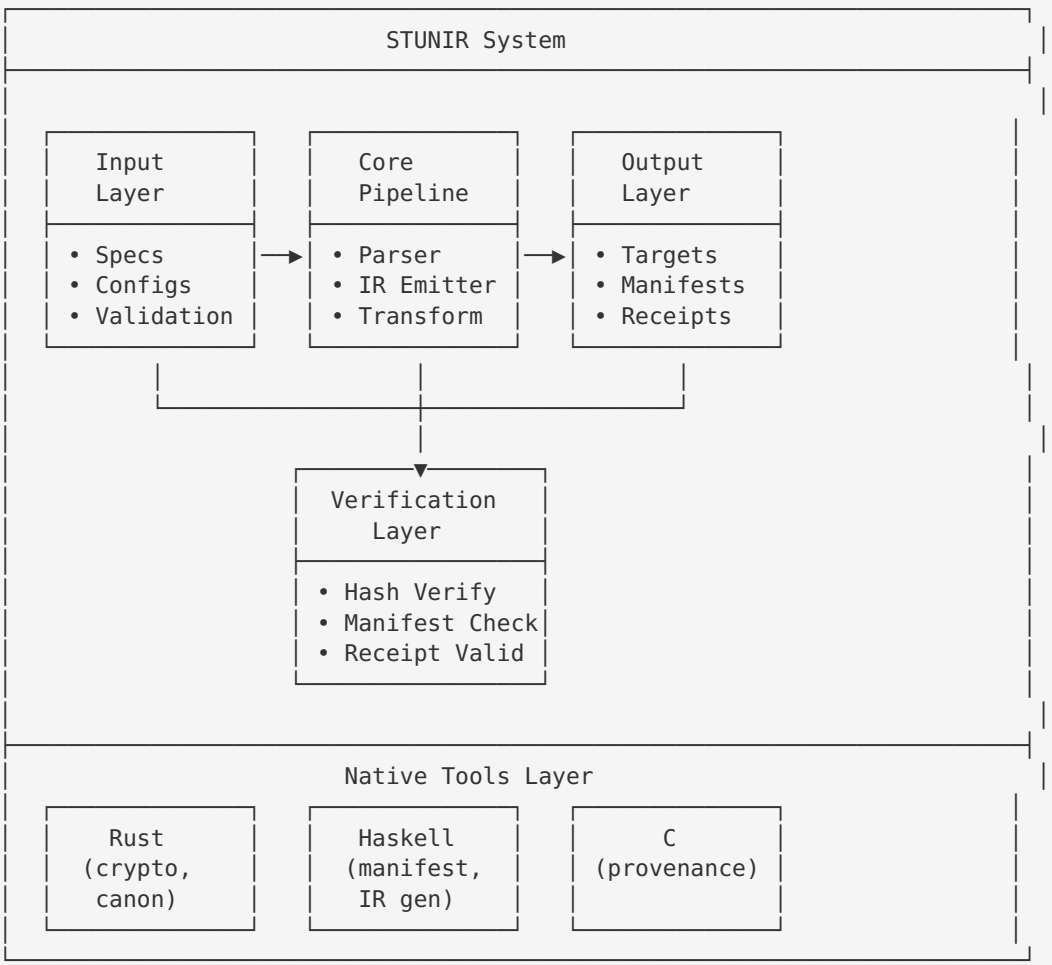
STUNIR Architecture

Technical architecture documentation for the STUNIR (Structured Toolchain for Unified Native IR) system.

