

Secure Authentication Scheme for Internet of Battlefield Things

*Project report submitted to the Amrita Vishwa Vidyapeetham in partial
fulfilment of the requirement for the Degree of*

BACHELOR of TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

Submitted by

SHOAIB AKHTAR - AM.EN.U4CSE20163

DEVU PAWAN ASHUTOSH - AM.EN.U4CSE20222

YESHWANTH REDDY - AM.EN.U4CSE20225

HARDIK KUMAR SINGH - AM.EN.U4CSE20232



**AMRITA SCHOOL OF COMPUTING
AMRITA VISHWA VIDYAPEETHAM
(Estd. U/S 3 of the UGC Act 1956)
AMRITAPURI CAMPUS**

KOLLAM -690525

MAY 2024

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
AMRITA VISHWA VIDYAPEETHAM
(Estd. U/S 3 of the UGC Act 1956)
Amritapuri Campus
Kollam -690525



BONAFIDE CERTIFICATE

This is to certify that the project report entitled "Secure Authentication Scheme for Internet of Battlefield Things" submitted by SHOAIB AKHTAR(AM.EN.U4CSE20163), DEVU PAWAN ASHUTOSH(AM.EN.U4CSE20222), YESHWANTH REDDY(AM.EN.U4CSE20225) and HARDIK KUMAR SINGH(AM.EN.U4CSE20232), in partial fulfillment of the requirements for the award of Degree of Bachelor of Technology in Computer Science and Engineering from Amrita Vishwa Vidyapeetham, is a bonafide record of the work carried out by them under my guidance and supervision at Amrita School of Computing, Amritapuri during Semester 8 of the academic year 2023-2024.

Dr. Nimmy K

Project Guide

Project Coordinator

Dr. Swaminathan J

Chairperson

Reviewer

Dept. of Computer Science & Engineering

Place : Amritapuri

Date : 15 May 2024

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

AMRITA VISHWA VIDYAPEETHAM

(Estd. U/S 3 of the UGC Act 1956)

Amritapuri Campus

Kollam -690525



DECLARATION

We, **SHOAIB AKHTAR(AM.EN.U4CSE20163)**, **DEVU PAWAN ASHUTOSH(AM.EN.U4CSE20222)**, **YESHWANTH REDDY(AM.EN.U4CSE20225)** and **HARDIK KUMAR SINGH(AM.EN.U4CSE20232)** hereby declare that this project entitled "Secure Authentication Scheme for Internet of Battlefield Things" is a record of the original work done by us under the guidance of **Dr. Nimmy K**, Dept. of Computer Science and Engineering, Amrita Vishwa Vidyapeetham, that this work has not formed the basis for any degree/diploma/associationship/fellowship or similar awards to any candidate in any university to the best of our knowledge.

Place : Amritapuri

Date : 17 May 2024

Signature of the Project Guide

Signature of the student

Acknowledgements

We wish to express our sincere gratitude to Mata Amritanandamayi Devi (Amma) for her divine blessings and unwavering inspiration throughout our project journey. Her compassionate guidance has served as a beacon of light, instilling in us the virtues of perseverance and dedication. We extend our heartfelt thanks to Dr. Swaminathan J, Chairperson of the Department of Computer Science & Engineering, for his steadfast support and encouragement. His guidance and mentorship have played a pivotal role in shaping our project and nurturing our academic growth. Our deepest appreciation goes to our esteemed mentor, Dr. Nimmy K, whose insightful feedback and constructive guidance have been instrumental in steering our project towards success. Her expertise and mentorship have motivated us to push our boundaries and strive for excellence. We are grateful to Ms. Namitha K, our project coordinator, for their consistent support and assistance throughout the project. Their organizational skills and dedication have ensured seamless coordination and timely completion of tasks. Special thanks are due to Mrs. Deepthi L.R, our reviewer, for her invaluable feedback and suggestions, which have significantly contributed to the refinement of our project. Her expertise and attention to detail have enhanced the quality and effectiveness of our work. We would also like to acknowledge the entire Department of Computer Science Engineering at Amrita Vishwa Vidyapeetham for providing us with a conducive academic environment. Their commitment to excellence has been a driving force behind our academic pursuits. We are deeply indebted to our families, friends, and all those who have supported us with their encouragement and patience throughout this journey. Their unwavering belief in us has been a constant source of motivation. Finally, we express our gratitude to all the individuals and entities who have directly or indirectly contributed to the success of our project.

Shoaib, Ashutosh, Yeshwanth, Hardik

Abstract

This research introduces a novel authentication protocol designed to enhance security within the Internet of Battlefield Things (IoBT). Traditional methods often struggle in the demanding conditions of combat environments. Our proposed protocol leverages fingerprint biometrics combined with the robust cryptographic technique of fuzzy extractors. This approach transforms variable fingerprint data into consistent cryptographic keys, ensuring reliable identification even in challenging scenarios. Additionally, geometric secret sharing safeguards these keys by distributing them between the user's device and the battlefield gateway. This multi-layered approach provides a continuous and non-disruptive authentication process, facilitating swift access to critical resources. Initial testing suggests notable improvements in both the speed and precision of authentication, potentially leading to heightened security and operational effectiveness within military settings. This paper delves into the protocol's architecture, underlying design principles, and preliminary results, underscoring its potential for future refinement and practical application in the IoBT landscape. The protocol's design prioritizes resilience in the face of adverse battlefield conditions, such as environmental factors impacting fingerprint quality or potential device damage. Our system is adaptable, accommodating various wearable sensor types and fingerprint acquisition methods. This flexibility allows for seamless integration into existing IoBT infrastructures, making it a practical solution for real-world military applications.

Keywords: Geometric Secret Sharing, Fuzzy Extractor, Internet of Battlefield Things (IoBT), Secure Authentication, Cryptography

Contents

Contents	i
List of Figures	iii
1 Introduction	1
1.1 Enhance Fingerprint Recognition Usability and Security using Fuzzy Extractor	1
1.2 Develop a Secure and Reliable Biometric Authentication Protocol for Battlefield Environments	2
2 Problem Definition	4
3 Related Work	5
4 Requirements	9
4.1 Hardware	9
4.1.1 Wearable Devices	9
4.2 Software	11
5 Proposed System	14
5.1 Registration Phase	15
5.2 Authentication Phase	18

5.3	Algorithms	20
5.3.1	Geometric Secret Sharing	20
5.3.2	Fuzzy Extractor with Code-Offset Construction	22
6	Result and Analysis	25
6.1	User	25
6.2	Device	27
6.3	Gateway	28
6.4	Output	30
6.5	Formal Security Analysis	32
6.5.1	AVISPA	32
6.5.2	Analysis of Security Features	33
7	Future Works	37
8	Conclusion	39
	References	41

List of Figures

5.1	Registration	16
5.2	Authentication	18
5.3	Enter Caption	21
5.4	Fuzzy extractor with code-offset	24
6.1	user code	25
6.2	Device code	27
6.3	Gateway code	29
6.4	output device	31
6.5	output user	31
6.6	output gateway	31

Chapter 1

Introduction

1.1 Enhance Fingerprint Recognition Usability and Security using Fuzzy Extractor

The first objective of this project focuses on the enhancement of fingerprint recognition technology, adapting it to meet the stringent requirements of battlefield conditions. In military operations, the effectiveness of biometric systems can be compromised by the harsh environment—dirt, debris, and sweat can all affect the accuracy of fingerprint sensors. Our goal is to develop a fingerprint recognition system that not only remains highly accurate under such conditions but is also easy for soldiers to use in high-pressure scenarios. This includes increasing the system’s tolerance to errors such as partial or smeared prints, ensuring reliable authentication even when ideal fingerprint capture is not possible.

In addition to improving usability and error tolerance, a critical component of our work involves safeguarding the biometric data itself. Utilizing fuzzy extractors, we plan to encrypt and store biometric data in a manner

that significantly enhances security. Fuzzy extractors help in generating a consistent cryptographic key from an input that is not exact but close enough, such as fingerprints, and thus are ideal for handling the variability inherent in biometric data. This method will prevent the potential reconstruction of fingerprint templates from stored data, addressing a major vulnerability in traditional biometric systems.

Furthermore, our approach includes robust measures to verify the authenticity of both the user and the sensor hardware, preventing impersonation by unauthorized parties. By ensuring that the sensors themselves can authenticate their identity, we mitigate the risk of spoofing attacks where fraudulent devices or data might otherwise gain unauthorized access to the system.

1.2 Develop a Secure and Reliable Biometric Authentication Protocol for Battlefield Environments

The second objective aims to develop a comprehensive authentication protocol that utilizes fingerprint biometrics, tailored specifically for the demanding conditions of battlefield environments. The protocol must not only be secure and reliable but also capable of operating effectively under the unique operational and environmental challenges faced by military personnel. Rapid, efficient, and unobtrusive authentication is crucial, as soldiers require immediate access to information and resources without compromising their safety or mission integrity.

To achieve this, the protocol will integrate advanced cryptographic techniques and robust security measures, ensuring the authentication process

is resistant to a wide range of threats, including cyberattacks and physical tampering. By leveraging the uniqueness of biometric data, particularly fingerprints, the protocol will provide a secure and non-transferable method of confirming a soldier's identity, ensuring that access to critical systems and information is tightly controlled.

The reliability of the system in adverse conditions is also a primary concern. The protocol will be designed to maintain functionality in environments with limited connectivity, in extreme weather, and in scenarios where conventional communication infrastructures might be compromised. This includes the capacity to operate independently of central servers when necessary, using decentralized or edge computing solutions to process and verify biometric data locally.

By addressing these challenges, the developed protocol will significantly enhance the operational capabilities of soldiers, allowing for secure and seamless interaction with the IoBT and supporting the strategic objectives of military operations with improved safety and efficiency.

Chapter 2

Problem Definition

This project aims to tackle the pressing issue of inadequate authentication protocols for soldiers utilizing the Internet of Battlefield Things (IoBT). In the rugged and dynamic environment of the battlefield, conventional authentication methods such as passwords and tokens prove to be unreliable and insecure. This creates significant vulnerabilities that compromise operational safety and security. To address this challenge, there is a pressing need for a robust, dependable, and swift authentication solution tailored specifically for the unique demands of military operations. Soldiers require a system that can swiftly and securely verify their identities, granting them access to critical battlefield information and devices without jeopardizing mission success or compromising sensitive data. By implementing such an authentication protocol, we can significantly enhance the resilience and efficiency of military communications and operations in challenging environments.

Chapter 3

Related Work

The paper titled "A Novel Secure Authentication Protocol for IoT and Cloud Servers" addresses the vulnerabilities present in current authentication methods for IoT environments, such as susceptibility to impersonation and offline password guessing attacks, while considering the computational burdens that robust security measures can introduce, particularly for resource-constrained IoT devices. The authors propose a new authentication scheme leveraging Elliptic Curve Cryptography (ECC), aiming to mitigate these challenges. ECC offers strong security with smaller key sizes, potentially making it suitable for IoT applications, and the authors claim that their protocol improves computational efficiency and reduces communication overhead compared to existing alternatives. However, the paper likely focuses more on theoretical analysis and the potential security advantages of the ECC-based protocol rather than extensive real-world implementation and testing. Additionally, while ECC is relatively efficient, it may still introduce overhead for extremely low-powered IoT devices, thus posing a limitation in certain hardware-constrained scenarios.

