/constant-time-testing

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

---

name: constant-time-testing type: domain description: > Constant-time testing detects timing side channels in cryptographic code. Use when auditing crypto implementations for timing vulnerabilities.

Constant-Time Testing

Timing attacks exploit variations in execution time to extract secret information from cryptographic implementations. Unlike cryptanalysis that targets theoretical weaknesses, timing attacks leverage implementation flaws - and they can affect any cryptographic code.

Background

Timing attacks were introduced by Kocher in 1996. Since then, researchers have demonstrated practical attacks on RSA (Schindler), OpenSSL (Brumley and Boneh), AES implementations, and even post-quantum algorithms like Kyber.

Key Concepts

Concept	Description
Constant-time	Code path and memory accesses independent of secret data
Timing leakage	Observable execution time differences correlated with secrets
Side channel	Information extracted from implementation rather than algorithm
Microarchitecture	CPU-level timing differences (cache, division, shifts)

Why This Matters

Timing vulnerabilities can:

- **Expose private keys** - Extract secret exponents in RSA/ECDH
- **Enable remote attacks** - Network-observable timing differences
- **Bypass cryptographic security** - Undermine theoretical guarantees
- **Persist silently** - Often undetected without specialized analysis

Two prerequisites enable exploitation:

1. **Access to oracle** - Sufficient queries to the vulnerable implementation
2. **Timing dependency** - Correlation between execution time and secret data

Common Constant-Time Violation Patterns

Four patterns account for most timing vulnerabilities:

```
// 1. Conditional jumps - most severe timing differences
if(secret == 1) { ... }
while(secret > 0) { ... }

// 2. Array access - cache-timing attacks
lookup_table[secret];
```

```
// 3. Integer division (processor dependent)
data = secret / m;

// 4. Shift operation (processor dependent)
data = a << secret;
```

Conditional jumps cause different code paths, leading to vast timing differences.

Array access dependent on secrets enables cache-timing attacks, as shown in AES cache-timing research.

Integer division and shift operations leak secrets on certain CPU architectures and compiler configurations.

When patterns cannot be avoided, employ masking techniques to remove correlation between timing and secrets.

Example: Modular Exponentiation Timing Attacks

Modular exponentiation (used in RSA and Diffie-Hellman) is susceptible to timing attacks. RSA decryption computes:

$$ct^{d} \mod{N}$$

where d is the secret exponent. The *exponentiation by squaring* optimization reduces multiplications to $\log{d}$:

$$ \begin{align*} & \textbf{Input: } \text{base }y,\text{exponent } d={d_n,\cdots,d_0}_2,\text{modulus } N \ & r = 1 \ & \textbf{for } i=|n| \text{ downto } 0: \ & \quad\textbf{if } d_i == 1: \ & \quad\quad r = r * y \mod{N} \ & \quad y = y * y \mod{N} \ & \textbf{return }r \end{align*} $$

The code branches on exponent bit d_i, violating constant-time principles. When $d_i = 1$, an additional multiplication occurs, increasing execution time and leaking bit information.

Montgomery multiplication (commonly used for modular arithmetic) also leaks timing: when intermediate values exceed modulus N, an additional reduction step is required. An attacker constructs inputs y and y' such that:

$$ \begin{align*} y^2 < y^3 < N \ y'^2 < N \leq y'^3 \end{align*} $$

For y, both multiplications take time t_1+t_1. For y', the second multiplication requires reduction, taking time t_1+t_2. This timing difference reveals whether d_i is 0 or 1.