Table of Contents

System Overview

High-Level Architecture



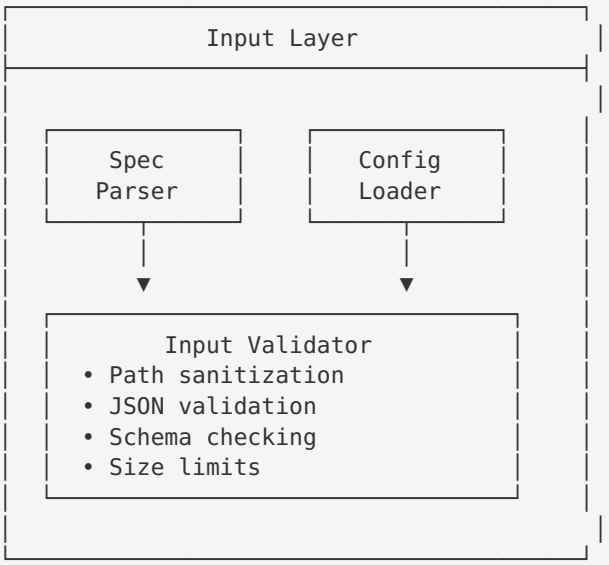
Core Principles

1. **Determinism**: All operations produce identical outputs for identical inputs
2. **Verifiability**: Every artifact can be cryptographically verified
3. **Polyglot**: Support for multiple programming languages
4. **Security**: Input validation and protection against common attacks
5. **Modularity**: Components can be used independently or together

---

Component Architecture

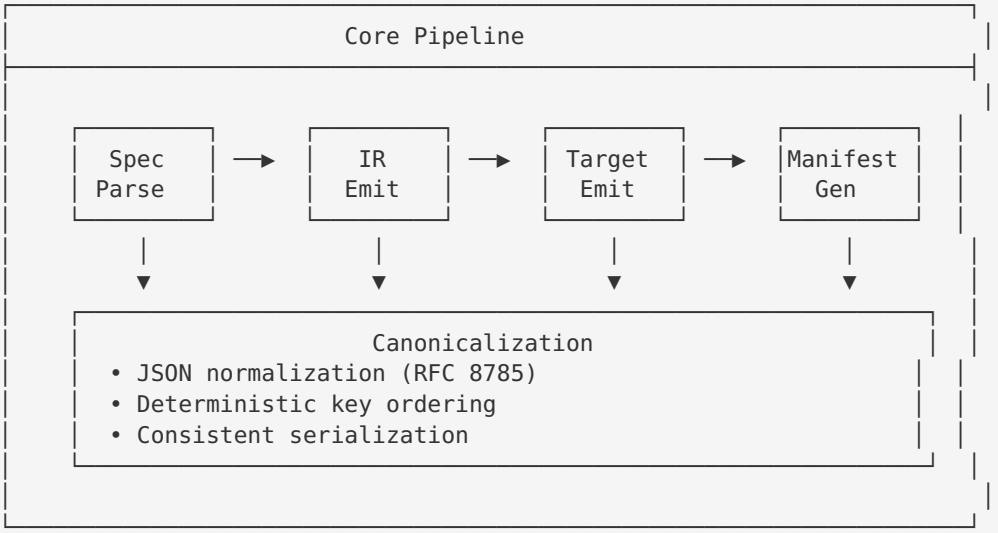
Input Layer



Components:

Component	Responsibility	Implementation
Spec Parser	Parse spec JSON files	Python + Rust
Config Loader	Load configuration	Python
Input Validator	Validate and sanitize inputs	Python (`tools/security/validation.py`)

