Cryptanalysis and Attacks

MAT364 - Cryptography Course

Instructor: Adil Akhmetov

University: SDU

Week 3

Press Space for next page →

What is Cryptanalysis?

Definition

Cryptanalysis is the art and science of breaking cryptographic systems without knowing the secret key.

Goals

- **Recover plaintext** from ciphertext
- **Determine the key** used for encryption
- **Find weaknesses** in cryptographic algorithms
- **Understand security** of cryptographic systems

Types of Attacks

- **Ciphertext-only** - Only ciphertext available
- **Known-plaintext** - Some plaintext-ciphertext pairs known
- **Chosen-plaintext** - Attacker can choose plaintext
- **Chosen-ciphertext** - Attacker can choose ciphertext

Attack Complexity

- **Computational complexity** - Time and space required
- **Data complexity** - Amount of data needed
- **Success probability** - Likelihood of success

Remember: Cryptanalysis helps us understand security weaknesses and improve cryptographic systems!

Brute Force Attacks

How It Works

- **Try every possible key** systematically
- **Test each key** against known plaintext
- **Stop when** correct key is found
- **Time complexity:** O(2^n) where n = key length

When It's Feasible

- **Small key spaces** (Caesar cipher: 25 keys)
- **Weak algorithms** with limited keys
- **Known plaintext** available for verification
- **Sufficient computational power**

Implementation

```
def brute_force_caesar(ciphertext, known_plaintext):
    """Brute force Caesar cipher"""
    for shift in range(26):
        decrypted = caesar_decrypt(ciphertext, shift)
        if known_plaintext.lower() in decrypted.lower(
            return shift, decrypted
    return None, None

def caesar_decrypt(text, shift):
    result = ""
    for char in text:
        if char.isalpha():
            ascii_offset = 65 if char.isupper() else 9
            letter_pos = ord(char) - ascii_offset
            new_pos = (letter_pos - shift) % 26
            result += chr(new_pos + ascii_offset)
```

Frequency Analysis

English Letter Frequencies

Most common letters:

- E: 12.7%, T: 9.1%, A: 8.2%, O: 7.5%
- I: 7.0%, N: 6.7%, S: 6.3%, H: 6.1%
- R: 6.0%, D: 4.3%, L: 4.0%, C: 2.8%

Least common letters:

- J: 0.2%, X: 0.2%, Q: 0.1%, Z: 0.1%

Method

1. **Count frequencies** in ciphertext
2. **Match to English** letter frequencies
3. **Use common patterns** (digraphs, trigraphs)
4. **Refine mapping** iteratively

Implementation

```
def frequency_analysis(ciphertext):
    """Analyze letter frequencies"""
    text = ciphertext.upper().replace(' ', '')
    frequencies = {}
    total_chars = len(text)

    for char in text:
        if char.isalpha():
            frequencies[char] = frequencies.get(char,

    # Convert to percentages
    for char in frequencies:
        frequencies[char] = (frequencies[char] / total

    return dict(sorted(frequencies.items(),
                       key=lambda x: x[1], reverse=True
```

Advanced Attack Techniques

Timing Attacks

How It Works

- **Measure execution time** of cryptographic operations
- **Correlate timing** with secret data
- **Extract information** from timing variations
- **Works on** software implementations

Vulnerable Operations

- **String comparison** - Early exit on mismatch
- **Modular exponentiation** - Different paths for 0/1 bits
- **Table lookups** - Cache timing differences
- **Branching** - Different execution paths

Example: String Comparison

```
def vulnerable_compare(a, b):
    """Vulnerable string comparison"""
    if len(a) ≠ len(b):
        return False

    for i in range(len(a)):
        if a[i] ≠ b[i]:  # Early exit reveals positio
            return False
    return True

def secure_compare(a, b):
    """Constant-time string comparison"""
    if len(a) ≠ len(b):
        return False

    result = 0
```

Side-Channel Attacks

Types of Side Channels

Power Analysis:

- **Simple Power Analysis (SPA)** - Direct power consumption
- **Differential Power Analysis (DPA)** - Statistical analysis
- **Correlation Power Analysis (CPA)** - Advanced statistical methods

Electromagnetic Analysis:

- **EM emissions** from cryptographic operations
- **Correlate with** secret data
- **Non-invasive** attack method

Cache Attacks

Cache Timing:

- **Monitor cache access** patterns
- **Extract information** from cache misses/hits
- **Works on** shared cache systems

Implementation:

```python
import time
import psutil

def cache_timing_attack():
    """Simple cache timing attack"""
    # Flush cache
    data = [0] * 1024 * 1024  # 1MB array

    # Measure access time
    start_time = time.perf_counter()
    _ = data[0]  # Should be cache hit
    end_time = time.perf_counter()
```

Differential Cryptanalysis

Basic Concept

- **Study differences** between plaintext pairs
- **Analyze how differences** propagate through cipher
- **Find patterns** that reveal key information
- **Works on** block ciphers and hash functions

Differential Characteristics

- **Input difference** - XOR of two plaintexts
- **Output difference** - XOR of corresponding ciphertexts
- **Probability** - Likelihood of characteristic
- **Round function** - How differences propagate