When to Use

Apply constant-time analysis when:

- Auditing cryptographic implementations (primitives, protocols)
- Code handles secret keys, passwords, or sensitive cryptographic material
- Implementing crypto algorithms from scratch
- Reviewing PRs that touch crypto code
- Investigating potential timing vulnerabilities

Consider alternatives when:

- Code does not process secret data
- Public algorithms with no secret inputs
- Non-cryptographic timing requirements (performance optimization)

Quick Reference

Scenario	Recommended Approach	Skill
Prove absence of leaks	Formal verification	SideTrail, ct-verif, FaCT
Detect statistical timing differences	Statistical testing	**dudect**
Track secret data flow at runtime	Dynamic analysis	**timecop**
Find cache-timing vulnerabilities	Symbolic execution	Binsec, pitchfork

Constant-Time Tooling Categories

The cryptographic community has developed four categories of timing analysis tools:

Category	Approach	Pros	Cons
Formal	Mathematical proof on model	Guarantees absence of leaks	Complexity, modeling assumptions
Symbolic	Symbolic execution paths	Concrete counterexamples	Time-intensive path exploration
Dynamic	Runtime tracing with marked secrets	Granular, flexible	Limited coverage to executed paths

Category	Approach	Pros	Cons
Statistical	Measure real execution timing	Practical, simple setup	No root cause, noise sensitivity

1. Formal Tools

Formal verification mathematically proves timing properties on an abstraction (model) of code. Tools create a model from source/binary and verify it satisfies specified properties (e.g., variables annotated as secret).

Popular tools:

- SideTrail
- ct-verif
- FaCT

Strengths: Proof of absence, language-agnostic (LLVM bytecode) **Weaknesses:** Requires expertise, modeling assumptions may miss real-world issues

2. Symbolic Tools

Symbolic execution analyzes how paths and memory accesses depend on symbolic variables (secrets). Provides concrete counterexamples. Focus on cache-timing attacks.

Popular tools:

- Binsec
- pitchfork

Strengths: Concrete counterexamples aid debugging **Weaknesses:** Path explosion leads to long execution times

3. Dynamic Tools

Dynamic analysis marks sensitive memory regions and traces execution to detect timing-dependent operations.

Popular tools:

- Memsan: Tutorial
- **Timecop** (see below)

Strengths: Granular control, targeted analysis **Weaknesses:** Coverage limited to executed paths

> **Detailed Guidance:** See the **timecop** skill for setup and usage.

4. Statistical Tools

Execute code with various inputs, measure elapsed time, and detect inconsistencies. Tests actual implementation including compiler optimizations and architecture.

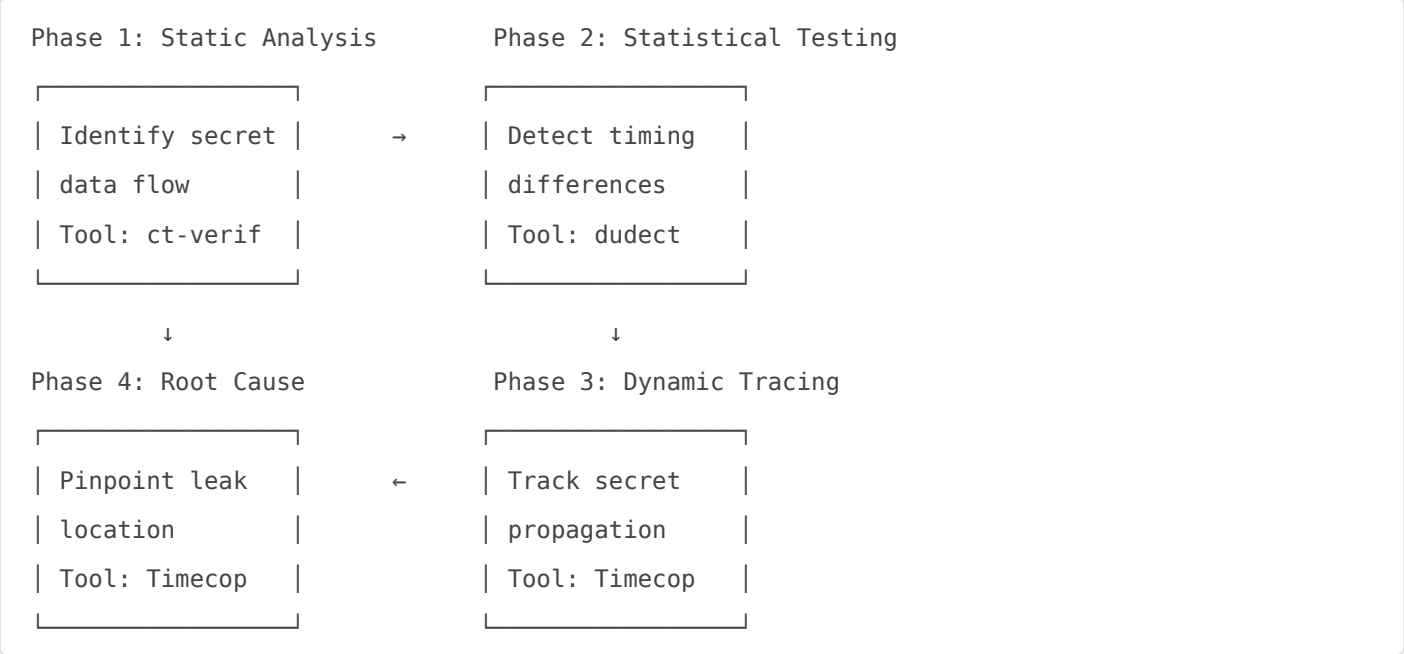
Popular tools:

- **dudect** (see below)
- tlsfuzzer

Strengths: Simple setup, practical real-world results **Weaknesses:** No root cause info, noise obscures weak signals

> **Detailed Guidance:** See the **dudect** skill for setup and usage.

Testing Workflow



Recommended approach:

1. **Start with dudect** - Quick statistical check for timing differences
2. **If leaks found** - Use Timecop to pinpoint root cause
3. **For high-assurance** - Apply formal verification (ct-verif, SideTrail)
4. **Continuous monitoring** - Integrate dudect into CI pipeline

Tools and Approaches

Dudect - Statistical Analysis

Dudect measures execution time for two input classes (fixed vs random) and uses Welch's t-test to detect statistically significant differences.

> **Detailed Guidance:** See the **dudect** skill for complete setup, usage patterns, and CI integration.

Quick Start for Constant-Time Analysis

```
#define DUDECT_IMPLEMENTATION
#include "dudect.h"

uint8_t do_one_computation(uint8_t *data) {
    // Code to measure goes here
}

void prepare_inputs(dudect_config_t *c, uint8_t *input_data, uint8_t *classes) {
    for (size_t i = 0; i < c->number_measurements; i++) {
        classes[i] = randombit();
        uint8_t *input = input_data + (size_t)i * c->chunk_size;
        if (classes[i] == 0) {
            // Fixed input class
        } else {
            // Random input class
        }
    }
}
```

Key advantages:

- Simple C header-only integration
- Statistical rigor via Welch's t-test
- Works with compiled binaries (real-world conditions)

Key limitations:

- No root cause information when leak detected
- Sensitive to measurement noise
- Cannot guarantee absence of leaks (statistical confidence only)

Timecop - Dynamic Tracing

Timecop wraps Valgrind to detect runtime operations dependent on secret memory regions.

> **Detailed Guidance:** See the **timecop** skill for installation, examples, and debugging.

Quick Start for Constant-Time Analysis

```
#include "valgrind/memcheck.h"

#define poison(addr, len) VALGRIND_MAKE_MEM_UNDEFINED(addr, len)
#define unpoison(addr, len) VALGRIND_MAKE_MEM_DEFINED(addr, len)

int main() {
    unsigned long long secret_key = 0x12345678;

    // Mark secret as poisoned
    poison(&secret_key, sizeof(secret_key));

    // Any branching or memory access dependent on secret_key
    // will be reported by Valgrind
    crypto_operation(secret_key);

    unpoison(&secret_key, sizeof(secret_key));
}
```

Run with Valgrind:

```
valgrind --leak-check=full --track-origins=yes ./binary
```

Key advantages:

- Pinpoints exact line of timing leak
- No code instrumentation required
- Tracks secret propagation through execution

Key limitations:

- Cannot detect microarchitecture timing differences
- Coverage limited to executed paths
- Performance overhead (runs on synthetic CPU)

Implementation Guide

Phase 1: Initial Assessment

Identify cryptographic code handling secrets:

- Private keys, exponents, nonces
- Password hashes, authentication tokens
- Encryption/decryption operations

Quick statistical check:

1. Write dudect harness for the crypto function
2. Run for 5-10 minutes with `timeout 600 ./ct_test`
3. Monitor t-value: high absolute values indicate leakage

Tools: dudect **Expected time:** 1-2 hours (harness writing + initial run)

Phase 2: Detailed Analysis

If dudect detects leakage:

Root cause investigation:

1. Mark secret variables with Timecop `poison()`
2. Run under Valgrind to identify exact line
3. Review the four common violation patterns
4. Check assembly output for conditional branches

Tools: Timecop, compiler output (`objdump -d`)

Phase 3: Remediation

Fix the timing leak:

- Replace conditional branches with constant-time selection (bitwise operations)
- Use constant-time comparison functions
- Replace array lookups with constant-time alternatives or masking
- Verify compiler doesn't optimize away constant-time code

Re-verify:

1. Run dudect again for extended period (30+ minutes)
2. Test across different compilers and optimization levels
3. Test on different CPU architectures

Phase 4: Continuous Monitoring

Integrate into CI:

- Add dudect tests to test suite
- Run for fixed duration (5-10 minutes in CI)
- Fail build if leakage detected

See the **dudect** skill for CI integration examples.

Common Vulnerabilities

Vulnerability	Description	Detection	Severity
Secret-dependent branch	`if (secret_bit) { ... }`	dudect, Timecop	CRITICAL
Secret-dependent array access	`table[secret_index]`	Timecop, Binsec	HIGH
Variable-time division	`result = x / secret`	Timecop	MEDIUM
Variable-time shift	`result = x << secret`	Timecop	MEDIUM
Montgomery reduction leak	Extra reduction when intermediate > N	dudect	HIGH

Secret-Dependent Branch: Deep Dive

The vulnerability: Execution time differs based on whether branch is taken. Common in optimized modular exponentiation (square-and-multiply).

How to detect with dudect:

```
uint8_t do_one_computation(uint8_t *data) {
    uint64_t base = ((uint64_t*)data)[0];
    uint64_t exponent = ((uint64_t*)data)[1]; // Secret!
    return mod_exp(base, exponent, MODULUS);
}

void prepare_inputs(dudect_config_t *c, uint8_t *input_data, uint8_t *classes) {
    for (size_t i = 0; i < c->number_measurements; i++) {
        classes[i] = randombit();
        uint64_t *input = (uint64_t*)(input_data + i * c->chunk_size);
        input[0] = rand(); // Random base
        input[1] = (classes[i] == 0) ? FIXED_EXPONENT : rand(); // Fixed vs random
    }
}
```

How to detect with Timecop:

```
poison(&exponent, sizeof(exponent));
result = mod_exp(base, exponent, modulus);
unpoison(&exponent, sizeof(exponent));
```

Valgrind will report:

```
Conditional jump or move depends on uninitialised value(s)
  at 0x40115D: mod_exp (example.c:14)
```

Related skill: dudect, **timecop**

Case Studies

Case Study: OpenSSL RSA Timing Attack

Brumley and Boneh (2005) extracted RSA private keys from OpenSSL over a network. The vulnerability exploited Montgomery multiplication's variable-time reduction step.

Attack vector: Timing differences in modular exponentiation **Detection approach:** Statistical analysis (precursor to dudect) **Impact:** Remote key extraction

Tools used: Custom timing measurement **Techniques applied:** Statistical analysis, chosen-ciphertext queries

Case Study: KyberSlash

Post-quantum algorithm Kyber's reference implementation contained timing vulnerabilities in polynomial operations. Division operations leaked secret coefficients.

