Ben Septer
Seminar in Applied Mathematics

I. Introduction

A. Topic:

The topic of this paper is to analyze the complexity and efficiency of two variations of the Diffie-Hellman Key Exchange (DHKE). This research will entail a comparative study of Big O Notation and an empirical performance analysis between Elliptical Curve Diffie-Hellman (ECDH) and Finite Field Diffie-Hellman (FFDH). The significance of this study is underscored by the increasing reliance on cryptographic protocols in securing digital communications, a pertinent issue in the evolving landscape of cybersecurity. My background in applied mathematics, cultivated through coursework at Southern New Hampshire University, provides a robust foundation in the mathematical principles underpinning this research. My career aspirations in the cybersecurity field, coupled with my CompTIA A+ and Security+ certifications, have equipped me with a nuanced understanding of cryptographic methods and their practical applications. In reviewing for this research paper, there is a plethora of sources to draw from (academic papers, professional websites, etc.) that reveal a tapestry of theoretical and practical explorations into DHKE. This paper aims to contribute to this body of knowledge by offering a detailed comparison of ECDH and FFDH, two prominent implementations of DHKE, focusing on their algorithmic efficiency and computational complexity both from a theoretical and practical standpoint. This comparison is not only academically interesting but also bears significant implications for the optimization and selection of cryptographic protocols in real-world applications. My particular interest in this topic stems from a fascination with the mathematical elegance of cryptographic algorithms and their critical role in safeguarding information. The objective is to elucidate the intricacies of ECDH and FFDH, potentially identifying areas for improvement or adaptation in varying cybersecurity contexts.

B. Research Question:

The central question I want to answer with this paper is:

What are the comparative complexities and efficiencies of ECDH and FFDH in terms of key exchange, key generation, and Big O Notation?

This question focuses on a specific aspect of cryptographic protocols; the algorithmic complexity and efficiency, measured by key generation/exchange, along with Big O Notation. To address this question within the time constraints of this course, I have conducted practical experimentation using Python scripts to measure the time taken for key generation and key exchange of both ECDH and FFDH algorithms. This empirical approach complements the theoretical analysis, providing a more comprehensive understanding of the algorithm's performance in real-world scenarios. The scope of this research, therefore, encompasses both the theoretical aspects of these algorithms, focusing on their mathematical foundations and algorithmic structures, and the empirical data gathered from the Python script experiments. This dual approach ensures a robust investigation, combining theoretical analysis with practical performance metrics.

C. Information:

Big O Notation:
Big O Notation is a fundamental concept in computer science used to describe the efficiency of algorithms. It's a part of complexity theory, which helps in analyzing algorithms based on the time and space they consume. To better understand Big O Notation, let's discuss a few key concepts involved:

Time Complexity: This refers to how the running time of an algorithm increases with the size of the input data ('n'). It's crucial for understanding how scalable an algorithm is. In the context of cryptographic algorithms like ECDH and FFDH, time complexity can indicate how quickly key generation and exchanges can be computed as key sizes increase.

Space Complexity: This deals with the amount of memory an algorithm requires. As cryptographic operations are often run on devices with limited resources, understanding space complexity is vital to their operation and implementation.

Worst-Case Scenario: Big O Notation describes the upper limit of performance (worst-case scenario) of an algorithm. This means it provides a ceiling for how long or how much space an algorithm will take, regardless of the specific inputs.

Ignoring Constants and Lower Order Terms: In Big O Notation, constants and smaller terms are usually ignored. This is because Big O Notation focuses on the growth rate of the algorithm's complexity, not the exact runtime. For instance, if an algorithm takes '5n^2 + 3n + 8' time, the Big O Notation will be O(n^2), since n^2 is the term that will most influence the running time for large values of 'n'.

Practical Relevance: While Big O Notation provides a theoretical measure, it is also applicable to practical scenarios. Understanding the Big O Complexity of ECDH and FFDH will help in predicting their performance in real-world applications, especially when dealing with large-scale systems where efficiency is paramount.

Comparative Analysis: In this study, comparing the Big O Notation of ECDH and FFDH will involve examining their respective time and space complexities. The objective is to determine which of the two algorithms scales better as key sizes increase, which is critical for applications requiring efficient cryptographic operations.

Let's review the different types of Big O Notation so that we may try to identify what kinds of complexity ECDH and FFDH may display:

Constant Time (O(1)): The runtime does not change regardless of the input size. This is typically represented by a horizontal line on a graph.

Logarithmic Time (O(log(n))): The runtime increases slowly as the input size increases and is represented by a curve that flattens out over time.

Linear Time (O(n)): The runtime increases linearly with the input size. This would be represented by a straight line with a positive slope.

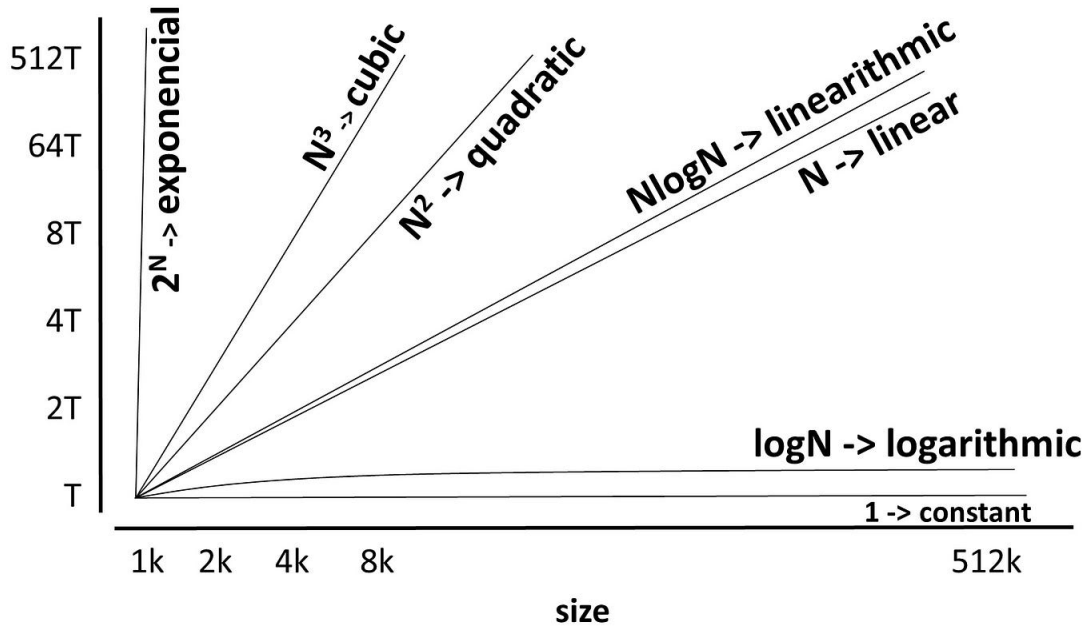
Linearithmic Time (O(n)log(n)): The runtime increases linearly multiplied by a logarithmic factor, usually represented by a curve that grows slower than quadratic but faster than linear.

Quadratic Time (O(n^2)): The runtime increases quadratically with the input size. This is represented by a parabolic curve.

Cubic Time (and Polynomial Time) (O(n^3)): This and other higher-order polynomials indicates that the runtime increases even more steeply with input size and is represented by steeper curves.

Exponential Time (O(2^n)): The runtime doubles with each additional element in the input

Factorial Time (O(n!)): The runtime grows extremely fast with any increase in input size.



Cipher Suites:
A cipher suite is a combination of cryptographic algorithms that secure network connections through various stages of the communication process. Each suite is a package of four main types of algorithms, each serving a distinct purpose in the secure transmission of data. The inclusion of DHKE in a cipher suite is particularly significant for establishing secure connections over the internet.

Overview of the Components in a Cipher Suite:
1. Key Exchange Algorithm: This is used for securely exchanging cryptographic keys between a client and a server. DHKE, including its variants like ECDH and FFDH, fits into this category. It facilitates the creation of a shared secret key between two parties over an insecure channel, without the shared secret ever being transmitted over the network.

2. Authentication Algorithm: This algorithm verifies the identity of the communicating parties. It ensures that the parties are indeed who they claim to be, preventing impersonation attacks.

3. Bulk Encryption Algorithm: Once a secure channel is established, this algorithm encrypts the data transmitted over the connection. It uses the keys generated by the key exchange algorithm to encrypt and decrypt data, ensuring confidentiality.

4. Message Authentication Code (MAC) Algorithm: This algorithm is responsible for maintaining the integrity and authenticity of the message. It generates a tag or a hash of the message, which can be used to detect tampering or alteration during transmission.

It should be noted that there are many variations and implementations of a cipher suite, and that this list should not be considered exhaustive. Rather, it is to give the reader a general idea of how the DHKE works with other cryptographic algorithms to facilitate secure communications.

How DHKE Fits Into the Cipher Suite:
Key Agreement Process: DHKE, in the context of a cipher suite, primarily handles the key agreement process. It allows two parties to establish a shared secret key, which is then used by the bulk encryption algorithm for encrypting data and by the MAC algorithm for authentication purposes.

Security and Efficiency:
The choice between ECDH and FFDH in a cipher suite can affect the overall security and efficiency of the communication. ECDH, known for its efficiency and smaller key sizes compared to FFDH, can provide the same level of security with less computational overhead.

Comparability and Flexibility:
The integration of DHKE into cipher suites offers flexibility in terms of compatibility with different protocols and systems. It allows for a secure method of key exchange without necessitating prior shared secrets, which is a fundamental aspect of many secure communication protocols like SSL/TLS (Secure Sockets Layer/Transport Layer Security).

Practical Implementation and Challenges:
The implementation of DHKE in cipher suites can vary, influencing the suite's overall security posture. For instance, ephemeral versions of DHKE (like DHE or ECDHE) provide forward secrecy, ensuring that the compromise of one session's keys does not affect the security of other sessions. Choosing the right cipher suite, including the appropriate version of DHKE, involves balancing security needs against performance requirements. This is especially crucial in resource constrained environments or high through-put scenarios.

Evolving Landscape of Cipher Suites:
The landscape of cipher suites is continually evolving, with newer algorithms being introduced and older ones being deprecated based on ongoing security research and evolving standards. Future advancements, such as quantum computing, also play a role in how cipher suites, including DHKE variants, are evaluated and chosen for secure communication.

Conclusion:
This paper will focus on comparing the computational efficiency of ECDH to FFDH, which is only one small component in how online traffic is encrypted on a daily basis by billions of people around the world.

