Zero-Knowledge Authentication

Joshua Mol
FAST
Sheridan College
Oakville, Canada
devr4ndom@gmail.com

x______________________

Hardil Kailey
FAST
Sheridan College
Oakville, Canada
hardil.kailey1@gmail.com

x______________________

***Abstract*— Within the study of security, authentication has received scrutinous attention from researchers. Throughout the existence of technology, authentication has remained the same, requiring users to submit credentials expecting the server to verify their correctness. These methods are outdated, placed in the past, where computers were slow, and communications were synchronous. Computers today possess powerful processing abilities while communicating asynchronously. Our project will focus on adapting an insecure zero-knowledge protocol known as Schnorr's Identification Scheme into a web application through restructuring the execution and responsibilities of both the prover and verifier. This scheme will require an additional third party who is neutral to the transaction of information to ensure communications integrity. Our team has successfully implemented the zero-knowledge protocol allowing for a user to authenticate with a server, disclosing only a single predetermined fictitious identifier. Future implementations of this protocol should use Non-Interactive-Zero-Knowledge Proofs to authenticate.**

I. INTRODUCTION

In modern-day computing, cybersecurity events occur nearly every day. Public relations then describe these attacks as breaches, leaks, and incidents that should not cause any alarm. Users are quietly instructed to change the password they used on the site and told everything afterward is handled with the greatest of care. Users are also informed about HTTPS and the padlock icon in their browser's URL bar; generally, if the user sees the padlock, they are considered safe. Users are then made aware that they should not use public WIFI because they could get their passwords stolen even if all the signs of being safe are present. With all these confusing messages and scary events occurring worldwide, users are more unsure about how security works regarding technology. Average users will often take ridiculous precautions to avoid being exploited and "hacked" and treat the topic of security as a form of magic rather than a science. Mass confusion comes before users learn about the cloud, database security practices, and how to make sense of any of it. It is becoming a requirement to be an expert in the field of information security even to gain a grasp of safe handling of technology. The reality of the matter is that hackers want either attention or money for their efforts in dumping a database or cracking an HTTPS connection. This often makes them turn to publish their findings or selling the information to people who will then empty the victim's bank account and cause many other forms of trouble with the information they purchased. These hacks are usually directed towards services such as Databases and HTTPS connections that are misconfigured or insecure. With hacks becoming commonplace in modern technology, users have developed mistrust in systems that possess personal information. Combating this mistrust, companies have invested billions into the security of their infrastructure, such as; user abnormality detection, password recovery, backup accounts, security questions, Multi-Factor authentication, and various other features to make their services more appealing. However, methodologies used to authenticate are rooted in the old infrastructure of systems and the limited capabilities of such systems. Our project implements a modern approach to handling user authentication, utilizing all resources that have been developed since the development of legacy authentication systems. Zero-Knowledge Proofs are a prime contender in replacing the current authentication model, as it removes the incentive for an attacker to justify making a time commitment in the attack. Utilizing Zero-Knowledge Proofs in authentication would replace information such as; CVV numbers, credit card numbers, and hashed passwords from databases. ZKP will reduce a company's exposure to data storage regulation, but it will also inspire confidence in the users, who are made aware that passwords and personal information are never transmitted or stored on the servers. Zero-Knowledge Proofs are also an ideal candidate for handling zero trust relations. When users come across sites, they may not be familiar with or a site that does not have the same funding as more significant sites, traditionally, this may cause a user to avoid using that particular site. Whereas with Zero-Knowledge Proofs the worry about a site securing users information is eliminated.

A. *Project Objective*

With our project, our team aims to demonstrate that Confidentiality, Integrity, and Accessibility are obtainable through the Zero-Knowledge Protocol ZKP. Our objective is to construct a web store application, bank processing API, and a lightweight client-facing implementation of Schnorr's Identification Scheme, that maintains confidentiality, integrity, and accessibility regardless of a malicious prover or malicious verifier being present.

B. *Problem Statement*

Our implementation of the Zero-Knowledge Protocol and systems will have to maintain proper identity access management within the systems autonomously. Users will have to be able to register an identity, verify they are such claimed identity while also managing malicious entities within the transactions.

II. PRELIMINARIES

A. Acronyms

TA, otherwise known as Trusted Authority, is an elected provider to provide RSA prime numbers in the use of encryptions algorithms.

ZKPs known as Zero-Knowledge Proofs are parameterized sets of information that prove the equivalency of truth rather than the actual truth. E.g., If a person were to select a secret black card from a deck of 52 cards, they could prove they took a black card by disclosing all the red cards they did not take. This equivalency of truth proves that the only cards left would be the 26 black cards, although this did not disclose which black card was selected.

