Cycle Based Hardware Simulation of Cache-Timing Attacks and Analysis of Countermeasures

Xavier Bear

(1) Problem Statement:

AES, like many cryptographic algorithms, is susceptible to side-channel attacks that exploit physical implementations rather than theoretical weaknesses. Among these, cache-timing attacks represent a significant threat, as they use the time variations in the execution of cryptographic operations to infer sensitive information about the secret key.

This project is motivated by work from Daniel J. Bernstein, who demonstrated the feasibility of extracting AES keys through cache-timing analysis. These attacks exploit the architectural characteristics of some CPUs to observe data access patterns and deduce information about the encryption processes. The vulnerability comes from the fact that hitting or missing the cache can lead to large, measurable differences in execution times. These differences can be analyzed to reveal information on possible encryption keys, thus compromising the encrypted data.

This project creates a simulation of the computer's hardware in order to emulate the mechanics of cache timing attacks. This allows a detailed exploration of the attack's dynamics and offers an insight on the differences in execution time which stem from cache hits and misses. In tandem with replication cache-timing attacks, this project evaluates simulations of various countermeasures proposed by Eran Tromer, Dag Arne Osvik, and Adi Shamir. These countermeasures include dynamic table storage, cache state normalization and process blocking.

(2) Relevance to Privacy:

The relevance of cache-timing attacks on AES to privacy is both technological and societal. Technologically, these attacks exploit the microarchitectural characteristics of modern processors, potentially allowing attackers to bypass the cryptography of AES without directly breaking the algorithm. This vulnerability is a significant risk to encrypted data's confidentiality, impacting a wide array of applications from web browsing to communication systems. Since our reliance on digital technology grows, ensuring the strength of cryptographic systems against such side-channel attacks becomes significant towards maintaining the privacy of digital communications.

Although there is no notable instance of a cache-timing attack in the real world, there have been similar attacks that have occurred. One of the more famous side-channel attacks that shares principles with cache-timing attacks is the "Spectre" and "Meltdown" vulnerabilities discovered in 2018. These hardware vulnerabilities affected almost all modern processors and allowed attackers to exploit speculative execution to access protected memory areas. While not directly cache-timing attacks, Spectre and Meltdown showed the potential for side-channel attacks to compromise system security at a hardware level, leading to widespread concern and prompting significant efforts towards mitigating these vulnerabilities in hardware.

(3) Simulation and Implementation of Cache-Timing attack:

The implementation of the cache-timing attack on AES encryption was through a simulation. This implementation required several components designed to mimic the condition under which a real-world attack might occur. Below is step-by-step plan detailing the process, including tool selection, environment setup, and data collection:

(3.1) Tool Selection:

- **Python**: Chosen for its simplicity and many libraries
- **Cryptography Library:** Used for its cryptographic primitives allowing me to simulate AES operations
- **Matplotlib & Numpy**: Used for data visualizations and numerical operations respectively.

(3.2) Environment Setup:

1. **Cycle Counter:** A class designed to simulate counting cpu cycles during the execution of cryptographic operations. Counting by "cycles" allowed me to remove noise from the online jupyter notebook I used compared to if I just used time.
2. **Cache Simulation:** The Cache is modeled as a collection of cache lines, with each line capable of storing data corresponding to a specific memory address. The cache also had a set access time for cache hits and access time for cache misses.
    a. **The Cache Access:** This method determines if the cache hit or missed which is simulated by checking if the given address matches the data stored in any cache line determined by the address modulo the cache size and adjusted for the cache's associativity.
    b. **Cache Replacement:** On a miss, the least recently used line within the set is replaced with new data, emulating a basic cache replacement policy.

3. **CPU:** The CPU class simulates the AES encryption operations, accessing the cache based on key/data inputs, and uses CycleCounter to measure execution time.

