Computer Science Department, University of Crete

CS-458: Introduction to Cryptography

Instructor: Harry Manifavas

Project

Evaluating AES Implementations for Side-Channel Resistance

Deadline: 10/01/2026, 23:59

v0.1

Notes:

- You will have approximately **2 months** to complete the project. There will be **no extension**. The project accounts for **30%** of the overall grade. It can be done in **teams of up to 3 people**.
- You are **allowed to use AI tools** or code found online to build a complete system. AI use will not affect grading provided the student demonstrates full understanding during the oral defense. At the end, there will be an **oral examination** and similarity checking.
- Due to the workload required, you should **start early** on the necessary research as well as the development. Ensure you **fully understand all the concepts** involved and **design the architecture properly** before starting the actual work. The main challenge isn't the amount of code you have to write, it's the conceptual difficulty of understanding the statistical analysis (Part 3) and the countermeasures (Part 4).

Table of Contents

Introduction 2
Project Structure 2
Part 1: Preparatory Phase 3
Part 2: Experimental Setup 4
Part 3: Side-Channel Analysis 5
Part 4: Countermeasures & Evaluation 5
Part 5 – Deliverables, Final Report and Presentation 6
Indicative Timetable and Rubric 7
Instructions on How AI Tools Can Help 8
References 9

Introduction

The Advanced Encryption Standard[1] (AES) is the most widely deployed symmetric-key block cipher today, used in Transport Layer Security protocol (TLS), disk encryption, and countless secure applications. While the algorithm itself is mathematically secure, **implementation flaws can leak information through physical or software side channels**, such as execution time or memory access patterns.

In this project, you will:

- Study how AES implementations may leak secret information.
- Select and analyze two AES implementations in **C** (one safe, one unsafe)**.**
- Design and run experiments to **test their resistance** to software side-channel attacks[2].
- Document your findings, propose countermeasures for unsafe implementations, and evaluate trade-offs between performance and security.

The project emphasizes **critical evaluation** rather than building new cryptographic algorithms. Your goal is to *measure, reason, and judge* the implementation's robustness.

Project Structure

Part	Title	Goal	Deliverable (report chapter)
1	Preparatory Phase	Understand AES internals and side-channel principles	Report on Part 1 tasks
2	Experimental Setup	Compile, instrument, and benchmark AES code	Report on Part 2 tasks
3	Side-Channel Measurements & Analysis	Collect and interpret timing data	Report on Part 3 tasks
4	Countermeasures & Evaluation	Propose or test mitigation strategies	Report on Part 4 tasks
5	Final Report & Presentation	Summarize results, insights, and lessons	Report + slides + source code

[1] https://en.wikipedia.org/wiki/Advanced_Encryption_Standard, 2526F-CS458-09a-Symmetric-Crypto-IV-AES-I.pdf, 2526F-CS458-09b-Symmetric-Crypto-IV-AES-I.pdf
[2] https://en.wikipedia.org/wiki/Side-channel_attack

Part 1: Preparatory Phase

Goal
To understand how AES can leak information during execution and define the experimental plan.

Tasks

1. Review:
   - AES structure (SubBytes, ShiftRows, MixColumns, AddRoundKey).
   - Software-observable[3] side channels (Timing[4] and cache-based[5])
   - "Constant-time coding" practices
2. Select one implementation that is known to be **non-constant-time** and one safe AES implementation (in C), e.g.:
   - **Tiny-AES-C[6]**
     - A very compact, portable C implementation of AES
     - Designed for simplicity and small footprint (embedded systems, small code size) rather than maximum hardening or constant-time behavior
     - A good "naïve" implementation (likely to show side-channel leakage) which you can analyze
   - **OpenSSL AES** (reference implementation)
     - Part of the widely used OpenSSL cryptographic library, which contains highly optimized AES implementations, often using hardware acceleration (AES-NI) when available
     - Performance-oriented, mature, and deployed in many real systems—so students analyzing it will get insight into "production grade" code
     - However, depending on build options and platform, constant-time guarantees via code structure may vary; thus it also makes an interesting target for side-channel testing
   - **BearSSL AES[7]** (constant-time reference)
     - BearSSL explicitly documents multiple AES variants: "big" (table-based, non-constant-time) and "ct" / "ct64" (constant-time) implementations
     - Using BearSSL's constant-time AES in the project gives you an implementation to test against, ideal for comparison with a "leaky" version
3. Define the **attack model**:
   - What will you measure (execution time, cache effects)?
   - What is the attacker's capability (control over inputs, observe execution time and cache behavior, access to partial plaintext/ciphertext)?
   - The attacks must be reproducible and infer information about the **secret key** or **plaintext**

---

[3] Physical side channels (power, EM, acoustic attacks, fault attacks) are out of scope as they require hardware equipment and specialized environments.
[4] https://en.wikipedia.org/wiki/Timing_attack
[5] https://en.wikipedia.org/wiki/Cache_timing_attack
[6] https://github.com/kokke/tiny-AES-c
[7] https://www.bearssl.org/constanttime.html?utm_source=chatgpt.com

Deliverables: one chapter

- Description of AES, chosen implementations, and attack model
- Initial references used and summary of concepts

Part 2: Experimental Setup

Goal

To create a controlled test environment for timing or cache analysis. Build a controlled, repeatable, and statistically meaningful measurement environment.

