/wycheproof

Source: `~/.claude/skills/tob-testing-handbook-skills/skills/wycheproof/SKILL.md`

---

name: wycheproof type: domain description: > Wycheproof provides test vectors for validating cryptographic implementations. Use when testing crypto code for known attacks and edge cases.

Wycheproof

Wycheproof is an extensive collection of test vectors designed to verify the correctness of cryptographic implementations and test against known attacks. Originally developed by Google, it is now a community-managed project where contributors can add test vectors for specific cryptographic constructions.

Background

Key Concepts

Concept	Description
Test vector	Input/output pair for validating crypto implementation correctness
Test group	Collection of test vectors sharing attributes (key size, IV size, curve)
Result flag	Indicates if test should pass (valid), fail (invalid), or is acceptable
Edge case testing	Testing for known vulnerabilities and attack patterns

Why This Matters

Cryptographic implementations are notoriously difficult to get right. Even small bugs can:

- Expose private keys
- Allow signature forgery
- Enable message decryption
- Create consensus problems when different implementations accept/reject the same inputs

Wycheproof has found vulnerabilities in major libraries including OpenJDK's SHA1withDSA, Bouncy Castle's ECDHC, and the elliptic npm package.

When to Use

Apply Wycheproof when:

- Testing cryptographic implementations (AES-GCM, ECDSA, ECDH, RSA, etc.)
- Validating that crypto code handles edge cases correctly
- Verifying implementations against known attack vectors
- Setting up CI/CD for cryptographic libraries
- Auditing third-party crypto code for correctness

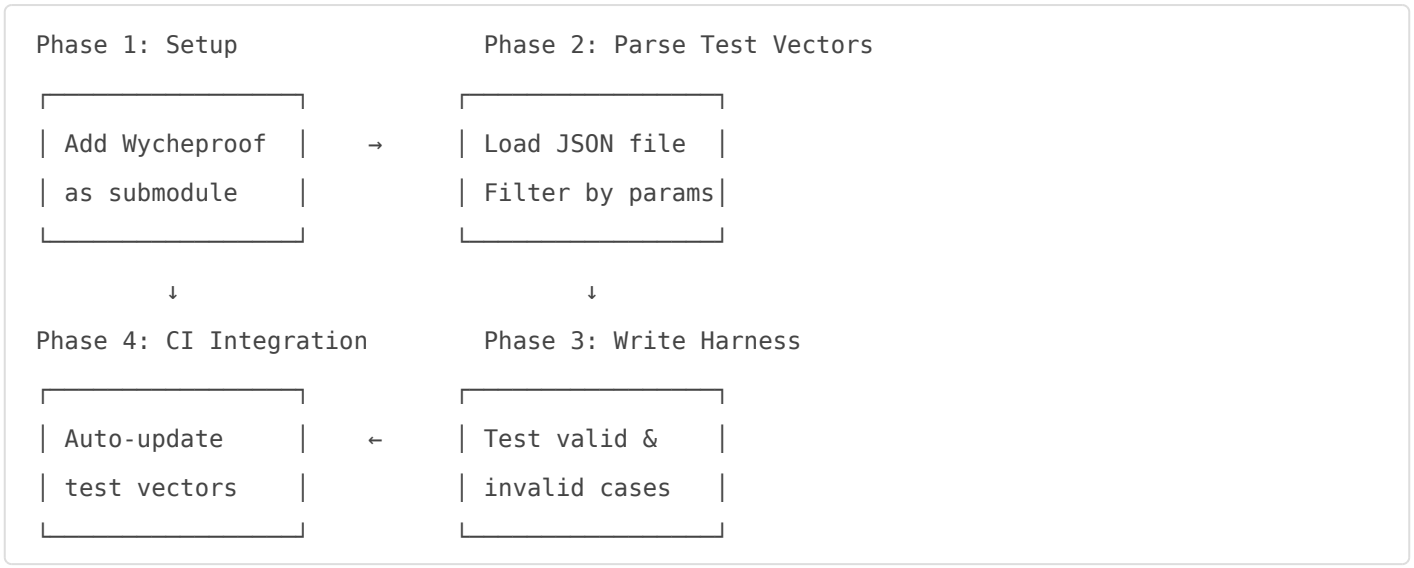
Consider alternatives when:

- Testing for timing side-channels (use constant-time testing tools instead)
- Finding new unknown bugs (use fuzzing instead)
- Testing custom/experimental cryptographic algorithms (Wycheproof only covers established algorithms)