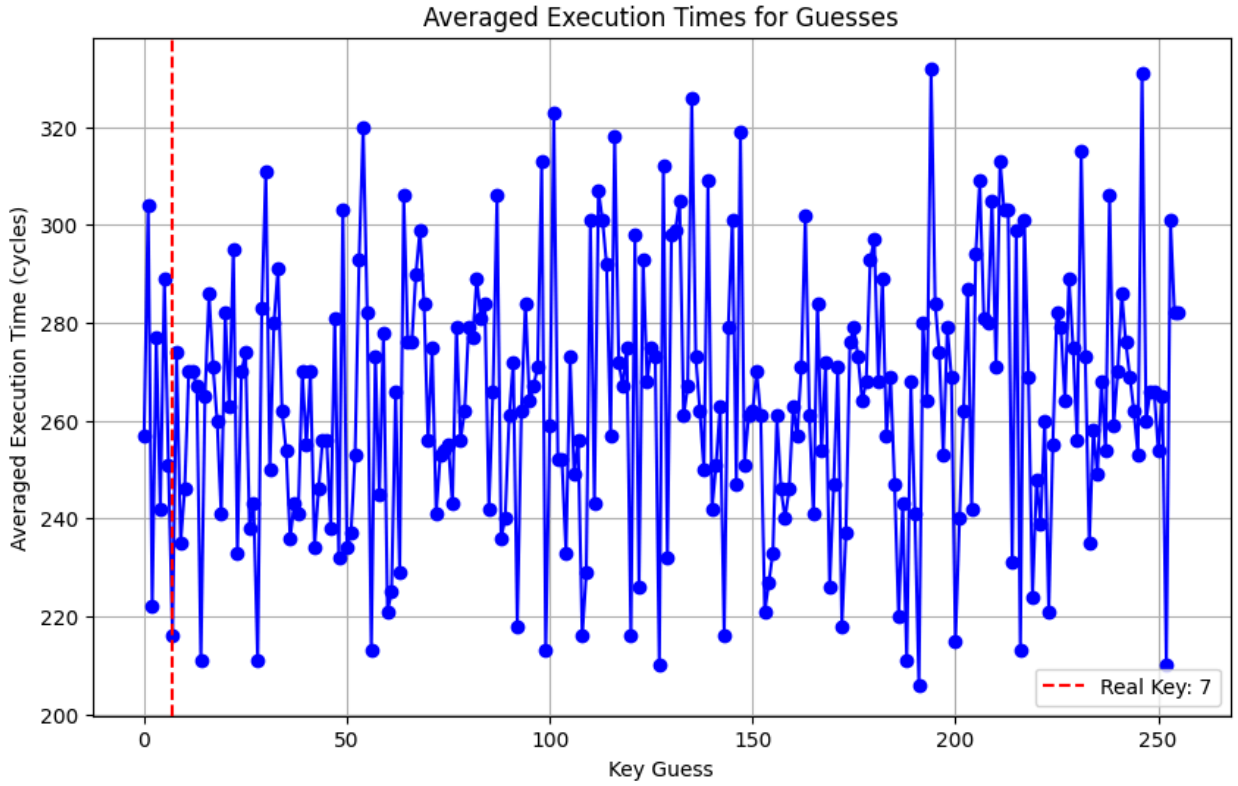
(3.3) Data Collection

1. **Simulation Class:** This class starts the setup, initializing the cache, cycle counter, and CPU simulation with a given real key. It also simulates the victim and attacker's actions.
2. **Execution Time Measurement:** For every possible key guess, the simulation executes a victim operation and measures the number of cycles it takes to complete using the attacker_monitor method. This process is repeated for each key guess to collect timing data.

3. **Visualization:** Using Matplotlib, the collected data is visualized to identify patterns that might indicate the correct key based on execution times.