Tasks

1. Write a small C program or Python wrapper that repeatedly encrypts data using your AES implementation
    - Call the AES encryption routine many times (e.g., 10,000–100,000 encryptions)
    - Structure the harness so that encryption dominates the runtime
2. Collect timing measurements using one or more of:
    - clock() from <time.h>, low accuracy
    - rdtsc (CPU cycle counter), high accuracy
    - std::chrono in C++, high accuracy
    - timeit in Python, medium accuracy
    - Use the same timer for all experiments
3. Generate random plaintexts and measure how execution time varies with input.
    - Fully random (baseline)
    - Fixed first byte, others random
    - Structured (e.g., all-zero plaintexts)
4. Use statistical analysis to detect possible correlations
    - Raw timing data is useless without statistics
    - Start with average and variance per input class
    - As side-channel leakage is often subtle, you may need thousands or tens of thousands of measurements to detect it
5. Visualize the results (Python matplotlib, pandas)
    - Histograms, Scatter plots, Boxplots, etc.

Deliverables: one chapter

- Source code for measurement harness
- Raw datasets of measurements (csv)
- Plots and visualizations showing timing distributions
- Short methodological description:
    - Timers used
    - Machine specs
    - Number of encryptions
    - Input generation strategy
    - Any noise-control measures taken

Part 3: Side-Channel Analysis

Goal
Examine the timing dataset collected in Part 2 and determine whether it reveals side-channel leakage, whether execution time depends on secret data (the key) or controlled data (plaintext). This phase requires careful thinking, not just computation. You must distinguish between:

- Noise (random fluctuations, OS scheduling, CPU load)
- Leakage (systematic timing differences caused by secret-dependent operations)

The goal is to evaluate whether the AES implementation behaves in a way that is consistent with constant-time coding or whether measurable patterns expose sensitive information.

Tasks

1. Process your group measurement data to identify leakage patterns
    - Does time depend on specific plaintext bytes or key bits?
    - Does the execution time increase or decrease for certain plaintext bytes?
    - Do some inputs cause larger variance?
    - The unsafe one should show clearer patterns
    - The constant-time one should show flat, uniform timing behavior
2. Conduct statistical tests
    - Statistical testing provides evidence that observed patterns are real and not due to random fluctuations
    - Welch's t-test or ANOVA on timing groups
    - Correlation between inputs and time
3. Interpret the results:
    - Explain what the data means, not just report a number
    - Explain why a pattern may exist (e.g., table-based AES implementation, secret-dependent branching)
    - Distinguish between random noise and meaningful leakage.
    - Operating systems introduce noise, small timing variations are normal and not every difference is meaningful
    - Articulate why some patterns likely represent random CPU jitter while other patterns are systematic differences caused by the implementation

Deliverables: one chapter

- Analysis section with statistical tests, plots, and conclusions.
- Discussion of whether an AES implementation is constant-time with supporting evidence
- How the results compare between safe and unsafe AES implementations

Part 4: Countermeasures & Evaluation

Goal
The goal of this part is not to write a completely new AES implementation, but rather:

- To examine practical mitigation techniques
- To test whether they improve constant-time behavior

- To understand the trade-offs engineers face between performance, security, and portability

Tasks

1. Explore or implement one of:
    - Pre-computed S-boxes in constant memory
    - Elimination of secret-dependent branching
    - Use of hardware AES instructions (AES-NI) if available
2. Re-run timing tests and compare leakage reduction. To ensure results are comparable, apply the same measurement harness as in previous parts:
    - Same plaintext generation
    - Same number of samples
    - Same timing function
    - Same CPU pinning (if used)
3. Security improvements often come at a cost. Measure performance impact. interpret these results and explain why the changes occur.
4. Discuss residual risks and how modern systems (e.g., OpenSSL, Linux kernel) mitigate them. Even after applying countermeasures, some risks remain. Discuss how real systems address these issues.

Deliverables: one chapter

- Modified code implementing mitigation measures.
- Comparative (before/after) results, plots and performance table.
- Discussion of real-world implications (e.g., TLS, disk encryption).

Part 5 – Deliverables, Final Report and Presentation

Goal

Proper documentation and presentation. Your grade also depends heavily on how clearly and accurately you document your methodology, present your findings, and demonstrate understanding during the oral examination.

Report structure

1. **Introduction**
    - This section sets the context of your project (motivation, goals, and scope)
2. **Preparatory phase**
    - Theory summary behind the practice and implementation choices. sets the stage for the experiments
3. **Experimental setup**
    - Describe how you collected data in enough details that another student could reproduce your measurements (environment, parameters, and instrumentation)
4. **Results and analysis**
    - Plots, tables, interpretation. “Does this AES implementation leak through timing? Why or why not?”
5. **Countermeasures**
    - Explain both security effects and performance trade-offs (implementation, evaluation, performance).

6. **Conclusions**
    - Lessons learned, remaining vulnerabilities, future work.
7. **Appendices**
    - Code snippets, AI prompts, references, additional plots or large tables.