NIZKPs known as Noninteractive Zero-Knowledge Proofs are parameterized sets of information that can be solved and proved by a single individual without a second party present in communications.

CZKP known as Constraint Zero-Knowledge Proofs, is a zero-knowledge proof that proves only one single fact of an object in reference. E.g., Proving the bank account number.

RZKP known as Range Zero-Knowledge Proofs, is zero-knowledge proofs that proof not a fact about an object but rather a range of possibilities about the object in reference. E.g., Proving income range rather than the income amount.

BPZKP known as Bullet Proofs Zero-Knowledge Proofs, is more complex post-processed CZKPs and RZKPs. CRTs are put together as inputs to a function that produces an outcome verified based on the pooled input and output values. This process takes the outputs of both CZKPs, and RZKPs sums them into a total before doing mathematical rearranging operations, to receive the same output sum as the input sum although obscuring the operational values produced by CZKP and RZKP.

MP known as Malicious Prover is a term used to describe an attacker trying to prove something, they know to be false or do not know is right.

MV known as Malicious Verifier is a term used to describe a verifier trying to compromise a provers information.

MITM known as Man-In-The-Middle is an attack that is conducted by a third party who listens to traffic actively or passively and can intercept and change the information sent to derive the secret used in the ZKP protocol.

TTC known as Two-Timing Challenges, is an attacker's method to get the prover with insecure protocol standards to disclose enough information to perform a reverse computation of the secret.

IA known as Impersonation attempts are attempts made by an MP to get verified as a prover who they are not.

DLP known as the Discrete Logarithm Problem, is a problem defined by mathematicians to be sufficiently tricky given large primes to derive the keys.

P is known as polynomial time and is an operation that can be handled within a reasonable amount of time while also being proved in a reasonable amount of time.

NP known as non-polynomial, is a time-complexity descriptor to acknowledge a problem that is difficult to solve but very easy to check for correctness. This problem can have exponential time complexity but have a CRT to prove P time's operation correctness in P time.

CRT known as the certificate is the proof generated from an NP or P problem that is composed of mathematical equivalency of the original computation to verify the validity of an assertion at the request of the verifier.

EXP is a term used in time complexity calculations where an event does not have a CRT. In EXP, an event or operation is of exponential time and takes exponential time to prove to another individual.

SIS known as Schnorr's Identification Scheme is a Zero-knowledge protocol to exchange knowledge of a variable to another party relying on the DLP security.

B. Equations

In ZKPs, there are three mandatory properties, completeness, soundness, non-disclosure. Within these three properties, there are rules that every ZKP needs to follow.

$$\textbf{Completeness:}\quad \forall \boldsymbol{x} \in \boldsymbol{L}, \exists \boldsymbol{\omega}\ |\boldsymbol{\omega}| \leq \mathbb{P}(|\boldsymbol{x}\,|)$$

$$: \boldsymbol{V}(\boldsymbol{x}, \boldsymbol{\omega}) = \textbf{True}$$

$$\textbf{Soundness:}\quad \forall \boldsymbol{x} \notin \boldsymbol{L}, \forall \boldsymbol{\omega}\ |\boldsymbol{\omega}| \leq \mathbb{P}(|\,\boldsymbol{x}|)$$

$$: \boldsymbol{V}(\boldsymbol{x}, \boldsymbol{\omega}) = \textbf{False}$$

$$\textbf{Zero-Knowledge: Non-disclosure of } \boldsymbol{s} \mid \boldsymbol{V}(\boldsymbol{x}, \boldsymbol{\omega})$$

$$\text{Given } \rho\ is\ a\ large\ prime \mid \rho \approx\ 2^{2048},$$
$$q\ is\ a\ large\ prime\ divisor\ p - 1\ |q \approx\ 2^{1023},$$
$$\alpha\ \in\ Z_\rho^* \mid q \mid \alpha^q\ mod(\rho) \equiv 1,\ and\ \alpha^{q/2}\ mod(\rho) \not\equiv 1,$$
$$\tau \mid 2^\tau < q, \kappa\ such\ that\ 0 \leq \kappa \leq q - 1,$$
$$\text{Verifier selects } r \mid 1 \leq r \leq 2^\tau.$$

$$\boldsymbol{Out^{P}_{<P,V>}(\rho,\ q, \alpha, r, s,\ \kappa,\ y, \vartheta, \gamma)}$$