Key Exchange:
To illustrate the process of key exchange in cryptography in a simplified, non-mathematical manner, let’s consider the well-known analogy involving two characters, Alice and Bob, who wish to communicate securely:

1. Setting the Stage for Secure Communication:
Alice and Bob decide to communicate securely over an insecure channel (i.e., the internet). Their goal is to establish a shared secret key that only they know, which can be used to encrypt and decrypt their messages.

2. Agreeing on Common Parameters:
They begin by agreeing on two public numbers: a large prime number and a generator. These numbers are not secrets; they can be openly shared or transmitted over the insecure channel.

3. Generating Private and Public Values:
Alice selects a private value, known only to her. She uses the agreed-upon generator and prime number to calculate a public value. This public value is derived from her private value but does not reveal it. Similarly, Bob chooses his own private value and computes a corresponding public value using the same generator and prime number.

4. Exchanging Public Values:
Alice sends her public value to Bob, and Bob sends his public value to Alice. These exchanges occur over the insecure channel.

5. Computing the Shared Secret Key:
Upon receiving Bob's public value, Alice combines it with her private value to compute the shared secret key. This computation typically involves exponentiation, using the prime number as a modulus. Bob performs a similar calculation, combining Alice's public value with his private value to compute the same shared secret key. Due to the mathematical properties of the key exchange algorithm, both Alice and Bob arrive at the same shared secret key independently.

6. Security Against Eavesdroppers:
If an eavesdropper, say Eve, is listening to their communication, she would only see the publicly exchanged values and the agreed-upon prime number and generator. However, without access to either Alice's or Bob's private values, Eve cannot compute the shared secret key. This cryptographic technique ensures that Alice and Bob have a secure key for their communication, despite the initial exchange occurring over an insecure channel.

7. Application in Cryptography:
This method, a simplified version of DHKE, is a cornerstone in modern cryptography. It allows two parties to establish a secure communication channel without the need for a prior shared secret, which is essential in many digital security protocols.

Mathematics of ECDH:
The mathematics of ECDH relies on the creation of elliptical curves. Specifically, it is based on the elliptic curve discrete logarithm problem (ECDLP) instead of the discrete logarithm problem (DLP, more on that later). In ECDH, two parties, Alice and Bob can establish a shared secret key over an insecure channel, each having an elliptic curve public-private key pair. The ECDLP involves finding a secret key S, given an elliptic curve group E of order n and a primitive element P in E. If two points Q

= [a]P and R = [b]P are given (where ‘a’ and ‘b’ are the private keys of Alice and Bob, respectively), the adversary’s challenge is to find S = [(ab)mod(n)]P. Please see Appendix A for two Python scripts that will go over the creation of elliptic curves and the ECDLP.

Mathematics of FFDH:
FFDH relies on the Discrete Logarithm Problem (DLP). The DLP involves computing logarithms within a group. In many cryptographic applications, the group operation is treated as a black box. This means that they do not rely on the specific properties of the group but only use the group operation to perform calculations. The DLP is considered hard to attack because, as of now, there is no known efficient method to solve it in polynomial time for large groups. This hardness underpins the security of cryptographic systems that use DLP (which includes FFDH). DLP involves the computation of logarithms in a group setting, where the main challenge is to find the exponent x in the expression g^x given g and g^x. Please see Appendix B for a practical example and walk-through.

D. Assessment:

The information provided above gives a solid theoretical and practical foundation for analyzing and conceptualizing the complexity and efficiency of ECDH and FFDH key exchange methods. This theoretical understanding is crucial for developing a mathematical model that accurately represents the algorithms’ performance characteristics. The overview of cipher suites and the role of DHKE within them contextualizes the practical application of these algorithms. It highlights the real-world significance of their efficiency and security, which is integral to the relevance of this paper. The theoretical concepts of Big O Notation will guide the development of a model that quantitatively compares the time and space complexities of ECDH and FFDH. This comparison is fundamental to answering our research question. Empirical data from Python script experiments (see Appendices A-G) will complement the theoretical analysis, providing a practical perspective on the algorithms’ performance. This data will help validate the theoretical model with real-world observations. As far as limitations go, the empirical data is limited to the specific conditions under which the Python scripts were executed. Different environments or configurations might yield different results. To account for this, we will run the same scripts on my desktop, laptop, and in a Google Colab environment. Overall, the gathered information is highly appropriate for developing a mathematical model to compare ECDH and FFDH.

II. Model

A. Selection:

For this research paper comparing the complexity and efficiency of ECDH and FFDH, I have decided to use an empirical performance model that measures key exchange times. This model is used to measure and compare the actual runtimes of the key exchange process using various key sizes. It is suitable for comparing the efficiency of different cryptographic algorithms under practical conditions (in this case, the ECDH and FFDH algorithms). This model allows for a direct comparison of performance metrics, which is essential for understanding the practical implications of using one algorithm over another. The model is adaptable to various environments and can easily incorporate different key sizes, making it highly relevant for real-world applications. The empirical data collected is straightforward and can be easily interpreted by readers of both mathematical and non-mathematical backgrounds. As far as limitations of the selected model, it may not account for all environmental factors that could affect performance, such as system load, hardware capabilities, or network conditions. To overcome the specific limitation of hardware, I have collected data from my desktop

(running a 12th Gen Intel® Core i7-12700, using 12 cores), my laptop (11th Gen Intel® Core i5-1135G7 @ 2.40GHz, using 4 cores), and from a Google Colab environment (using a V100 GPU for its runtime processor). Another limitation is that performance can vary across runs due to factors like CPU scheduling or background processes, leading to non-deterministic results. To overcome this limitation, I have run (on each environment) the same test five times each. I will average these together in order to provide a more accurate result. The empirical performance model relates to Big O Notation because while Big O Notation provides a theoretical measure of algorithmic complexity (usually representing the worst-case scenario as the input size grows), the empirical model measures actual runtimes for specific input sizes (i.e., key sizes). This allows us to observe the practical impacts of this theoretical complexity and enables us to identify what kind of Big O complexity is applicable to the algorithm being measured. The graphs produced can be seen as a practical illustration of Big O Notation complexities. For instance, if the runtime grows linearly with the key size, it may suggest a Big O Notation of O(n). We will draw conclusions from our data later on in this paper. Comparing the empirical results from our model to the expected Big O Notation for ECDH and FFDH can validate whether the theoretical complexity classes are reflective of actual performance. Discrepancies might indicate that the Big O Notation does not account for all practical considerations, such as constant factors or lower-order terms that are disregarded but can affect real-world performance. Lastly, the chosen model will help us understand that while Big O Notation provides a high-level understanding of algorithmic efficiency, it does not always translate directly into real-world performance due to practical considerations such as system architecture, compiler optimizations, or other runtime environment factors.

B. Creation:

Our empirical performance model utilized here will conduct actual measurements of key exchange times and construct the visual representation of the data to analyze the runtime performance of both ECDH and FFDH key exchange processes. To address variations in performance due to different system capabilities, I collected data across three separate environments (see above). Each script was executed five separate times, and the results were averaged to mitigate the effects of CPU scheduling and background processes. As far as tools go, the ‘cryptography’ Python library provided the necessary cryptographic primitives for ECDH and FFDH key generation and exchange. The ‘time’ module was used to record precise measurements of the duration of the key exchange processes. The ‘matplotlib’ library and its complementary ‘scienceplots’ package were utilized to graphically represent the data, allowing for clear visualization of performance trends (the ‘scienceplots’ package was not used on Google Colab, but this has no effect on the data). The empirical data is expected to provide insight into the Big O Notation trends by correlating the key size with the runtime performance. While the complexity of ECDH and FFDH algorithms are theoretically represented by Big O Notation, the empirical performance model will allow us to observe the actual impacts of this complexity. An important factor to consider is that the model’s reliability is contingent upon the accuracy and precision of the timing mechanisms in the given Python environment. Also, while the results collected are promising, they are not exhaustive and would benefit from additional research, more comparisons between other cryptographic algorithms, and further testing in diverse computational settings. The creation of this empirical performance model has been meticulously executed, taking into consideration various environmental factors and utilizing appropriate mathematical and technological tools.

C. Process:

In crafting the empirical performance model to compare the runtimes between ECDH and FFDH key exchange algorithms, the process involved a series of deliberate choices and methodological decisions.

ECDH and FFDH were selected due to their widespread use in securing communications. Research indicates that both are based on DHKE. However, they differ in their mathematical underpinnings. As explained above, ECDH relies on elliptic curve cryptography, whereas FFDH relies on finite field arithmetic. The decision to compare these two is rooted in the need to understand the practical implications of their theoretical differences. As for the selection of key sizes, they were chosen to reflect common standards in cryptographic practice. For ECDH, the key sizes 256, 384, and 521 bits were selected, and for the FFDH, the key sizes 2048, 3072, and 4096 bits were chosen. This aligns with recommendations from institutions like the National Institute of Standards and Technology (NIST). These specific key sizes are suggested because they offer a balance of comparative efficiency and security. To ensure the robustness of the model, the tests were executed across various computational environments. This was influenced by literature that emphasizes the importance of environmental factors on cryptographic and algorithmic performance. By running tests in different environments, the model accounts for variability in computational resources, which is crucial for its applicability. Each key exchange was performed multiple times so that the results can be averaged and compared to each other. This approach mitigates the influence of transient system states and background processes, as suggested by the performance analysis methodologies in this field of study. It is also considered a standard practice to enhance the reliability of performance measurements. Python was chosen due to its readability, widespread use in both academic and industry settings, and the rich ecosystem of libraries it offers. The ‘cryptography’ and ‘NumPy’ libraries were used for their well-documented and secure implementations of cryptographic primitives and mathematical functionalities (respectively), making them a trustworthy choice for modeling. The empirical data collected will be analyzed and compared to a Big O Notation to determine what kind of complexity the algorithm has. This comparison will help to validate the practical performance against the theoretical complexity, a method supported by computational complexity research.