The next paper titled "A Secure, Lightweight, and Anonymous User Au-

thentication Protocol for IoT Environments,” the paper addresses the prevalent security vulnerabilities in existing authentication schemes tailored for IoT settings. These schemes often demand substantial computational resources, rendering them impractical for resource-constrained IoT devices. Moreover, the protocol identified a lack of emphasis on user anonymity and untraceability, pivotal for privacy in IoT contexts. By primarily utilizing hash functions and XOR operations, the protocol ensures efficiency, particularly suitable for low-power IoT devices. Notably, it incorporates a three-factor authentication system involving passwords, smart cards, and biometric data, enhancing security. While the paper focuses on countering network-based attacks, it acknowledges the susceptibility of IoT devices to physical breaches. However, it foresees potential bottlenecks in large-scale scenarios with numerous concurrent users, a consideration absent in theoretical analyses.

The 2013 paper titled ”Fuzzy Extractors from Fingerprints” addresses the challenge of utilizing noisy biometric data like fingerprints for cryptographic purposes. Existing fuzzy extractor constructions at the time suffered from high computational overhead, rendering them impractical for user authentication systems. The authors likely prioritize minimizing the size of the ”helper data” generated by the fuzzy extractor to reduce storage overhead. They ensure key indistinguishability, preventing adversaries from learning substantial information about the secret key derived from the fingerprint, even with access to the helper data. Crucially, they achieve constant computational cost during the identification phase, vital for real-time authentication scenarios, particularly on resource-constrained devices. However, the performance of the fuzzy extractor may vary depending on the type of fingerprint sensor used, with optimizations for one sensor potentially not generalizing perfectly to others. Additionally, while theoretical security analyses focus on

core fuzzy extractor algorithms, practical implementations could introduce side-channel vulnerabilities if not carefully secured, such as timing attacks.

The paper titled "Lightweight and Privacy-Preserving Remote User Authentication for Smart Homes," published in IEEE Access in December 2021, presents a user authentication protocol tailored for smart home environments. It addresses the shortcomings of existing protocols, particularly their vulnerability to smart home attacks and computational overhead. Utilizing Photo Response Non-Uniformity (PRNU) for resilience against smartphone capture and phishing attacks, the protocol ensures lightweight deployment on resource-constrained IoT devices. Geometric secret sharing facilitates mutual authentication among entities. While the scheme boasts a 20 percent reduction in communication overhead compared to existing methods on smart devices, it heavily relies on the consistency of PRNU, potentially susceptible to variability in image sensor noise. Additionally, the protocol assumes single-device interaction scenarios, potentially necessitating further considerations for multi-device authentication flows. The paper primarily focuses on remote attacks, neglecting discussions on the protocol's robustness against physical side-channel attacks. Furthermore, scalability concerns regarding computational overhead in large-scale smart home environments remain inadequately addressed.

"IoBT-OS: Optimizing the Sensing-to-Decision Pipeline for the Internet of Battlefield Things," presented at ICCCN 2022, addresses the critical need for fast and reliable decision-making in battlefield environments. Existing systems often suffer from sluggishness and dependency on remote computing resources, posing risks in scenarios where communication can be severed. IoBT-OS is introduced as an operating system specifically designed to enhance decision-making at the tactical edge by prioritizing resource efficiency.

It integrates several key components, including an edge AI efficiency library, real-time data management algorithms informed by mission objectives, digital twin support, and offline training capabilities. The edge AI efficiency library employs techniques such as compressed neural networks, confidence estimation, and compressive offloading to minimize processing time and energy consumption. Digital twins enable complex computations to be performed in the cloud while maintaining operational functionality. Notably, the paper focuses on optimizing edge efficiency and lacks discussion on resilience or tailored intelligence. Challenges include accurately modeling latency to describe the relationship between model execution time and neural network parameters affecting output quality.

Chapter 4

Requirements

The design of this project contains both hardware and software. The specifications are listed below.

4.1 Hardware

4.1.1 Wearable Devices

- A wearable device, such as a smartwatch or a wristband, equipped with secure communication capabilities can serve as a user device in an IoBT environment. These devices, designed to be worn on the body, often incorporate sensors for biometric authentication, such as fingerprint or heart rate sensors, ensuring secure access to the device and network. They can communicate with gateways or other devices using low-power wireless technologies, enabling the exchange of data and commands. In a battlefield scenario, such a wearable could display critical information like maps or enemy locations, while securely transmitting real-time data on the soldier's health and location back to command. The device's form factor is optimized for comfort, durability, and discreetness,

making it suitable for use in demanding environments.

- **Fingerprint Scanner** : A fingerprint scanner designed solely for recording fingerprints operates by capturing images of individuals' fingerprints upon contact with its surface. These scanners employ various sensor technologies like optical, capacitive, or ultrasonic to acquire high-resolution images of fingerprint patterns. Once captured, these images are processed and stored securely in a database for future reference. This process ensures that each individual's unique fingerprint is accurately recorded and can be accessed when needed for verification or identification purposes.
- **Arduino** : Arduino is an open-source electronics platform that simplifies the creation of interactive projects. It consists of a microcontroller at its core, which processes instructions and interfaces with various input and output devices. Users write code, called sketches, in the Arduino IDE using a simplified version of C++, which controls the behavior of the microcontroller. These sketches enable the interaction with sensors, LEDs, motors, and other components connected to the Arduino board. Once uploaded, the microcontroller executes the instructions, allowing users to prototype and experiment with electronics projects easily. Arduino's accessibility and versatility make it a popular choice for hobbyists, students, and professionals exploring a wide range of applications, from robotics and home automation to art installations and scientific experiments.

4.2 Software

- **VS CODE** : Visual Studio Code (VS Code) is a lightweight yet powerful source code editor developed by Microsoft. It is renowned for its versatility, ease of use, and extensive customization options, making it a popular choice among developers across various programming languages and platforms. VS Code provides features such as syntax highlighting, code completion, debugging capabilities, and Git integration, enhancing productivity and facilitating collaborative development. Its rich ecosystem of extensions further extends its functionality, allowing users to tailor the editor to their specific needs and preferences. With its fast performance and cross-platform support for Windows, macOS, and Linux, Visual Studio Code has become an indispensable tool for developers worldwide.
- **Fuzzy Extractor** : Fuzzy extractors are cryptographic tools designed to derive stable cryptographic keys from biometric data like fingerprints or iris scans, even in the presence of noise or variations. They work by first capturing and processing the biometric data to create a template. Error correction techniques are then employed to handle variations, ensuring that the data remains reliable. From this corrected data, a secure cryptographic key is generated using one-way functions. During authentication, captured biometric data is similarly processed, error-corrected, and matched against the stored key. Fuzzy extractors offer a robust solution for secure biometric authentication while maintaining privacy by not storing raw biometric data directly.
- **Hashing Algorithms**:
 - **AES** : The Advanced Encryption Standard (AES) is a widely-used

cryptographic algorithm for encryption and decryption of data. However, it's not a hashing algorithm. AES operates on blocks of data, typically 128 bits, using a symmetric key, which means the same key is used for both encryption and decryption. It employs a substitution-permutation network, where data undergoes multiple rounds of substitution and permutation operations, including byte substitution, row shifting, column mixing, and key addition. The number of rounds depends on the key size, with AES supporting key sizes of 128, 192, or 256 bits. Due to its security, efficiency, and standardization by the National Institute of Standards and Technology (NIST), AES is commonly utilized in various applications such as secure communication protocols, disk encryption, and digital signatures.

- SHA-256 : part of the SHA-2 (Secure Hash Algorithm 2) family, is a widely-used cryptographic hash function that generates a fixed-size 256-bit (32-byte) hash value from input data of arbitrary size. As a one-way function, SHA-256 is designed to be computationally infeasible to reverse, meaning it's practically impossible to retrieve the original input data from its hash. SHA-256 operates through a series of logical operations, including bitwise operations, modular addition, and rotations, applied to the input data in successive blocks. This process results in a unique, deterministic hash value for each unique input, making it suitable for various security applications such as data integrity verification, digital signatures, password hashing, and blockchain technology. The cryptographic strength and widespread adoption of SHA-256 make it a fundamental component of modern information security

protocols.

- **Socket Programming :** Socket programming in Python allows communication between processes over a network using sockets. Python's 'socket' module provides a straightforward interface for creating and interacting with sockets. To establish a connection, a server creates a socket, binds it to a specific address and port, and then listens for incoming connections. Clients initiate a connection by creating a socket and specifying the server's address and port. Once connected, data can be transmitted bidirectionally between the client and server using methods like 'send()' and 'recv()'. Python's socket programming enables the implementation of various network protocols such as TCP/IP and UDP, making it suitable for building client-server applications, distributed systems, and network services. Its simplicity and versatility have made it a popular choice for network communication in Python applications.

Chapter 5

Proposed System

Robust yet user-friendly biometric authentication is crucial in various contexts, particularly in high-stakes environments like battlefield settings. By integrating fuzzy extractors with traditional fingerprint recognition, a more secure authentication mechanism is achieved. This fusion accommodates natural variations in biometric data, mitigating the need for perfect scans every time. Such adaptability not only enhances security but also significantly improves user experience, a critical factor in real-world applications.

Moreover, the incorporation of mutual authentication further fortifies security measures. Utilizing geometric secret sharing facilitates two-way authentication between users (and their devices) and the server. This method acts as a deterrent against man-in-the-middle attacks and server impersonation, thereby elevating the overall security level of the Internet of Battlefield Things (IoBT) system. This mutual authentication mechanism establishes trust between the communicating entities, ensuring that only legitimate connections are established, and sensitive data remains protected from unauthorized access or tampering.

In essence, the combined approach of integrating fuzzy extractors with

traditional fingerprint recognition and employing geometric secret sharing for mutual authentication presents a comprehensive solution that not only enhances security but also maintains user-friendliness. This is particularly advantageous in dynamic and high-pressure environments such as battlefield settings, where the reliability and efficiency of authentication mechanisms are paramount.

5.1 Registration Phase

During the registration phase of the project, a meticulous process unfolds to establish a secure and reliable means of user authentication. At the forefront of this process is the utilization of a fingerprint sensor, a sophisticated device designed to capture and digitize the unique fingerprint patterns of individuals. This initial step sets the foundation for the subsequent stages of authentication, ensuring that only authorized users gain access to the system.

Upon activation, the fingerprint sensor diligently collects the fingerprint data of the user. This raw data serves as the primary input for the authentication process, acting as a digital representation of the user's biometric identity. However, before proceeding further, the collected fingerprint undergoes a preliminary examination to verify the identity of the user. This crucial step acts as the first line of defense, preventing unauthorized individuals from gaining entry into the system.

