STUNIR User Guide

A comprehensive guide to using the STUNIR (Structured Toolchain for Unified Native IR) system.

Table of Contents

Introduction

What is STUNIR?

STUNIR is a deterministic build and verification toolchain that ensures reproducible outputs across different platforms and runtimes. It provides:

- **Deterministic Builds**: Same inputs always produce same outputs
- **Cryptographic Verification**: SHA-256 hashing of all artifacts
- **Multi-Language Support**: Python, Rust, Haskell, and C tooling
- **Manifest Management**: Track and verify all build artifacts
- **Security First**: Built-in input validation and sanitization

Key Concepts

Concept	Description
Spec	Input specification defining modules and source code
IR	Intermediate Representation - normalized, target-agnostic format
Receipt	Cryptographic proof linking build inputs to outputs
Manifest	Index of artifacts with their hashes
Target	Output code in a specific language (Rust, C, etc.)

Installation

Prerequisites

- **Python**: 3.8 or later
- **Rust**: 1.70 or later (for native tools)
- **Haskell**: GHC 9.0 or later (optional, for Haskell tools)

Install from Source

```
# Clone the repository
git clone https://github.com/your-org/stunir.git
cd stunir

# Install Python dependencies
pip install -r requirements.txt

# Build Rust native tools
cd tools/native/rust/stunir-native
cargo build --release

# Add to PATH
export PATH="$PATH:$(pwd)/target/release"
```

Verify Installation

```
# Check Python tools
python -c "from tools.ir_emitter import emit_ir; print('Python OK')"

# Check Rust tools
stunir-native --version

# Run tests
pytest tests/ -v
```

Quick Start

5-Minute Tutorial

Step 1: Create a Spec File

Create `example_spec.json`:

```
{
  "kind": "spec",
  "modules": [
    {
      "name": "hello",
      "source": "print('Hello, STUNIR!')",
      "lang": "python"
    }
  ],
  "metadata": {
    "author": "Your Name",
    "version": "1.0.0"
  }
}
```

Step 2: Generate IR

```
python -m tools.ir_emitter.emit_ir example_spec.json example_ir.json
```

Output:

```
Generated IR for module: hello
Output written to: example_ir.json
SHA-256: abc123...
```

Step 3: Verify the IR

```
# Hash the IR file
stunir-native hash example_ir.json
```

Step 4: Generate Manifest

```
# Create receipts directory
mkdir -p receipts

# Generate manifest
python -m manifests.ir.gen_ir_manifest --output receipts/ir_manifest.json
```

Step 5: Verify Build

```
python -m manifests.ir.verify_ir_manifest receipts/ir_manifest.json
```

Basic Workflow

The STUNIR Pipeline

```
┌─────────────┐     ┌─────────────┐     ┌─────────────┐     ┌─────────────┐
│    Spec     │ --> │     IR      │ --> │   Target    │ --> │  Manifest   │
│   (JSON)    │     │   (JSON)    │     │   (Code)    │     │   (JSON)    │
└─────────────┘     └─────────────┘     └─────────────┘     └─────────────┘
      │                   │                   │                   │
      v                   v                   v                   v
  Parse &             Emit &             Generate           Verify &
  Validate         Canonicalize            Code               Sign
```

Step-by-Step

1. Write Specification

```
{
  "kind": "spec",
  "modules": [
    {
      "name": "calculator",
      "source": "def add(a, b): return a + b",
      "lang": "python"
    },
    {
      "name": "utils",
      "source": "def log(msg): print(f'[LOG] {msg}')",
      "lang": "python"
    }
  ]
}
```

2. Convert to IR

```
python -m tools.ir_emitter.emit_ir spec.json ir.json
```

The IR normalizes the spec into a target-agnostic format:

```json
{
  "kind": "ir",
  "generator": "stunir-python",
  "ir_version": "v1",
  "module_name": "calculator",
  "functions": [
    {
      "name": "add",
      "body": [
        {"op": "return", "args": ["a + b"]}
      ]
    }
  ]
}
```

3. Generate Target Code

```bash
# Generate Rust code
python -m tools.emitters.emit_code --target rust ir.json output/

# Generate C code
python -m tools.emitters.emit_code --target c99 ir.json output/
```