Attack vector: Secret-dependent division timing **Detection approach:** Dynamic analysis and statistical testing **Impact:** Secret key recovery in post-quantum cryptography

Tools used: Timing measurement tools **Techniques applied:** Differential timing analysis

Advanced Usage

Tips and Tricks

Tip	Why It Helps
Pin dudect to isolated CPU core (`taskset -c 2`)	Reduces OS noise, improves signal detection
Test multiple compilers (gcc, clang, MSVC)	Optimizations may introduce or remove leaks
Run dudect for extended periods (hours)	Increases statistical confidence
Minimize non-crypto code in harness	Reduces noise that masks weak signals
Check assembly output (`objdump -d`)	Verify compiler didn't introduce branches
Use `-O3 -march=native` in testing	Matches production optimization levels

Common Mistakes

Mistake	Why It's Wrong	Correct Approach
Only testing one input distribution	May miss leaks visible with other patterns	Test fixed-vs-random, fixed-vs-fixed-different, etc.
Short dudect runs (< 1 minute)	Insufficient measurements for weak signals	Run 5-10+ minutes, longer for high assurance
Ignoring compiler optimization levels	`-O0` may hide leaks present in `-O3`	Test at production optimization level
Not testing on target architecture	x86 vs ARM have different timing characteristics	Test on deployment platform
Marking too much as secret in Timecop	False positives, unclear results	Mark only true secrets (keys, not public data)

Related Skills

Tool Skills

Skill	Primary Use in Constant-Time Analysis
dudect	Statistical detection of timing differences via Welch's t-test
timecop	Dynamic tracing to pinpoint exact location of timing leaks