$$\vartheta = \alpha^{-s}\ (mod\ p)$$

$$\vartheta = \alpha^{q-s}\ (mod\ p)$$

$$\vartheta = 70322^{1031-755}\ (mod\ 88667)$$

$$\vartheta = 13136$$

$$\gamma = \alpha^\kappa (mod\ \rho)$$

$$\gamma = 70322^{543}\ (mod\ 88667)$$

$$\gamma = 84109$$

$$y = \kappa + sr\ (mod\ q)$$

$$y = 543 + 755 * 1000\ (mod\ 1031)$$

$$y = 851$$

$$\boldsymbol{Out^{V}_{<P,V>}(\rho,\ q,\tau,\alpha,r,\ y,\ \vartheta,\gamma)}$$

$$\begin{aligned}\alpha^{y}\vartheta^{r} &\equiv \alpha^{k+\mathcal{s}r}\vartheta^{r}(mod\ \rho)\\ &\equiv \alpha^{k+\mathcal{s}r}\alpha^{-\mathcal{s}r}\ (mod\ \rho)\\ &\equiv \alpha^{k}\ (mod\ \rho)\\ &\equiv \gamma\ (mod\ \rho)\end{aligned}$$

…………………………………………………………….

$$r = GenSecureRandom(1,\ 2^{\tau})$$
$$r = 1000$$
$$\gamma \equiv \alpha^{y}\vartheta^{r}\ (mod\ \rho)$$
$$84109 \equiv 70322^{851}13136^{1000}\ (mod\ 88667)$$
$$84109 \equiv 84109\ (mod\ 88667)$$

C. *Common Mistakes*

Insufficient bit length in "r" can allow the attacker to precompute a commitment to a predicted "r" value if two possibilities of challenges isn't enough. Precomputation, by predicting "r". e.g., If τ = 1, then the random challenge will reside within space 2 = {0, 1}; this permits an impersonation with a 50% success rate, as there are only two possible fake commitment that can be selected.

- Not rotating variables for recalculating CRTs with the same agreed public parameters. This can lead to manipulating the captured response, which would derive the private secret using the two sets of captures responses.

$$\boldsymbol{\gamma \equiv \alpha^{y_1}\vartheta^{r_1}\ \equiv \alpha^{y_2}\vartheta^{r_2}(mod\ \rho)}$$
$$\boldsymbol{\alpha^{y_1-y_2} \equiv \vartheta^{r_2-r_1}(mod\ \rho)}$$
$$Since\ \vartheta\ \equiv\ \alpha^{-\mathcal{s}}\ we\ can\ substitute$$
$$\boldsymbol{\alpha^{y_1-y_2} \equiv \alpha^{-\mathcal{s}(r_2-r_1)}(mod\ \rho)}$$
$$Since\ \alpha\ has\ order\ q\ we\ can\ now\ drop\ the\ \alpha\ and\ mod\ by\ q$$
$$\boldsymbol{y_1 - y_2 \equiv \mathcal{s}(r_1 - r_2)(mod\ q)}$$
$$\boldsymbol{\mathcal{s} = (y_1 - y_2)\ (r_1 - r_2)^{-1}(mod\ q)}$$

$$\boldsymbol{\mathcal{s} = (y_1 - y_2)\ (r_1 - r_2)^{-1}(mod\ q)}$$
$$\boldsymbol{\mathcal{s} = (851 - 23)(1000\ - 1003)^{-1}(mod\ 1031)}$$
$$\boldsymbol{\mathcal{s} = 755}$$

III. LITERATURE REVIEW

[1] The article's study relates to our capstone project significantly since the authors are implementing ZKP for their web servers. The authors of the text focus on issues regarding the security of information passed through the web servers. Since the communication channel uses the web servers, it is necessary to keep user information confidential and secure. Developers use passwords and cryptography to hide the messages transmitting through the network to mitigate any attack. However, if an attacker can run a man in the middle attack and successfully gains access to the keys, they can bypass the security and decrypt confidential information. The authors talk about implementing a Zero-Knowledge Proof with other technologies such as Ban Logic and Chaotic Maps technology to prevent such attacks from taking place. It will protect the servers from running into a man in the middle attack and eavesdropping. The implementation of ZKP protocol has roots originating from Schnorr's algorithm of ZKP. The algorithm security is dependent on the discrete logarithms problem where the prime p generator must be a sufficiently large prime for ex p ≥ 2512. The verifier must send out a randomly generated prime number to prove their identity to the server to access the information. Our team is focusing on using a similar approach for the capstone project, and instead of securing server information, our team is developing a web application that will use ZKP to register and allow access to our website.