D. Tools:

In the creation of this empirical performance model, several tools and techniques were utilized, each chosen for its suitability to the task at hand and its relevance to the overarching goal of accurately measuring and comparing the key exchange times for ECDH and FFDH. The Python programming language was chosen for its widespread acceptance in scientific computing due to its readability and simplicity. Its extensive library ecosystem, particularly for data analysis and cryptography, makes it an excellent tool for empirical research. The language’s popularity in both academic and industrial research supports reproducibility and validation of results by the broader community. The ‘cryptography’ library is appropriate for this model due to its rigorous security standards and ease of use, which are essential for conducting accurate cryptographic measurements. Its implementations of ECDH and FFDH are industry standard, ensuring that the model reflects the performance of algorithms as they would be deployed in real-world applications. The ‘time’ module was used to record the duration of key exchanges. The NumPy library is considered the most reliable Python library for mathematical calculations and is used in both academic and industry settings. ‘matplotlib’ was chosen because it is a powerful plotting library that is widely used in the scientific community for its versatility and ability to produce publication-quality figures. The complementary ‘scienceplots’ package provides a set of styles tailored for scientific plotting, enhancing the readability and professional appearance of the graphs produced. The selected tools are appropriate for the empirical model type due to their ability to accurately measure the performance of cryptographic algorithms, a key factor in assessing the complexity and efficiency in practice. The ‘cryptography’ library aligns with the best practices in the field of cybersecurity research, where precision, security, and reliability are paramount. The use of these tools and techniques is supported by their prevalence in published research and their endorsement by the cryptographic community. The tools and techniques chosen are not only standard in

computational and cryptographic research but also offer the necessary functionality to address the specific needs of this empirical performance model.

E. Analysis:

To analyze the results, we will provide a detailed analysis for the Python scripts used.
- See Appendix C for the Python script that measures the key generation and key exchange time between ECDH and FFDH.
- See Appendix D for the Python script that visually represents the key generation and exchange time of ECDH.
- See Appendix E for the Python script that visually represents the key generation and exchange time of FFDH.
- See Appendix F for the Python script that creates the visual representation of runtime analysis between ECDH and FFDH.
- See Appendix G for the Python scrip that creates a visual representation of runtime analysis of ECDH

Analysis of ECDH and FFDH key generation and exchange times generated by Appendix C:
On average, ECDH key generations and exchanges are faster than FFDH key generations and exchanges across all environments. This aligns with theoretical expectations, as ECDH (which relies on elliptic curve cryptography) is generally considered more efficient than FFDH, particularly at smaller key sizes when compared to the equivalent security level of FFDH. The key exchange times varied across the desktop, laptop, and Google Colab environments, reflecting the impact of different hardware capabilities and system resources on cryptographic operations. Despite this variability, the trend of ECDH being faster remains consistent, supporting the model's findings.

Desktop (12^{th} Gen Intel® Core i7-12700, using 12 cores):
Average ECDH: 0.0006818 seconds
Average FFDH: 0.004796 seconds

Laptop (11^{th} Gen Intel® Core i5-1135G7 @ 2.40GHz, using 4 cores):
Average ECDH: 0.0012058 seconds
Average FFDH: 0.0059273 seconds

Google Colab (V100 GPU):
Average ECDH: 0.0033806 seconds
Average FFDH: 0.0085918 seconds

Total Average Across All Environments:
ECDH: 0.001756 seconds
FFDH: 0.006438 seconds

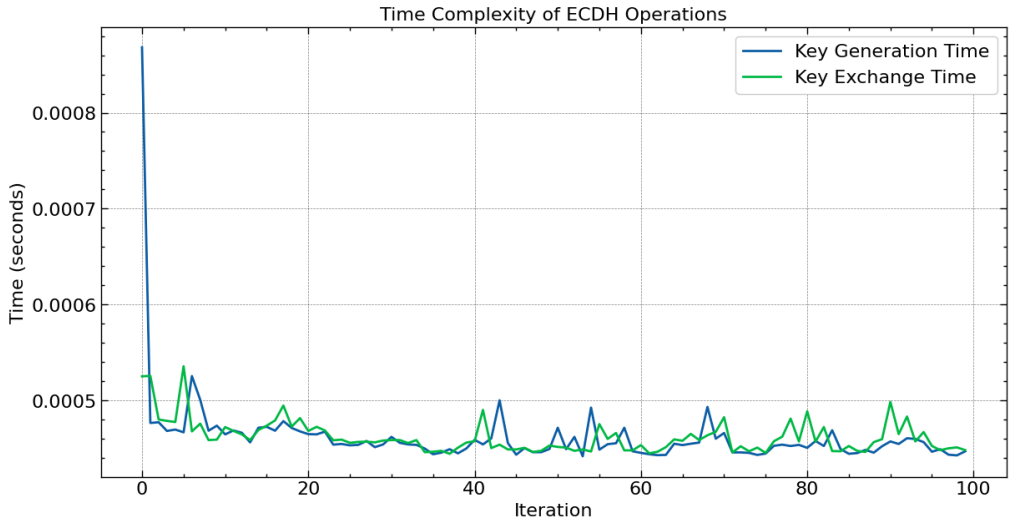
The consistency of ECDH's faster performance in all tests supports the conclusion that ECDH is more efficient in terms of computational time and resources, fitting the research question and the information gathered. The empirical performance model effectively demonstrates that ECDH key exchanges are consistently faster than FFDH key exchanges across various computational environments. The results fit well with our research question, providing a clear answer based on practical performance metrics rather than theoretical complexity alone. The findings are supported by existing research that also highlights the efficiency of ECDH over FFDH in practical implementations. While the results are clear, they are limited to the specific key sizes and environments tested. Further research could expand the

range of key sizes and include more varied environments, but these will not be conducted for the purposes of this research paper. Additional tests can also be conducted to assess the impact of network latency and other real-world factors on key exchange times, but, again, will not be conducted for the purposes of this research paper.

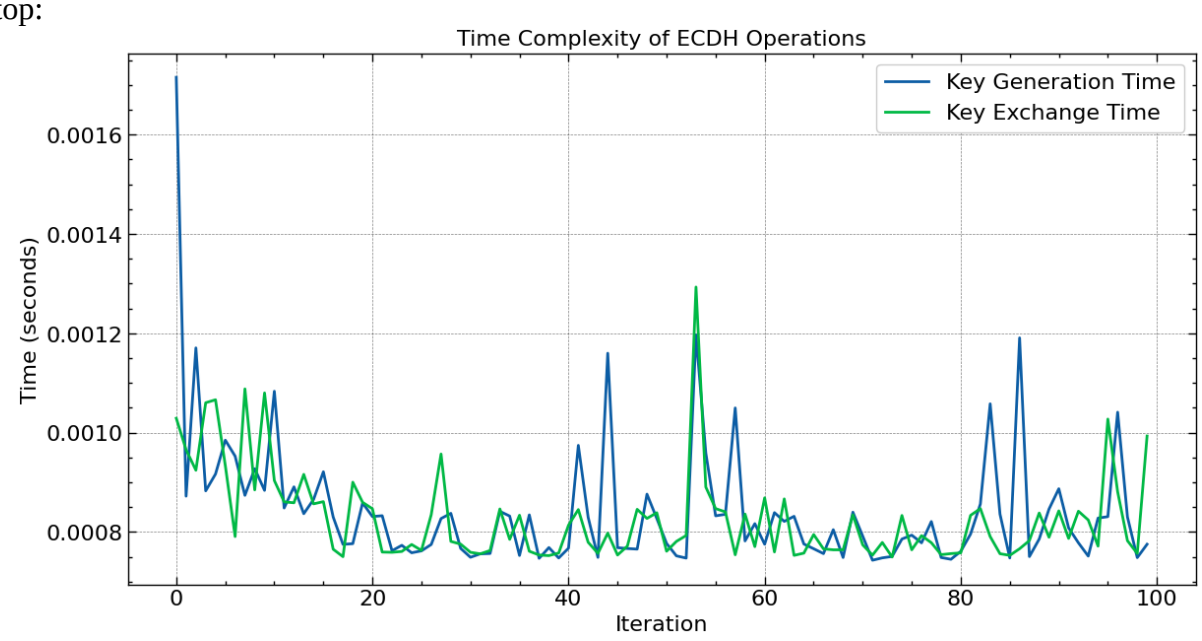
Now, let's look at the information gathered from Appendix D and E. These scripts measured the key generation and exchange times of ECDH and FFDH and visually represented them. The results are very interesting. This research paper will only use one graph from each runtime environment. The presentation portion will contain all graphs and will delve deeper into the empirical performance model analysis.

Here are visual representations of key generations and key exchanges for ECDH:

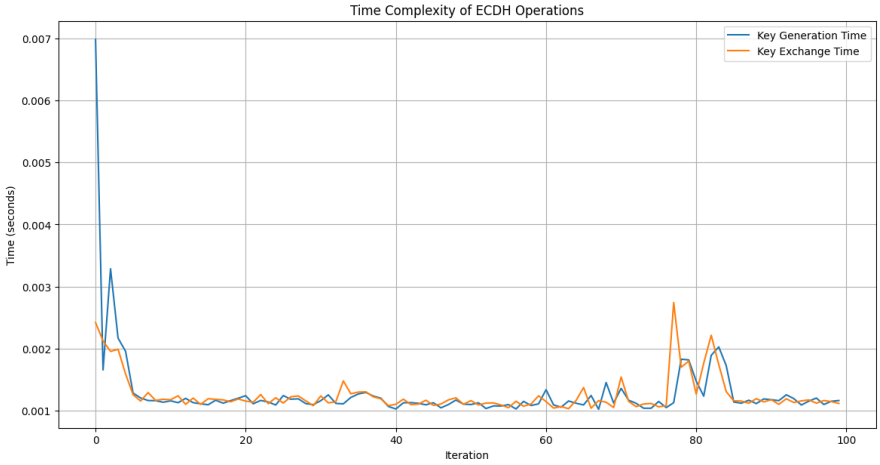
Desktop:



Laptop:



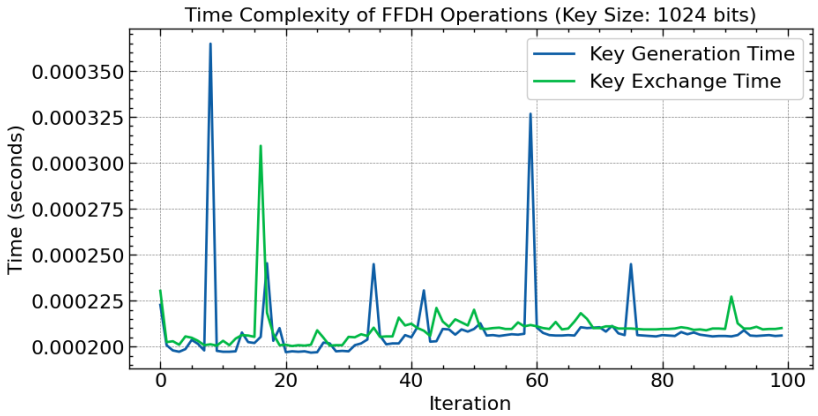
Colab:



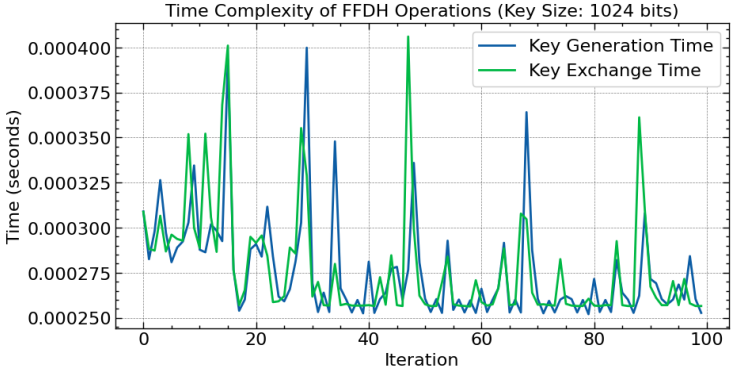
All the graphs created show that there is an initial high computational time taken to generate the keys before dropping off to what is essentially Constant Time (O(1)) complexity. The reasons for this will be discussed in the section that deals with Big O Notation.