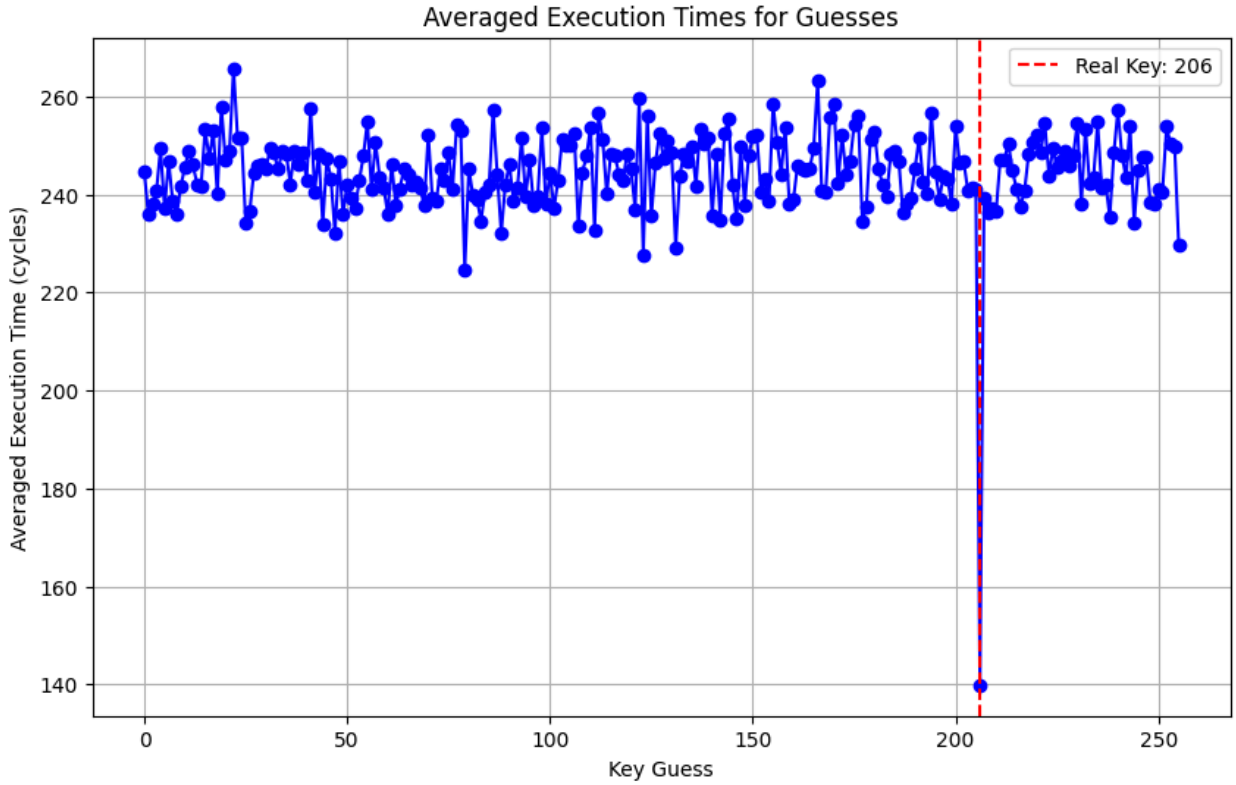
(4) Analysis of Initial Cache-Timing Attack:

Fig 1. (Guesses for 8 Bit key, (One measurement per guess) for Standard AES)



In the simulation, the execution times observed for operations using the real AES keys were generally lower when compared to those using the incorrect keys. With this graph showing one measurement per key guess, the data collected shows that execution of the real key fell lower than most, but was not significant enough to stand out due to noise added to the simulation.

Fig 2. (Guesses for 8 Bit key, (10 measurements (averaged) per guess) for Standard AES)



By taking ten measurements for each key guess, with the results averaged, the real key stands out amongst the guesses. By aggregating multiple measurements, the impact of the outliers and randomly added noise is diminished. This is a consistent pattern of cache behavior that is more accurately captured through repeated observations. With noise reduced, the difference in the execution times are amplified.

(5) Implementation of Countermeasures

(5.1) Dynamic Table Storage

1. **Implementation:** The DynamicTableAESCPU class simulates dynamic table storage by randomly shuffling the S-Box. The shuffling represents the remapping of sensitive data in memory, making it difficult for an attacker to find patterns based on key guesses.
2. **Simulation Details:** Before the AES operation, the S-Box is shuffled adding an overhead of 50 cycles to simulate the time it would take to remap these tables in a real system. During the AES operation, cache access is made to the S-Box with each access showing the updated positions.

(5.2) Cache State Normalization

1. **Implementation:** The NormalizedCacheAESCPU class simulates cache state normalization. This implementation accesses all possible values related to the S-Box before the actual encryption operation. Pre-accessing data ensures that the cache is consistent.
2. **Simulation Details:** The simulation accesses each S-Box value to "normalize" the cache state. This is simulated by accessing all S-Box values sequentially, followed by the AES operation, where cache accesses are made.

(5.3) Process Blocking:

1. **Implementation:** The CPUBlockProcess class simulates process blocking by adding a fixed number of cycles for each AES operation, regardless of the actual cache behavior. This approach ensures that each operation takes the same amount of time which effectively minimizes timing variations at the high cost of performance.
2. **Simulation Details:** Regardless of the key guess, the method adds a predetermined number of cycles to the cycle counter. This simulates uniform execution time for the encryption operation.