Once the user's identity is confirmed through the preliminary examination, the collected fingerprint data undergoes a series of security measures to fortify its integrity. One such measure involves the generation of a fuzzy key, a cryptographic technique that adds an additional layer of complexity

to the stored data, making it resistant to unauthorized access or tampering. Alternatively, the fingerprint data may be stored as a user nonce, a unique identifier assigned to each user for authentication purposes. Additionally, the data may undergo hashing and encryption processes to further enhance its security, rendering it virtually indecipherable to unauthorized entities. With

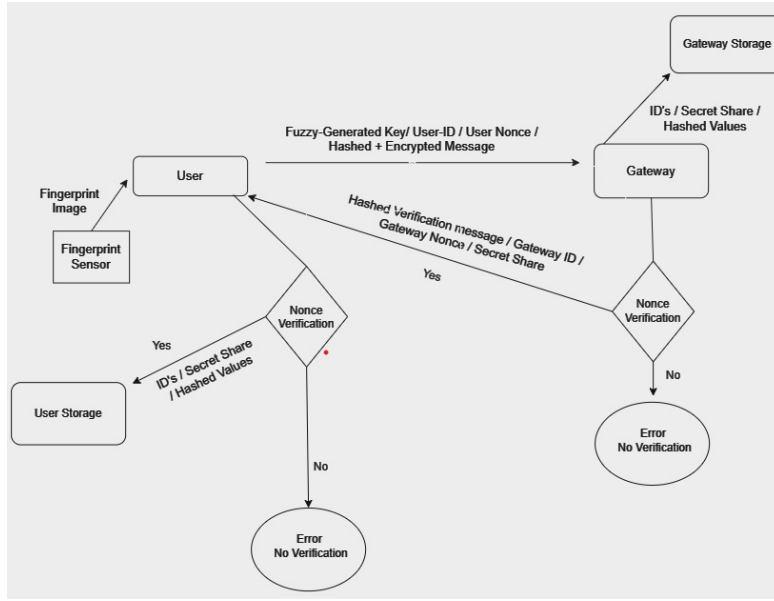


Figure 5.1: Registration

the fingerprint data securely processed and encrypted, it is transmitted to the gateway for centralized storage and processing. The gateway serves as a central hub within the system, responsible for managing user authentication requests and storing critical authentication data securely. Upon receiving the encrypted fingerprint data, the gateway stores it along with corresponding identifiers, secret shares, and hashed values in its storage repository. This meticulous storage process ensures that the user's biometric data remains protected and accessible only to authorized personnel.

Simultaneously, the gateway initiates a nonce verification process to validate the authenticity of the transmitted data. The nonce, a random or pseudo-random number generated for each authentication session, serves as

a unique identifier and authentication token. By verifying the nonce associated with the transmitted data, the gateway ensures that the data originates from a legitimate source and has not been intercepted or tampered with en route.

If the nonce verification process yields a positive result, indicating the authenticity of the transmitted data, the gateway responds by sending back a hashed verification message to the user. This message serves as confirmation of successful authentication, providing the user with assurance that their identity has been successfully verified. Additionally, the gateway includes its own identifier, nonce, and secret share in the response, further enhancing the security of the authentication process.

However, in the event of unsuccessful nonce verification, the gateway promptly notifies the user by issuing an error message. This prompt response mechanism ensures that any potential authentication issues are addressed promptly, minimizing the risk of unauthorized access or security breaches within the system.

In conclusion, the registration phase of the project encompasses a comprehensive authentication process designed to establish a secure and reliable means of user verification. From the meticulous collection of fingerprint data to the centralized storage and processing at the gateway, every step is carefully orchestrated to safeguard user identities and ensure the integrity of the authentication process. Through the integration of advanced cryptographic techniques and robust security measures, the registration phase lays the groundwork for a secure and efficient authentication system capable of meeting the stringent security requirements of modern applications.

5.2 Authentication Phase

During the authentication phase, the system orchestrates a sophisticated process to verify the identity of users and devices securely. At the heart of this phase lies the fingerprint sensor, which plays a pivotal role in capturing and identifying the unique fingerprint images of users. Once the fingerprint image is collected, the sensor proceeds to identify the user based on the captured data.

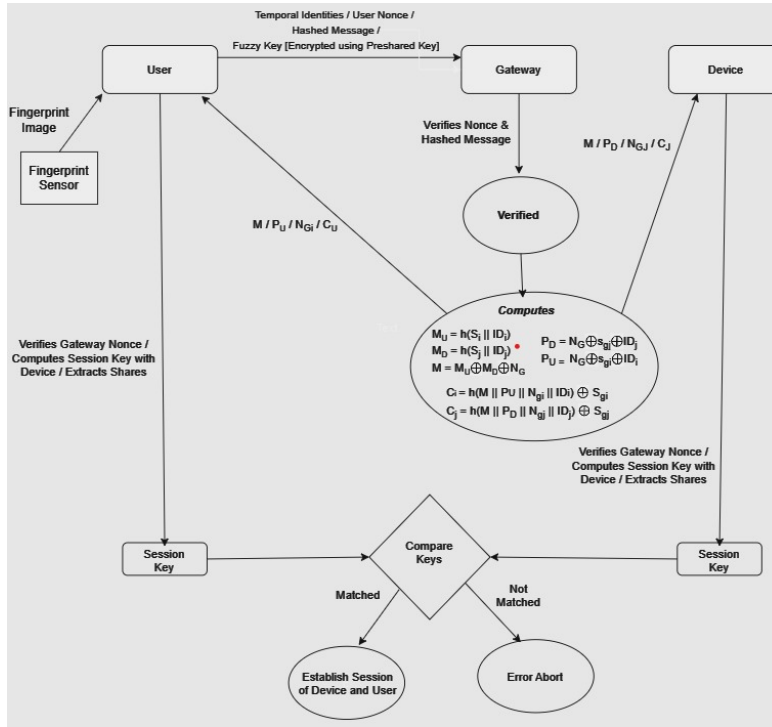


Figure 5.2: Authentication

Upon successful identification, the sensor generates temporal identities for the user, along with essential authentication elements such as user nonces, hashed messages, and encrypted fuzzy keys using a pre-shared key. These components form the crux of the authentication process, ensuring that only authorized users and devices gain access to the system.

Subsequently, the sensor transmits these authentication elements to the

gateway for further verification. At the gateway, a meticulous verification process ensues, wherein the nonce and hashed message received from the sensor are scrutinized for authenticity. By verifying these critical components, the gateway confirms the legitimacy of the authentication data and proceeds with the authentication process.

Following successful verification, the gateway computes and generates distinct messages for both the user and the device. These messages, denoted as Message for User (M_u) and Message for Device (M_d), contain essential authentication information crucial for establishing a secure connection between the user and the device. Additionally, the messages incorporate encrypted shares for both the user (P_u) and the device (P_d), further enhancing the security of the authentication process.

the gateway decrypts the image I and verifies integrity and freshness of the image by performing hash comparison and image analysis respectively. Upon successful verification, it extracts $Prnu$ and Im and compares with the stored values. Gateway then proceeds to perform face recognition of the user. After successful verification, the gateway retrieves $MD=h(S_j||ID_j)$ and $MU=h(S_i||ID_i)$ from its memory. To establish a session key at both user and device end, Gateway computes $M=MU\oplus MD\oplus NG$ using the newly generated nonce NG . It then encrypts the shares as follows $PU=NG\oplus sGi\oplus ID_i$, $PD=NG\oplus sGj\oplus ID_j$. Further, gateway computes $Cj=h(M||PD||NGj||IDj)\oplus sGj$ and $Ci=h(M||PU||NGi||IDi)\oplus sGi$ for verification of the received message at both ends with newly generated NGj and NGi . Gateway then sends a message $\{M, PD, NGj, Cj\}$ to the device and sends another message $\{M, PU, NGi, Ci\}$ to the user.

The gateway also computes hashed values (Ci and Cj) of the messages, serving as additional security measures to safeguard against potential tam-

pering or unauthorized access.

Upon completion of the authentication process at the gateway, the authenticated messages, along with their corresponding hashed values, are transmitted to both the user and the device. Upon receiving the authentication data, both the user and the device independently generate their own gateway nonces and compute session keys with the device.

Furthermore, each entity extracts and compares their respective shares derived from the received messages. If the comparison yields a match, indicating the successful authentication of both the user and the device, the system proceeds to establish a secure session between the user and the device. Conversely, if any inconsistencies are detected during the comparison process, the system promptly aborts the authentication attempt, safeguarding against unauthorized access or potential security breaches.

In summary, the authentication phase of the system employs a robust and intricate process to verify the identities of users and devices securely. From the initial fingerprint identification to the generation and verification of authentication elements at the gateway, every step is meticulously executed to ensure the integrity and security of the authentication process. Through the integration of advanced cryptographic techniques and stringent security measures, the authentication phase establishes a reliable foundation for secure communication and interaction within the system.

5.3 Algorithms

5.3.1 Geometric Secret Sharing

- This algorithm is for Geometric Secret Sharing, where a secret \overline{S} is divided into multiple shares such that at least t shares are required to

1 Geometric Secret Sharing Algorithm

Algorithm 1 Geometric Secret Sharing

```

1: procedure SECRETSHARING( $Sec, p$ )
2:   Generate random numbers  $Rand$  and  $Sec$ 
3:   Compute shares:
4:    $share1 \leftarrow (Sec + Rand) \bmod p$             $\triangleright p$  is a large prime number
5:    $share2 \leftarrow (Sec + 2Rand) \bmod p$         $\triangleright Rand$  is a random number
6:   return  $share1, share2$ 
7: end procedure

```

Algorithm 2 Reconstruction of Secret

```

1: procedure RECONSTRUCTSECRET( $share1, share2, p$ )
2:   Reconstruct the secret  $Sec$ :
3:    $Sec \leftarrow (2 \times share1 - share2) \bmod p$     $\triangleright p$  is a large prime number
4:   return  $Sec$ 
5: end procedure

```

Figure 5.3: Enter Caption

reconstruct the secret.

1. Setup:

In this step, we select a large prime number p . This prime number serves as the modulus for the arithmetic operations performed during secret sharing, ensuring computational security.

2. Secret Sharing:

Two random numbers, $Rand$ and Sec , are generated. Sec represents the secret that needs to be shared among participants, while $Rand$ is a randomly chosen number used in the secret sharing process. These random numbers are crucial for the security of the scheme.

3. Compute shares:

Two shares, $share1$ and $share2$, are computed using the secret Sec and the random number $Rand$. $share1$ is calculated by adding Sec and $Rand$ together and taking the result modulo p . Similarly, $share2$ is computed by adding Sec and $2Rand$ together and taking the result modulo p . These shares are distributed among the participants.

4. Reconstruction of Secret:

To reconstruct the original secret Sec , the shares $share1$ and $share2$ are combined using a reconstruction formula. By subtracting $share2$ from twice $share1$, and then taking the result modulo p , we can retrieve the original secret Sec . This process ensures that only authorized subsets of participants can reconstruct the secret.

5.3.2 Fuzzy Extractor with Code-Offset Construction

1. **Input Parameters:** The algorithm takes input parameters such as the desired key length (`length`), maximum allowable Hamming error (`ham_err`), and the probability of key reproduction error (`rep_err`), along with the source value (`value`) for which a key needs to be generated.
2. **Initialization:** The algorithm initializes the Fuzzy Extractor with the provided input parameters and parses any additional arguments required for the digital lockers.
3. **Key Generation:** This step involves generating a random key of the specified length (`length`) and initializing arrays for nonces, masks, and digests, each with a size equal to the number of helpers (`num_helpers`). For each helper, a random nonce and mask are generated, and bitwise AND operation is performed between the mask and the source value to

obtain vectors. These vectors are then hashed using PBKDF2-HMAC to produce digests. Finally, the digests are XORed with the padded key to generate ciphertexts.