Here are the graphs used to visually represent the key generation and exchange times for FFDH:

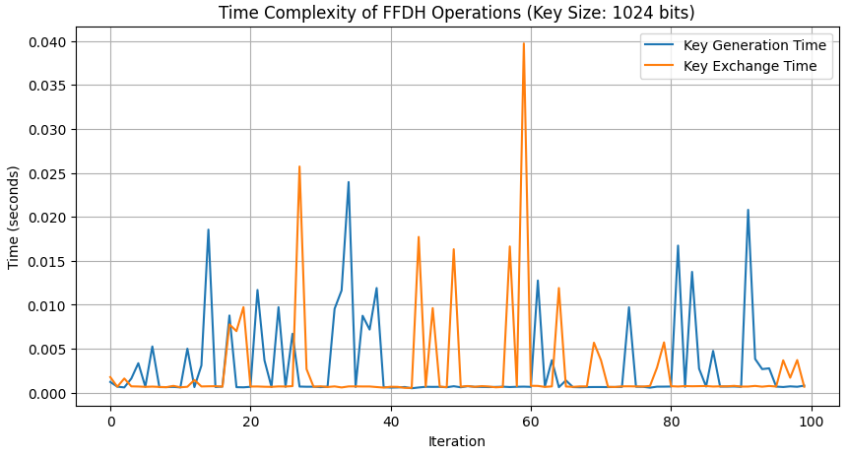
Desktop:



Laptop:



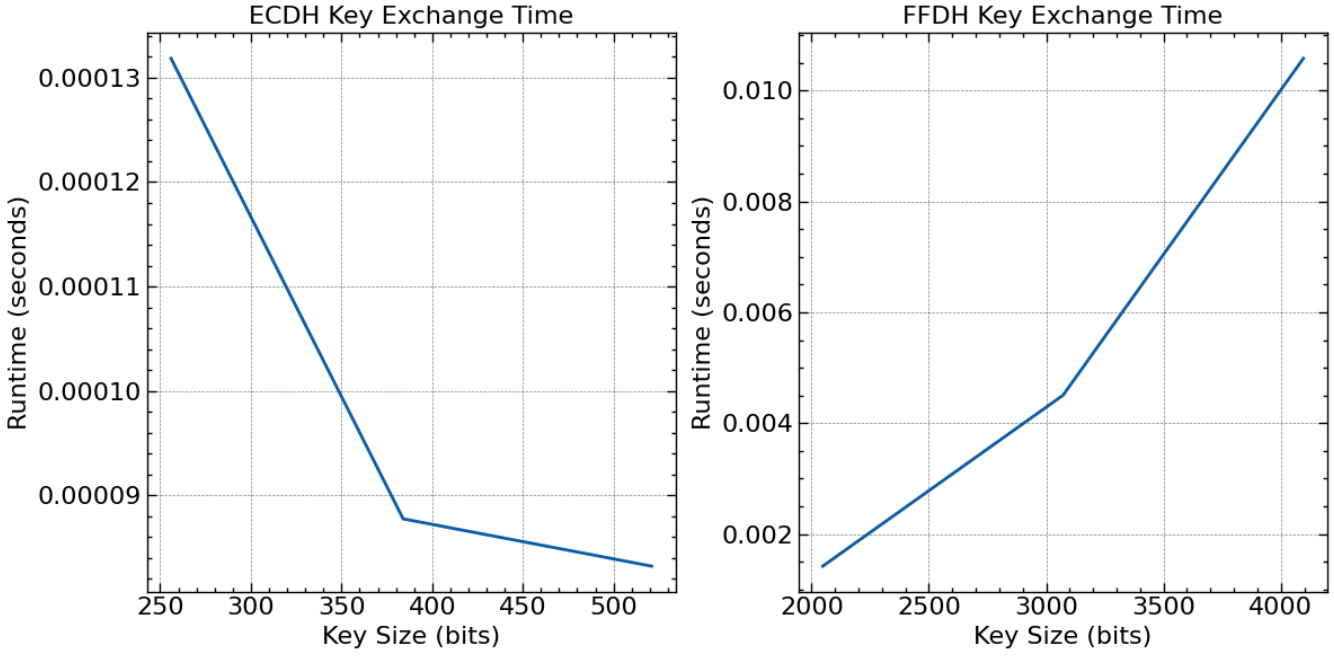
Colab:



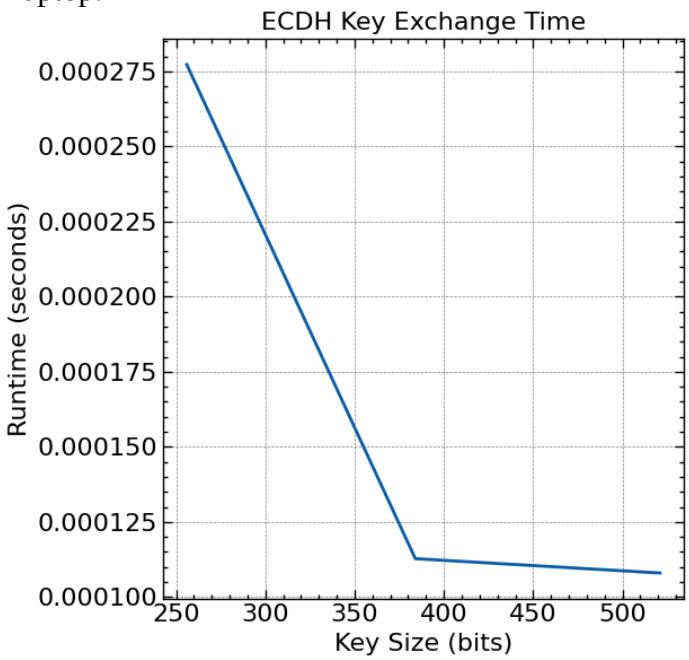
As we can see, there is no “dropping off” in computational resources like there is in ECDH. Instead, there are sharp spikes and drops throughout the key generation/exchange process. This indicates a Linear Time (O(n)) or possibly a Quadratic Time (O(n^2)) complexity. More on this below.

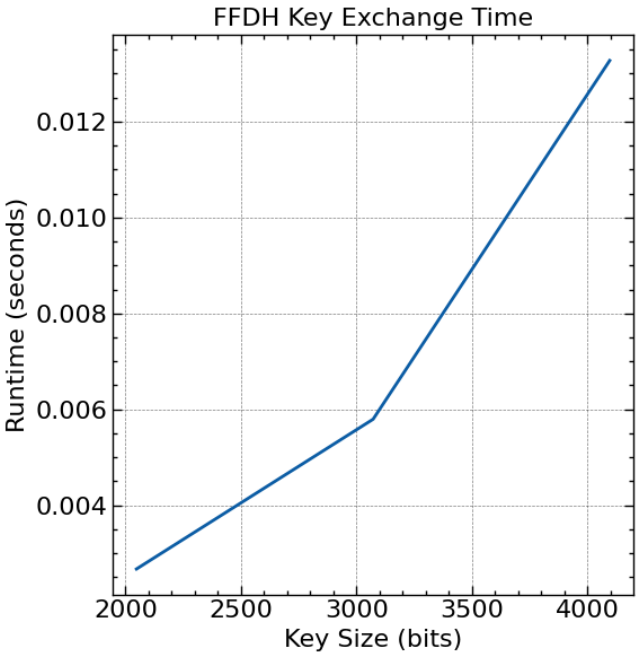
Now, let us look at some of the information gleaned from Appendix F. This directly compares the runtime between ECDH and FFDH. It is from here that we may be able to draw a conclusion as to the applicable Big O Notation.