(6) Countermeasure Results

Fig 3. (Guesses for 8 Bit key, (One measurement per guess) for Dynamic Table Storage)

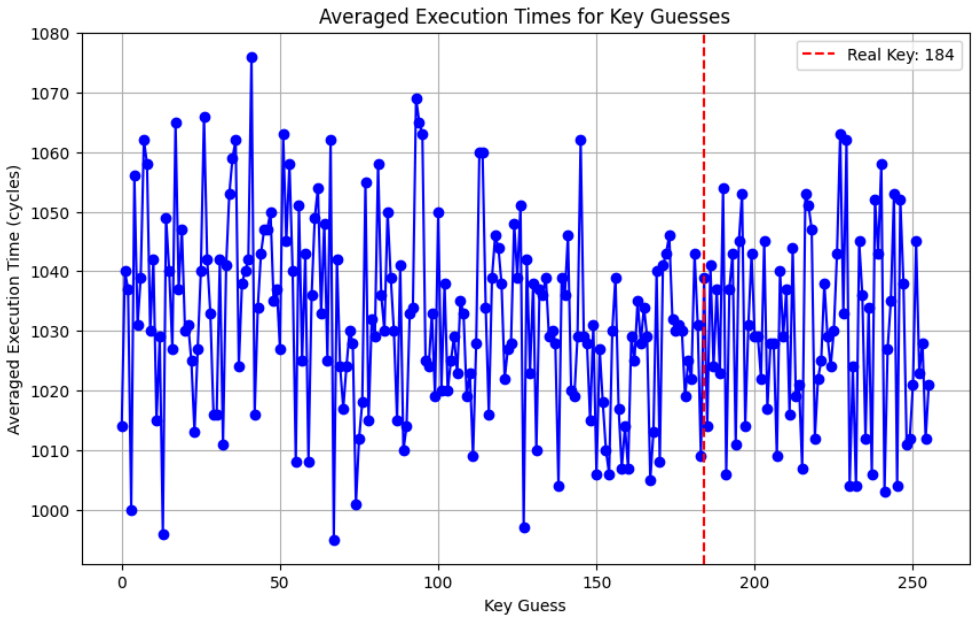


Fig 4. (Guesses for 8 Bit key, (10 measurements (averaged) per guess) for Dynamic Table Storage)