4. Generate Receipts

```bash
# Run build with receipt generation
./scripts/build.sh --receipts
```

5. Verify Build

```bash
# Strict verification
./scripts/verify_strict.sh --strict
```

Common Use Cases

Use Case 1: Reproducible CI/CD Builds

```yaml
# .github/workflows/build.yml
name: STUNIR Build

on: [push, pull_request]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Setup STUNIR
        run: |
          pip install -r requirements.txt
          cd tools/native/rust/stunir-native
          cargo build --release

      - name: Build with Receipts
        run: ./scripts/build.sh --receipts

      - name: Verify Build
        run: ./scripts/verify_strict.sh --strict

      - name: Upload Receipts
        uses: actions/upload-artifact@v4
        with:
          name: build-receipts
          path: receipts/
```

Use Case 2: Multi-Target Code Generation

```
#!/bin/bash
# generate_all_targets.sh

SPEC="myproject/spec.json"
IR="build/ir.json"
TARGETS="rust c89 c99 x86 arm wasm"

# Generate IR
python -m tools.ir_emitter.emit_ir "$SPEC" "$IR"

# Generate all targets
for target in $TARGETS; do
    echo "Generating $target..."
    python -m tools.emitters.emit_code \
        --target "$target" \
        "$IR" \
        "build/targets/$target/"
done

# Generate manifest
python -m manifests.targets.gen_targets_manifest \
    --targets-dir build/targets/ \
    --output receipts/targets_manifest.json

echo "Done! All targets generated."
```

Use Case 3: Verify Third-Party Builds

```python
#!/usr/bin/env python3
"""Verify a third-party STUNIR build."""

from manifests.ir.verify_ir_manifest import IrManifestVerifier
from tools.errors import VerificationError

def verify_build(manifest_path: str, artifacts_dir: str) -> bool:
    verifier = IrManifestVerifier()

    try:
        result = verifier.verify(manifest_path, artifacts_dir)

        if result.valid:
            print("✓ Build verified successfully")
            print(f"  Artifacts: {result.artifact_count}")
            print(f"  Manifest hash: {result.manifest_hash[:16]}...")
            return True
        else:
            print("✗ Verification failed")
            for error in result.errors:
                print(f"  - {error}")
            return False

    except VerificationError as e:
        print(f"✗ Error: {e.message}")
        print(f"  Suggestion: {e.suggestion}")
        return False

if __name__ == "__main__":
    import sys
    success = verify_build(sys.argv[1], sys.argv[2])
    sys.exit(0 if success else 1)
```

Use Case 4: Input Validation

```python
#!/usr/bin/env python3
"""Validate user-provided spec files."""

from tools.security.validation import (
    InputValidator,
    validate_json_input,
    validate_identifier,
)
from tools.errors import ValidationError, ErrorHandler

def validate_spec_file(path: str) -> bool:
    handler = ErrorHandler(verbose=True)
    validator = InputValidator(max_file_size=10*1024*1024)  # 10MB limit

    # Validate file path
    file_result = validator.validate_file(path)
    if not file_result.valid:
        for error in file_result.errors:
            handler.collect(ValidationError("E3000", error, path=path))
        handler.report()
        return False

    # Validate JSON content
    json_result = validator.validate_json_file(path)
    if not json_result.valid:
        for error in json_result.errors:
            handler.collect(ValidationError("E2001", error, path=path))
        handler.report()
        return False

    spec = json_result.value

    # Validate spec structure
    if spec.get("kind") != "spec":
        handler.collect(ValidationError(
            "E3003",
            "Invalid spec kind",
            field="kind",
            value=spec.get("kind"),
            expected="spec"
        ))

    # Validate module names
    for i, module in enumerate(spec.get("modules", [])):
        name_result = validate_identifier(module.get("name", ""))
        if not name_result.valid:
            for error in name_result.errors:
                handler.collect(ValidationError(
                    "E3001",
                    error,
                    field=f"modules[{i}].name",
                    value=module.get("name")
                ))

    if handler.has_errors():
        handler.report()
        return False

    print("✓ Spec file is valid")
    return True

if __name__ == "__main__":
    import sys
    validate_spec_file(sys.argv[1])
```

Configuration

Environment Variables

Variable	Description	Default
`STUNIR_MAX_FILE_SIZE`	Maximum file size in bytes	100MB
`STUNIR_MAX_JSON_DEPTH`	Maximum JSON nesting depth	50
`STUNIR_RECEIPTS_DIR`	Directory for build receipts	`receipts/`
`STUNIR_VERBOSE`	Enable verbose output	`false`

