STUNIR Testing Strategy

This document outlines the overall testing approach, test categories, coverage goals, and roadmap for test expansion.

Overview

STUNIR employs a multi-layer testing strategy to ensure correctness, determinism, and security across all components.

Testing Pyramid

```
                    ┌─────────────┐
                    │  E2E Tests  │  ← Full pipeline verification
                    ├─────────────┤
                    │ Integration │  ← Component interaction
             ┌──────┴─────────────┴──────┐
             │        Unit Tests         │  ← Individual functions
        ┌────┴───────────────────────────┴────┐
        │           Security Tests            │  ← Cross-cutting
        └─────────────────────────────────────┘
```

Test Categories

1. Unit Tests

Purpose: Test individual functions and modules in isolation.

Languages:
- **Rust:** `cargo test` in `tools/native/rust/stunir-native/tests/`
- **Python:** `pytest` in `tests/`
- **Haskell:** `cabal test` (future)

Coverage Target: 80%+

Key Areas:
- Cryptographic functions (SHA-256, Merkle trees)
- JSON canonicalization
- IR parsing and validation
- Serialization/deserialization
- Manifest generation

2. Integration Tests

Purpose: Test interaction between components.

Location: `tests/integration/`

Coverage Target: Critical paths

Key Scenarios:
- Spec → IR → Targets pipeline

- Manifest generation and verification
- Receipt creation and validation
- Multi-target generation

3. End-to-End Tests

Purpose: Verify complete workflows from input to output.

Key Workflows:
- Full build pipeline execution
- Cross-tool verification
- Determinism verification

4. Security Tests

Purpose: Verify security properties and catch vulnerabilities.

Location: `tests/security/`

Key Areas:
- Input validation
- Path traversal prevention
- Hash collision resistance
- Symlink handling

5. Determinism Tests

Purpose: Ensure reproducible outputs.

Key Properties:
- Same input → same output
- Cross-platform consistency
- Stable serialization

Coverage Goals

By Module

Module	Current	Target	Priority
crypto	70%	90%	High
ir_v1	60%	85%	High
canonical	80%	95%	High
manifests	50%	80%	Medium
emitters	40%	75%	Medium
tools	30%	70%	Medium

By Category

Category	Tests	Status
Rust Unit	25+	✅ Implemented
Python Unit	10+	🟡 Partial
Integration	10+	✅ Implemented
Security	5+	✅ Implemented
E2E	3+	🟡 Planned

Test Infrastructure

CI/CD Integration

```
# GitHub Actions workflows
.github/workflows/
├── ci.yml        # Main CI - runs on every push/PR
├── security.yml  # Security scans - weekly + on push
└── docs.yml      # Documentation checks
```

Test Frameworks

Language	Framework	Runner
Rust	built-in	`cargo test`
Python	pytest	`pytest`
Haskell	HUnit/QuickCheck	`cabal test`

Test Utilities

Rust: `tests/common/mod.rs`
- Temp directory creation
- Test file generation
- Sample data constants
- Hash computation helpers

Python: `tests/integration/utils.py`
- JSON utilities
- Hash computation
- Verification helpers
- Test data generators

Determinism Verification

Requirements

All STUNIR outputs must be deterministic:

1. **Canonical JSON:** Keys sorted, no extra whitespace
2. **Stable hashes:** Same content = same hash
3. **Ordered iteration:** BTreeMap in Rust, sorted() in Python
4. **No timestamps in core outputs:** Timestamps only in metadata

Verification Tests

```python
def test_determinism():
    results = [generate_output(input) for _ in range(5)]
    assert all(r == results[0] for r in results)
```

Roadmap for Test Expansion

Phase 1: Foundation (Current)

- [x] Rust unit test framework
- [x] Python integration test framework
- [x] CI/CD pipeline
- [x] Security test basics

Phase 2: Coverage Expansion

- [ ] Increase Rust coverage to 90%
- [ ] Add property-based tests (QuickCheck)
- [ ] Add fuzzing for parsers
- [ ] Cross-platform testing

Phase 3: Advanced Testing

- [ ] Performance benchmarks
- [ ] Memory leak detection
- [ ] Concurrency tests
- [ ] Chaos engineering

Phase 4: Compliance

- [ ] Formal verification (selected modules)
- [ ] Audit trail testing
- [ ] Regulatory compliance checks

Writing Effective Tests

Test Naming Convention

```
test_<module>_<scenario>_<expected_result>

Examples:
- test_hash_file_empty_returns_known_hash
- test_parse_spec_invalid_json_fails
- test_manifest_generation_deterministic
```

Test Structure (AAA Pattern)

```rust
#[test]
fn test_feature() {
    // Arrange - Set up test data
    let input = create_test_input();

    // Act - Execute the code under test
    let result = function_under_test(input);

    // Assert - Verify the results
    assert!(result.is_ok());
    assert_eq!(result.unwrap(), expected_value);
}
```

Critical Properties to Test

1. **Determinism:** Same input always produces same output
2. **Idempotency:** Repeated operations have no additional effect
3. **Reversibility:** Serialize/deserialize roundtrips
4. **Error handling:** Invalid inputs are rejected gracefully
5. **Edge cases:** Empty inputs, large inputs, special characters

Running Tests

Quick Verification

```bash
# Run all Rust tests
cd tools/native/rust/stunir-native && cargo test

# Run all Python tests
pytest tests/ -v

# Run integration tests only
pytest tests/integration/ -v

# Run security tests only
pytest tests/security/ -v
```

Full Test Suite

```
# Run everything with coverage
pytest tests/ -v --cov=tools --cov=manifests --cov-report=html

# Run Rust with verbose output
cargo test -- --nocapture
```

CI Simulation

```
# Simulate CI workflow locally
act -j python-test  # Requires 'act' tool
```

Test Documentation

- **Rust Tests:** `tools/native/rust/stunir-native/TESTING.md`
- **Integration Tests:** `tests/integration/README.md`
- **Security Tests:** `tests/security/README.md`
- **CI/CD:** `.github/workflows/README.md`

Troubleshooting

Common Issues

1. **Tests fail locally but pass in CI**
   - Check Python version compatibility
   - Verify dependencies are installed

2. **Flaky tests**
   - Check for timing dependencies
   - Ensure proper test isolation

3. **Coverage gaps**
   - Use `--cov-report=term-missing` to find uncovered lines
   - Prioritize high-risk code paths

Contact

For questions about testing:
- Review existing tests for patterns
- Check CI logs for failure details
- Consult SECURITY.md for security test requirements