Quick Reference

Scenario	Recommended Approach	Notes
AES-GCM implementation	Use `aes_gcm_test.json`	316 test vectors across 44 test groups
ECDSA verification	Use `ecdsa_*_test.json` for specific curves	Tests signature malleability, DER encoding
ECDH key exchange	Use `ecdh_*_test.json`	Tests invalid curve attacks
RSA signatures	Use `rsa_*_test.json`	Tests padding oracle attacks
ChaCha20-Poly1305	Use `chacha20_poly1305_test.json`	Tests AEAD implementation

Testing Workflow



Repository Structure

The Wycheproof repository is organized as follows:

```
├ □□README.md       : Project overview
├ □□doc             : Documentation
├ □□java            : Java JCE interface testing harness
├ □□javascript      : JavaScript testing harness
├ □□schemas         : Test vector schemas
├ □□testvectors     : Test vectors
└ □□testvectors_v1  : Updated test vectors (more detailed)
```

The essential folders are `testvectors` and `testvectors_v1`. While both contain similar files, `testvectors_v1` includes more detailed information and is recommended for new integrations.

Supported Algorithms

Wycheproof provides test vectors for a wide range of cryptographic algorithms:

Category	Algorithms
Symmetric Encryption	AES-GCM, AES-EAX, ChaCha20-Poly1305
Signatures	ECDSA, EdDSA, RSA-PSS, RSA-PKCS1
Key Exchange	ECDH, X25519, X448
Hashing	HMAC, HKDF
Curves	secp256k1, secp256r1, secp384r1, secp521r1, ed25519, ed448

Test File Structure

Each JSON test file tests a specific cryptographic construction. All test files share common attributes:

```
"algorithm"         : The name of the algorithm tested
"schema"            : The JSON schema (found in schemas folder)
"generatorVersion"  : The version number
"numberOfTests"     : The total number of test vectors in this file
"header"            : Detailed description of test vectors
"notes"             : In-depth explanation of flags in test vectors
"testGroups"        : Array of one or multiple test groups
```

Test Groups

Test groups group sets of tests based on shared attributes such as:

- Key sizes
- IV sizes
- Public keys
- Curves

This classification allows extracting tests that meet specific criteria relevant to the construction being tested.

Test Vector Attributes

Shared Attributes

All test vectors contain four common fields:

- **tcId**: Unique identifier for the test vector within a file
- **comment**: Additional information about the test case
- **flags**: Descriptions of specific test case types and potential dangers (referenced in `notes` field)
- **result**: Expected outcome of the test

The `result` field can take three values:

Result	Meaning
valid	Test case should succeed
acceptable	Test case is allowed to succeed but contains non-ideal attributes
invalid	Test case should fail

Unique Attributes

Unique attributes are specific to the algorithm being tested:

Algorithm	Unique Attributes
AES-GCM	`key`, `iv`, `aad`, `msg`, `ct`, `tag`
ECDH secp256k1	`public`, `private`, `shared`
ECDSA	`msg`, `sig`, `result`
EdDSA	`msg`, `sig`, `pk`

Implementation Guide

Phase 1: Add Wycheproof to Your Project

Option 1: Git Submodule (Recommended)

Adding Wycheproof as a git submodule ensures automatic updates:

```
git submodule add https://github.com/C2SP/wycheproof.git
```

Option 2: Fetch Specific Test Vectors

If submodules aren't possible, fetch specific JSON files:

```
#!/bin/bash

TMP_WYCHEPROOF_FOLDER=".wycheproof/"
TEST_VECTORS=('aes_gcm_test.json' 'aes_eax_test.json')
BASE_URL="https://raw.githubusercontent.com/C2SP/wycheproof/master/testvectors_v1/"

# Create wycheproof folder
mkdir -p $TMP_WYCHEPROOF_FOLDER

# Request all test vector files if they don't exist
for i in "${TEST_VECTORS[@]}"; do
  if [ ! -f "${TMP_WYCHEPROOF_FOLDER}${i}" ]; then
    curl -o "${TMP_WYCHEPROOF_FOLDER}${i}" "${BASE_URL}${i}"
    if [ $? -ne 0 ]; then
      echo "Failed to download ${i}"
      exit 1
    fi
  fi
done
```

Phase 2: Parse Test Vectors

Identify the test file for your algorithm and parse the JSON:

Python Example:

```
import json

def load_wycheproof_test_vectors(path: str):
    testVectors = []
    try:
        with open(path, "r") as f:
            wycheproof_json = json.loads(f.read())
    except FileNotFoundError:
        print(f"No Wycheproof file found at: {path}")
        return testVectors

    # Attributes that need hex-to-bytes conversion
    convert_attr = {"key", "aad", "iv", "msg", "ct", "tag"}
```

```
    for testGroup in wycheproof_json["testGroups"]:
        # Filter test groups based on implementation constraints
        if testGroup["ivSize"] < 64 or testGroup["ivSize"] > 1024:
            continue

        for tv in testGroup["tests"]:
            # Convert hex strings to bytes
            for attr in convert_attr:
                if attr in tv:
                    tv[attr] = bytes.fromhex(tv[attr])
            testVectors.append(tv)

    return testVectors
```

JavaScript Example:

```
const fs = require('fs').promises;

async function loadWycheproofTestVectors(path) {
  const tests = [];

  try {
    const fileContent = await fs.readFile(path);
    const data = JSON.parse(fileContent.toString());

    data.testGroups.forEach(testGroup => {
      testGroup.tests.forEach(test => {
        // Add shared test group properties to each test
        test['pk'] = testGroup.publicKey.pk;
        tests.push(test);
      });
    });
  } catch (err) {
    console.error('Error reading or parsing file:', err);
    throw err;
  }

  return tests;
}
```

Phase 3: Write Testing Harness

Create test functions that handle both valid and invalid test cases.

Python/pytest Example:

```
import pytest
from cryptography.hazmat.primitives.ciphers.aead import AESGCM

tvs = load_wycheproof_test_vectors("wycheproof/testvectors_v1/aes_gcm_test.json")

@pytest.mark.parametrize("tv", tvs, ids=[str(tv['tcId']) for tv in tvs])
def test_encryption(tv):
    try:
        aesgcm = AESGCM(tv['key'])
        ct = aesgcm.encrypt(tv['iv'], tv['msg'], tv['aad'])
    except ValueError as e:
        # Implementation raised error - verify test was expected to fail
        assert tv['result'] != 'valid', tv['comment']
        return

    if tv['result'] == 'valid':
        assert ct[:-16] == tv['ct'], f"Ciphertext mismatch: {tv['comment']}"
        assert ct[-16:] == tv['tag'], f"Tag mismatch: {tv['comment']}"
    elif tv['result'] == 'invalid' or tv['result'] == 'acceptable':
        assert ct[:-16] != tv['ct'] or ct[-16:] != tv['tag']

@pytest.mark.parametrize("tv", tvs, ids=[str(tv['tcId']) for tv in tvs])
def test_decryption(tv):
    try:
        aesgcm = AESGCM(tv['key'])
        decrypted_msg = aesgcm.decrypt(tv['iv'], tv['ct'] + tv['tag'], tv['aad'])
    except ValueError:
        assert tv['result'] != 'valid', tv['comment']
        return
    except InvalidTag:
        assert tv['result'] != 'valid', tv['comment']
        assert 'ModifiedTag' in tv['flags'], f"Expected 'ModifiedTag' flag: {tv['comment']}"
        return
```

```
    assert tv['result'] == 'valid', f"No invalid test case should pass: {tv['comment']}"
    assert decrypted_msg == tv['msg'], f"Decryption mismatch: {tv['comment']}"
```

JavaScript/Mocha Example:

```
const assert = require('assert');

function testFactory(tcId, tests) {
  it(`[${tcId + 1}] ${tests[tcId].comment}`, function () {
    const test = tests[tcId];
    const ed25519 = new eddsa('ed25519');
    const key = ed25519.keyFromPublic(toArray(test.pk, 'hex'));

    let sig;
    if (test.result === 'valid') {
      sig = key.verify(test.msg, test.sig);
      assert.equal(sig, true, `[${test.tcId}] ${test.comment}`);
    } else if (test.result === 'invalid') {
      try {
        sig = key.verify(test.msg, test.sig);
      } catch (err) {
        // Point could not be decoded
        sig = false;
      }
      assert.equal(sig, false, `[${test.tcId}] ${test.comment}`);
    }
  });
}

// Generate tests for all test vectors
for (var tcId = 0; tcId < tests.length; tcId++) {
  testFactory(tcId, tests);
}
```

Phase 4: CI Integration

Ensure test vectors stay up to date by:

1. **Using git submodules**: Update submodule in CI before running tests
2. **Fetching latest vectors**: Run fetch script before test execution

