# /constant-time-analysis

**Source:** `~/.claude/skills/tob-constant-time-analysis/skills/constant-time-analysis/SKILL.md`

---

name: constant-time-analysis
description: Detects timing side-channel vulnerabilities in cryptographic code. Use when implementing or reviewing crypto code, encountering division on secrets, secret-dependent branches, or constant-time programming questions in C, C++, Go, Rust, Swift, Java, Kotlin, C#, PHP, JavaScript, TypeScript, Python, or Ruby.

## Constant-Time Analysis

Analyze cryptographic code to detect operations that leak secret data through execution timing variations.

# When to Use

```
User writing crypto code? ──yes──> Use this skill
         |
         no
         |
         v
User asking about timing attacks? ──yes──> Use this skill
         |
         no
         |
         v
Code handles secret keys/tokens? ──yes──> Use this skill
         |
         no
         |
         v
Skip this skill
```

**Concrete triggers:**

- User implements signature, encryption, or key derivation
- Code contains `/` or `%` operators on secret-derived values
- User mentions "constant-time", "timing attack", "side-channel", "KyberSlash"
- Reviewing functions named `sign`, `verify`, `encrypt`, `decrypt`, `derive_key`

# When NOT to Use

- Non-cryptographic code (business logic, UI, etc.)
- Public data processing where timing leaks don't matter
- Code that doesn't handle secrets, keys, or authentication tokens
- High-level API usage where timing is handled by the library

# Language Selection

Based on the file extension or language context, refer to the appropriate guide:

| Language | File Extensions | Guide |
|---|---|---|
| C, C++ | `.c`, `.h`, `.cpp`, `.cc`, `.hpp` | references/compiled.md |
| Go | `.go` | references/compiled.md |
| Rust | `.rs` | references/compiled.md |
| Swift | `.swift` | references/swift.md |
| Java | `.java` | references/vm-compiled.md |
| Kotlin | `.kt`, `.kts` | references/kotlin.md |
| C# | `.cs` | references/vm-compiled.md |
| PHP | `.php` | references/php.md |
| JavaScript | `.js`, `.mjs`, `.cjs` | references/javascript.md |
| TypeScript | `.ts`, `.tsx` | references/javascript.md |
| Python | `.py` | references/python.md |
| Ruby | `.rb` | references/ruby.md |

# Quick Start

```
# Analyze any supported file type
uv run {baseDir}/ct_analyzer/analyzer.py <source_file>

# Include conditional branch warnings
uv run {baseDir}/ct_analyzer/analyzer.py --warnings <source_file>

# Filter to specific functions
uv run {baseDir}/ct_analyzer/analyzer.py --func 'sign|verify' <source_file>

# JSON output for CI
uv run {baseDir}/ct_analyzer/analyzer.py --json <source_file>
```

# Native Compiled Languages Only (C, C++, Go, Rust)

```
# Cross-architecture testing (RECOMMENDED)
uv run {baseDir}/ct_analyzer/analyzer.py --arch x86_64 crypto.c
uv run {baseDir}/ct_analyzer/analyzer.py --arch arm64 crypto.c

# Multiple optimization levels
uv run {baseDir}/ct_analyzer/analyzer.py --opt-level O0 crypto.c
uv run {baseDir}/ct_analyzer/analyzer.py --opt-level O3 crypto.c
```

# VM-Compiled Languages (Java, Kotlin, C#)

```
# Analyze Java bytecode
uv run {baseDir}/ct_analyzer/analyzer.py CryptoUtils.java

# Analyze Kotlin bytecode (Android/JVM)
uv run {baseDir}/ct_analyzer/analyzer.py CryptoUtils.kt

# Analyze C# IL
uv run {baseDir}/ct_analyzer/analyzer.py CryptoUtils.cs
```

Note: Java, Kotlin, and C# compile to bytecode (JVM/CIL) that runs on a virtual machine with JIT compilation. The analyzer examines the bytecode directly, not the JIT-compiled native code. The `--arch` and `--opt-level` flags do not apply to these languages.

# Swift (iOS/macOS)

```
# Analyze Swift for native architecture
uv run {baseDir}/ct_analyzer/analyzer.py crypto.swift

# Analyze for specific architecture (iOS devices)
uv run {baseDir}/ct_analyzer/analyzer.py --arch arm64 crypto.swift

# Analyze with different optimization levels
uv run {baseDir}/ct_analyzer/analyzer.py --opt-level O0 crypto.swift
```

Note: Swift compiles to native code like C/C++/Go/Rust, so it uses assembly-level analysis and supports `--arch` and `--opt-level` flags.