Desktop:

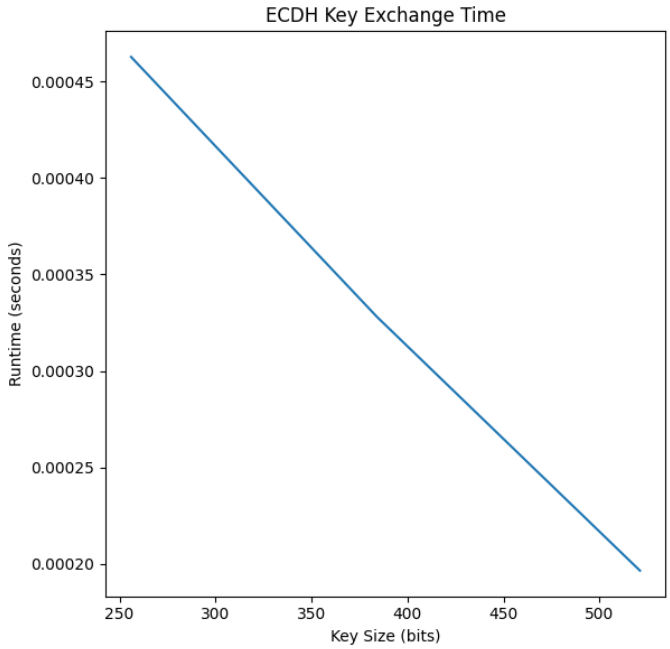


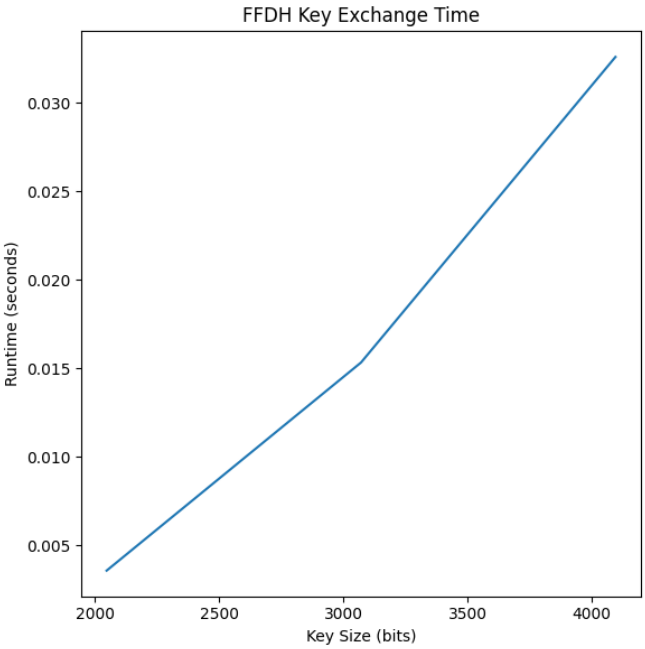
Laptop:





Google Colab:

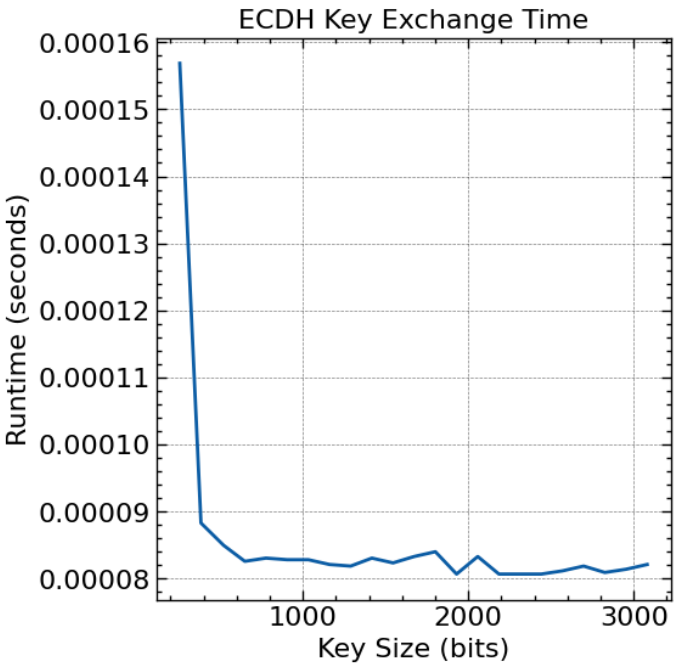




ECDH Performance Trends:

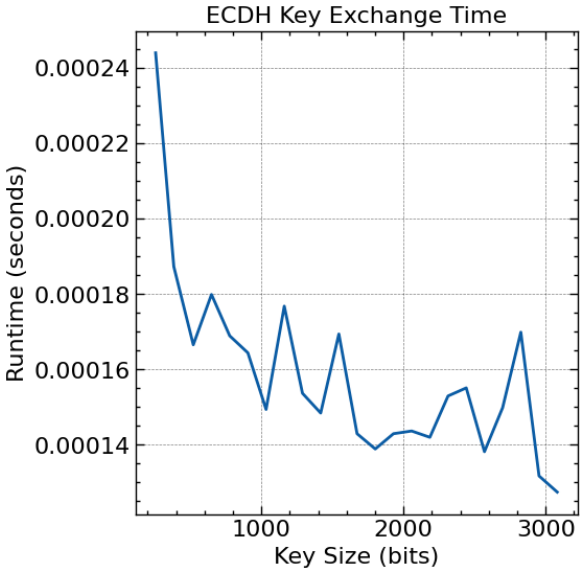
At first, the curve of the ECDH tends to decrease as the key size increases. While the key generation and key exchange times were as expected, there was some trouble in pinpointing why the computational complexity would seem to decrease even as the key size became greater in length. A modified Python script that tested greater and greater lengths of the ECDH was ran in order to see if this trend continued. Please see Appendix G for the modified version of the code. This was initially ran on the desktop environment.

Output:

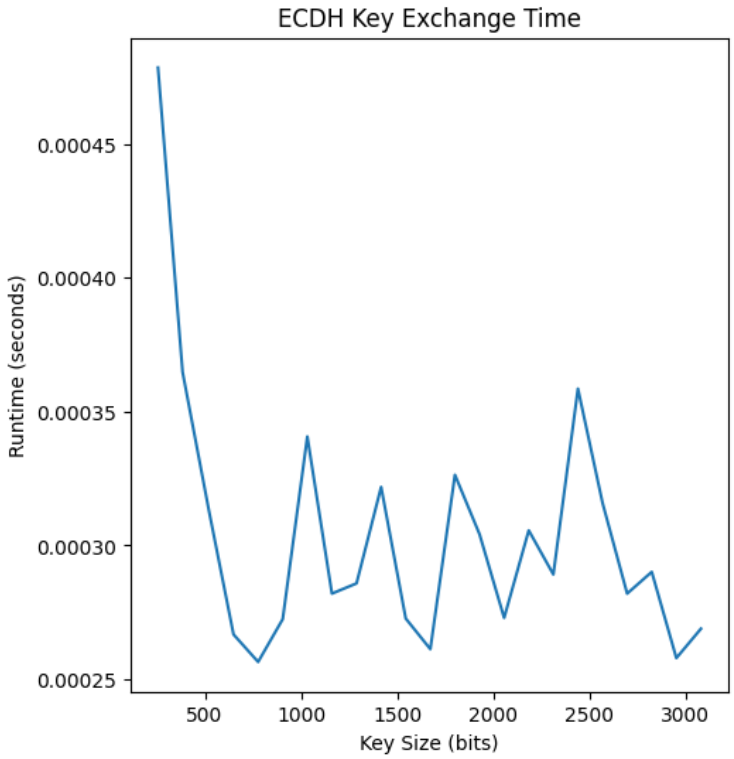


Based on this (and other graphs that will be presented in the presentation) it appears that the Big O Notation of ECDH can be described as consuming a high amount of computational resources (comparatively speaking) before sharply dropping off to what is essentially Constant Time (O(1)) complexity. This confirms and shows why ECDH is computationally less intensive than other cryptographic algorithms. The primary computational demands of ECDH are concentrated in the initial generation of elliptic curves. Once these curves are established, the size of the keys has a minimal impact on computational requirements. This means that even as key sizes increase, ECDH maintains a more consistent and manageable computational load, unlike other key exchange algorithms that exhibit more dramatic increases in computational demands with larger keys. Additionally, the core of ECDH is founded on scalar operations on elliptic curves, a process that is computationally less burdensome than the exponentiation or logarithmic operations characteristic of many alternative cryptographic systems. To confirm this finding (created on the desktop environment) I ran the same code on both the laptop and Google Colab environments and produced roughly the same results:

Laptop:



Google Colab:



These same results were also gathered using different variations of ECDH. The full analysis will be available in the presentation portion of this research paper.

FFDH Performance Trends:
FFDH key exchange times show a more consistent and clear growth as key sizes increase. This behavior is more aligned with what might expect from a traditional Big O Notation perspective, which is typical for algorithms where the difficulty increases significantly with larger key sizes. Based on the available models that we've run, it appears that the Big O Notation of the FFDH key exchange can be described as having a Linear Time Complexity (O(n)), bordering on Quadratic Time Complexity (O(n^2)).

The variability in key exchange times across different environments highlights that real-world performance is also influenced by hardware and system capabilities, which are not accounted for in Big O Notation. Big O Notation abstracts away such details, focusing on growth rates as input sizes increase, independent of machine-specific characteristics. However, despite these environmental differences, ECDH consistently shows faster key generation/exchange times than FFDH, which supports the notion that ECDH is more efficient at generating and exchanging cryptographic keys. This aligns with the theoretical understanding that elliptic curve cryptography can achieve comparable levels of security with smaller keys and less computational work than finite field cryptography.

F. Limitations:

As far as computational complexity goes, FFDH is much more resource heavy than ECDH. The implementation of FFDH dramatically increased the time needed in order to find the key generation/exchange times and to create the graphs. This can lead to inefficiency and scalability issues, particularly for large-scale systems or those that require frequent key exchanges. The model used here

may not account for all real-world operating conditions, such as system load, network latency, or other environmental factors that can affect performance. As far as other limitations are concerned, the data points may lack granularity, especially if the key sizes tested do not cover a wider range or if the increments between key sizes are too large. The measurements taken may suffer from sampling errors if the number of trials for each key size is too low or if there is significant variance in the times recorded that is not accounted for. This model takes sufficient precaution to try and safeguard against these concerns, but there is always the possibility that something may have been overlooked.

G. Approach:

The following steps were taken to achieve the above results:
1. Selection of Algorithms: ECDH and FFDH were chosen due to their prevalent use in secure key generation and exchanges.
2. Identification of Metrics: It was determined that the algorithmic runtime would be the primary metric for comparison and would be used to determine what kind of Big O Notation the tested algorithm has.
3. Environmental Setup: Several computational environments were set up, including a desktop (running a 12^{th} Gen Intel® Core i7-12700, using 12 cores), laptop (11^{th} Gen Intel® Core i5-1135G7 @ 2.40GHz, using 4 cores), and a Google Colab environment (using the V100 GPU). This was to ensure that the results would not be hardware-specific and could be generalized.
4. Key Size Variations: I decided on a range of key sizes to test each algorithm. For ECDH, this involved different bit lengths and different curve variations (this will be covered in the presentation portion of the research paper). For FFDH, a range starting from 2000 bits upward was chosen. This was based on industry standards of cryptographic strengths as set forth by NIST.
5. Implementation of Algorithms: The algorithms were utilized using existing implementations of ECDH and FFDH in each environment using the ‘cryptography’ Python library. This was to ensure that the implementations were correct and secure.
6. Benchmarking: The key generation/exchange algorithms were executed multiple times for each key size to gather data. This involved measuring the runtime of the key generation/exchange process from start to finish.
7. Data Collection: The runtime data was collected and recorded for further analysis.
8. Analysis of Results: Graphs were plotted to show the relationship between key size and runtime for both ECDH and FFDH.
9. Big O Notation Analysis: The shapes and trends of ECDH and FFDH were interpreted in the graphs in order to theorize the Big O Complexity of the key generation/exchange times for both algorithms. It appears that ECDH uses a high amount of computational resources (comparatively speaking) before sharply dropping off to what can be accurately described as a Constant Time Complexity (O(1)). The FFDH can be said to have a Linear Time Complexity (O(n)), bordering on what can also be described as Quadratic Time (O(n^2)) Complexity.
10. Comparative Analysis: Comparing the runtimes of ECDH and FFDH across different environments helps us to understand the impact of computational resources on performance.
11. Testing for Limitations: The empirical performance model was evaluated for potential errors, including environmental factors, data sampling errors, and graphical resolution errors.
12. Conclusion Formulation: Based on the analysis, ECDH is a much simpler and more efficient cryptographic algorithm than FFDH. It is not said that FFDH is less “secure” because a large enough FFDH would be comparatively “as secure” or “more secure” than a very weak implementation of ECDH. However, all things being equal, ECDH is a much more efficient algorithm than FFDH, making it ideal for any environment where computational resources are at a premium or limited.