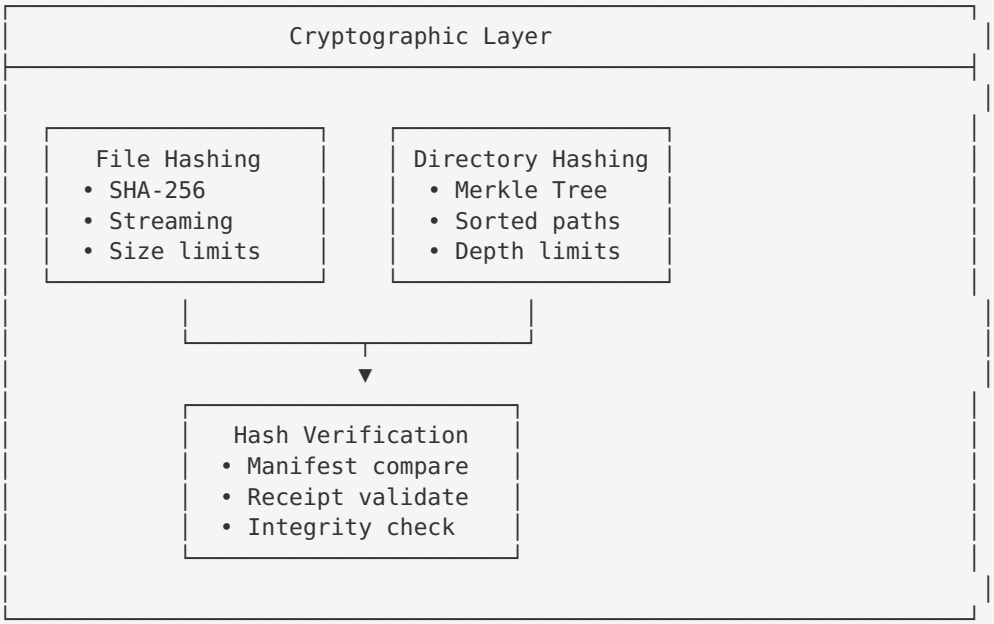
Core Pipeline



Pipeline Stages:

1. **Spec Parse**: Convert input spec to internal representation
2. **IR Emit**: Generate normalized Intermediate Representation
3. **Target Emit**: Generate target-specific code
4. **Manifest Gen**: Create artifact manifests