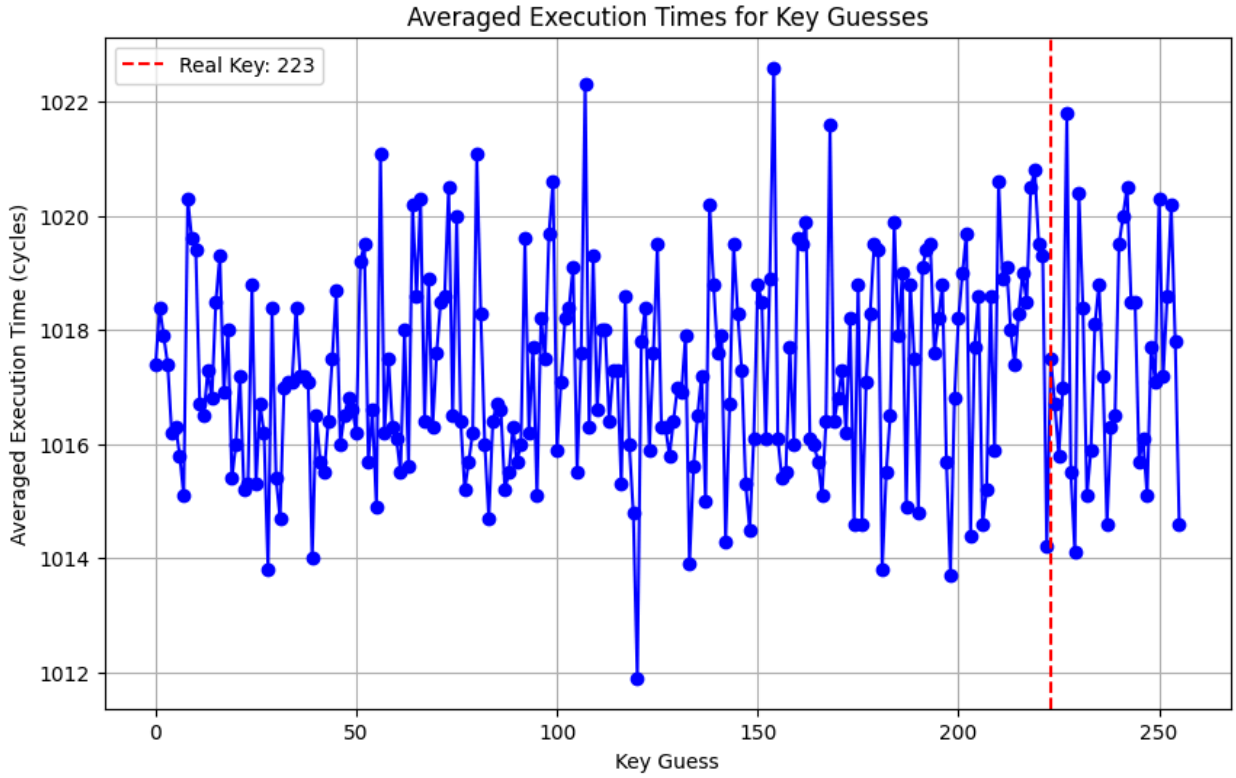


Fig 5. (Guesses for 8 Bit key, (One measurement per guess) for Cache State Normalization)

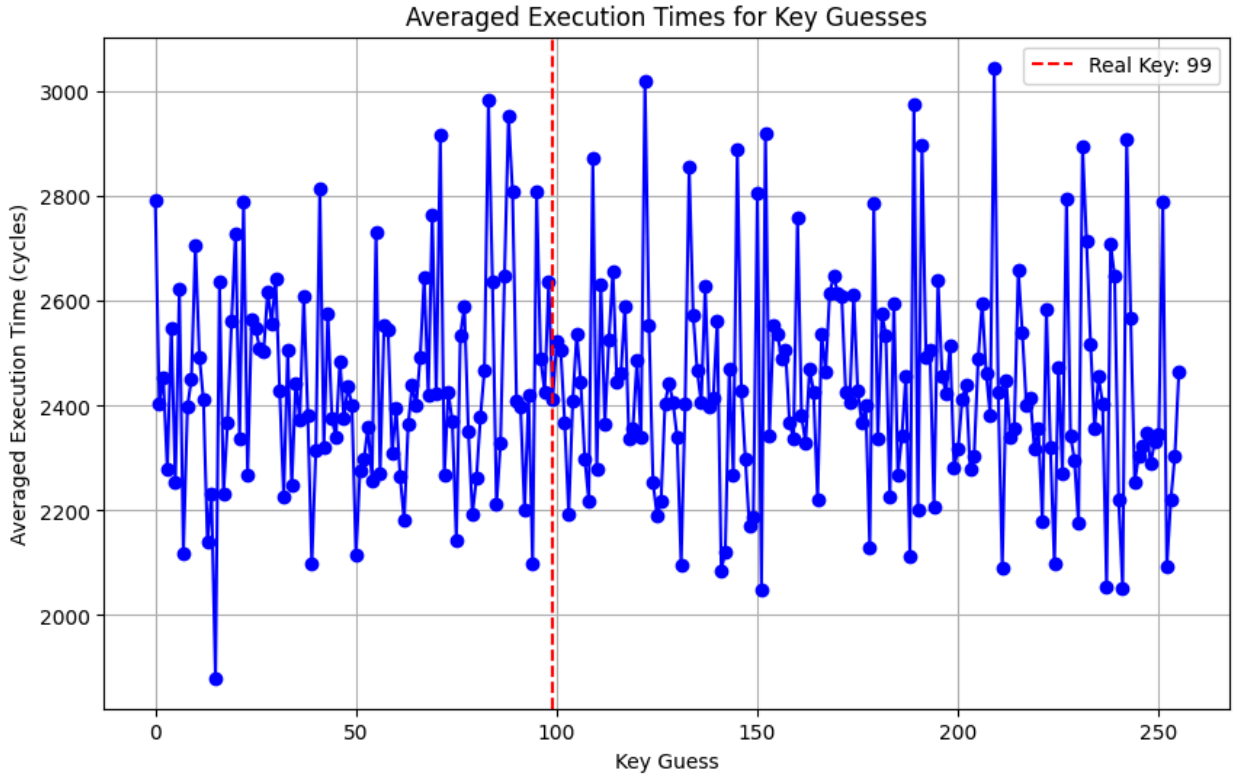


Fig 6. (Guesses for 8 Bit key, (10 measurements (averaged) per guess) for Cache State Normalization)