3. **Scheduled updates**: Set up weekly/monthly updates to catch new test vectors

Common Vulnerabilities Detected

Wycheproof test vectors are designed to catch specific vulnerability patterns:

Vulnerability	Description	Affected Algorithms	Example CVE
Signature malleability	Multiple valid signatures for same message	ECDSA, EdDSA	CVE-2024-42459
Invalid DER encoding	Accepting non-canonical DER signatures	ECDSA	CVE-2024-42460, CVE-2024-42461
Invalid curve attacks	ECDH with invalid curve points	ECDH	Common in many libraries
Padding oracle	Timing leaks in padding validation	RSA-PKCS1	Historical OpenSSL issues
Tag forgery	Accepting modified authentication tags	AES-GCM, ChaCha20-Poly1305	Various implementations

Signature Malleability: Deep Dive

Problem: Implementations that don't validate signature encoding can accept multiple valid signatures for the same message.

Example (EdDSA): Appending or removing zeros from signature:

```
Valid signature:   ...6a5c51eb6f946b30d
Invalid signature: ...6a5c51eb6f946b30d0000  (should be rejected)
```

How to detect:

```
# Add signature length check
if len(sig) != 128:  # EdDSA signatures must be exactly 64 bytes (128 hex chars)
    return False
```

Impact: Can lead to consensus problems when different implementations accept/reject the same signatures.

Related Wycheproof tests:

- EdDSA: tcId 37 - "removing 0 byte from signature"
- ECDSA: tcId 06 - "Legacy: ASN encoding of r misses leading 0"

Case Study: Elliptic npm Package

This case study demonstrates how Wycheproof found three CVEs in the popular elliptic npm package (3000+ dependents, millions of weekly downloads).

Overview

The elliptic library is an elliptic-curve cryptography library written in JavaScript, supporting ECDH, ECDSA, and EdDSA. Using Wycheproof test vectors on version 6.5.6 revealed multiple vulnerabilities:

- **CVE-2024-42459**: EdDSA signature malleability (appending/removing zeros)
- **CVE-2024-42460**: ECDSA DER encoding - invalid bit placement
- **CVE-2024-42461**: ECDSA DER encoding - leading zero in length field

Methodology

1. **Identify supported curves**: ed25519 for EdDSA
2. **Find test vectors**: `testvectors_v1/ed25519_test.json`
3. **Parse test vectors**: Load JSON and extract tests
4. **Write test harness**: Create parameterized tests
5. **Run tests**: Identify failures
6. **Analyze root causes**: Examine implementation code
7. **Propose fixes**: Add validation checks

Key Findings

EdDSA Issue (CVE-2024-42459):

- Missing signature length validation
- Allowed trailing zeros in signatures
- Fix: Add `if(sig.length !== 128) return false;`

ECDSA Issue 1 (CVE-2024-42460):

- Missing check for first bit being zero in DER-encoded r and s values
- Fix: Add `if ((data[p.place] & 128) !== 0) return false;`

ECDSA Issue 2 (CVE-2024-42461):

- DER length field accepted leading zeros
- Fix: Add `if(buf[p.place] === 0x00) return false;`

Impact

All three vulnerabilities allowed multiple valid signatures for a single message, leading to consensus problems across implementations.

Lessons learned:

- Wycheproof catches subtle encoding bugs
- Reusable test harnesses pay dividends
- Test vector comments and flags help diagnose issues
- Even popular libraries benefit from systematic test vector validation

Advanced Usage

Tips and Tricks

Tip	Why It Helps
Filter test groups by parameters	Focus on test vectors relevant to your implementation constraints
Use test vector flags	Understand specific vulnerability patterns being tested
Check the `notes` field	Get detailed explanations of flag meanings
Test both encrypt/decrypt and sign/verify	Ensure bidirectional correctness
Run tests in CI	Catch regressions and benefit from new test vectors
Use parameterized tests	Get clear failure messages with tcId and comment

Common Mistakes

Mistake	Why It's Wrong	Correct Approach
Only testing valid cases	Misses vulnerabilities where invalid inputs are accepted	Test all result types: valid, invalid, acceptable
Ignoring "acceptable" result	Implementation might have subtle bugs	Treat acceptable as warnings worth investigating
Not filtering test groups	Wastes time on unsupported parameters	Filter by keySize, ivSize, etc. based on your implementation
Not updating test vectors	Miss new vulnerability patterns	Use submodules or scheduled fetches
Testing only one direction	Encrypt/sign might work but decrypt/verify fails	Test both operations