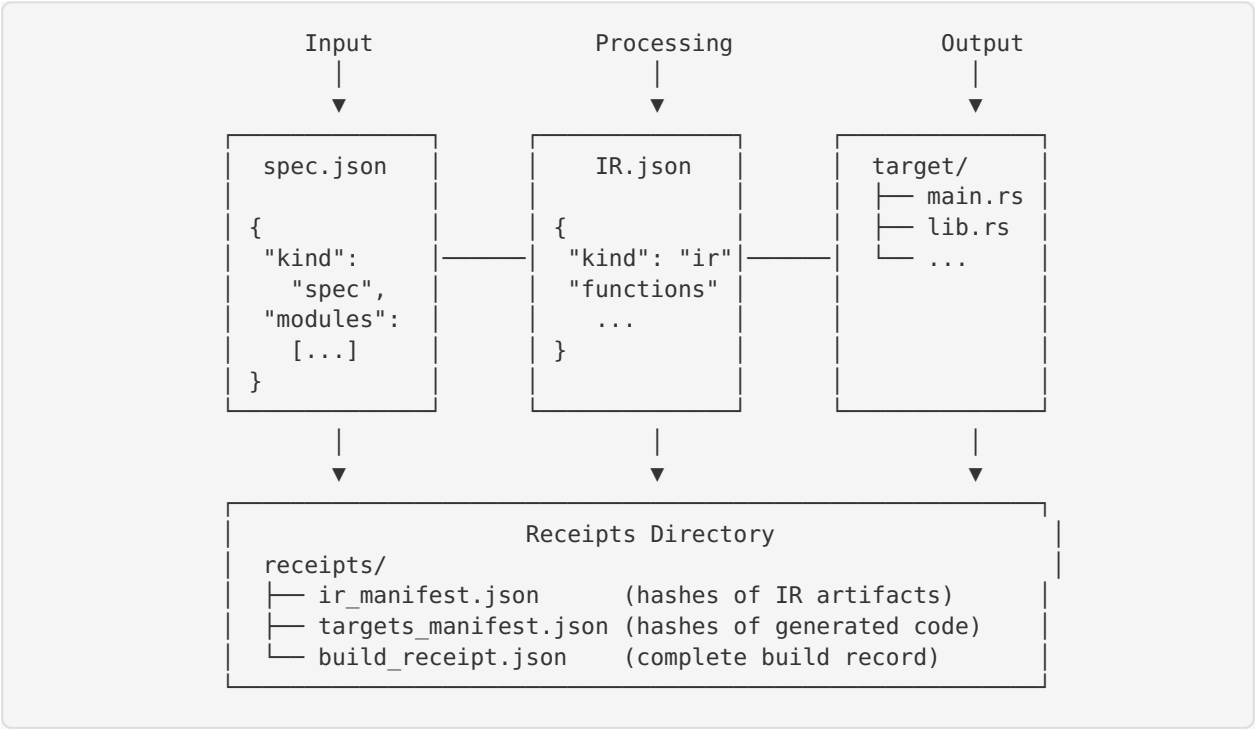
Cryptographic Layer



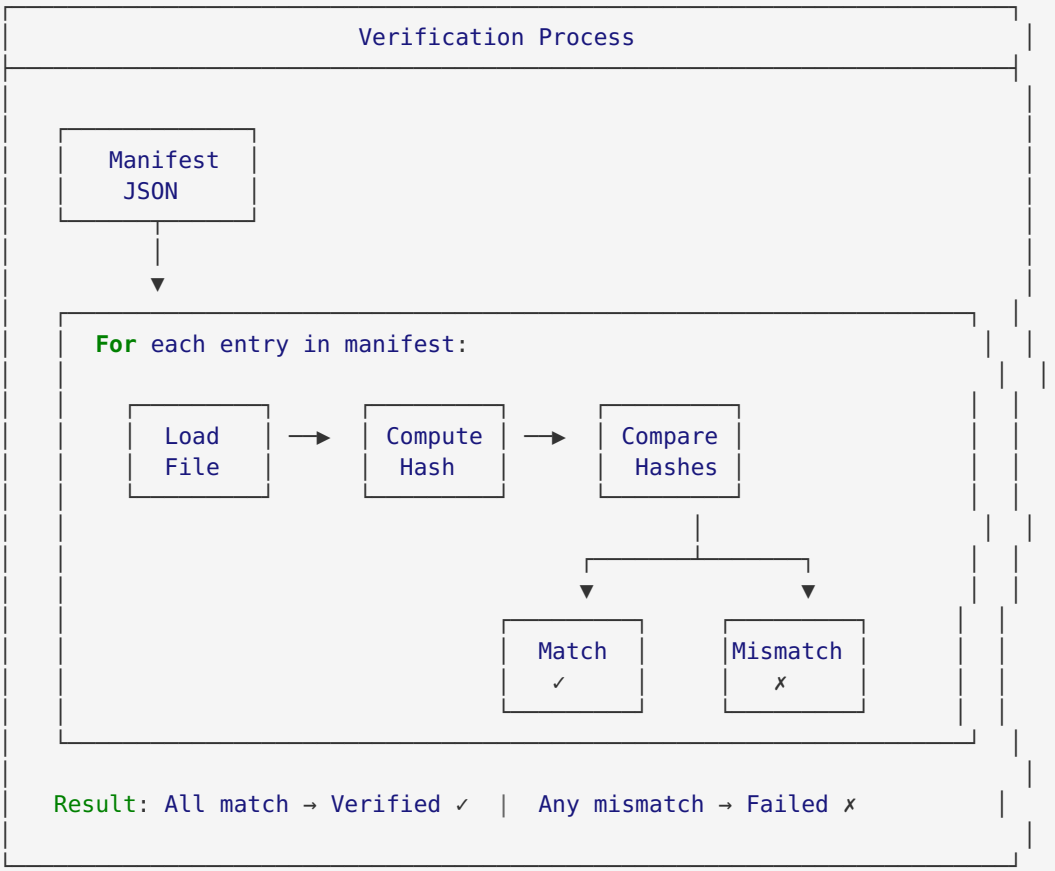
Implementation: Primarily in Rust (`stunir-native`) for performance and memory safety.