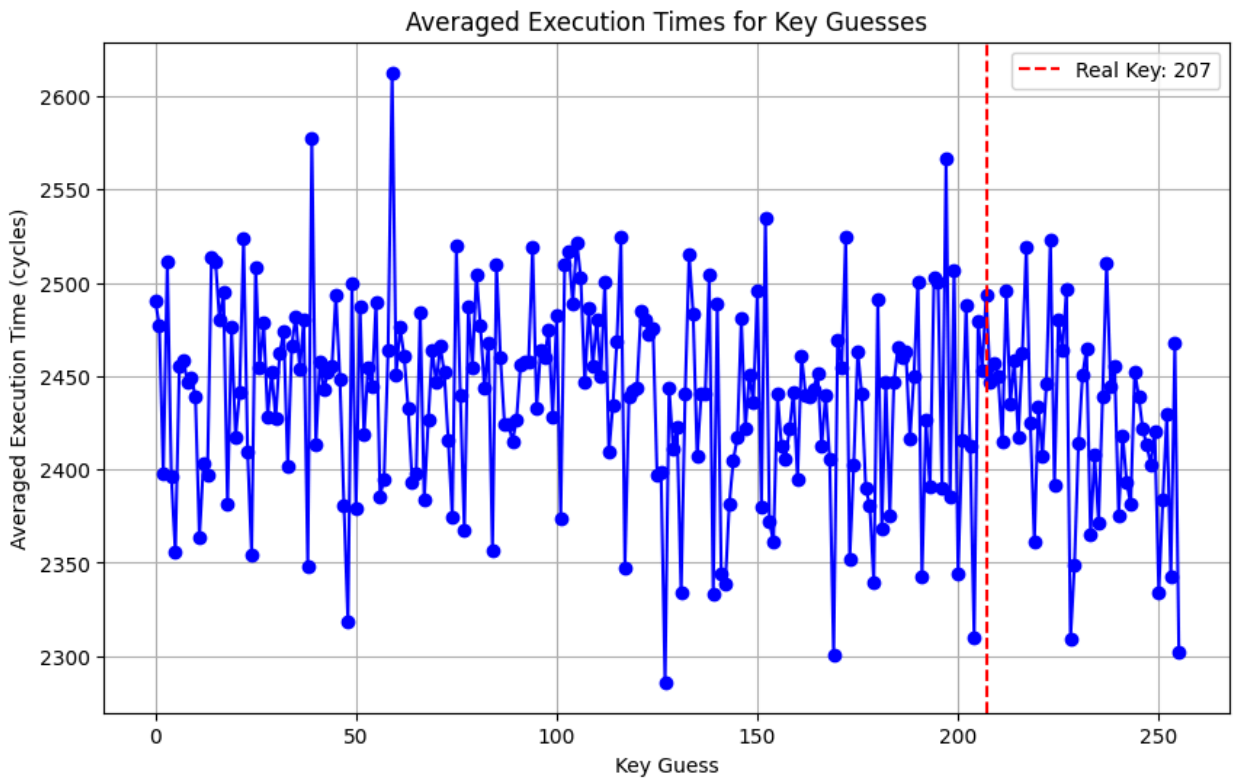


Fig 7. (Guesses for 8 Bit key, (One measurement per guess) for Process Blocking)