H. Applicability:

The purpose of this empirical performance model is to analyze and compare the efficiency of two key generation/exchange algorithms, ECDH and FFDH, in terms of their computational runtime and complexity. The model is directly applicable to the research question, which seeks to understand the performance characteristics and computational complexity of these cryptographic algorithms under varying conditions. The model is designed to measure the specific aspect of the cryptographic algorithms that the research question targets, which is the time efficiency of key generation/exchanges. By focusing on runtime measurements across a wide range of key sizes, the model directly probes the heart of the research question. The model also provides empirical data that reflects the actual performance of the algorithms in realistic scenarios. The data gathered from multiple environments also helps in generalizing the findings. A quantitative comparison of ECDH and FFDH is also conduced, offering a clear picture of how each algorithm scales with increasing key sizes. By analyzing this scaling, the research question can be answered with evidence-backed conclusions about which algorithm is more efficient for different key sizes and what kind of Big O Notation can be identified. The empirical performance model aligns well with the research question because it employs a methodological approach to gather data that can be used to infer the computational complexity (Big O Notation) of the algorithms. The graphical representations of the runtime against key size give a visual and intuitive understanding of how well each algorithm performs, which is central to answering the research question. The model highlights patterns, trends, and anomalies that could indicate the practical limitations and strengths of the algorithms, thus providing insights into their suitability for different cryptographic applications.

In conclusion, the empirical performance model chosen serves a critical tool for investigating the research question. It not only measures the required data but also organizes and presents them in a way that facilitates understanding of the underlying complexities of ECDH and FFDH. This systematic approach ensures that the conclusions drawn are robust and directly relevant to the research question.

Appendix A:

```
import matplotlib.pyplot as plt
import numpy as np
```

This line import necessary Python libraries

```
def plot_elliptic_curve(a, b):
```

This line defines a function to plot an elliptic curve, taking parameters ‘a’ and ‘b’, which are the coefficients in the curve equation.

```
x = np.linspace(-5, 5, 400)
```

Creates a NumPy array of 400 points evenly spaced between -5 and 5. These points are used as x-coordinates for plotting the curve

```
y_square = x**3 + a * x + b
```

For each x-coordinate, this calculates y^2 using the elliptic curve equation. The result is an array of y^2 values corresponding to each x-coordinate.

```
y_square_real = y_square[y_square >= 0]
```

Filters the array to include only those y^2 values that are non-negative (real when taking the square root).

```
x_real = x[y_square >= 0]
```

Filters the x-coordinates to correspond to the non-negative y^2 values

```
y_positive = np.sqrt(y_square_real)
```

Calculates the positive square root of the non-negative y^2 values

```
y_negative = -y_positive
```

Negates the positive y-values to get the negative square root values

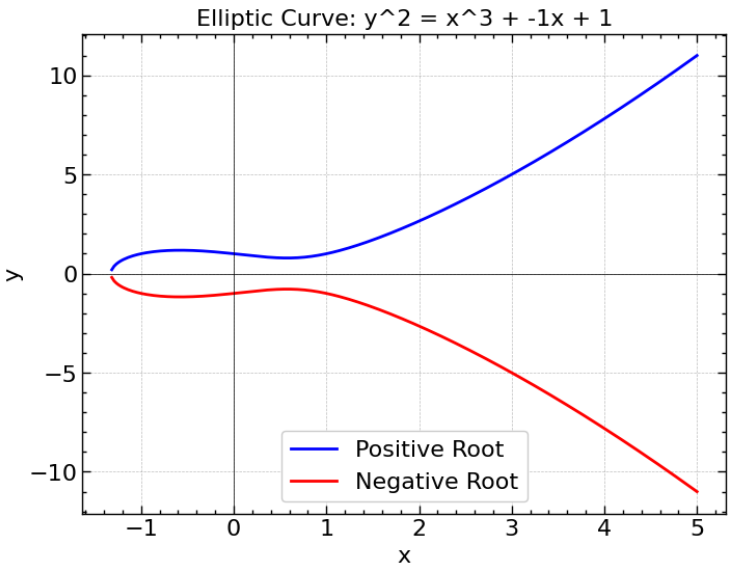
```
plt.figure(figsize=(8, 6))
plt.plot(x_real, y_positive, label="Positive Root", color="blue")
plt.plot(x_real, y_negative, label="Negative Root", color="red")
plt.axhline(0, color="black", linewidth=0.5)
plt.axvline(0, color="black", linewidth=0.5)
plt.title("Elliptic Curve: y^2 = x^3 + {}x + {}".format(a, b))
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid(color="gray", linestyle="--", linewidth=0.5)
plt.show()
```

This section provides the code necessary to plot the elliptical curve.

```
a = -1 # Coefficient of x
b = 1 # Constant term
```

These lines assign specific values to the coefficients 'a' and 'b' for the elliptic curve equation

```
plot_elliptic_curve(a, b)
```
Output:



And now, the ECDLP:

```
import numpy as np
import matplotlib.pyplot as plt
```
Imports the necessary Python libraries

```
def plot_elliptic_curve(a, b, x_range):
```
Defines a function to plot an elliptical curve of the form y^2 = x^3 + ax + b. ‘a’ and ‘b’ are the coefficients in the curve equation, and ‘x_range’ specifies the range of x-values for the plot.

```
x = np.linspace(x_range[0], x_range[1], 1000)
y_squared = x**3 + a * x + b
```
Generates 1000 linearly spaced values between the start and end of ‘x_range’. ‘y_squared’ calculates y^2 for each value of ‘x’ using the curve equation.

```
y_positive = np.sqrt(y_squared)
y_negative = -y_positive
```
Calculates the positive and negative square roots of ‘y_squared’, representing the two possible values of ‘y’ for each ‘x’

```
plt.figure(figsize=(8, 6))
plt.plot(x, y_positive, "b")
plt.plot(x, y_negative, "b")
plt.title(f"Elliptic Curve: y^2 = x^3 + {a}x + {b}")
plt.xlabel("x")
plt.ylabel("y")
plt.grid(True)
plt.axhline(0, color="black", linewidth=0.5)
plt.axvline(0, color="black", linewidth=0.5)
plt.show()
```
This plots the elliptical curve

```
def elliptic_add(p, q, a, b):
```
Defines a function to add two points ‘p’ and ‘q’ on the elliptic curve.

```
if p is None:
return q
if q is None:
return p
```
These two for-loops handle the special case where either of the points is the identity element (represented as ‘None’)

```
if p[0] == q[0] and p[1] == -q[1]:
return None
```
This checks if ‘p’ and ‘q’ are inverses of each other. If so, their sum is the identity element.

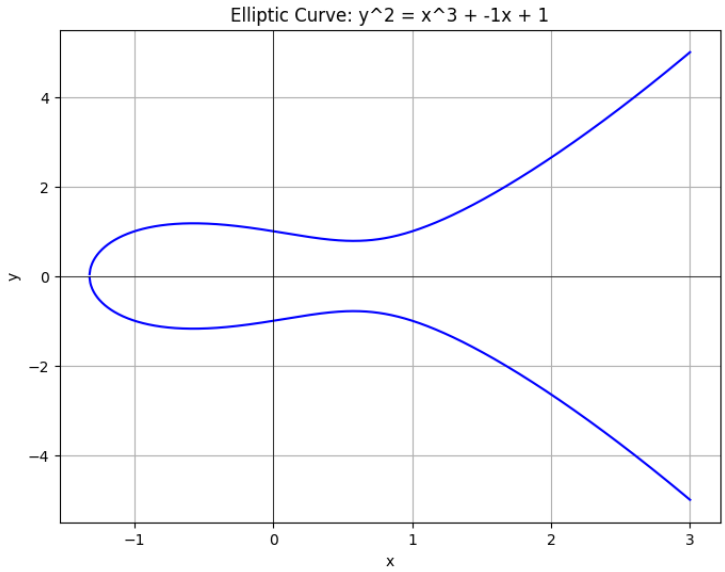
```
if p != q:
m = (q[1] - p[1]) / (q[0] - p[0])
else:
m = (3 * p[0] ** 2 + a) / (2 * p[1])
```
Calculates the slope ‘m’ of the line between ‘p’ and ‘q’, or the tangent ‘p’ if they are the same.

```
rx = m**2 - p[0] - q[0]
ry = m * (p[0] - rx) - p[1]
return (rx, ry)
```
Computes the resulting point after addition, following the elliptic curve point addition rules.