Data Flow

Spec to Output Flow



Verification Flow



Design Decisions

Why Multiple Languages?

STUNIR uses multiple implementation languages for specific purposes:

Language	Use Case	Rationale
Python	Orchestration, tooling	Rapid development, ecosystem
Rust	Crypto, performance-critical	Memory safety, speed
Haskell	IR generation, type safety	Strong types, purity
C	Provenance, low-level	Universal compatibility

Why Canonical JSON?

Problem: Standard JSON serialization is non-deterministic (key ordering varies).

Solution: RFC 8785 JSON Canonicalization Scheme (JCS):
- Lexicographically sorted keys
- No insignificant whitespace
- Consistent number formatting
- Minimal string escaping

```
# Non-deterministic (standard)
{"z": 1, "a": 2}  # or {"a": 2, "z": 1} - depends on implementation

# Canonical (STUNIR)
{"a":2,"z":1}     # Always this exact output
```

Why Merkle Trees for Directories?

Benefits:
1. **Partial Verification**: Verify subset without full directory scan
2. **Change Detection**: Quickly identify which files changed
3. **Parallelization**: Hash individual files concurrently
4. **Determinism**: Sorted paths ensure consistent ordering

Structure:

```
Directory Root Hash
         │
         ┴
    ┌────┴────┐
    │         │
  file1     dir1
  (hash)      │
            ┌─┴─┐
          file2   file3
          (hash)  (hash)

Root = SHA256(SHA256(file1) || SHA256(dir1_root))
```

Why Separate Receipts from Manifests?

Aspect	Manifest	Receipt
Purpose	Index of artifacts	Proof of build
Contents	Paths + hashes	Inputs, outputs, tools, time
Mutability	Regenerated on changes	Append-only history
Scope	Single artifact type	Entire build

Technology Stack

Core Technologies

```
┌──────────────────────────────────────────────────────────────────────┐
│                        Technology Stack                              │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│  Layer           Technology              Purpose                     │
│  ──────────────────────────────────────────────────────────────────  │
│  CLI             Python (argparse)       User interface              │
│  Orchestration   Python                  Workflow coordination       │
│  IR Processing   Python + Haskell        Spec → IR transformation    │
│  Cryptography    Rust (sha2, hex)        Hashing, verification       │
│  Serialization   serde (Rust), json (Py) Data encoding/decoding      │
│  Validation      Python (custom)         Input sanitization          │
│  Build           Cargo, Cabal, pip       Package management          │
│  Testing         pytest, cargo test      Automated testing           │
│                                                                      │
└──────────────────────────────────────────────────────────────────────┘
```

Dependencies

Python:
- `hashlib` - SHA-256 hashing
- `json` - JSON parsing
- `pathlib` - Path handling
- `dataclasses` - Data structures

Rust:
- `sha2` - SHA-256 implementation
- `serde` / `serde_json` - Serialization
- `anyhow` - Error handling
- `thiserror` - Error types
- `clap` - CLI parsing