4. **Reproduction:** In this step, the algorithm reproduces the key based on a given source value (`value`) and previously generated helpers. It initializes arrays for ciphertexts, masks, and nonces using the provided helpers. Bitwise AND operation is performed between the masks and the source value to obtain vectors, which are then hashed using PBKDF2-HMAC to produce digests. These digests are XORed with the ciphertexts to obtain plaintexts. The algorithm checks if the sum of the last `sec_len` bytes of each plaintext is zero, indicating a successful unlock. If so, it returns the corresponding key; otherwise, it returns `None`.

Algorithm 3 Fuzzy Extractor

```

1: Input: length, ham_err, rep_err, value
2: Output: key
3: procedure FUZZYEXTRACTOR(length, ham_err, rep_err, value)
4:   lockerArgs  $\leftarrow$  Parse arguments for digital lockers
5:   Initialize FuzzyExtractor with length, ham_err, rep_err, lockerArgs
6:   key, (ciphers, masks, nonces)  $\leftarrow$  Generate(value)
7:   return key
8: end procedure
9:
10: procedure GENERATE(value)
11:   key  $\leftarrow$  Generate random key of length length
12:   Initialize arrays nonces, masks, digests with size num_helpers
13:   for helper in num_helpers do
14:     nonces[helper]  $\leftarrow$  Generate random nonce of length nonce_len
15:     masks[helper]  $\leftarrow$  Generate random mask of length length
16:     vectors[helper]  $\leftarrow$  Bitwise AND operation between masks[helper]
and value
17:   end for
18:   for helper in num_helpers do
19:     d_vector  $\leftarrow$  vectors[helper]
20:     d_nonce  $\leftarrow$  nonces[helper]
21:     digest  $\leftarrow$  Compute PBKDF2-HMAC(d_vector, d_nonce)
22:     digests[helper]  $\leftarrow$  digest
23:   end for
24:   ciphers  $\leftarrow$  Bitwise XOR operation between digests and padded key
25:   return key, (ciphers, masks, nonces)
26: end procedure
27:
28: procedure REPRODUCE(value, helpers)
29:   Initialize arrays ciphers, masks, nonces from helpers
30:   vectors  $\leftarrow$  Bitwise AND operation between masks and value
31:   for helper in num_helpers do
32:     d_vector  $\leftarrow$  vectors[helper]
33:     d_nonce  $\leftarrow$  nonces[helper]
34:     digest  $\leftarrow$  Compute PBKDF2-HMAC(d_vector, d_nonce)
35:     digests[helper]  $\leftarrow$  digest
36:   end for
37:   plains  $\leftarrow$  Bitwise XOR operation between digests and ciphers
38:   for check in num_helpers do
39:     if  $\sum plains[check][-sec\_len :] = 0$  then
40:       return plains[check][: -sec\_len]
41:     end if
42:   end for
43:   return None
44: end procedure

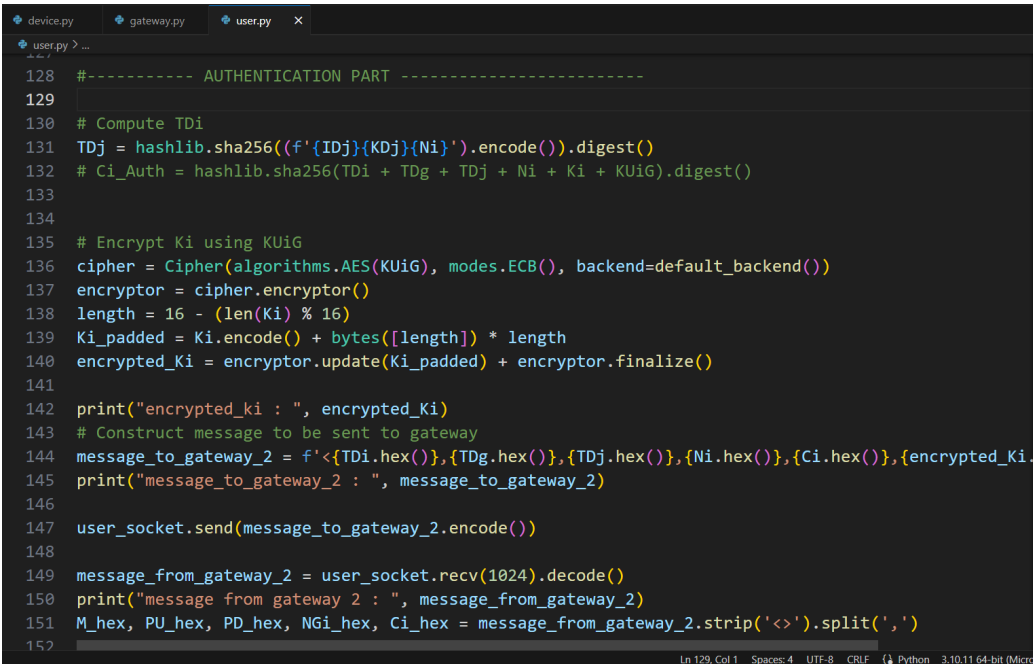
```

Figure 5.4: Fuzzy extractor with code-offset

Chapter 6

Result and Analysis

6.1 User

A screenshot of a code editor with three tabs: 'device.py', 'gateway.py', and 'user.py'. The 'user.py' tab is active, showing Python code for user authentication and message exchange. The code includes comments and uses the hashlib library for SHA-256 hashing and the Cipher module for AES encryption. It calculates TDj, Ci_Auth, and encrypted_Ki, then constructs a message to send to the gateway and receives a response back.

```
128 #----- AUTHENTICATION PART -----
129
130 # Compute TDi
131 TDj = hashlib.sha256((f'{IDj}{KDj}{Ni}').encode()).digest()
132 # Ci_Auth = hashlib.sha256(TDi + TDg + TDj + Ni + Ki + KUiG).digest()
133
134
135 # Encrypt Ki using KUiG
136 cipher = Cipher(algorithms.AES(KUiG), modes.ECB(), backend=default_backend())
137 encryptor = cipher.encryptor()
138 length = 16 - (len(Ki) % 16)
139 Ki_padded = Ki.encode() + bytes([length]) * length
140 encrypted_Ki = encryptor.update(Ki_padded) + encryptor.finalize()
141
142 print("encrypted_ki : ", encrypted_Ki)
143 # Construct message to be sent to gateway
144 message_to_gateway_2 = f'<{TDi.hex()}, {TDg.hex()}, {TDj.hex()}, {Ni.hex()}, {Ci.hex()}, {encrypted_Ki}
145 print("message_to_gateway_2 : ", message_to_gateway_2)
146
147 user_socket.send(message_to_gateway_2.encode())
148
149 message_from_gateway_2 = user_socket.recv(1024).decode()
150 print("message from gateway 2 : ", message_from_gateway_2)
151 M_hex, PU_hex, PD_hex, NGi_hex, Ci_hex = message_from_gateway_2.strip('<>').split(',')
152
```

Figure 6.1: user code

This code snippet seems to be part of a communication protocol involving cryptographic operations between a client and a gateway. Let's break it down step by step:

1. **TDj Calculation:** TDj is calculated using the SHA-256 hash function. It takes the concatenation of three elements: IDj, KDj, and Ni. This operation generates a cryptographic hash digest.

2. **Ci_Auth Calculation:** Ci_Auth is computed using SHA-256 hash function as well. It takes the concatenation of several elements including TDj, TDg, Ni, Ki, and KUiG. This hash will be used for authentication purposes.

3. **Encrypting Ki:** Ki, which seems to be a key, is encrypted using the AES algorithm with ECB mode. ECB mode is used for simplicity here, though it's generally not recommended for secure communication due to its vulnerability to certain attacks. The key used for encryption is KUiG. The length of Ki is padded to a multiple of 16 bytes (AES block size), and then it's encrypted.

4. **Constructing Message to Gateway:** A message is constructed using the calculated TD values, Ni, Ci_Auth, and the encrypted Ki. This message is formatted as a string.

5. **Sending Message to Gateway:** The constructed message is sent to the gateway using a user socket.

6. **Receiving and Parsing Message from Gateway:** The client waits to receive a response from the gateway. Upon receiving the response, it is decoded and split into different components: M_hex, PU_hex, PD_hex, NGi_hex, and Ci_hex.

This code snippet seems to be part of a client-server communication protocol where the client sends a message to the gateway, expecting a response. The communication involves cryptographic operations like hashing and encryption to ensure data integrity and confidentiality. Additionally, parsing of the received message allows the client to extract relevant information from the gateway's response.

6.2 Device



```

1 import socket
2 import hashlib
3 from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
4 from cryptography.hazmat.backends import default_backend
5
6
7 # Define the gateway address and port
8 gateway_address = ('localhost', 12346)
9
10 # Connect to the gateway
11 device_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
12 device_socket.connect(gateway_address)
13
14 IDi = "xyz"
15 IDj = "ghi"
16
17 message_from_gateway_1 = device_socket.recv(1024).decode()
18 M_hex, PU_hex, PD_hex, NGj_hex, Cj_hex = message_from_gateway_1.strip('<>').split(',')
19
20
21 message_from_gateway_2 = device_socket.recv(1024).decode() # Decode the received bytes to a string
22 Nj_hex, R_hex = message_from_gateway_2.strip('<>').split(',')
23 Nj = bytes.fromhex(Nj_hex)
24 R = bytes.fromhex(R_hex)
25 print("Nj Received:", Nj.hex())
26 print("R received:", R.hex())

```

Figure 6.2: Device code

Certainly, let's delve into the theoretical explanation of each part of the code:

1. Imports: - 'socket': This module provides access to the BSD socket interface, which is used for network communication. - 'hashlib': It offers hashing algorithms like SHA-256 for cryptographic operations. - 'Ciphers', 'algorithms', 'modes' from 'cryptography.hazmat.primitives.ciphers': These are part of the 'cryptography' library, which provides cryptographic algorithms for encryption and decryption. - 'default_backend' from 'cryptography.hazmat.backends': It provides access to the default cryptographic backend for performing cryptographic operations.

2. Gateway Address and Port: - 'gateway_address': It defines the address and port of the gateway to which the client will connect. In this case, it's set to 'localhost' and port '12346'.

3. Connecting to the Gateway: - A TCP socket (`'device_socket'`) is created using `'socket.socket()'`, specifying the address family (`'AF_INET'` for IPv4) and socket type (`'SOCK_STREAM'` for TCP). - `'connect()'` is used to establish a connection to the gateway using the provided address and port.

4. Receiving Messages from the Gateway: - Two messages are received from the gateway using `'device_socket.recv(1024).decode()'`. The `'recv()'` method waits to receive data from the gateway, and `'decode()'` converts the received bytes into a string. - The received messages are then split into components using the `'split()'` method based on the delimiter `','`.

5. Converting Hexadecimal Strings: - The hexadecimal strings `'Nj_hex'` and `'R_hex'` are converted into bytes objects (`'Nj'` and `'R'`) using the `'bytes.fromhex()'` method. This is necessary to work with binary data in Python.