[2] The authors of the article have found a way to make Zero-Knowledge proofs of membership even more secure since zero bits of information will be revealed. In the Zero-Knowledge proofs of membership, the prover must provide one bit of information to the verifier to authenticate successfully. This method is not an exact zero-knowledge protocol as information shared can be intercepted by the malicious user who can use this one bit of knowledge to falsify the user's identity and gain access to the system. The authors of the article state that the true ZKP will allow the user to prove that he/she knows the secret without having to send out any additional information and prove they know the secret to make secure communication without revealing the secret. This concept is what makes the zero-knowledge proofs very useful. The authors called this scheme the Efficient Identification Scheme, where the prover must convince the verifier that they know the key without revealing it to the other party. This method uses a trust center scheme where a modulo n is published of two sufficiently large prime numbers using the following formula 4r+3. Once the n has been published, the trust center can close, and unlike RSA, where n must change every single time, the application can use the same n value since the factorization of n is not available to anyone. This way, the prover must prove to the verifier that they know the value of n. Our team uses a similar approach where the Trusted Agent generates a prime for the user and uses asymmetric cryptography. The secret is calculated to give the user access to our website once they finish with the registration process.

[3] The authors of the article are not using the same approach as us when using the Zero-knowledge protocol for their web application. The authors of the article create a web application that will store Z-RSA representations of the user credentials in a MySQL database. Z-RSA will prevent any password sniffing or cross-site scripting attacks from gaining access to user passwords. They are using the Fiat, Shamir ZKP algorithm to further develop their work and added RSA's encryption to secure the central server. Since the application security is stored at the central server, it is easier to maintain, and less chance of any attack since Z-RSA is being implemented. One thing I would recommend them doing is to use Amazon Web Service and use their Dynamo DB database. It is a no SQL storage database that will read and write to the application from whichever layer, thus providing more flexibility.

[4] In this article, the authors employ graphs to implement the zero-knowledge protocol is yet another web application. However, rather than using the ZKP to protect the user's password, they use the ZKP protocol to protect username and password pairs. In their implementation, they utilize HTML and JavaScript to generate graph challenges and solve challenges generated by the server to authenticate and validate the timeliness of responses. Since the password never leaves the browser it is much safer than the traditional use of the password; we are accustomed to right now. The private key is generated for the user once they type in the correct password. Similarly, the author suggests that if the user enters the wrong password, it will generate a wrong private key, and thus the challenge graphs sent to the server will be invalid regarding the users' credentials. This will prevent the user from authenticating with the server and accessing the website.

[5] This is yet another article that mentions Zero-Knowledge proof authentication in their web application. The authors of the article want to tackle sending sensitive information such as usernames and passwords over the internet. They have implemented the Zero-Knowledge protocol using isomorphic graphs. Since isomorphism cannot be calculated through any known polynomial-time algorithm, this makes the zero-knowledge protocol implemented in their web application mathematically infeasible to break. The authors use the username and password fields to create two isomorphic graphs to create a permutation using the graphs. This way, the password, and username are kept a secret, but the permutation always remains the same. The authors also used Ajax and XML to create their web app since it does not require web browsers to generate any coprime or multiplicative inverses. Since the use of XML and AJAX does not require any web plugins, hence the claim it will keep the web application safe from any attacks. This method provides a unique approach to handling the Zero-Knowledge protocol requirements, although in this implementation, heavy user computations are required.

[6] Within this paper, the author describes the various types of zero-knowledge protocols and how they differ. The article covers each protocol variant's drawbacks, relating to why we have chosen to move forward with interactive proofs rather than noninteractive. Noninteractive proofs are an excellent way to verify a changing unknown variable, which discourages the incentive to execute a brute force attack as the statistics of resetting variables at a given interval of time. NIZKP makes the statistical probability in the brute force attack computationally infeasible. Whereas within our application, the user's password does not change instead, it remains a constant meaning that if a noninteractive proof were to be employed, the security on such proof would have to be sufficient to guarantee the security of the user password for the lifetime of the proof. The article also points out that many avenues for utilizing zero-knowledge proofs are coming forward, including but not limited to quantum computing variants.

[7] The text describes and focuses on the methodologies of batch processing of ZKPs. ZKPs alone can be very expensive, regarding the time it takes to accomplish the verification process. Due to the time complexity in solving a single ZKP the text describes batching a process in which multiple ZKPs can be processed at the same time complexity as a single ZKP. Having batch processing can reduce server load to manageable levels in this computationally intensive process, although this assumes that all ZKPs are valid. In the case one of many ZKPs is invalid, the batch process returns an invalid response requiring individual computation or resubmission of all ZKPs from the client. This batch processing is a form of bulletproofs, although it only takes in IZKPs rather than a mix of IZKPs and NIZKPs. It can accurately assess and compute according to values for a batch of queued poof verifications and process them in a batch of 5. Although this can be a net negative of the site if an attack were to occur in a login system, one malicious prover within a batch makes the entire batch unusable. Utilization of batch processing means that a Dos attack may be possible on the application's authentication system.