Example: Simple Block Cipher

```python
def simple_block_cipher(plaintext, key, rounds=4):
    """Simple block cipher for demonstration"""
    state = plaintext

    for round in range(rounds):
        # XOR with round key
        state = state ^ key[round % len(key)]

        # Simple substitution
        state = ((state << 1) | (state >> 7)) & 0xFF

        # Simple permutation
        state = ((state & 0x0F) << 4) | ((state & 0xF0

    return state
```

Modern Attack Vectors

Fault Attacks

How It Works

- **Introduce faults** during computation
- **Analyze faulty outputs** to extract secrets
- **Methods:** Voltage glitching, clock glitching, laser attacks
- **Targets:** Smart cards, embedded devices

Types of Faults

- **Single-bit faults** - Flip one bit
- **Multi-bit faults** - Flip multiple bits
- **Timing faults** - Skip operations
- **Instruction faults** - Skip instructions

Example: RSA Fault Attack

```python
def rsa_sign(message, private_key, n, d):
    """RSA signature with potential fault"""
    m_hash = hash(message) % n

    # Simulate fault injection
    if fault_injection_point():
        d = d ^ 0x1000  # Flip a bit in private key

    signature = pow(m_hash, d, n)
    return signature

def fault_attack_rsa():
    """Extract RSA private key using fault attack"""
    # This is a simplified example
    n = 0x1234567890ABCDEF  # Public modulus
    d = 0x9876543210FEDCBA  # Private exponent
```

Cache Attacks

Cache-Based Attacks

Flush+Reload:

- **Flush** target memory from cache
- **Trigger** cryptographic operation
- **Reload** and measure access time
- **Determine** which data was accessed

Prime+Probe:

- **Prime** cache with known data
- **Trigger** cryptographic operation
- **Probe** cache to see what was evicted
- **Infer** secret data from cache state

Implementation Example

```python
import time
import mmap
import os

class CacheAttacker:
    def __init__(self, target_size=4096):
        self.target_size = target_size
        self.cache_line_size = 64

    def flush_cache(self, address):
        """Flush specific cache line"""
        # Use clflush instruction (simplified)
        pass

    def measure_access_time(self, address):
        """Measure memory access time"""
```

Spectre and Meltdown

Spectre Attack

Speculative Execution:

- **CPU predicts** future execution paths
- **Executes instructions** speculatively
- **Leaves traces** in cache and branch predictors
- **Attacker can read** these traces

Variants:

- **Spectre v1** - Bounds check bypass
- **Spectre v2** - Branch target injection
- **Spectre v4** - Speculative store bypass

Meltdown Attack

Memory Access:

- **Access kernel memory** from user space
- **Speculative execution** allows access
- **Cache side effects** reveal data
- **Works on** most Intel processors

Mitigations

Hardware:

- **Intel CET** - Control-flow Enforcement Technology
- **ARM Pointer Authentication**
- **AMD Shadow Stack**

Software:

Practical Attack Tools

Attack Frameworks

Popular Tools

Cryptanalysis:

- **Cryptool** - Educational cryptanalysis
- **John the Ripper** - Password cracking
- **Hashcat** - GPU-accelerated cracking
- **Aircrack-ng** - WiFi security testing

Side-Channel:

- **ChipWhisperer** - Hardware security testing
- **Oscilloscope** - Power analysis
- **Logic analyzer** - Signal analysis

Python Libraries

```python
# Cryptographic attacks
import hashlib
import hmac
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import

# Timing attacks
import time
import statistics

# Frequency analysis
from collections import Counter
import matplotlib.pyplot as plt

# Example: Password cracking
def crack_password_hash(target_hash, wordlist):
```

Frequency Analysis Tool

Complete Implementation

```python
import matplotlib.pyplot as plt
from collections import Counter
import string

class FrequencyAnalyzer:
    def __init__(self):
        self.english_freq = {
            'E': 12.7, 'T': 9.1, 'A': 8.2, 'O': 7.5, '
            'N': 6.7, 'S': 6.3, 'H': 6.1, 'R': 6.0, 'D
            'L': 4.0, 'C': 2.8, 'U': 2.8, 'M': 2.4, 'W
            'F': 2.2, 'G': 2.0, 'Y': 2.0, 'P': 1.9, 'B
            'V': 1.0, 'K': 0.8, 'J': 0.2, 'X': 0.2, 'Q
        }

    def analyze(self, text):
        """Analyze letter frequencies in text"""
```

Usage Example

```python
# Example usage
analyzer = FrequencyAnalyzer()

# Analyze ciphertext
ciphertext = "KHOOR ZRUOG"
cipher_freq = analyzer.analyze(ciphertext)

print("Ciphertext frequencies:")
for letter, freq in cipher_freq.items():
    print(f"{letter}: {freq:.2f}%")

# Plot comparison
analyzer.plot_frequencies(cipher_freq, "Caesar Cipher

# Find most likely mapping
def find_mapping(cipher_freq, english_freq):
```