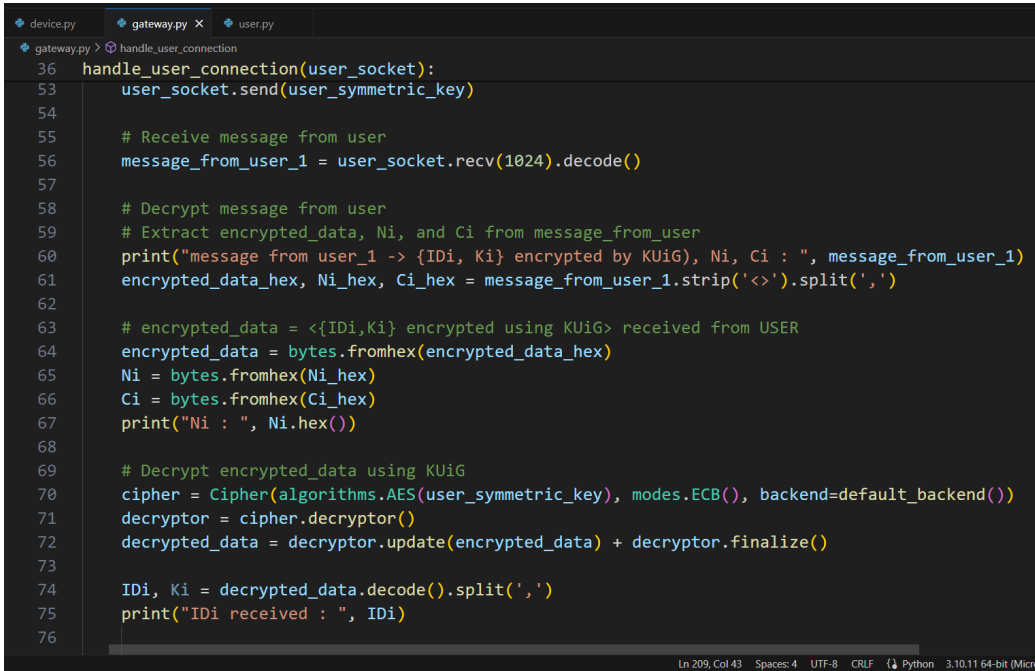
6. Printing Received Data: - The received `'Nj'` and `'R'` values are printed in hexadecimal format using `'hex()'`.

Overall, the code establishes a connection to a gateway, receives messages from it, extracts relevant data from the messages, and prints out some of this data. This is a typical pattern in client-server communication where the client interacts with the server by sending and receiving messages over a network connection.

6.3 Gateway

This code snippet appears to handle the communication from a user to the server-side, decrypting the message received from the user. Let's break down the steps:

1. Receiving the Message from the User: - The code receives a message from the user via the `'user_socket'` connection using `'user_socket.recv(1024).decode()'`.



```

36 handle_user_connection(user_socket):
53     user_socket.send(user_symmetric_key)
54
55     # Receive message from user
56     message_from_user_1 = user_socket.recv(1024).decode()
57
58     # Decrypt message from user
59     # Extract encrypted_data, Ni, and Ci from message_from_user
60     print("message from user_1 -> {IDi, Ki} encrypted by KUiG), Ni, Ci : ", message_from_user_1)
61     encrypted_data_hex, Ni_hex, Ci_hex = message_from_user_1.strip('<>').split(',')
62
63     # encrypted_data = <{IDi,Ki} encrypted using KUiG> received from USER
64     encrypted_data = bytes.fromhex(encrypted_data_hex)
65     Ni = bytes.fromhex(Ni_hex)
66     Ci = bytes.fromhex(Ci_hex)
67     print("Ni : ", Ni.hex())
68
69     # Decrypt encrypted_data using KUiG
70     cipher = Cipher(algorithms.AES(user_symmetric_key), modes.ECB(), backend=default_backend())
71     decryptor = cipher.decryptor()
72     decrypted_data = decryptor.update(encrypted_data) + decryptor.finalize()
73
74     IDi, Ki = decrypted_data.decode().split(',')
75     print("IDi received : ", IDi)
76

```

Figure 6.3: Gateway code

- The received message is decoded into a string and stored in ‘message_from_user_1’.

2. Extracting Components from the Message: - The received message is assumed to be in a specific format where it contains encrypted data, a nonce (Ni), and some additional authentication data (Ci). - The message is split into components based on a delimiter (‘,’ in this case) using ‘split(’,)’’. - The extracted components, ‘encrypted_data_hex’, ‘Ni_hex’, and ‘Ci_hex’, are assigned to respective variables.

3. Converting Hexadecimal Strings: - The hexadecimal strings ‘encrypted_data_hex’, ‘Ni_hex’, and ‘Ci_hex’ are converted into bytes objects (‘encrypted_data’, ‘Ni’, and ‘Ci’) using ‘bytes.fromhex()’ method.

4. Decrypting the Encrypted Data: - The encrypted data (‘encrypted_data’) is decrypted using the symmetric key (‘user_symmetric_key’) with the AES algorithm and ECB mode. - A cipher object is created using ‘Cipher’ from the ‘cryptography.hazmat.primitives.ciphers’ module. - A decryptor object is created using ‘decryptor = cipher.decryptor()’. - The encrypted data is

decrypted using `'decrypted_data = decryptor.update(encrypted_data) + decryptor.finalize()'`.

5. Extracting IDi and Ki: - The decrypted data is assumed to be in a specific format where it contains IDi and Ki separated by a delimiter (',' in this case). - The decrypted data is decoded into a string and split based on the delimiter. - The extracted IDi and Ki are assigned to respective variables.

6. Printing Received IDi: - The received IDi is printed to the console using `'print("IDi received: ", IDi)'`.

This code segment essentially handles the decryption of a message received from a user, extracting relevant information such as the user's identity (IDi) and a key (Ki), assuming they were encrypted with a symmetric key (`user_symmetric_key`). It demonstrates a typical pattern in server-side processing of encrypted user messages in a network communication scenario.

6.4 Output

The session key equality between the user and device sides ensures secure and synchronized communication, enabling both parties to encrypt and decrypt messages using a shared secret key. This mutual agreement on the session key enhances data confidentiality and integrity during the communication process, contributing to overall system security.

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Nj Received: 0d3c35659f377415d94aa2b313f3cef2
R received: b74c719e54420aa90f404e27f69bdf0a
Received M: 2481d23202e5ac372f87586d9634646e
Received PU: a8a4a7
Received PD: b7b5b4
Received NGj: 0e8e243220d14d5f80b719b673020e88
Received Cj: 20f06b8e73caabac9cb883a5c95bcb0015ece00869469f1d14562d10db827e1c
Cj XOR SGj = 327222e9f88c8d22a38d9ddf28af8f9c73bb439c7399ddc37ec488bb10afc37
hash_value = 327222e9f88c8d22a38d9ddf28af8f9c73bb439c7399ddc37ec488bb10afc37
Integrity of the message verified.
Reconstructed Secret = Sj : b'%\x04\x92\xcf\x16\x8cM\x1c~j<\xf4\xf7\xa2g\xf2\xea\xe0%mh\x05\x01\x8e\xde6\x02\t}\x83d\xd2'
Vi = 2481d2
NG = 1f1113
Vj = cc2a279a8220f11d28357c5764207392e5af261e24f5aa6efd47b4eae749d3de
Session key (KS) on the device side: 36d6220655432da3d7961c271d5e12dcd9171ccd6ead33aba87ea0b8efc630c0
PS C:\Users\shoai\Desktop\Secure Authentication>

```

Figure 6.4: output device

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Received NGi: 1a7a244beddd9f20444dc0215dc7e6d9
Received Ci: b508bddd9f20444dc0215dc7e6d9
Integrity of the message verified.
Ci XOR SGi = 78a1a8d81e84024ed0c1fe43ff2b2ede01abaac8959b8ec6c36664766fd8b9b7
hash_value = 78a1a8d81e84024ed0c1fe43ff2b2ede01abaac8959b8ec6c36664766fd8b9b7

SGi : b'\xcd\xa9\x15c\xac\x8r\\LR\x11+] \x01\xf9$z\t\x88\x08x21]\xf0c\xce\xfb:\x14\xf3\xe8\x10'
Vi = cc2a279a8220f11d28357c5764207392e5af261e24f5aa6efd47b4eae749d3de
NG = 1f1113
Vi = 2481d2
Session key (KS) on the user side: 36d6220655432da3d7961c271d5e12dcd9171ccd6ead33aba87ea0b8efc630c0
Since, Session Key on user side is equal to Session key on device side
HENCE, SESSION KEY VERIFIED -> AUTHENTICATION SUCCESSFUL
PS C:\Users\shoai\Desktop\Secure Authentication>