[8] In this article, the authors discuss the growing industry of online gaming and microtransactions. Now more than ever, virtual economies are growing in the number of transactions processed daily, with the author's goal to create a system where the users can make safe microtransactions with the server. With the Zero-Knowledge Proofs, the users of the MMORPG server can make their online transactions without needing to provide their credentials. The paper's authors also use isomorphism to generate graphs using the usernames and passwords received during the payment process. The computations will complete using matrices and natural numbers; thus, the most basic web browsers would process the ZKP. The use of complex systems like elliptic curves is deemed as an alternative and not necessary to the integrity of this protocol.

IV. DIAGRAMS

A. *Network Topology*

Figure 1.0 illustrates our network topology configured within AWS. This diagram provides a visual representation of the site's configuration to route and handles a client's web requests. A novel bank application is employed to represent how a banking authority may implement zero-knowledge into this new infrastructure. Route 53 is a DNS routing service provided by AWS, which has an A record pointing to different CloudFront distributions to handle respective subdomains and private content. CloudFront is a DDOS and traffic management service that can manage connections to an S3 endpoint caching requests in edge locations and reducing load and access times. CloudFront also acts as a security mechanism protecting assets from unauthorized access, signed URL's with predetermined expiries are issued to access the protected endpoint assets securing against unauthorized access. Clients will post a request to API gateway when interacting with the static website hosted from S3, which gets routed to the proper Lambda functions written in python, where only lambda has access to protected endpoints such as bank servers and databases. The protected resources within S3 are by CloudFront's restricted and signed URLs requiring lambda to generate a time-bound Signed URL returning it to the user to access particular resources within S3.

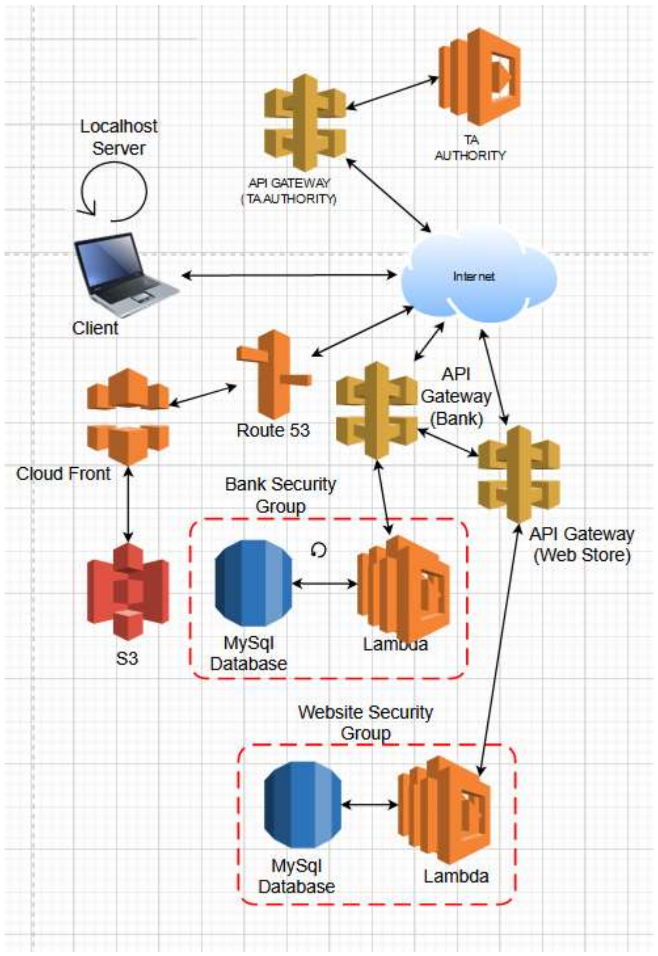


Fig 1.0

B. Database entity diagrams

The chosen values entered into the database are integral to the serverless architecture selected to implement this project. Storing these values in the database are necessary since our architecture is stateless, these values allow the serverless architecture to verify that the commitment sent before the challenge and after the challenge is the same. Failure to implement this properly could allow an attacker to resend an altered commitment and response, thus authenticating the victim user without proper credentials. Values stored in the database do not represent the password, but instead, represent a public declaration of knowing a password. These values carry no representation of the password or way of deriving the password that would be less than the Discrete Log problem's time complexity. These values stored in the database relieve the user from requiring any information regarding the key exchange on their local computer. This design allows the users to use multiple devices to authenticate with the server without worrying about lost or damaged data, which would make an account irrecoverable. Similarly, our web application and our online banking system also follows the same protocol to protect the user's information from jeopardizing.