Configuration File

Create `.stunir.json` in your project root:

```
{
  "version": "1.0",
  "receipts_dir": "build/receipts",
  "targets": ["rust", "c99"],
  "validation": {
    "max_file_size": 52428800,
    "max_json_depth": 30,
    "allow_symlinks": false
  },
  "security": {
    "strict_mode": true,
    "verify_on_build": true
  }
}
```

Troubleshooting

Common Issues

Error: E1001 - File Not Found

Problem: Required file does not exist.

```
[E1001] IO Error: spec.json: file not found
```

Solution:
1. Check the file path is correct
2. Ensure you’re in the right directory
3. Check file permissions

```
# Verify file exists
ls -la spec.json

# Check current directory
pwd
```

Error: E2001 - Invalid JSON Syntax

Problem: JSON file has syntax errors.

```
[E2001] JSON Error: Invalid JSON syntax: Expecting property name
```

Solution:
1. Validate JSON syntax
2. Check for trailing commas
3. Ensure all strings are quoted

```
# Validate JSON
python -m json.tool spec.json

# Common issues:
# - Trailing commas: {"key": "value",}  ← remove comma
# - Single quotes: {'key': 'value'}  ← use double quotes
# - Unquoted keys: {key: "value"}  ← quote the key
```

Error: E4001 - Hash Mismatch

Problem: File has been modified since manifest was generated.

```
[E4001] Verification Failed: Hash mismatch for output.json
  Expected: abc123...
  Actual: def456...
```

Solution:
1. Check if file was intentionally modified
2. Regenerate manifest if changes are expected
3. Investigate unauthorized changes

```
# Regenerate manifest
python -m manifests.ir.gen_ir_manifest --output receipts/ir_manifest.json

# Or check git for changes
git diff output.json
```

Error: E6001 - Path Traversal Detected

Problem: Input path tries to escape project directory.

```
[E6001] Security Error: Path traversal detected: ../etc/passwd
```

Solution:
1. Use paths within the project directory

2. Use absolute paths if needed
3. Don't include `..` in paths

```
# Instead of:
python script.py ../other/file.json

# Use absolute path:
python script.py /full/path/to/file.json

# Or relative from project:
python script.py data/file.json
```

Debug Mode

Enable debug output for detailed information:

```
# Python tools
STUNIR_VERBOSE=true python -m tools.ir_emitter.emit_ir spec.json ir.json

# Rust tools
RUST_LOG=debug stunir-native hash file.json
```

FAQ

General Questions

Q: What makes STUNIR "deterministic"?

A: STUNIR ensures that:
- JSON output uses canonical form (sorted keys, no extra whitespace)
- File hashes use SHA-256 with consistent encoding
- Timestamps are excluded from artifacts (stored in receipts)
- All tools use the same algorithms across platforms

Q: Can I use STUNIR with existing projects?

A: Yes! STUNIR can be integrated incrementally:
1. Start by adding manifest generation
2. Add verification to your CI/CD pipeline
3. Gradually convert build steps to use STUNIR tools

Q: How do I handle binary files?

A: Binary files are hashed directly without modification. Use the crypto module:

```
from tools.security.validation import compute_sha256

with open("binary.dat", "rb") as f:
    hash_value = compute_sha256(f.read())
```

Performance Questions

Q: How fast is directory hashing?

A: Benchmarks on a typical system:
- 100 files (1KB each): ~10ms
- 1000 files (1KB each): ~100ms
- 10 files (1MB each): ~50ms

Q: Can I hash large directories?

A: Yes, but with limits:
- Maximum 100 directory depth
- Maximum 1GB per file
- Uses streaming for large files

Security Questions

Q: What security measures are built-in?

A: STUNIR includes:
- Path traversal prevention
- Symlink blocking (configurable)
- File size limits
- Input validation
- No shell command injection

Q: How do I report a security issue?

A: See SECURITY.md (../SECURITY.md) for responsible disclosure guidelines.

Next Steps

- API Reference (API_REFERENCE.md) - Complete API documentation
- Architecture (ARCHITECTURE.md) - System design details
- Contributing (../CONTRIBUTING.md) - How to contribute
- Deployment (DEPLOYMENT.md) - Production deployment guide

Last updated: January 2026