```

Figure 6.5: output user

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\shoai\Desktop\Secure Authentication> & C:\Users\shoai\AppData\Local\Microsoft\WindowsApps\python3.10.exe "c:/Users/shoai/Desktop/Secure Authentication/gateway.py"
Gateway is listening for connections...
KUi : 8273880ace320fc56449745b10ef529bfe0baa86f05cf3a85d5dd0767f0647cf2baf30829e6346e0e5a0cf5a6f431a9c78b4e0e7f90eb421286bf3419889fa0
KUiG - Symmetric Key: b'\x1bb\x3j\x3\x2-\x83V\x9c5\xef_\xc5\x7\xef'
message from user_1 -> {IDi, Ki} encrypted by KUiG, Ni, Ci : <1855d74b47792f28d80c2cfff46d50b5d,40419e9d51eb749b11a10d8b314429e3,6f7c3ed131b971c
c0303d64a4e0bad832f51222c8bb486adb206c00bfdeb936f>
Ni : 40419e9d51eb749b11a10d8b314429e3
IDi received : xyz
TDi : 95680bd6b5a535303772077f5a7939a972af79ca4d9559c5f5feaf0bc15e1a
IDi: xyz
hash(Si + IDi): d54a6e16cf15e34b3f226845352ca1a8ec46bb23e06634be7f68a14322e1b9dd
SGi : 0x1c8d49e1364bb010d746d69fa59bb8d4e
sent PU1 = 3946c92252541795aab5bbf5cf9fb7f
sent PU2 = 94709cf7ebbad4f011a6d5b9f7a5c2ac
sent NGi = 1a7a244beddd9f20444dc0215dc7e6d9

```

Figure 6.6: output gateway

6.5 Formal Security Analysis

6.5.1 AVISPA

V. FORMAL SECURITY ANALYSIS USING AVISPA This section presents the formal security analysis using AVISPA and analysis of various security features that are essential to cryptographic protocols.

A. FORMAL SECURITY ANALYSIS USING AVISPA We use the SPAN+AVISPA (Security Protocol Animator for Automated Validation of Internet Security Protocols and Applications) tool for performing the formal analysis of the proposed protocol [50] [51]. Experimental results on many internet security protocols show that the AVISPA tool is state of the art for automated validation of security protocols. AVISPA uses a High-Level Protocol Specification Language (HLPSL) to represent the cryptographic protocols. The HLPSL2IF translator translates the protocol to Intermediate Format (IF) specifications. IF specification is then provided to the back-end modules for analysis. There are mainly four back-ends in the AVISPA tool: which include OFMC, an On-the-fly Model-Checker which detects all known attacks, CL-AtSe, a Constraint-Logic-based Attack Searcher, SATMC, an SAT-based Model-Checker and TA4SP, a Tree Automata based on Automatic Approximations for the Analysis of Security Protocols. AVISPA analyze the protocol under the assumption that the network is under the control of the Dolev-Yao intruder over which is the exchange of messages happens. We translate both the registration and authentication and key establishment phases of the proposed protocol to HLPSL. The actions of each entity are represented as basic roles. Further, these basic roles are combined to represent the composed role, representing the interactions among them. The entities are represented as user U , gateway G and the smart device D . The entities communicate us-

ing two different channels: SND and RCV. Finally, an environment role is defined as shown in Fig. 5, which contains global constants and composition of one or more sessions. Besides, it describes the intruder i who plays the role of a legitimate user. The intruder knowledge is specified in the environment session. Finally, the CL AtSe and OFMC back-ends found the protocol SAFE; in other words, the proposed protocol is secure against the Dolev-Yao threat model used in AVISPA. The implementation also includes the simulation of the intruder attack with the publicly known parameters. The intruder gains no knowledge after capturing the message sent by the user. This shows that the proposed protocol is secure against Man-in-the-middle attacks and replay attacks. In other words, the security goals are satisfied by the proposed protocol as specified in the environment.

6.5.2 Analysis of Security Features

Anonymity: User U_i , gateway G , and device D_j have temporal identities to preserve the anonymity of the communicating entities. They use their public keys for computing the temporal identities. The user, gateway, and device compute the temporal identities as $T_{Di} = h(ID_i ||| KU_i ||| Ni)$, $T_{DG} = h(ID_i ||| KG ||| NG)$, and $T_{Dj} = h(ID_j ||| Kj ||| Nj)$ respectively. Hence, adversary A who is eavesdropping on the channel will not be able to identify the communicating entities, thus preserving the privacy of all communicating entities.

Key Freshness: Key freshness is of paramount importance to a key establishment protocol that ensures that each session's key is randomly generated. In the proposed protocol, to generate a session key both U_i and D_j compute the secrets to obtain $h(S_i ||| ID_i)$ and $h(S_j ||| ID_j)$ respectively. Finally, they compute the session key as $Ks = h(h(S_i ||| ID_i) ||| NG ||| h(S_j ||| ID_j))$ where

NG is a newly generated nonce for each session. Hence, the freshness of the key is ensured by the presence of nonce generated by G, which is a trusted entity.

Forward Secrecy: Forward secrecy ensures that the session keys established are not compromised when the long-term key is compromised. Suppose A steals the share s_j of the device D_j . A will not be able to determine the secret S as it requires the knowledge of each of the shares and hence will not compute the session keys. Therefore, in this proposed protocol, compromise of any long-term key does not compromise the session keys.

Fake Gateway Attacks: In this attack, A adds a fake gateway to steal the credentials, such as stored keys, or hijacks communication between the user and the smart device. The fake gateway won't be able to decrypt the message $\langle T_{Di}, T_{DG}, T_{Dj}, Ni, Ci, \{Ki\}KU_iG \rangle$ as it doesn't own the key KU_iG . Moreover, it won't be able to compute the message $\langle M, PU, NG_i, Ci \rangle$. Further, if the fake gateway replays a previously sent message $\langle M, PU, NG_i, Ci \rangle$, the message will be discarded by the smartphone because of old nonce NG or unmatched Ci value.

Man-In-The-Middle Attacks: In a man-in-the-middle attack (MITM), A intercepts the messages and possibly alters the communications between two entities. Suppose, A relays the message $\langle T_{Di}, T_{DG}, T_{Dj}, Ni, Ci, \{Ki\}KU_iG \rangle$ to gateway G, G will discard the message upon verification of NA and CA. Besides, A will not be able to produce a similar response to force user U_i to compute a key which is known to A. The key is computed as $Ks = h(Vi || NG || Vj)$ where V_i and V_j are neither stored at the user end nor at the device end. Both entities compute the session key using their corresponding shares. Moreover, suppose A manipulates the content of the message $\langle M', PU, NG_i, Ci \rangle$ where M is replaced by M' , the proposed pro-

protocol will be aborted by the U_i when $h(S_i || IDG)$ is not matched. Similarly, A will not succeed in establishing a session key with the device by relaying or modifying the message. Hence, the proposed protocol is resilient to MITM attacks.

Replay Attacks: In a replay attack A interferes by replaying a message or a part of a message that was sent previously in any protocol run. Our proposed protocol detects replay attacks through the verification of nonce and integrity. Suppose, A replays the message $\langle T_{Di}, T_{DG}, T_{Dj}, Ni, Ci, \{Ki\}KUiG \rangle$ which was previously sent by the user with a modified Ni . G will abort the protocol as Ci won't match with the received value. Moreover, the rest of the messages include the nonces in their hash values. Hence, the proposed protocol is resilient to replay attacks.

User Impersonation Attacks: Suppose A impersonates as user U_i . A cannot generate the message $\langle T_{Di}, T_{DG}, T_{Dj}, Ni, Ci, \{Ki\}KUiG \rangle$ as A does not possess the public keys of U_i and G and the pre-shared key $KUiG$. Hence, the proposed protocol is resilient to user impersonation attacks.

Smart Device Impersonation Attacks: Suppose A tries to add a device AD_j to the IoBT network. AD_j will neither be able to compute the session key as it does not possess a share to reconstruct the secret with G nor be able to compute $h(M || PD || NGj || IDj)$ to extract the share of the gateway as it requires the knowledge of ID_j . Hence, the proposed protocol is resilient to smart device impersonation attacks.

Denial of Service Attacks: There are two types of denial-of-service attacks (DoS) mainly connection depletion attack and resource depletion attack. Connection depletion attack can be mitigated using local authentication on the user side. However, it is difficult to mitigate a resource depletion

attack completely. The nonce and hash verification prevent this attack to a great extent. For instance, A can send a sp [label=0)]

Chapter 7

Future Works

- **Performance Optimization:** Investigate methods to optimize the protocol's performance in terms of computation and communication overhead, especially in resource-constrained IoT devices.
- **Scalability Analysis:** Evaluate the protocol's scalability in larger IoT networks with a higher number of devices and gateways. Assess its performance under different network conditions and traffic loads.
- **Usability Studies:** Conduct user studies to evaluate the protocol's usability and user experience, particularly in terms of ease of use, setup, and configuration for both technical and non-technical users.
- **Privacy Enhancements:** Investigate additional privacy-enhancing techniques, such as differential privacy or homomorphic encryption, to further protect user data and sensitive information.
- **Standardization Efforts:** Participate in standardization efforts to contribute the protocol's design and implementation to broader industry standards for IoT security and authentication.

- **Dynamic Environments:** Extend the protocol to handle dynamic IoT environments where devices join and leave the network frequently, ensuring seamless authentication and key management.
- **Machine Learning for Anomaly Detection:** Integrate machine learning techniques to analyze network traffic and device behavior patterns, enabling the detection of anomalies and potential security threats.
- **Multi-Factor Authentication:** Enhance the protocol with additional authentication factors, such as biometrics (e.g., voice recognition, gait analysis) or behavioral biometrics (e.g., typing patterns), to further strengthen security.
- **Fault Tolerance:** Design mechanisms to ensure the protocol's resilience against various faults and failures, such as device malfunctions, network outages, or communication disruptions.

Chapter 8

Conclusion

This project introduces an authentication scheme tailored for the demanding security requirements of the Internet of Battlefield Things (IoBT). The battlefield environment, characterized by its challenges and unpredictability, necessitates authentication solutions that surpass traditional methods such as passwords or physical tokens.

This scheme integrates two key techniques to address these demands effectively. Firstly, fuzzy extractors are utilized to accommodate the potential variability of biometric inputs in adverse battlefield conditions. This technique allows for minor discrepancies in biometric data, ensuring reliable authentication of soldiers even when readings are imperfect due to environmental stressors. Secondly, geometric secret sharing enhances security by distributing secret shares across multiple IoBT devices. This strategy significantly increases resilience against compromise, as an adversary would need to obtain a substantial number of shares to reconstruct the secret.

The combination of these techniques offers a significant advantage within the IoBT context, bolstering data security, streamlining operations, and ensuring adaptability to the dynamic battlefield environment. In conclusion,

this project demonstrates a successful approach for achieving secure, efficient, and battlefield-ready authentication within the IoBT. Furthermore, the insights and methodologies developed in this work hold the potential to guide the creation of secure authentication solutions applicable to other high-risk, dynamic environments.

References

- [1] M. Wazid, A. K. Das, N. Kumar, and J. J. Rodrigues, "Lightweight and privacy-preserving remote user authentication for smart homes," in *IEEE Access*, vol. 10, pp. 138512-138523, 2021, doi: 10.1109/ACCESS.2021.3138664.
- [2] N. Li, F. Guo, Y. Mu, W. Susilo and S. Nepal, "Fuzzy Extractors for Biometric Identification," 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), Atlanta, GA, USA, 2017, pp. 667-677, doi: 10.1109/ICDCS.2017.107.
- [3] Iqbal, Ummer Tandon, Aditya Gupta, Sonali Yadav, Arvind Neware, Rahul Gelana, Fraol. (2022). A Novel Secure Authentication Protocol for IoT and Cloud Servers. *Wireless Communications and Mobile Computing*. 2022. 1-17. 10.1155/2022/7707543.
- [4] D. Liu et al., "IoBT-OS: Optimizing the Sensing-to-Decision Loop for the Internet of Battlefield Things," 2022 International Conference on Computer Communications and Networks (ICCCN), Honolulu, HI, USA, 2022, pp. 1-10, doi: 10.1109/ICCCN54977.2022.9868920.
- [5] K. Nimmy, S. Sankaran, K. Achuthan and P. Calyam, "Lightweight and Privacy-Preserving Remote User Authentication for Smart Homes,"

- in IEEE Access, vol. 10, pp. 176-190, 2022, doi: 10.1109/ACCESS.2021.3137175.
- [6] Y. Glouche, T. Genet, and E. Houssay, “SPAN: A security protocol animator for AVISPA—User manual,” IRISA/Univ. Rennes 1, Tech. Rep., 2006.
- [7] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuéllar, and P. H. Drielsma, “The AVISPA tool for the automated validation of internet security protocols and applications,” in Proc. Int. Conf. Comput. Aided Verification. Springer, 2005, pp. 281–285.
- [8] S. Challa, A. K. Das, V. Odelu, N. Kumar, S. Kumari, M. K. Khan, and A. V. Vasilakos, “An efficient ECC-based provably secure three-factor user authentication and key agreement protocol for wireless healthcare sensor networks,” Comput. Elect. Eng., vol. 69, pp. 534–554, Jul. 2018.
- [9] Rahamathullah, U., Karthikeyan, E. A lightweight trust-based system to ensure security on the Internet of Battlefield Things (IoBT) environment. Int J Syst Assur Eng Manag (2021). <https://doi.org/10.1007/s13198-021-01250-4>

Gateway.py

```
import socket
import threading
import ecdsa
import hashlib
import secrets

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend

curve = ecdsa.NIST256p
message_to_device_1 = "mmm"
IDi = "abc"
IDg = "def"
Ki = "xyz"

def generate_ecc_keys_bytes():
    # Generate private key
    private_key = ecdsa.SigningKey.generate(curve=curve)

    # Generate public key from private key
    public_key = private_key.verifying_key

    # Convert keys to bytes
    private_key_bytes = private_key.to_string()
    public_key_bytes = public_key.to_string()

    return private_key_bytes, public_key_bytes

# Function to generate symmetric key
def generate_symmetric_key():
    return secrets.token_bytes(16) # Generate a 16-byte (128-bit) random symmetric key
```