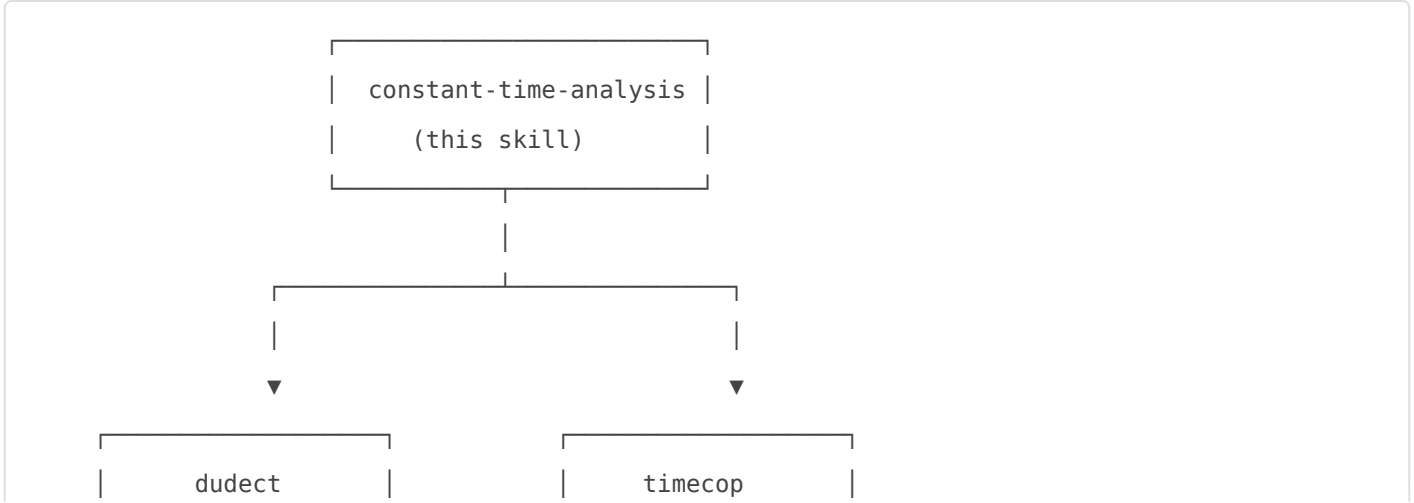
Technique Skills

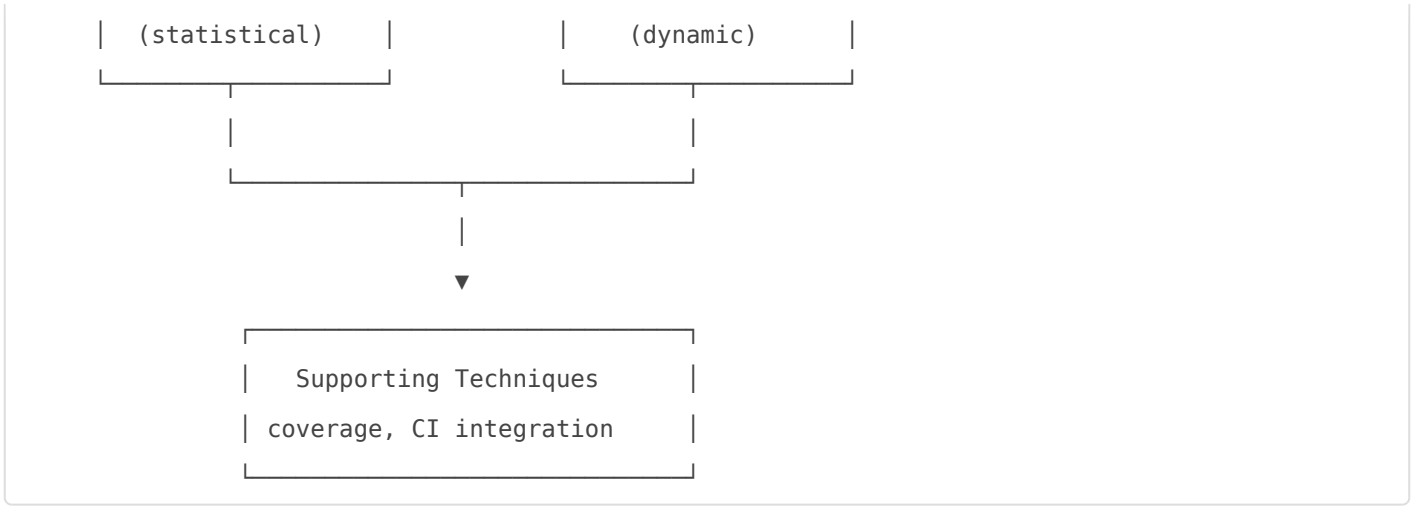
Skill	When to Apply
coverage-analysis	Ensure test inputs exercise all code paths in crypto function
ci-integration	Automate constant-time testing in continuous integration pipeline

Related Domain Skills

Skill	Relationship
crypto-testing	Constant-time analysis is essential component of cryptographic testing
fuzzing	Fuzzing crypto code may trigger timing-dependent paths

Skill Dependency Map





Resources

Key External Resources

These results must be false: A usability evaluation of constant-time analysis tools
Comprehensive usability study of constant-time analysis tools. Key findings: developers struggle with false positives, need better error messages, and benefit from tool integration. Evaluates FaCT, ct-verif, dudect, and Memsan across multiple cryptographic implementations. Recommends improved tooling UX and better documentation.

List of constant-time tools - CROCS Curated catalog of constant-time analysis tools with tutorials. Covers formal tools (ct-verif, FaCT), dynamic tools (Memsan, Timecop), symbolic tools (Binsec), and statistical tools (dudect). Includes practical tutorials for setup and usage.

Paul Kocher: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems Original 1996 paper introducing timing attacks. Demonstrates attacks on modular exponentiation in RSA and Diffie-Hellman. Essential historical context for understanding timing vulnerabilities.

Remote Timing Attacks are Practical (Brumley & Boneh) Demonstrates practical remote timing attacks against OpenSSL. Shows network-level timing differences are sufficient to extract RSA keys. Proves timing attacks work in realistic network conditions.

Cache-timing attacks on AES Shows AES implementations using lookup tables are vulnerable to cache-timing attacks. Demonstrates practical attacks extracting AES keys via cache timing side channels.

KyberSlash: Division Timings Leak Secrets Recent discovery of timing vulnerabilities in Kyber (NIST post-quantum standard). Shows division operations leak secret coefficients. Highlights that constant-time issues persist even in modern post-quantum cryptography.

Video Resources

- Trail of Bits: Constant-Time Programming - Overview of constant-time programming principles and tools

---

Revision #5
Created 2026-02-18 08:40:10 UTC by John
Updated 2026-06-21 20:01:22 UTC by John