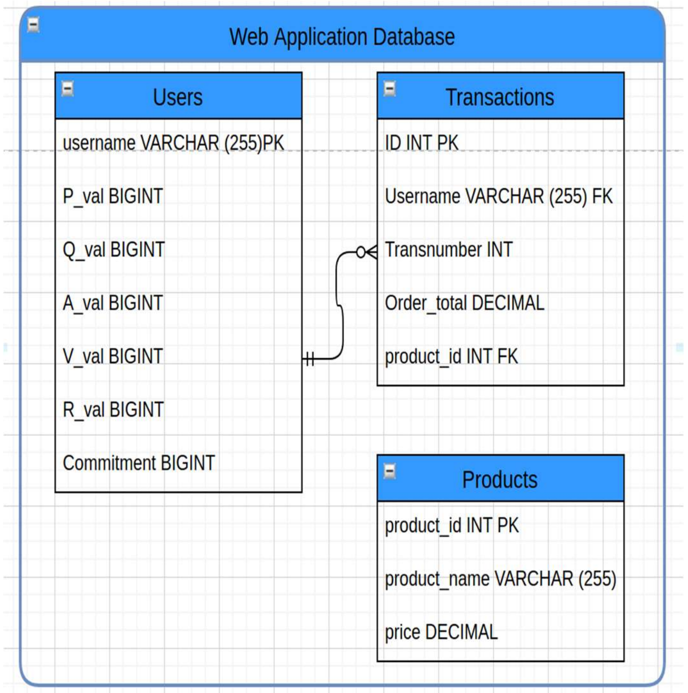


Fig 1.1

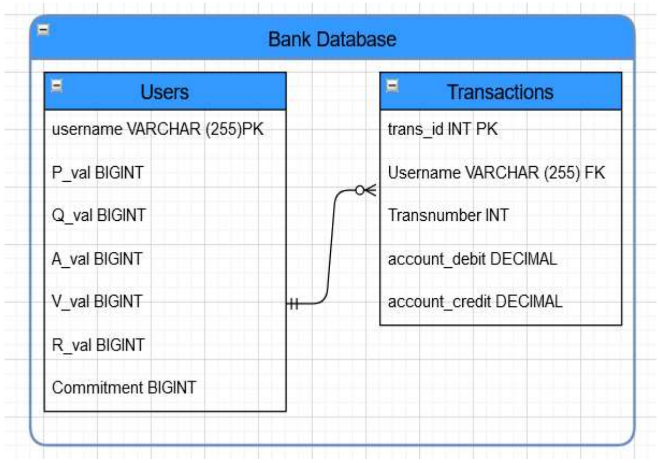


Fig 1.2

C. *Class Diagrams*

Our code regarding the Zero-Knowledge protocol is a lightweight client-facing algorithm, offloading all intensive processing on the server. We have implemented a local server in python due to the limitations and numeric processing by languages used commonly in web application development. This local server is responsible for converting a password string into a numerical number reliably and accurately, which abides by the Zero-Knowledge protocol's criteria. Within the AuthApi python script and the BankApi python script, intensive calculations are not processed on the user's personal device, enabling low powered devices to be compatible with this protocol. Many security features are implemented on the server to mitigate attack strategies and known vulnerabilities of the Zero-Knowledge protocol.

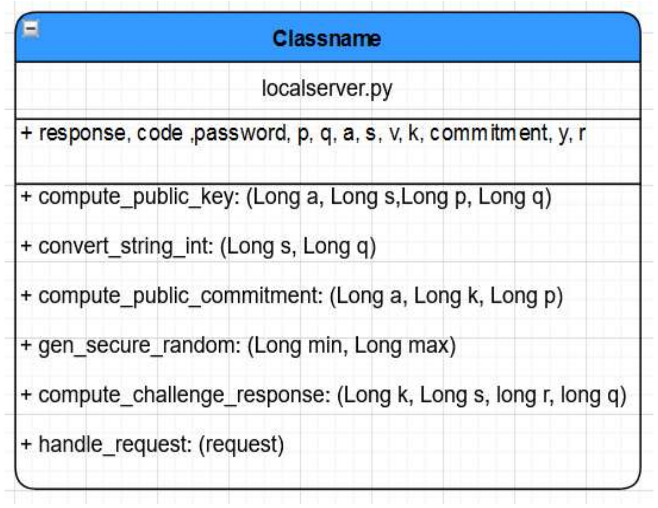

Classname
localserver.py
+ response, code ,password, p, q, a, s, v, k, commitment, y, r
+ compute_public_key: (Long a, Long s,Long p, Long q)
+ convert_string_int: (Long s, Long q)
+ compute_public_commitment: (Long a, Long k, Long p)
+ gen_secure_random: (Long min, Long max)
+ compute_challenge_response: (Long k, Long s, long r, long q)
+ handle_request: (request)