```

def handle_user_connection(user_socket):

    try:

        user_private_key, KUi = generate_ecc_keys_bytes()

        gateway_private_key, KG = generate_ecc_keys_bytes()    # KG = GATEWAY PUBLIC
KEY

        # print("KG : ", KG)

        # Generate symmetric key for user-gateway communication
        KUiG = generate_symmetric_key()
        # print("KUiG Type : ", type(KUiG))
        # print("KUiG- Symmetric Key: ", KUiG.hex())

        # Send public key and symmetric key to the user
        user_socket.send(KG)
        user_socket.send(KUi)
        user_socket.send(KUiG)

        # message_from_user_1 = user_socket.recv(1024).decode()
        # Ni, Ci = message_from_user_1.strip('<>').split(',')
        Ni = user_socket.recv(16)
        Ci = user_socket.recv(32).decode()

        IDi_bytes = IDi.encode()
        Ki_bytes = Ki.encode()
        IDg_bytes = IDg.encode()

```

```
Ci_generated = hashlib.md5(IDi_bytes + Ki_bytes + Ni + KUiG).hexdigest()
```

```
print("Ci received : ", Ci)
```

```
print("Ci generated: ", Ci_generated)
```

```
R = secrets.token_bytes(16)
```

```
NGi = secrets.token_bytes(16)
```

```
Si = bytes(a ^ b for a, b in zip(NGi, Ni))
```

```
SGi = (int.from_bytes(Si, 'big') + 2 * int.from_bytes(R, 'big')) % curve.order
```

```
first_hash_input = Si + IDi_bytes
```

```
first_hash_result = hashlib.md5(first_hash_input).digest()
```

```
second_hash_input = first_hash_result + Ni
```

```
M = hashlib.md5(second_hash_input).hexdigest()
```

```
concat_IDi_KUi_Ni = IDi_bytes + KUi + Ni
```

```
TDi = hashlib.md5(concat_IDi_KUi_Ni).hexdigest() #STRING TYPE
```

```
PU1 = bytes(a ^ b ^ c for a, b, c in zip(R, TDi.encode(), KUiG)) # PU1  
encrypting R
```

```
PU2 = bytes(a ^ b ^ c for a, b, c in zip(NGi, TDi.encode(), KUiG))
```

```
NG_2 = secrets.token_bytes(16)
```

```
concat_IDg_KG_NG_2 = IDg_bytes + KG + NG_2
```

```
TDg = hashlib.md5(concat_IDg_KG_NG_2).hexdigest()
```

```

Cg = hashlib.md5(TDg.encode() + M.encode() + PU1 + PU2 + NG_2).hexdigest()

    # Store data in GatewayStorage.txt
with open('GatewayStorage.txt', 'a') as f:
    f.write(f'<{IDi},{hashlib.md5(Si + IDi_bytes).hexdigest()},{SGi}>\n')

print("\n\nM : ", M)
print("PU1 : ", PU1)
print("PU2: ", PU2)
print("NG_2: ", NG_2)
print("Cg : ", Cg)
print("KUiG : ", KUiG)
print("\nR generated gateway side: ", R)

print("\nTDi gateway side : ", TDi)
print("TDi type : ", type(TDi))
print("TDg : ", TDg)
print("TDg type : ", type(TDg))

NGi_computed = bytes(a ^ b ^ c for a, b, c in zip(PU2, TDi.encode(), KUiG))
print("NGi computed - gateway side: ", NGi_computed)

print("Si gateway: ", Si)
print("Ni gateway: ", Ni)

print("\nPU2 : ", PU2)
print("TDi : ", TDi)
print("KUiG : ", KUiG)
print("NGi : ", NGi)
print("M gateway : ", M)
# user_socket.send(message_to_user_1.encode())
user_socket.send(M.encode())
user_socket.send(PU1)
user_socket.send(PU2)

```

```
user_socket.send(NG_2)
user_socket.send(Cg.encode())
```

```
global message_to_device_1
message_to_device_1 = "zzz"
```

```
except ConnectionResetError:
    print("Connection closed by the remote host.")
finally:
    # Close the connection
    user_socket.close()
```

```
# Function to handle device connections
```

```
def handle_device_connection(device_socket):
```

```
    global message_to_device_1
    print("Sending message to device: ", message_to_device_1)
    device_socket.send(message_to_device_1.encode())
```

```
    device_socket.close()
```

```
user_gateway_address = ('localhost', 12345)
device_gateway_address = ('localhost', 12346)
```

```
# Create sockets for user and device connections
```

```
user_gateway_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
device_gateway_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
# Bind sockets to their respective addresses and ports
```

```
user_gateway_socket.bind(user_gateway_address)
device_gateway_socket.bind(device_gateway_address)
```

```
# Listen for connections
user_gateway_socket.listen(5)
device_gateway_socket.listen(5)

print("Gateway is listening for connections...")

# Function to accept user connections
def accept_user_connections():
    while True:
        user_socket, user_address = user_gateway_socket.accept()
        user_thread = threading.Thread(target=handle_user_connection,
args=(user_socket,))
        user_thread.start()

# Function to accept device connections
def accept_device_connections():
    while True:
        device_socket, device_address = device_gateway_socket.accept()
        device_thread = threading.Thread(target=handle_device_connection,
args=(device_socket,))
        device_thread.start()

# Start threads for accepting user and device connections
user_accept_thread = threading.Thread(target=accept_user_connections)
device_accept_thread = threading.Thread(target=accept_device_connections)

# Start the threads
user_accept_thread.start()
device_accept_thread.start()
```


User.py

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
import socket
import hashlib
import secrets
```

```
IDi = "abc"
IDg = "def"
IDi_bytes = IDi.encode()
Ni = secrets.token_bytes(16)

print("type of Ni : ", type(Ni))
Ki = "xyz"
```

```
gateway_address = ('localhost', 12345)
user_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
user_socket.connect(gateway_address)
```

```
KG = user_socket.recv(64)
```

```
KUi = user_socket.recv(64).strip()
print("KUi - received : ", KUi.hex())
```

```
KUiG = user_socket.recv(16).strip() # Symmetric key is 16 bytes
print("KUiG type: ", type(KUiG))
print("KUiG - Symmetric Key (received): ", KUiG.hex())
```

```
Ci= hashlib.md5(IDi.encode() + Ki.encode() + Ni + KUiG).hexdigest()
```

```
user_socket.send(Ni)
```

```
user_socket.send(Ci.encode())
```

```
print("IDi : ", IDi)
```

```
print("Ki : ", Ki)
```

```
print("Ni : ", Ni)
```

```
print("Ci : ", Ci)
```

```
M_received = user_socket.recv(32)
```

```
PU1 = user_socket.recv(32)
```

```
PU2 = user_socket.recv(32)
```

```
NG_2 = user_socket.recv(16)
```

```
Cg = user_socket.recv(32)
```

```
print("\n\nM : ", M_received)
```

```
print("PU1 : ", PU1)
```

```
print("PU2: ", PU2)
```

```
print("NG_2: ", NG_2)
```

```
print("Cg : ", Cg)
```

```
print("KUiG : ", KUiG)
```

```
concat_IDi_KUi_Ni = IDi_bytes + KUi + Ni
```

```
TDi = hashlib.md5(concat_IDi_KUi_Ni).hexdigest() #STRING TYPE
```

```
print("\nTDi user side : ", TDi )
```

```
print("TDi type: ", type(TDi))
```

```
concat_IDg_KG_NG_2 = IDg.encode() + KG + NG_2
```

```

TDg = hashlib.md5(concat_IDg_KG_NG_2).hexdigest()
print("TDg : ", TDg)
print("TDg type : ", type(TDg))

import ecdsa
curve = ecdsa.NIST256p

R_computed = bytes(a ^ b ^ c for a, b, c in zip(PU1, TDi.encode(), KUiG))
NGi_computed = bytes(a ^ b ^ c for a, b, c in zip(PU2, TDi.encode(), KUiG))

Si = bytes(a ^ b for a, b in zip(NGi_computed, Ni))

si = (int.from_bytes(Si, 'big') + int.from_bytes(R_computed, 'big')) % curve.order

print("\nPU2 : ", PU2)
print("TDi : ", TDi)
print("KUiG : ", KUiG)
print("\nR computed : ", R_computed)
print("NGi_computed: ", NGi_computed)

print("Ni user: ", Ni)

first_hash_input = Si + IDi_bytes
first_hash_result = hashlib.md5(first_hash_input).digest()
second_hash_input = first_hash_result + Ni
M_generated = hashlib.md5(second_hash_input).hexdigest()
print("M generated : ", M_generated)

if M_received == M_generated.encode():
    print("\nM received matches M generated.")
else:
    print("\nM received doesn't match M generated.")

# Store data in UserStorage.txt
with open('UserStorage.txt', 'a') as f:

```

```
f.write(f'<{IDi},{IDg},{hashlib.md5(Si + IDg.encode()).hexdigest()},{si}>\n')
```

Device.py

```
import socket
import ecdsa
import hashlib
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend

gateway_address = ('localhost', 12346)
device_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
device_socket.connect(gateway_address)

message_from_gateway_1 = device_socket.recv(1024).decode()
print("message from gateway 1 : ",message_from_gateway_1)

device_socket.close()
```

