coverage-guided (invariant) modes. We use all of them.

2 Go Fuzzing for Cryptographic Primitives

2.1 The Go Fuzzing Framework

Go 1.18 introduced native fuzzing in the `testing` package. A fuzz test is a function that accepts `*testing.F` and calls `f.Fuzz`:

Listing 1: Basic Go fuzz test for NTT round-trip

```
1 func FuzzNTTRoundTrip(f *testing.F) {
2     // Seed corpus: known interesting inputs
```

```
    f.Add([]byte{0, 0, 0, 0})
    f.Add([]byte{255, 255, 255, 255})

    f.Fuzz(func(t *testing.T, data []byte) {
        if len(data) < PolyDegree*4 {
            t.Skip()
        }

        // Decode input as polynomial coefficients
        poly := decodePoly(data[:PolyDegree*4])

        // Apply NTT then INTT
        nttPoly := NTT(poly)
        recovered := INTT(nttPoly)

        // Property: round-trip is identity
        for i := range poly {
            if recovered[i] != poly[i] {
                t.Fatalf("NTT round-trip failed at index %d: "+
                    "got %d, want %d", i, recovered[i], poly[i])
            }
        }
    })
}
```

2.2 Running Go Fuzz Tests

Listing 2: Running Go fuzz tests

```
# Run fuzz test for 60 seconds
go test -fuzz FuzzNTTRoundTrip -fuzztime 60s

# Run with race detector (slower but catches data races)
go test -fuzz FuzzNTTRoundTrip -fuzztime 60s -race

# Run with address sanitizer (catches memory bugs in cgo)
CGO_ENABLED=1 go test -fuzz FuzzNTTRoundTrip -fuzztime 60s \
    -asan

# Reproduce a failing input from the corpus
go test -run FuzzNTTRoundTrip/corpus_entry_name
```

2.3 Coverage-Guided Mutation

Go's fuzzer uses coverage-guided mutation: it instruments the code to track which branches are taken, then mutates inputs to explore new branches. This is critical for cryptographic code where interesting behavior often hides behind conditional branches (e.g., modular reduction triggered only when a coefficient exceeds q).

The fuzzer maintains a *corpus*: a set of inputs that collectively maximize branch coverage. New inputs are added to the corpus only if they cover a previously uncovered branch.

2.4 Properties for Polynomial Arithmetic

Listing 3: Property-based fuzz tests for polynomial arithmetic

```
// Commutativity: a * b == b * a
func FuzzPolyMulCommutative(f *testing.F) {
    f.Fuzz(func(t *testing.T, seed1, seed2 []byte) {
        a := polyFromSeed(seed1)
        b := polyFromSeed(seed2)

        ab := PolyMul(a, b)
        ba := PolyMul(b, a)

        requirePolyEqual(t, ab, ba)
    })
}

// Distributivity: a * (b + c) == a*b + a*c
func FuzzPolyMulDistributive(f *testing.F) {
    f.Fuzz(func(t *testing.T, s1, s2, s3 []byte) {
        a := polyFromSeed(s1)
        b := polyFromSeed(s2)
        c := polyFromSeed(s3)

        lhs := PolyMul(a, PolyAdd(b, c))
        rhs := PolyAdd(PolyMul(a, b), PolyMul(a, c))

        requirePolyEqual(t, lhs, rhs)
    })
}

// NTT preserves multiplication:
// NTT(a * b) == NTT(a) . NTT(b) (pointwise)
func FuzzNTTHomomorphism(f *testing.F) {
    f.Fuzz(func(t *testing.T, s1, s2 []byte) {
        a := polyFromSeed(s1)
        b := polyFromSeed(s2)

        // Multiply then transform
        ab := PolyMul(a, b)
        nttAB := NTT(ab)

        // Transform then pointwise multiply
        nttA := NTT(a)
        nttB := NTT(b)
        nttProd := PointwiseMul(nttA, nttB)

        requirePolyEqual(t, nttAB, nttProd)
    })
}
```