Presentation

- 20 - 25 slides summarizing your methodology and findings.
- Each team will present and answer questions during oral defense.

Software

Students must submit **all software artifacts** required to reproduce their experiments. This includes:

- **AES implementations** used or modified during the project (both the "safe" and "unsafe" versions).
    - Clearly indicate whether they are original, modified, or third-party (e.g., Tiny-AES-C, OpenSSL AES, BearSSL AES).
- **Measurement harness**:
    - Source code (C or Python) used to perform encryption runs and collect timing or cache data.
    - Configuration parameters (e.g., number of runs, key/plaintext sizes, timing function used).
- **Data analysis scripts**:
    - Scripts (Python, etc.) used to process measurements, perform statistical analysis, and generate plots.
- **Build and run instructions**:
    - A *Readme file* describing compilation, execution, and reproduction steps on a standard Linux or Windows system.
    - Mention required libraries, compiler versions, or Python dependencies.
- **Results folder**:
    - Collected datasets (e.g., CSV timing files) and final plots.
- All submitted code must be self-contained, runnable, and organized into a logical folder structure, for example:

/AES-SideChannel-Project

```
├── src/          # C source code
├── data/         # Collected measurement files
├── analysis/     # Python scripts
├── results/      # Plots and figures
└── README.md     # Setup and run instructions
```

Indicative Timetable and Rubric

Weeks	Phase	Criteria	Weight	Description
1–2	Preparatory phase	Understanding of AES and side channels	**10%**	Demonstrates comprehension of AES internals and leakage principles.

3–4	Experimental setup	Functional measurement harness	**20%**	Accurate and repeatable test environment; proper data collection.
5–6	Analysis	Valid statistical reasoning	**25%**	Correct interpretation of timing data and clear justification of conclusions.
6–7	Countermeasures	Mitigation and evaluation	**25%**	Implementation or discussion of countermeasures; performance comparison.
7–8	Final Report & presentation	Clarity, structure, and teamwork	**10%**	Professional report and oral defense.
—	Creativity & originality	Novel approaches or deeper insight beyond baseline expectations.	**10%**	
—	**Total**		**100%**	

For the **creativity** criteria, you may consider the following options:

Auxiliary tasks	
Category	**Example Tasks**
Automation	Write a script to automatically test different AES variants or compilers.
Visualization	Create interactive graphs of timing variance or cache traces.
Comparison	Evaluate several AES implementation (e.g., OpenSSL vs. Tiny-AES).
Hardware awareness	Compare results with AES-NI on/off (where available).
Statistical rigor	Apply correlation power analysis (CPA) principles on timing data.

Instructions on How AI Tools Can Help

AI tools (ChatGPT, Gemini, Windows copilot, GitHub Copilot, code tools, etc.) can greatly enhance **efficiency** and **creativity** in this project. Their use is **allowed and encouraged** if transparent. Here are phase-specific instructions:

Guidelines for ethical use of AI

1. **Transparency**: Always cite AI-generated content, whether text, code, or visualizations. As already stated, there is no penalty for AI use.
2. **Supplement, not replace**: Use AI as a helper but *ensure original understanding and critical thinking*.
3. **Collaboration**: Share AI findings within the team to foster mutual learning and avoid over-reliance by individuals

Project Phase	Recommended AI Use
Preparatory phase	Summarize papers on AES side-channels, constant-time programming.
Experimental setup	Generate example code for timing loops, input generation, and plotting.
Analysis	Suggest statistical tests and visualization formats.
Countermeasures	Ask AI to propose ways to eliminate lookup-table leaks.

General use

- **Experiment** with various AI tools to identify those best suited to specific phases of the project
- **Collaboration**: You may utilize AI tools for sharing notes, task delegation, tracking progress, and maintaining team communication.
- **Presentation preparation**: Create slide content or draft talking points for the final presentation (e.g., based on the contents of your report as well as the implemented code).
- **Document formatting**: Automate the creation of professional-looking documents (e.g., for the roadmap) using timelines, flowcharts, tables, etc.
- **Debugging**: Leverage AI to troubleshoot buggy code
- **Code checking**: Validate or optimize code using AI tools for enhanced efficiency.

References

- **Essential:**
  - FIPS-197, *Advanced Encryption Standard (AES)*
  - Kocher et al., *Timing Attacks on Implementations of Diffie–Hellman, RSA, DSS, and Other Systems* (CRYPTO 1996)
  - Daniel J. Bernstein, Cache-timing attacks on AES, 2005, https://cr.yp.to/antiforgery/cachetiming-20050414.pdf
  - Osvik et al., *Cache Attacks and Countermeasures: The Case of AES* (CT-RSA 2006)
  - BearSSL constant-time coding guidelines: https://www.bearssl.org/constanttime.html

- **Further reading / tools:**
  - https://github.com/kokke/tiny-AES-c
  - https://github.com/openssl/openssl
  - https://github.com/IAIK/constant-time-test
  - https://cryptojedi.org/peter/data/attack.pdf
  - *Understanding Cryptography*, Paar & Pelzl, Springer 2010 / 2024