Related Skills

Tool Skills

Skill	Primary Use in Wycheproof Testing
pytest	Python testing framework for parameterized tests
mocha	JavaScript testing framework for test generation
constant-time-testing	Complement Wycheproof with timing side-channel testing
cryptofuzz	Fuzz-based crypto testing to find additional bugs

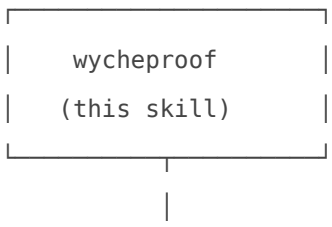
Technique Skills

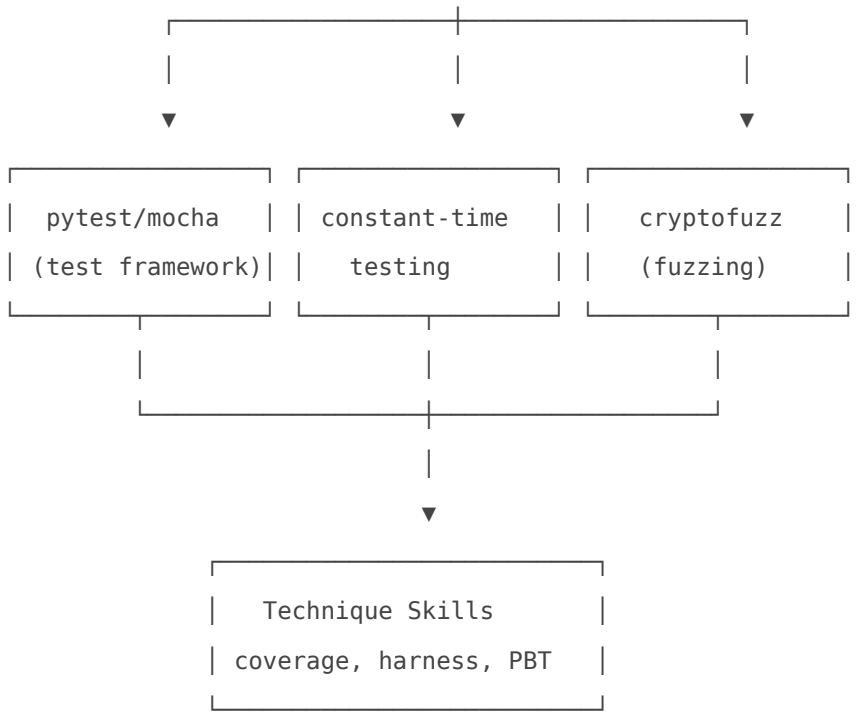
Skill	When to Apply
coverage-analysis	Ensure test vectors cover all code paths in crypto implementation
property-based-testing	Test mathematical properties (e.g., encrypt/decrypt round-trip)
fuzz-harness-writing	Create harnesses for crypto parsers (complements Wycheproof)

Related Domain Skills

Skill	Relationship
crypto-testing	Wycheproof is a key tool in comprehensive crypto testing methodology
fuzzing	Use fuzzing to find bugs Wycheproof doesn't cover (new edge cases)

Skill Dependency Map





Resources

Official Repository

Wycheproof GitHub Repository

The official repository contains:

- All test vectors in `testvectors/` and `testvectors_v1/`
- JSON schemas in `schemas/`
- Reference implementations in Java and JavaScript
- Documentation in `doc/`

Real-World Examples

pycryptodome

The pycryptodome library integrates Wycheproof test vectors in their test suite, demonstrating best practices for Python crypto implementations.

Community Resources

- C2SP Community - Cryptographic specifications and standards community maintaining Wycheproof
- Wycheproof issues tracker - Report bugs in test vectors or suggest new constructions

Summary

Wycheproof is an essential tool for validating cryptographic implementations against known attack vectors and edge cases. By integrating Wycheproof test vectors into your testing workflow:

1. Catch subtle encoding and validation bugs
2. Prevent signature malleability issues
3. Ensure consistent behavior across implementations
4. Benefit from community-contributed test vectors
5. Protect against known cryptographic vulnerabilities

The investment in writing a reusable testing harness pays dividends through continuous validation as new test vectors are added to the Wycheproof repository.

Revision #5
Created 2026-02-18 08:40:13 UTC by John
Updated 2026-06-21 20:01:31 UTC by John