# Prerequisites

| Language | Requirements |
|---|---|
| C, C++, Go, Rust | Compiler in PATH (`gcc`/`clang`, `go`, `rustc`) |
| Swift | Xcode or Swift toolchain (`swiftc` in PATH) |
| Java | JDK with `javac` and `javap` in PATH |
| Kotlin | Kotlin compiler (`kotlinc`) + JDK (`javap`) in PATH |
| C# | .NET SDK + `ilspycmd` (`dotnet tool install -g ilspycmd`) |
| PHP | PHP with VLD extension or OPcache |
| JavaScript/TypeScript | Node.js in PATH |
| Python | Python 3.x in PATH |
| Ruby | Ruby with `--dump=insns` support |

**macOS users**: Homebrew installs Java and .NET as "keg-only". You must add them to your PATH:

```
# For Java (add to ~/.zshrc)
export PATH="/opt/homebrew/opt/openjdk@21/bin:$PATH"

# For .NET tools (add to ~/.zshrc)
export PATH="$HOME/.dotnet/tools:$PATH"
```

See references/vm-compiled.md for detailed setup instructions and troubleshooting.

# Quick Reference

| Problem | Detection | Fix |
|---|---|---|
| Division on secrets | DIV, IDIV, SDIV, UDIV | Barrett reduction or multiply-by-inverse |
| Branch on secrets | JE, JNE, BEQ, BNE | Constant-time selection (cmov, bit masking) |
| Secret comparison | Early-exit memcmp | Use `crypto/subtle` or constant-time compare |
| Weak RNG | rand(), mt_rand, Math.random | Use crypto-secure RNG |
| Table lookup by secret | Array subscript on secret index | Bit-sliced lookups |

# Interpreting Results

**PASSED** - No variable-time operations detected.

**FAILED** - Dangerous instructions found. Example:

```
[ERROR] SDIV
  Function: decompose_vulnerable
  Reason: SDIV has early termination optimization; execution time depends on operand values
```

# Verifying Results (Avoiding False Positives)

**CRITICAL**: Not every flagged operation is a vulnerability. The tool has no data flow analysis - it flags ALL potentially dangerous operations regardless of whether they involve secrets.

For each flagged violation, ask: **Does this operation's input depend on secret data?**

1. **Identify the secret inputs** to the function (private keys, plaintext, signatures, tokens)
2. **Trace data flow** from the flagged instruction back to inputs
3. **Common false positive patterns**:

   ```
   // FALSE POSITIVE: Division uses public constant, not secret
   int num_blocks = data_len / 16;  // data_len is length, not content

   // TRUE POSITIVE: Division involves secret-derived value
   int32_t q = secret_coef / GAMMA2;  // secret_coef from private key
   ```

4. **Document your analysis** for each flagged item

# Quick Triage Questions

| Question | If Yes | If No |
|---|---|---|
| Is the operand a compile-time constant? | Likely false positive | Continue |
| Is the operand a public parameter (length, count)? | Likely false positive | Continue |
| Is the operand derived from key/plaintext/secret? | **TRUE POSITIVE** | Likely false positive |
| Can an attacker influence the operand value? | **TRUE POSITIVE** | Likely false positive |

# Limitations

1. **Static Analysis Only**: Analyzes assembly/bytecode, not runtime behavior. Cannot detect cache timing or microarchitectural side-channels.
2. **No Data Flow Analysis**: Flags all dangerous operations regardless of whether they process secrets. Manual review required.
3. **Compiler/Runtime Variations**: Different compilers, optimization levels, and runtime versions may produce different output.

# Real-World Impact

- **KyberSlash (2023)**: Division instructions in post-quantum ML-KEM implementations allowed key recovery
- **Lucky Thirteen (2013)**: Timing differences in CBC padding validation enabled plaintext recovery
- **RSA Timing Attacks**: Early implementations leaked private key bits through division timing

# References


- Cryptocoding Guidelines - Defensive coding for crypto
- KyberSlash - Division timing in post-quantum crypto
- BearSSL Constant-Time - Practical constant-time techniques


---


Revision #4
Created 2026-02-18 08:40:05 UTC by John
Updated 2026-05-31 20:01:46 UTC by John