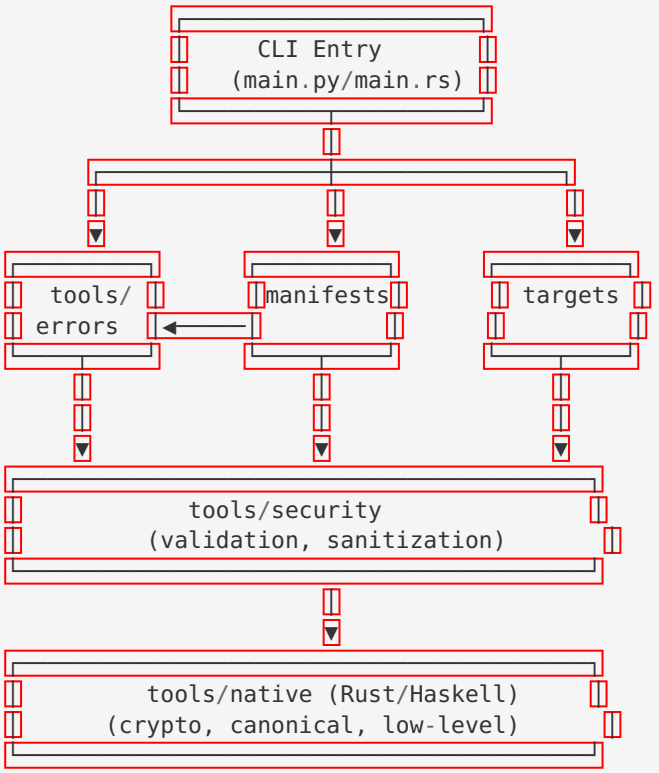
Haskell:
- `aeson` - JSON parsing
- `cryptonite` - Cryptographic primitives
- `bytestring` - Binary data

Module Organization

Repository Structure

```
stunir/
├── docs/                       # Documentation
│   ├── API_REFERENCE.md
│   ├── ARCHITECTURE.md         # This file
│   ├── USER_GUIDE.md
│   └── DEPLOYMENT.md
│
├── tools/                      # Python tooling
│   ├── ir_emitter/            # Spec → IR conversion
│   ├── emitters/              # Target code generators
│   ├── security/              # Input validation
│   │   ├── validation.py      # Sanitization
│   │   └── __init__.py
│   ├── errors.py              # Error system
│   └── native/                # Native tool sources
│       ├── rust/              # Rust implementation
│       │   └── stunir-native/
│       │       ├── src/
│       │       │   ├── lib.rs
│       │       │   ├── crypto.rs
│       │       │   ├── canonical.rs
│       │       │   └── ...
│       │       └── Cargo.toml
│       └── haskell/           # Haskell implementation
│
├── manifests/                  # Manifest generators
│   ├── base.py                # Base classes
│   ├── ir/                    # IR manifests
│   ├── receipts/              # Receipt manifests
│   └── targets/               # Target manifests
│
├── targets/                    # Target emitters
│   ├── polyglot/              # High-level languages
│   │   ├── rust/
│   │   └── c_base.py
│   └── assembly/              # Low-level targets
│       ├── x86/
│       └── arm/
│
├── tests/                      # Test suites
│   ├── security/              # Security tests
│   │   └── test_fuzzing.py
│   └── ...
│
├── scripts/                    # Build scripts
│   ├── build.sh
│   └── verify_strict.sh
│
└── receipts/                   # Build outputs (generated)
```