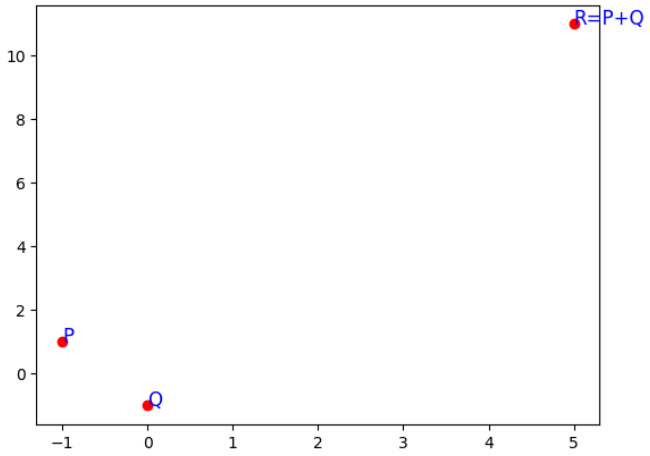
```
a = -1
b = 1
x_range = (-2, 3)
p = (-1, 1)
q = (0, -1)
r = elliptic_add(p, q, a, b)
```
Defines the curve parameters ‘a’ and ‘b’, the range for ‘x’, and the two points ‘p’ and ‘q’. Additionally, these lines calculate the sum ‘r’ of ‘p’ and ‘q’.

```
plot_elliptic_curve(a, b, x_range)
plt.scatter([p[0], q[0], r[0]], [p[1], q[1], r[1]], color="red")
plt.text(p[0], p[1], "P", fontsize=12, color="blue")
plt.text(q[0], q[1], "Q", fontsize=12, color="blue")
plt.text(r[0], r[1], "R=P+Q", fontsize=12, color="blue")
plt.show()
```

Output:



This is the elliptic curve that was created by the first half of the Python script.



Seen here, P and Q are two points on the elliptic curve, with R as the result of adding P and Q. This addition is a core operation in elliptic curve cryptography. The plot visually represents how the point R is derived from P and Q. To illustrate the ECDLP, let's consider a scenario where we know P and R, and we need to find an integer k such that R = kP. In a practical cryptographic setting, this is an extraordinarily challenging problem, as there's no efficient way to compute k given P and R. This hardness is what secures the elliptic curve-based cryptographic algorithms.

Appendix B:

```
import random
```
Python library that provides functions for generating random numbers

```
def modular_pow(base, exponent, modulus):
```
Defines a function with parameters 'base', 'exponent', and 'modulus'. This function will perform modular exponentiation

```
result = 1
```
This variable will store the outcome of the modular exponentiation

```
base = base % modulus
```
Reduces 'base' modulo 'modulus' to ensure it is within range of the finite field defined by 'modulus'

```
while (exponent > 0):
```
Begins a while-loop that will continue as long as 'exponent' is greater than 0

```
if (exponent % 2 == 1):
```
Checks if current 'exponent' is odd (using modulo operator). This is part of the binary exponentiation algorithm

```
result = (result * base) % modulus
```
If 'exponent' is odd, multiplies 'result' by 'base' and applies modulo 'modulus' to keep result within finite field

```
exponent = (exponent >> 1)
```
Right shifts the 'exponent' by one bit (equivalent to dividing by 2 and discarding the remainder). This is a step in binary exponentiation

```
base = (base * base) % modulus
```
Squares 'base' and applies modulo 'modulus'. This is another step in binary exponentiation algorithm.

```
return result
```
One loop completes, returns final 'result' of modular exponentiation

```
def discrete_logarithm_problem_ex():
```
Define a prime modulus and a base (generator)

```
prime_modulus = 23
base = 5
exponent = random.randint(1, prime_modulus - 1)
```
Randomly select an exponent (private key in cryptographic use). The numbers for the prime modulus and base were arbitrarily selected.

```
resultant = modular_pow(base, exponent, prime_modulus)
```
Compute the resultant (public key in cryptographic use)

```
print(f"Given prime modulus (p): {prime_modulus}")
print(f"Given base (g): {base}")
print(f"Computed resultant (h = g^x mod p): {resultant}")
print(f"Unknown exponent (x): {exponent}")
print("\nChallenge: Find the exponent x given g, h, and p.")
discrete_logarithm_problem_ex()
```

Output:

```
Given prime modulus (p): 23
Given base (g): 5
Computed resultant (h = g^x mod p): 14
Unknown exponent (x): 21
Challenge: Find the exponent x given g, h, and p.
```

In the context of DLP, the prime modulus is a prime number that defines the finite field (or group) in which the calculations are performed. For this example, the operations in this group are modulo 23. The base (often denoted as ‘g’ in cryptography) is the generator of the group. In DLP, the base is a number within the finite field defined by the prime modulus. Here, the base is 5, which means that calculations are based on powers of 5, modulo 23. The computed result of raising base ‘g’ to the power of ‘x’ (the exponent) and then taking the result modulo ‘p’ (the prime modulus) is 14, calculated as 5^21 mod 23. This resultant is often referred to as ‘h’ in DLP and represents the public key in cryptographic applications. As we just mentioned, the exponent ‘x’ is the discrete logarithm that the DLP seeks to find. In cryptographic terms, this exponent is often the private key. Here, the unknown exponent (x) is 21, which was a randomly chosen integer between 1 and 22 (since the prime modulus is 23). The challenge of DLP is to find the exponent ‘x’ given the base ‘g’, the resultant ‘h’, and the prime modulus ‘p’.

Appendix C:

```
from cryptography.hazmat.primitives.asymmetric import ec, dh
from cryptography.hazmat.backends import default_backend
import time
```

Imports the necessary Python libraries

```
def time_ecdh_key_exchange():
```

Defines a function to measure the time taken for an ECDH key exchange

```
private_key = ec.generate_private_key(ec.SECP384R1(), default_backend())
```

Generates private key for ECDH using the SECP384R1 curve

```
peer_public_key = ec.generate_private_key(ec.SECP384R1(), default_backend()).public_key()
```

Generates a peer's public key for the key exchange

```
start_time = time.time()
```

Records current time before starting key exchange

```
shared_key = private_key.exchange(ec.ECDH(), peer_public_key)
```

Performs ECDH key exchange to compute the shared key

```
end_time = time.time()
```

Records the current time immediately after the key exchange

```
return end_time - start_time
```

Calculates and returns the time taken for ECDH key exchange

```
def time_ffdh_key_exchange():
```

Defines a function to measure the time taken for a FFDH key exchange

```
parameters = dh.generate_parameters(
generator=2, key_size=3072, backend=default_backend())
```

Generates FFDH parameters using a 3072-bit key size

```
private_key = parameters.generate_private_key()
```

Generates a private key for FFDH

```
peer_public_key = parameters.generate_private_key().public_key()
```

Generates a peer's public key for the key exchange

```
start_time = time.time()
```

Records current time before starting key exchange

```
shared_key = private_key.exchange(peer_public_key)
```

Performs FFDH key exchange to compute shared key

```
end_time = time.time()
```

Records the current time immediately after the key exchange

```
return end_time - start_time
```
Calculates and returns the time taken for FFDH key exchange

```
ecdh_time = time_ecdh_key_exchange()
print(f"ECDH Key Exchange Time: {ecdh_time} seconds")
```
Assigns the time taken by the ECDH key exchange and prints it to the terminal

```
ffdh_time = time_ffdh_key_exchange()
print(f"FFDH Key Exchange Time: {ffdh_time} seconds")
```
Assigns the time taken by the FFDH key exchange and prints it to the terminal

ECDH and FFDH functions are called and their execution times are printed. This compares the performance of both cryptographic variations of DHKE.

Appendix D:

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives import serialization
import numpy as np
import matplotlib.pyplot as plt
import time
import scienceplots
plt.style.use(["science", "notebook", "grid"])
```

Imports the necessary Python libraries

```
def measure_ecdhe_operations(iterations):
```

This function will measure the time taken for key generation and exchange operations over a specified number of iterations.

```
time_data = np.zeros((iterations, 2))
```

Creates an array to store time for key generation and exchange

```
for i in range(iterations):
start_time = time.time()
private_key = ec.generate_private_key(ec.SECP384R1(), default_backend())
public_key = private_key.public_key()
time_data[i, 0] = time.time() - start_time
```

This for-loop will run for the number of times specified by 'iterations'. It records the current time before starting key generation. ‘private_key’ and ‘public_key’ both generate a private key using the same elliptic curve and backend as before, and then derives the public key. ‘time_data’ calculates the time taken for key generation by subtracting the start time from the current time and stores it in the time data array

```
peer_private_key = ec.generate_private_key(ec.SECP384R1(), default_backend())
peer_public_key = peer_private_key.public_key()
```

These lines generate another pair of private and public keys, simulating a second participant in the key exchange

```
start_time = time.time()
```

Current time is recorded again before starting the key exchange

```
private_key.exchange(ec.ECDH(), peer_public_key)
```

Key exchange operation performed using the first participant's private key and the second participant's public key

```
time_data[i, 1] = time.time() - start_time
```

Calculates the time taken for the key exchange and stores it in the time data array

```
return time_data
```

This function returns the time data array with the timing information for further analysis

```
iterations = 100
time_data = measure_ecdhe_operations(iterations)
```
Sets the number of iterations to be performed, calls the function, and stores the returned time data array

```
plt.figure(figsize=(14, 7))
plt.plot(time_data[:, 0], label="Key Generation Time")
plt.plot(time_data[:, 1], label="Key Exchange Time")
plt.xlabel("Iteration")
plt.ylabel("Time (seconds)")
plt.title("Time Complexity of ECDH Operations")
plt.legend()
plt.grid(True)
plt.show()
```
This section of the code produces the graph

Appendix E:

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import dh
import numpy as np
import matplotlib.pyplot as plt
import scienceplots
plt.style.use(['science', 'notebook', 'grid'])
import time
```

Imports the necessary Python libraries

```
key_size = 1024
iterations = 100
```

This sets the key size (1024 bits) and the number of iterations that the key generation and exchange process will be repeated.

```
def measure_ffdh_operations(iterations, key_size):
time_data = np.zeros((iterations, 2))
```

Defines a function to measure the time taken for key generation and exchange for FFDH. Initializes array with dimensions 'iterations' * 2, filled with 0s. This array will store timing data for key generation and key exchange

```
parameters = dh.generate_parameters(
generator=2, key_size=key_size, backend=default_backend())
```

Generates DH parameters using a predefined prime number (generator=2) and the specified key size. These parameters are generated only once and used for all iterations

```
for i in range(iterations):
start_time = time.time()
private_key1 = parameters.generate_private_key()
public_key1 = private_key1.public_key()
time_data[i, 0] = time.time() - start_time
```

This section records the current time before starting the key generation process. It then calculates the time taken for key generation and stores it in the first column for 'time_data' array for the current iteration. Being measured is the time it takes to generate a private key for the first party and derive its corresponding public key using the previously generated DH parameters.

```
private_key2 = parameters.generate_private_key()
public_key2 = private_key2.public_key()
start_time = time.time()
private_key1.exchange(public_key2)
time_data[i, 1] = time.time() - start_time
return time_data
```