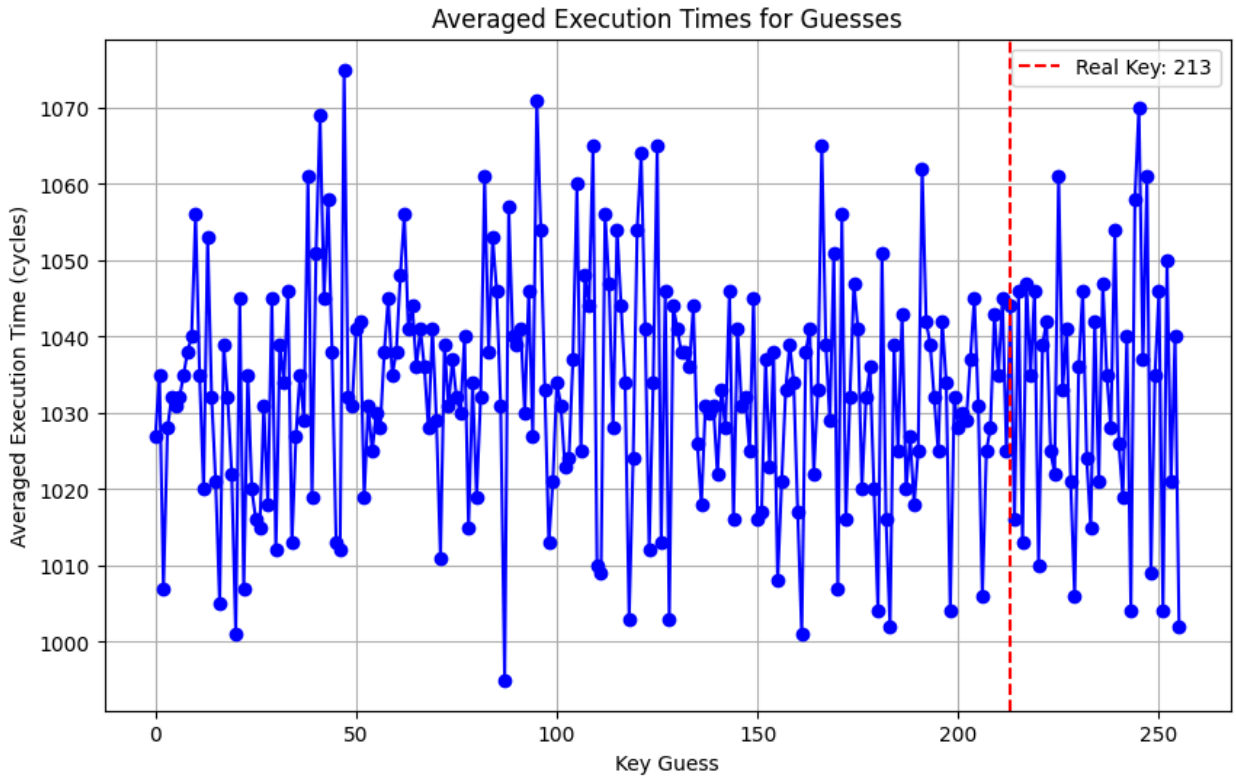
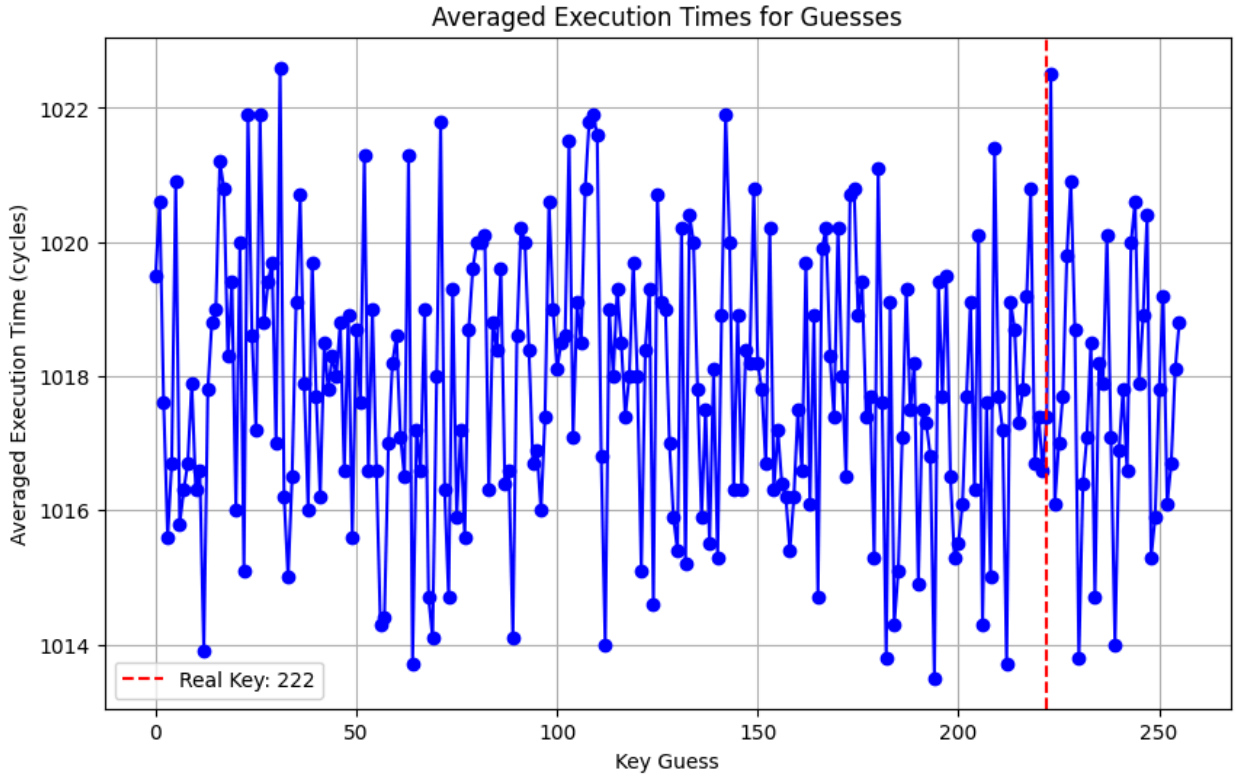