Module Dependencies

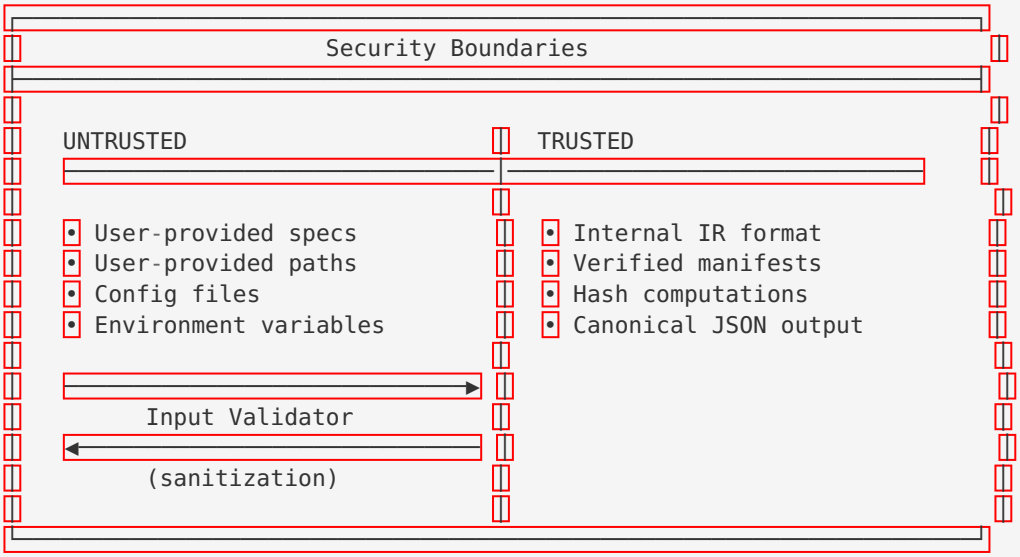


Security Architecture

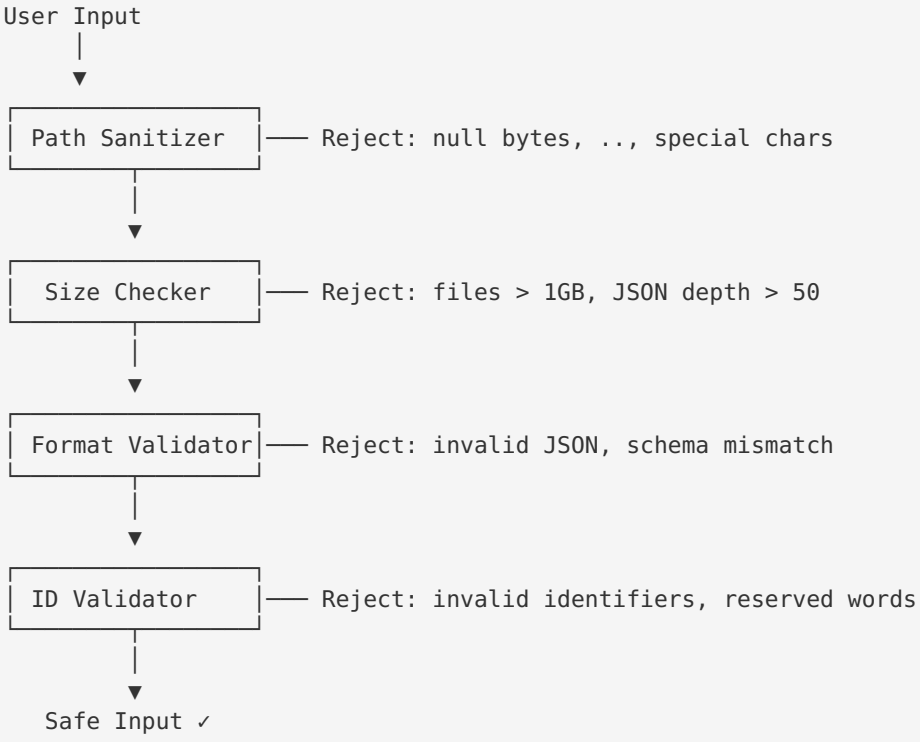
Threat Model

Threat	Mitigation
Path traversal	Path sanitization, base dir check
Symlink attacks	No symlink following by default
DoS via large files	File size limits (1GB max)
DoS via deep nesting	Directory depth limit (100)
JSON bomb	JSON depth limit (50), size limit
Command injection	No shell=True, input validation
Hash collision	SHA-256 (256-bit security)
Manifest tampering	Verification against fresh hashes

Security Boundaries



Input Validation Pipeline



See Also

- API Reference (API_REFERENCE.md) - Complete API documentation
- User Guide (USER_GUIDE.md) - Getting started tutorial
- Contributing (../CONTRIBUTING.md) - Development guidelines
- Deployment (DEPLOYMENT.md) - Production deployment guide
- Security Policy (../SECURITY.md) - Security guidelines