2.5 FHE Noise Budget Properties

FHE (Fully Homomorphic Encryption) ciphertexts carry a *noise budget*: each operation increases noise, and when the budget is exhausted, decryption fails. We fuzz test noise properties:

Listing 4: FHE noise budget fuzz test

```
func FuzzFHENoiseGrowth(f *testing.F) {
    f.Fuzz(func(t *testing.T, plaintext1, plaintext2 uint64) {
        // Constrain to valid plaintext space
        p1 := plaintext1 % PlaintextModulus
```

```
        p2 := plaintext2 % PlaintextModulus

        // Encrypt
        ct1 := Encrypt(publicKey, p1)
        ct2 := Encrypt(publicKey, p2)

        // Add ciphertexts
        ctSum := Add(ct1, ct2)

        // Noise must not exceed budget
        noise := EstimateNoise(ctSum, secretKey)
        if noise > NoiseBudget {
            t.Fatalf("noise budget exceeded after addition: "+
                "noise=%d, budget=%d", noise, NoiseBudget)
        }

        // Decrypt must succeed
        result := Decrypt(secretKey, ctSum)
        expected := (p1 + p2) % PlaintextModulus
        if result != expected {
            t.Fatalf("decryption failed: got %d, want %d",
                result, expected)
        }
    })
}
```

3 Foundry Fuzz Testing for Smart Contracts

3.1 Random Fuzz Testing

Foundry's fuzz mode generates random inputs for each test parameter:

Listing 5: Foundry fuzz test with bound

```
function testFuzz_swapNeverIncreasesK(
    uint256 amountIn
) public {
    amountIn = bound(amountIn, 1, pool.reserve0() / 2);

    uint256 kBefore = pool.reserve0() * pool.reserve1();
    pool.swap(amountIn, 0, address(this));
    uint256 kAfter = pool.reserve0() * pool.reserve1();

    assertGe(kAfter, kBefore);
}
```

Key functions:

- `bound(x, min, max)`: Clamps `x` to $[\texttt{min}, \texttt{max}]$. Unlike `vm.assume`, this does not discard inputs—it transforms them.
- `vm.assume(cond)`: Discards the input if `cond` is false. Use sparingly: too many rejections waste fuzz cycles.

3.2 Stateful Invariant Testing

Invariant testing is the most powerful fuzz technique for smart contracts. Foundry generates random sequences of function calls (actions) and checks invariants after each action.

The key insight: bugs often require specific sequences of operations. A deposit followed by a withdraw is fine; a deposit, a price change, and then a withdraw may violate an invariant. Invariant testing explores these sequences.

3.3 Coverage Metrics

Listing 6: Foundry coverage report

```
$ forge coverage
| File                | % Lines | % Stmts | % Branch | % Funcs |
|---------------------|---------|---------|----------|---------|
| src/Pool.sol        |  96.2%  |  94.8%  |  91.3%   |  100%   |
| src/Vault.sol       |  93.7%  |  92.1%  |  88.9%   |  97.4%  |
| src/Bridge.sol      |  95.1%  |  93.4%  |  90.2%   |  100%   |
| src/Token.sol       |  98.3%  |  97.6%  |  95.1%   |  100%   |
```

We require $\geq$85% branch coverage. Uncovered branches must be justified (e.g., unreachable error paths, platform-specific code).

4 Halmos Symbolic Testing

Where fuzzing samples the input space randomly, Halmos explores it exhaustively (within loop bounds). We use Halmos for the strongest properties where fuzzing alone is insufficient:

Listing 7: Halmos symbolic test for access control

```
function check_onlyOwnerCanSetFee(
    address caller,
    uint256 newFee
) public {
    vm.assume(caller != owner);
    vm.prank(caller);

    // Must revert for non-owner
    try pool.setFee(newFee) {
        assert(false); // Should not reach here
    } catch {
        // Expected: revert
    }
}
```

4.1 Complementary Techniques

Property Type	Foundry Fuzz	Foundry Invariant	Halmos
Single-tx invariant	Good	N/A	Best
Multi-tx invariant	N/A	Good	N/A
Edge case detection	Good	Good	Best
Sequence-dependent	N/A	Best	N/A
Proof guarantee	No	No	Yes

Table 2: When to use each technique.