Fig 8. (Guesses for 8 Bit key, (10 measurements (averaged) per guess) for Process Blocking)



In the simulation of countermeasures against cache-timing attacks on AWS, none of the implemented countermeasures allowed for easy identification of the real key based on execution time analysis. The range of cycles observed for Dynamic Table Storage and Process blocking was very similar, clustering around 1012-1022. On the other hand, Cache State Normalization ran at a significantly higher range of 2200-2600 cycles due to all the pre-accessing of possible S-BOx values. This approach, though effective against cache timing, was extremely expensive performance wise.

*Please note that the fixed number of cycles for process blocking was made arbitrarily, chosen to significantly mask any timing discrepancies. With further analysis, you can most likely find a more optimal fixed number for your specific implementation.

(7) Reflection

I can say that I learned more on my own from this project than any project that I have ever done. With the process spanning 25-30 hours, I learned about the complexities of the cache, the vulnerabilities of AES against timing-attacks, and how to abstract hardware to a way that can be implemented as a simulation in software.

One of the significant lessons I learned was how hardware and software interface, especially how cryptographic processes interact with a computer's architecture. When creating

the simulations of the countermeasures, I had to try my best to understand how they prevented the cache-timing attack vulnerability in order to properly simulate them in my implementation.

I will admit that I strayed some ways away from the original project proposal. I chose to simulate rather than execute cache-timing attacks on real hardware because I quickly realized that I was not experienced enough in OS-level programming as well as hosting servers. I also did not have proper time to understand how to implement a bit sliced AES in the simulation so that part was left out completely.

(I) Argument for Assessment Matrix

Privacy Competency: I complete hours of research, and read multiple papers to have a good enough understanding of the attack for implementation in the simulation.

Future Impact: Although the project does not focus on a very modern topic, it does explain the dangers present when sharing data and explains possible mitigations in relative depth.

Broader Perspective: This project definitely leaned towards a more technical side of things therefore my analysis of broader perspectives was limited to privacy of users and performance costs of algorithms. I hope I can go deeper in this subject on a further project

Technical Expertise: I believe that my implementation was realistic and painted a picture as to how the attack and its countermeasures worked. I was required to learn in depth about the architecture of the cache for simulation as well as how to implement the attack and countermeasures.

Personal Growth: I believe that I properly applied what I learned in class at a much deeper level than what was taught in class. I believe that I went from knowing surface level knowledge about the topic to likely being able to hold a conversation about the topic with an industry professional.

(II)Works Cited:

Bernstein, Daniel J. *Cache-Timing Attacks on AES*, Department of Mathematics, Statistics, and Computer Science, cr.yp.to/antiforgery/cachetiming-20050414.pdf.

GfG. “Multilevel Cache Organisation.” *GeeksforGeeks*, GeeksforGeeks, 12 May 2023, www.geeksforgeeks.org/multilevel-cache-organisation/.

Tromer, Eran, et al. “Efficient Cache Attacks on AES, and Countermeasures.” *Efficient Cache Attacks on AES, and Countermeasures*, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, www.cs.tau.ac.il/~tromer/papers/cache-joc-official.pdf. Accessed 29 Feb. 2024.