Similarly, this section generates a private key for the second party and derives its public key. It performs the key exchange operation: first party's private key is used to compute a shared secret using second party's public key. This section also records the time taken to do so.

```
time_data = measure_ffdh_operations(iterations, key_size)
```

Calls 'measure_ffdh_operations' function with the specified number of iterations and key size, storing the result in 'time_data'

```
plt.figure(figsize=(10, 5))
plt.plot(time_data[:, 0], label="Key Generation Time")
plt.plot(time_data[:, 1], label="Key Exchange Time")
plt.xlabel("Iteration")
plt.ylabel("Time (seconds)")
plt.title(f"Time Complexity of FFDH Operations (Key Size: {key_size} bits)")
plt.legend()
plt.grid(True)
plt.show()
```

Outputs the graph

Appendix F:

```
from cryptography.hazmat.primitives.asymmetric import ec, dh
from cryptography.hazmat.backends import default_backend
import time
import matplotlib.pyplot as plt
import scienceplots
plt.style.use(["science", "notebook", "grid"])
```

Imports the necessary Python libraries

```
def time_ecdh_key_exchange(key_size):
```

Defines a function that takes 'key_size' as an argument to measure the time taken for ECDH key exchange

```
curve = {256: ec.SECP256R1, 384: ec.SECP256R1 , 521: ec.SECP256R1 }[key_size]
```

Creates a dictionary to map 'key_size' to the elliptic curve (specifically, the SECP256R1 curve variation) and then selects the appropriate elliptic curve based on provided 'key_size'

```
private_key = ec.generate_private_key(curve(), default_backend())
```

Generates a private key for ECDH using the selected elliptic curve

```
peer_public_key = ec.generate_private_key(curve(), default_backend()).public_key()
```

Generates a peer's public key for the key exchange by first generating a private key and then deriving its public key

```
start_time = time.time()
```

Records the current time before starting the key exchange

```
shared_key = private_key.exchange(ec.ECDH(), peer_public_key)
```

Performs ECDH key exchange to compute the shared key

```
end_time = time.time()
```

Records the current time immediately after the key exchange

```
return end_time - start_time
```

Calculates and returns the time taken for ECDH key exchange

```
def time_ffdh_key_exchange(key_size):
parameters = dh.generate_parameters(
generator=2, key_size=key_size, backend=default_backend())
private_key = parameters.generate_private_key()
peer_public_key = parameters.generate_private_key().public_key()
start_time = time.time()
shared_key = private_key.exchange(peer_public_key)
end_time = time.time()
return end_time – start_time
```

The above code block contains similar steps as above, but for FFDH

```
key_sizes = [256, 384, 521]
```

Key sizes for ECDH

```
key_sizes_ffdh = [2048, 3072, 4096]
```
Key sizes for FFDH

```
ecdh_times = [time_ecdh_key_exchange(size) for size in key_sizes]
ffdh_times = [time_ffdh_key_exchange(size) for size in key_sizes_ffdh]
```
Define arrays of key sizes for ECDH and FFDH and then uses list comprehension to measure and store the time taken for key exchanges at each key size

```
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(key_sizes, ecdh_times)
plt.xlabel("Key Size (bits)")
plt.ylabel("Runtime (seconds)")
plt.title("ECDH Key Exchange Time")
```
Plotting information for ECDH key exchange

```
plt.subplot(1, 2, 2)
plt.plot(key_sizes_ffdh, ffdh_times)
plt.xlabel("Key Size (bits)")
plt.ylabel("Runtime (seconds)")
plt.title("FFDH Key Exchange Time")
```
Plotting information for FFDH key exchange

```
plt.tight_layout()
plt.show()
```

Appendix G:

This code is based off of the script in Appendix F, but without FFDH.

```
from cryptography.hazmat.primitives.asymmetric import ec, dh
from cryptography.hazmat.backends import default_backend
import time
import matplotlib.pyplot as plt
import scienceplots
plt.style.use(["science", "notebook", "grid"])
def time_ecdh_key_exchange(key_size):
curve = {
256: ec.SECP256R1, 384: ec.SECP256R1, 521: ec.SECP256R1, 649: ec.SECP256R1,
777: ec.SECP256R1, 905: ec.SECP256R1, 1033: ec.SECP256R1, 1161: ec.SECP256R1,
1289: ec.SECP256R1, 1417: ec.SECP256R1, 1545: ec.SECP256R1, 1673: ec.SECP256R1,
1801: ec.SECP256R1, 1929: ec.SECP256R1, 2057: ec.SECP256R1, 2185: ec.SECP256R1,
2313: ec.SECP256R1, 2441: ec.SECP256R1, 2569: ec.SECP256R1, 2697: ec.SECP256R1,
2825: ec.SECP256R1, 2953: ec.SECP256R1, 3081: ec.SECP256R1}[key_size]
private_key = ec.generate_private_key(curve(), default_backend())
peer_public_key = ec.generate_private_key(curve(), default_backend()).public_key()
start_time = time.time()
shared_key = private_key.exchange(ec.ECDH(), peer_public_key)
end_time = time.time()
return end_time - start_time
key_sizes = [
256, 384, 521, 649, 777, 905, 1033, 1161, 1289, 1417, 1545, 1673, 1801, 1929, 2057, 2185, 2313,
2441, 2569, 2697, 2825, 2953, 3081]
ecdh_times = [time_ecdh_key_exchange(size) for size in key_sizes]
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(key_sizes, ecdh_times)
plt.xlabel("Key Size (bits)")
plt.ylabel("Runtime (seconds)")
plt.title("ECDH Key Exchange Time")
```

Sources:

1. Python Community Cryptography Contributors. (2023). Cryptography 42.0.0.dev1 documentation. Cryptography. https://cryptography.io/en/latest/
This is the URL to the ‘cryptography’ Python library documentation. This is crucial because it provides comprehensive guidelines and references on how to implement secure cryptographic operations. This library is pivotal for ensuring the integrity and security of data within the models, especially when dealing with sensitive information or executing key exchanges like ECDH and FFDH, as seen in the research. Understanding and applying the best practices from this documentation not only helps in constructing a robust cryptographic system but also in validating the reliability and efficiency of the empirical performance model, which is essential for the credibility of the research paper.

2. NumPy Developers. (2022). NumPy user guide — NumPy v1.26 Manual. NumPy. https://numpy.org/doc/stable/user/index.html#user
This is the URL to the ‘NumPy’ Python library documentation. This is essential as it offers detailed information on utilizing this powerful library for numerical computing in Python. NumPy provides the foundational array data structure and computational capabilities needed for scientific computing tasks.

3. Diffie, W., & Hellman, M. E. (1976). New Directions in Cryptography. IEEE Transactions on Information Theory, IT-22(6)
This is the seminal research paper published by Whitfield Diffie and Martin E. Hellman that set the foundation for the DHKE. It proposes methods to address challenges in cryptography and highlights the evolving role of communication and computation theories in solving longstanding cryptographic problems. It was the publication of this paper that led to many of our modern-day cryptographic communications techniques.

4. Corrigan-Gibbs, H., & Kogan, D. (2021). The Discrete-Logarithm Problem with Preprocessing. Stanford University.
This explores and analyzes the discrete logarithm problem (DLP). The DLP is fundamental in cryptography, underpinning key exchange protocols, public-key encryption, and digital signatures.

5. Haakegaard, R., & Lang, J. (2015). The Elliptic Curve Diffie-Hellman (ECDH).
This paper delves into the intricacies and security aspects of ECDH protocols. It discusses the distinction between ECDH and general DH, emphasizing ECDH’s reliance on the elliptic curve discrete logarithm problem (ECDLP) and its role in secure key exchange over insecure channels.

6. Polk, W. T., Dodson, D. F., Burr, W. E., Ferraiolo, H., & Cooper, D. 2018. NIST Special Publication 800-87-4, Cryptographic Algorithms and Key Sizes for Personal Identity Verification. National Institute of Standards and Technology.
The National Institute of Standards and Technology (NIST) is considered one of the leading agencies that establishes industry standards within the field of cybersecurity. This publication is where the recommended key sizes for ECDH and FFDH were derived from.

7. Mala, F. A., & Ali, R. (2022). The Big-O of Mathematics and Computer Science. Journal of Applied Mathematics and Computation, 6(1), 1-3. DOI: 10.26855/jamc.2022.03.001
This paper analyzes the complexity and efficiency of Big O Notation. It provides a comprehensive review of Big O Notation, which is pivotal in the study of computational complexity of algorithms. It covers the historical development of algorithmic analysis, the fundamental principles of Big O Notation, and its application in evaluating the computational complexities of algorithms and functions.

8. Miller, V. (1985). Use of Elliptic Curves in Cryptography. Conference Paper presented at the Center for Communications Research - Princeton
This paper, published in 1985, is considered one of the seminal academic works that propelled the use of elliptical curves in cryptographic communications. It covers much of the basic mathematics that are commonly used in elliptical curve cryptographic algorithms today.

9. Sung, S. (2020). An Introduction to Cipher Suites. Keyfactor. Retrieved December 2, 2023, from https://www.keyfactor.com/blog/cipher-suites-explained/
This article was immensely helpful in summarizing the information needed for describing cipher suites and how they pertain to DHKE.

10. Lake, J. (2023). Demystifying Diffie-Hellman key exchange and explaining how it works. Comparitech. Retrieved December 2, 2023, from https://www.comparitech.com/blog/information-security/diffie-hellman-key-exchange/
This article gave a step-by-step analysis of the DHKE and was helpful for summarizing information found in multiple academic papers.

11. GeeksforGeeks. (n.d.). Implementation of Diffie-Hellman Algorithm. Retrieved November 20, 2023, from https://www.geeksforgeeks.org/implementation-diffie-hellman-algorithm/?ref=gcse

12. GeeksforGeeks. (n.d.). Analysis of Algorithms | Big O Analysis. Retrieved November 20, 2023, from https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/

13. GeeksforGeeks. (n.d.). Big O Notation Interview Questions & Answers. Retrieved November 20, 2023, from https://www.geeksforgeeks.org/big-o-notation-interview-questions-answers/
GeeksforGeeks is considered one of the industry standard professional websites that many IT and computer science professionals go to for a valuable resource or to contribute some knowledge of their own. It was very helpful in providing articles on Big O Notation and the DHKE.

Picture(s):
1. Moses Gamio. Interview Noodle. (2021). Big O Notation. Retrieved November 20, 2023, from https://interviewnoodle.com/big-o-notation-e1aaca926c2
This was the website for the Big O Notation graph used on Page 3.