Timing Attack Implementation

Vulnerable Implementation

```python
import time
import random
import statistics

def vulnerable_compare(a, b):
    """Vulnerable string comparison"""
    if len(a) != len(b):
        return False

    for i in range(len(a)):
        if a[i] != b[i]:  # Early exit reveals positio
            return False
    return True

def timing_attack(target, charset, max_length=8):
    """Extract string using timing attack"""
```

Secure Implementation

```python
def secure_compare(a, b):
    """Constant-time string comparison"""
    if len(a) != len(b):
        return False

    result = 0
    for i in range(len(a)):
        result |= ord(a[i]) ^ ord(b[i])

    return result == 0

def hmac_verify(message, signature, key):
    """Secure HMAC verification"""
    expected = hmac.new(key, message, hashlib.sha256).
    return secure_compare(signature, expected)
```

Defensive Measures

Countermeasures

Against Timing Attacks

Constant-Time Implementation:

- **Avoid early returns** in comparisons
- **Use bitwise operations** instead of branches
- **Add random delays** to mask timing
- **Use hardware** constant-time instructions

Code Example:

```
def constant_time_compare(a, b):
    """Constant-time string comparison"""
    if len(a) != len(b):
        return False

    result = 0
    for i in range(len(a)):
        result |= ord(a[i]) ^ ord(b[i])

    return result == 0
```

Against Side-Channel Attacks

Power Analysis Protection:

- **Randomize** execution order
- **Add noise** to power consumption
- **Use masking** techniques
- **Implement** countermeasures in hardware

Cache Attack Protection:

- **Flush sensitive data** from cache
- **Use dedicated** cache lines
- **Implement** cache partitioning
- **Monitor** cache access patterns

Secure Programming Practices

General Principles

Input Validation:

- **Validate all inputs** before processing
- **Use whitelist** approach when possible
- **Sanitize data** before cryptographic operations
- **Implement** proper error handling

Memory Management:

- **Clear sensitive data** from memory
- **Use secure memory** allocation
- **Implement** memory protection
- **Avoid** memory leaks

Cryptographic Best Practices

Key Management:

- **Generate keys** using secure random
- **Store keys** securely
- **Rotate keys** regularly
- **Use different keys** for different purposes

Implementation:

- **Use established** cryptographic libraries
- **Follow** security guidelines
- **Test thoroughly** for vulnerabilities
- **Keep libraries** updated

```
import secrets
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import

def secure_key_generation():
```

Practical Exercises

Exercise 1: Frequency Analysis

Task: Break a monoalphabetic substitution cipher

1. **Implement** frequency analysis tool
2. **Analyze** given ciphertext
3. **Find** most likely letter mapping
4. **Decrypt** the message
5. **Verify** correctness

Ciphertext: "WKH TXLFN EURZQ IRA MXPSV RYHU WKH ODCB GRJ"

Exercise 2: Timing Attack

Task: Extract password using timing attack

1. **Implement** vulnerable password check
2. **Create** timing attack tool

Exercise 3: Side-Channel Analysis

Task: Analyze power consumption patterns

1. **Implement** simple cryptographic function
2. **Simulate** power consumption
3. **Analyze** patterns in power data
4. **Extract** secret information
5. **Implement** countermeasures

Exercise 4: Differential Analysis

Task: Find differential characteristics

1. **Implement** simple block cipher
2. **Test** different input pairs
3. **Find** differential characteristics
4. **Analyze** probability distributions
5. **Use** characteristics for key recovery

Common Vulnerabilities

Implementation Errors

❌ **Weak Random Number Generation:**

- Using `random()` instead of `secrets`
- Predictable seeds
- Insufficient entropy

❌ **Timing Vulnerabilities:**

- Early returns in comparisons
- Different execution paths
- Cache-dependent operations

❌ **Memory Issues:**

- Not clearing sensitive data
- Buffer overflows
- Memory leaks

Design Flaws

❌ **Weak Algorithms:**

- Using deprecated algorithms
- Insufficient key lengths
- Poor key scheduling

❌ **Protocol Issues:**

- Reusing keys
- Weak authentication
- Missing integrity checks

❌ **Side-Channel Leakage:**

- Power consumption patterns
- Electromagnetic emissions
- Cache access patterns

Modern Attack Trends

Emerging Threats

AI-Powered Attacks:

- **Machine learning** for pattern recognition
- **Automated** vulnerability discovery
- **Enhanced** side-channel analysis
- **Adaptive** attack strategies

Quantum Computing:

- **Shor's algorithm** breaks RSA/ECC
- **Grover's algorithm** halves key strength
- **Post-quantum** cryptography needed
- **Hybrid** classical-quantum attacks

Defense Strategies

AI Defense:

- **ML-based** anomaly detection
- **Automated** vulnerability scanning
- **Intelligent** threat response
- **Adaptive** security measures

Quantum Resistance:

- **Lattice-based** cryptography
- **Code-based** cryptography
- **Multivariate** cryptography
- **Hash-based** signatures

Future: Cryptanalysis is evolving with AI and quantum computing - we must stay ahead of the curve!

Questions?

Let's discuss cryptanalysis! 💬

Next Week: We'll explore stream ciphers and learn about modern symmetric encryption!

Assignment: Implement and test various attack techniques on classical ciphers!