AUTHENTICATION

gateway_auth.py

```
import socket
import threading
import ecdsa
import hashlib
import secrets

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend

curve = ecdsa.NIST256p
message_to_device_1 = "mmm"
IDi = "abc"
IDg = "def"
IDj = "ghi"

Si = "secret of user"
Sj = "secret of device"

Ki = "xyz"

M = "mmm"
PD = "pd"
NGj = "ngj"
```

```
Cj = 'cj'
```

```
R = "r"
```

```
def generate_ecc_keys_bytes():  
    # Generate private key  
    private_key = ecdsa.SigningKey.generate(curve=curve)  
  
    # Generate public key from private key  
    public_key = private_key.verifying_key  
  
    # Convert keys to bytes  
    private_key_bytes = private_key.to_string()  
    public_key_bytes = public_key.to_string()  
  
    return private_key_bytes, public_key_bytes
```

```
# Function to generate symmetric key
```

```
def generate_symmetric_key():  
    return secrets.token_bytes(16)
```

```
def handle_user_connection(user_socket):
```

```
    try:
```

```
        TDi = user_socket.recv(32)  
        TDg = user_socket.recv(32)  
        TDj = user_socket.recv(32)  
        Ni = user_socket.recv(16)  
        Ci = user_socket.recv(32)  
        Ki = user_socket.recv(32)  
        KUiG = user_socket.recv(16)
```

```
        print("TDi recv: ", TDi)  
        print("TDg recv: ", TDg)  
        print("TDj recv: ", TDj)
```

```
print("Ni recv: ", Ni.hex())
print("Ci recv: ", Ci)
print("Ki recv: ", Ki)
print("KUiG recv: ", KUiG.hex())
```

```
from random import randint
p = curve.order
global R
```

```
SGi = (int.from_bytes(Si.encode(), 'big') + 2 * int.from_bytes(R, 'big')) %
curve.order
```

```
si = (int.from_bytes(Si.encode(), 'big') + int.from_bytes(R, 'big')) %
curve.order
```

```
SGj = (int.from_bytes(Sj.encode(), 'big') + 2 * int.from_bytes(R, 'big')) %
curve.order
```

```
sj = (int.from_bytes(Sj.encode(), 'big') + int.from_bytes(R, 'big')) %
curve.order
```

```
testing_Secret_generate_Si = ((2 * si) - SGi) % p
```

```
print("R : ", R)
print("p : ", p)
print("SGi : ", SGi)
print("share(si) : ", si)
print("Secret(Si) : ", int.from_bytes(Si.encode(), 'big'))
print("testing secret generation: ", testing_Secret_generate_Si)
```

```
NG = secrets.token_bytes(16)
global NGj
```

```
NGj = secrets.token_bytes(16)
```

```
NGi = secrets.token_bytes(16)
```

```
MU = hashlib.md5(Si.encode() + IDi.encode()).hexdigest()
```

```
MD = hashlib.md5(Sj.encode() + IDj.encode()).hexdigest()
```

```
global M
```

```
M = bytes(a ^ b ^ c for a, b, c in zip(MU.encode(), MD.encode(), NG))
```

```
# PU = bytes(a ^ b ^ c for a, b, c in zip(NG, SGi.encode(), IDi.encode()))
```

```
PU = bytes(a ^ b ^ c for a, b, c in zip(NG, SGi.to_bytes((SGi.bit_length() + 7) // 8, byteorder='big'), IDi.encode()))
```

```
global PD
```

```
PD = bytes(a ^ b ^ c for a, b, c in zip(NG, SGj.to_bytes((SGj.bit_length() + 7) // 8, byteorder='big'), IDj.encode()))
```

```
hash_device = hashlib.md5(M + PD + NGj + IDj.encode()).hexdigest()
```

```
global Cj
```

```
Cj = bytes(a ^ b for a, b in zip(hash_device.encode(), SGj.to_bytes((SGj.bit_length() + 7) // 8, byteorder='big')))
```

```
hash_user = hashlib.md5(M + PU + NGi + IDi.encode()).hexdigest()
```

```
Ci = bytes(a ^ b for a, b in zip(hash_user.encode(), SGi.to_bytes((SGi.bit_length() + 7) // 8, byteorder='big')))
```

```
SGi_computed = bytes(a ^ b for a, b in zip(Ci, hash_user.encode()))
```

```
Ci_XOR_SGi = bytes(a ^ b for a, b in zip(Ci, SGi_computed))
```

```
print("\n-----GATEWAY-----")
```

```
print("M : ", M.hex())
```



```
print("PU : ", PU.hex())
print("NGi : ", NGi.hex())
print("Ci : ", Ci.hex())
print("Hash user: ", hash_user)
print("SGi: ", SGi)
print("SGi_computed (Ci XOR hash_user): ", SGi_computed.hex())
print("Ci_XOR_SGi : ", Ci_XOR_SGi)
```

```
print("PD : ", PD)
```

```
print("Cj : ", Cj)
```

```
print("NGj ", NGj)
```

```
f'<{M.hex()}, {PU.hex()}, {NGi.hex()}, {Ci.hex()}, {R.hex()}>'
```

```
user_socket.send(M)
user_socket.send(PU)
user_socket.send(NGi)
user_socket.send(Ci)
user_socket.send(R)
```

```
user_socket.close()
```

```
except ConnectionResetError:
    print("Connection closed by the remote host.")
finally:
    # Close the connection
    user_socket.close()
```

```
def handle_device_connection(device_socket):
```

```
global M
device_socket.send(M)
global PD
device_socket.send(PD)
global NGj
device_socket.send(NGj)
global Cj
device_socket.send(Cj)
global R
device_socket.send(R)
```

```
print("----- DEVICE HANDLER -----")
print("M : ", M)
```

```
print("PD : ", PD)
print("NGj ", NGj)
```

```
print("Cj : ", Cj)
print("R: ", R)
```

```
device_socket.close()
```

```
user_gateway_address = ('localhost', 12345)
device_gateway_address = ('localhost', 12346)
```

```
# Create sockets for user and device connections
user_gateway_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
device_gateway_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
# Bind sockets to their respective addresses and ports
user_gateway_socket.bind(user_gateway_address)
device_gateway_socket.bind(device_gateway_address)
```

```

# Listen for connections
user_gateway_socket.listen(5)
device_gateway_socket.listen(5)

print("Gateway is listening for connections...")

# Function to accept user connections
def accept_user_connections():
    while True:
        user_socket, user_address = user_gateway_socket.accept()
        user_thread = threading.Thread(target=handle_user_connection,
args=(user_socket,))
        user_thread.start()

# Function to accept device connections
def accept_device_connections():
    while True:
        device_socket, device_address = device_gateway_socket.accept()
        device_thread = threading.Thread(target=handle_device_connection,
args=(device_socket,))
        device_thread.start()

# Start threads for accepting user and device connections
user_accept_thread = threading.Thread(target=accept_user_connections)
device_accept_thread = threading.Thread(target=accept_device_connections)

# Start the threads
user_accept_thread.start()
device_accept_thread.start()

```

user_auth.py

```

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
import socket

```

```
import hashlib
import secrets
import ecdsa
import time

gateway_address = ('localhost', 12345)
user_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
user_socket.connect(gateway_address)


curve = ecdsa.NIST256p


IDi = "abc"
IDg = "def"
IDj = "ghi"
Si = "secret of user"


Ki = "xyz" # KEY generated by FUZZY


def generate_symmetric_key():
    return secrets.token_bytes(16)


def generate_ecc_keys_bytes():
    # Generate private key
    private_key = ecdsa.SigningKey.generate(curve=curve)

    # Generate public key from private key
    public_key = private_key.verifying_key

    # Convert keys to bytes
    private_key_bytes = private_key.to_string()
    public_key_bytes = public_key.to_string()

    return private_key_bytes, public_key_bytes


user_private_key, KUi = generate_ecc_keys_bytes()
```

```
gateway_private_key, KG = generate_ecc_keys_bytes()    # KG = GATEWAY PUBLIC KEY
device_private_key, KDj = generate_ecc_keys_bytes()
```

```
KUiG = secrets.token_bytes(16)
```

```
# NOTE -> PUBLIC KEY GENERATED IS NOT IN THE FORM OF BYTE (ITS IN STRING)
```

```
Ni = secrets.token_bytes(16)
```

```
TDi = hashlib.md5(IDi.encode() + KUi + Ni).hexdigest()
```

```
TDg = hashlib.md5(IDg.encode() + KG + Ni).hexdigest()
```

```
TDj = hashlib.md5(IDj.encode() + KDj + Ni).hexdigest()
```

```
Ci = hashlib.md5(TDi.encode() + TDg.encode() + TDj.encode() + Ni).hexdigest()
```

```
print("TDi : ", TDi)
```

```
print("TDg : ", TDg)
```

```
print("TDj : ", TDj)
```

```
print("Ni : ", Ni.hex())
```

```
print("Ci : ", Ci)
```

```
print("Ki : ", Ki)
```

```
print("KUiG : ", KUiG.hex())
```

```
user_socket.send(TDi.encode())
```

```
user_socket.send(TDg.encode())
```

```
user_socket.send(TDj.encode())
```

```
user_socket.send(Ni)
```

```
user_socket.send(Ci.encode())
```

```
user_socket.send(Ki.encode())
```

```
user_socket.send(KUiG)
```

```
M = user_socket.recv(32)
```

```
PU = user_socket.recv(3)
```

```
NGi = user_socket.recv(32)
```

```
Ci = user_socket.recv(32)
```

```
R = user_socket.recv(32)
```

```

print("\n-----USER-----")
print("M : ", M.hex())
print("PU : ", PU.hex())
print("NGi : ", NGi.hex())
print("Ci : ", Ci.hex())
hash_user = hashlib.md5(M + PU + NGi + IDi.encode()).hexdigest()
SGi = bytes(a ^ b for a, b in zip(Ci, hash_user.encode()))#,
SGi.to_bytes((SGi.bit_length() + 7) // 8, byteorder='big'))
Ci_XOR_SGi = bytes(a ^ b for a, b in zip(Ci, SGi))
print("SGi : ", SGi)

# SECRET GENERATION
si = (int.from_bytes(Si.encode(), 'big') + int.from_bytes(R, 'big')) % curve.order
Secret_generated = ((2 * si) - int.from_bytes(SGi, 'big')) % curve.order
print("Secret Generated: ", Secret_generated)
print("Si : ", int.from_bytes(Si.encode(), 'big'))

if(int.from_bytes(Si.encode(), 'big') == Secret_generated):
    print("SECRET SUCCESSFULLY VERIFIED")
else:
    print("WRONG SECRET")

Vi = hashlib.md5(Si.encode() + IDi.encode()).hexdigest()
NG = bytes(a ^ b ^ c for a, b, c in zip(PU, SGi, IDi.encode()))

Vj = bytes(a ^ b ^ c for a, b, c in zip(M, Vi.encode(), NG))
KS = hashlib.md5(Vi.encode() + NG + Vj).digest()

specific_bytes = Vi[0:3]

# print("Vi encode : ", Vi.encode())
print("Vi = ", Vi)
print("NG = ", NG)
print("Vj = ", Vj)

```

```

print("specific_byte - Vi",specific_bytes)
KS_new = hashlib.md5(specific_bytes.encode() + NG + Vj).digest()
KS_new_str = str(KS_new.hex())
with open("Session_Key.txt", "w") as f:
    f.write(KS_new_str)

print("Session key (KS) on the USER side:", KS_new.hex())

user_socket.close()

```

device_auth.py

```

import socket
import ecdsa
import hashlib
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
import time

gateway_address = ('localhost', 12346)
device_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
device_socket.connect(gateway_address)

curve = ecdsa.NIST256p

IDj = "ghi"
Sj = "secret of device"

M = device_socket.recv(32)
PD = device_socket.recv(3)
NGj = device_socket.recv(32)
Cj = device_socket.recv(32)
R = device_socket.recv(32)

print("M : ", M.hex())
print("PD : ", PD.hex())

```

```

print("NGj : ", NGj.hex())
print("Cj : ", Cj.hex())

hash_device = hashlib.md5(M + PD + NGj + IDj.encode()).hexdigest()
SGj = bytes(a ^ b for a, b in zip(Cj, hash_device.encode()))#,
SGi.to_bytes((SGi.bit_length() + 7) // 8, byteorder='big'))
Cj_XOR_SGj = bytes(a ^ b for a, b in zip(Cj, SGj))

print("SGj : ", SGj)

# SECRET GENERATION
sj = (int.from_bytes(Sj.encode(), 'big') + int.from_bytes(R, 'big')) % curve.order
Secret_generated = ((2 * sj) - int.from_bytes(SGj, 'big')) % curve.order
print("Secret Generated: ", Secret_generated)
print("Si : ", int.from_bytes(Sj.encode(), 'big'))

if(int.from_bytes(Sj.encode(), 'big') == Secret_generated):
    print("SECRET SUCCESSFULLY VERIFIED")
else:
    print("WRONG SECRET")

Vj = hashlib.md5(Sj.encode() + IDj.encode()).hexdigest()
NG = bytes(a ^ b ^ c for a, b, c in zip(PD, SGj, IDj.encode()))

Vi = bytes(a ^ b ^ c for a, b, c in zip(M, Vj.encode(), NG))
KS = hashlib.md5(Vi + NG + Vj.encode()).digest()
print("Vi = ", Vi)
print("NG = ", NG)
print("Vj = ", Vj)
specific_bytes = Vj[0:3]
print("specific_byte - Vj",specific_bytes)
KS_new = hashlib.md5(Vi + NG + specific_bytes.encode()).digest()
print("Session key (KS) on the DEVICE side:", KS_new.hex())

device_socket.close()

```