Fig: 1.3

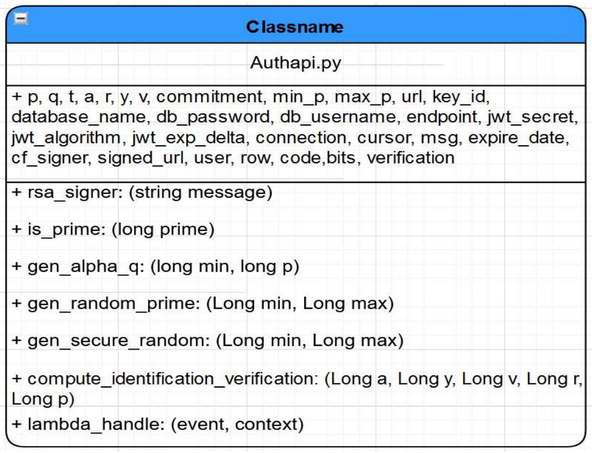

Classname
Authapi.py
+ p, q, t, a, r, y, v, commitment, min_p, max_p, url, key_id, database_name, db_password, db_username, endpoint, jwt_secret, jwt_algorithm, jwt_exp_delta, connection, cursor, msg, expire_date, cf_signer, signed_url, user, row, code,bits, verification
+ rsa_signer: (string message)
+ is_prime: (long prime)
+ gen_alpha_q: (long min, long p)
+ gen_random_prime: (Long min, Long max)
+ gen_secure_random: (Long min, Long max)
+ compute_identification_verification: (Long a, Long y, Long v, Long r, Long p)
+ lambda_handle: (event, context)

Fig 1.4

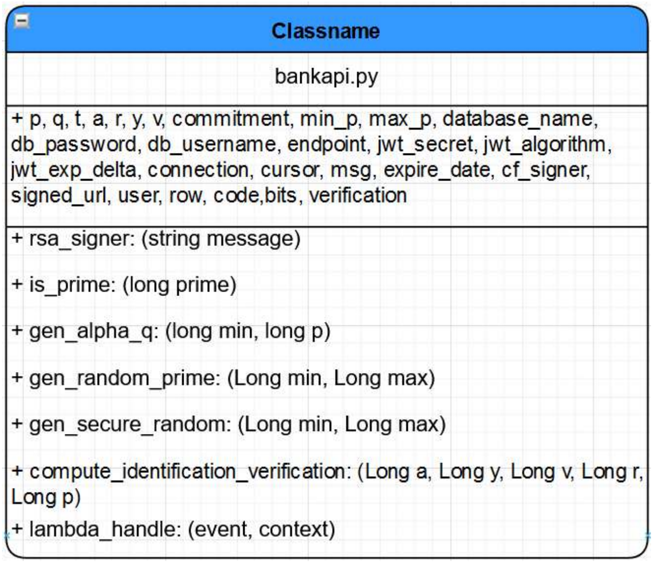

Classname
bankapi.py
+ p, q, t, a, r, y, v, commitment, min_p, max_p, database_name, db_password, db_username, endpoint, jwt_secret, jwt_algorithm, jwt_exp_delta, connection, cursor, msg, expire_date, cf_signer, signed_url, user, row, code,bits, verification
+ rsa_signer: (string message)
+ is_prime: (long prime)
+ gen_alpha_q: (long min, long p)
+ gen_random_prime: (Long min, Long max)
+ gen_secure_random: (Long min, Long max)
+ compute_identification_verification: (Long a, Long y, Long v, Long r, Long p)
+ lambda_handle: (event, context)

Fig 1.5

D. *Flow Chart*

Due to the Zero-knowledge protocol, our website is advantageous to the user since they do not need to confirm their identity. Many websites require a user to confirm their email and further login a second time before accessing the content of the website. In the Zero-Knowledge protocol, a unique username and a password are required for the initial registration and subsequent logins to maintain confidentiality, accountability, and authorization within our systems.