5 Bugs Found

Bug	Component	Technique	Severity
NTT coefficient overflow at $q-1$	Pulsar	Go fuzz	High
INTT missing final reduction	NTT engine	Go fuzz	High
Montgomery mul wrong for $a=0$	NTT engine	Go fuzz	Medium
Polynomial add modular wrap	Polynomial lib	Go fuzz	Medium
FHE noise undercount for $n=1$	FHE core	Go fuzz	High
Vault share rounding to zero	Yield vault	Foundry fuzz	Critical
AMM fee bypass at max uint	AMM pool	Foundry fuzz	High
Bridge nonce gap on revert	Teleport	Foundry invariant	High
Staking reward overflow	Staking	Foundry fuzz	Medium
Token approve race condition	ERC-20	Halmos	Low
... 13 more findings ...			
Total			**23**

Table 3: Selected bugs found by fuzzing and property-based testing.

6 Shrinking and Reproducibility

When a fuzzer finds a failing input, it is typically large and random. *Shrinking* reduces the input to the smallest value that still triggers the failure.

Go's fuzzer performs automatic shrinking: it binary-searches the input space for simpler inputs that still fail. Foundry does the same. The result is a minimal reproducer that is easy to understand and debug.

All failing inputs are saved to the corpus directory. Running the test suite without the `-fuzz` flag replays all corpus entries, ensuring that regressions are caught immediately.

Listing 8: Corpus directory structure

```
testdata/fuzz/FuzzNTTRoundTrip/
    corpus/
        seed_0              # Initial seed
        seed_1              # Another seed
        abc123def456...     # Auto-generated corpus entry
    failing/
        xyz789...           # Minimized failing input
```

7 Coverage-Guided Fuzzing Internals

7.1 How Coverage Guidance Works

1. The fuzzer instruments the binary to track which basic blocks (or edges) are executed.
2. For each input, the fuzzer runs the code and records the set of covered edges.
3. If an input covers a new edge (one not covered by any previous input in the corpus), the input is added to the corpus.
4. New inputs are generated by mutating corpus entries (bit flips, byte insertions, arithmetic operations on integer fields).

7.2 Why It Matters for Crypto

Cryptographic code has many conditional branches that depend on input values:

- Modular reduction: triggered when $a \geq q$.

- Carry propagation: triggered by specific bit patterns.
- Branch-free constant-time code: still has compiler-generated branches.

Coverage-guided fuzzing systematically explores these branches, while random testing may miss rare conditions.

8 Conclusion

Property-based testing and fuzzing are the most cost-effective techniques for finding bugs in cryptographic code. They require minimal setup—write a property, run the fuzzer—and find bugs that unit tests and code review miss. Our methodology, applied to Lux Network's cryptographic stack (Go NTT, polynomial arithmetic, FHE, smart contracts), found 23 bugs including 3 critical issues.

The key insight: write properties, not test cases. A property like "NTT round-trip is identity" tests more effectively than a thousand hand-picked test vectors, because the fuzzer explores the input space guided by code coverage rather than human intuition.

Start with the round-trip property. If your code has an encode/decode, encrypt/decrypt, or NTT/INTT pair, write the round-trip fuzz test first. It takes five minutes and catches the most common class of bugs.

References

[1] Go Team. "Go Fuzzing." The Go Blog, 2022.

[2] Paradigm. "Foundry: A Blazing Fast, Portable and Modular Toolkit for Ethereum Application Development." 2022.

[3] Park, D. et al. "Halmos: Symbolic Testing for EVM Smart Contracts." a16z Crypto, 2023.

[4] Claessen, K. and Hughes, J. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs." ICFP, 2000.

[5] Zalewski, M. "American Fuzzy Lop (AFL)." 2014.

[6] LLVM Project. "libFuzzer: A Library for Coverage-Guided Fuzz Testing." 2015.