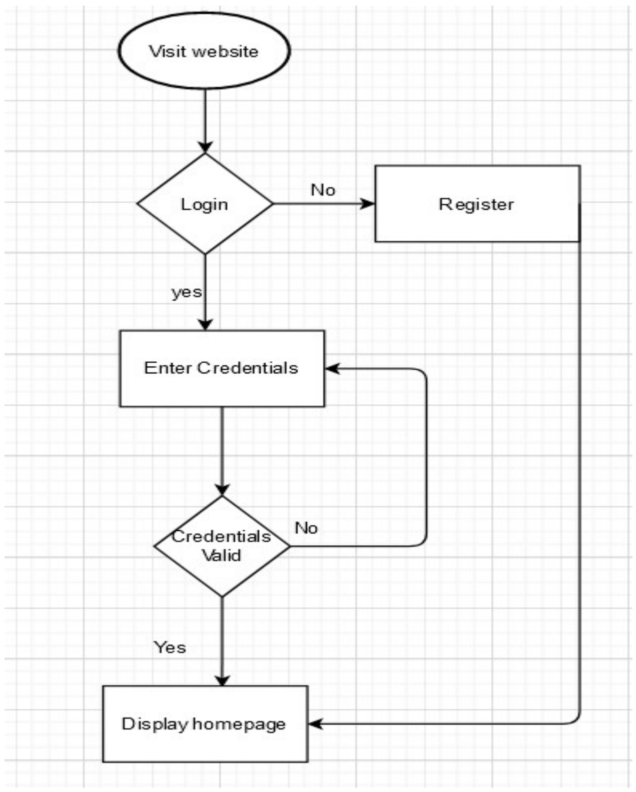


Fig 1.4

V. METHODOLOGY

A. Libraries used

[9] PyMySQL is a package that contains MySQL client libraries that interfaces with the MySQL API based on PEP 249. This allows us to make high-level queries and execute high-level commands within our MySQL database.

[10] Jwt is an open standard in python that meets the RFC7519 that defined a way to securely transmitting information between client and server using a signed JSON object. The JWT token is then verified based on the coding of the JSON object using the secret with the HNAC algorithm or using an RSA key pair.

[11] Rsa Python RSA supports encryption, decryption, signing, and verification as well as key generation according to PKCS#1. We utilized python RSA to sign the CloudFront URLs with generated private keys and verify the validity of the signatures with the public keys once accessed.

[12] Boto is an Amazon web services API package that allows us to interface with the various Amazon web services such as CloudFront, S3, and Route53. We utilized Boto to generate pre-signed URLs for CloudFront to access secured S3 buckets once authenticated.

[13] Boto3 is a software development kit for python that allows developers to write software that communicates between AWS services. This library gives our team the ability to write a lambda function to communicate with the banking entities lambda function to process payments within the web application.

[14] DateTime is a library that allows for the generation of various time formats. The web application utilizes this import to generate signed URLs with a time delta to implement the expiry feature. This import is also used to generate expiries for the JSON web token for accessing restricted lambda functions.

B. Design & Implementation

The decision to make the server-side implementation of the Zero-Knowledge protocol on serverless architecture was made to reduce costs associated with hosting servers limiting run time to on-demand only. The serverless architecture was a good choice for this protocol not only for cost but also show the versatility that Zero-Knowledge offers as it's merely an exchange of preprocessed variables.

Implementing the serverless architecture requires storage, state, and processing, to fulfill the storage requirement S3 is used to host static website content, along with MYSQL RDS to provide state-fullness and handle client data storage. CloudFront was added to provide stateful security, protecting the content from unauthorized access and DOS attacks. CloudFront, along with JWT tokens, is used to grant access only to users who are logged in or registered. Lambda, in combination with API Gateway, is used to fulfill the processing needs of the site during user authentication.

Lastly, Route 53 is used to give the website a DNS domain name allowing for proper SSL and use of user experience. The client is required to run a localhost python server to manage requests and process responses, future integration of this protocol with the browser would eliminate the need to run the server. Protocol execution order is critical to maintaining the security of the protocol; if one variable is sent to early, then the security of all user's accounts can be compromised. The serverless design implementation allows us to execute sandboxed code preventing variable stealing and injection.

C. Development Challenges

Implementing this zero-knowledge application, our team has various problems with math, architecture, and integration with AWS. During the development of this application, our team had to revise our plans of development as AWS doesn't handle serverless applications that require custom implementations of authentication protocols. Instead, they push their own services pre-configured for you to use. One of the first challenges our team faced was the 3 Mb limit of dependency uploads, although AWS allows dependency uploads of 10Mb, locally debugging features are disabled if you exceed 3Mb.

This forced us to use python imports based on their size rather than the ease of use and documentation. AWS also offers a feature of accessing dependencies beyond 10Mb that are stored in S3. This option didn't fit the team's needs as S3 eventual consistency would cause the execution of old code, causing confusion in development. S3 also caused the delay between POST requests and response to drastically increase as the data needed to be retrieved decompressed then executed. This caused the web application login to lag and thought to cause user confusion in production. For the development of cookies and signed URLs, our team had to discover and enable a legacy feature in CloudFront to allow for reduced caching time, previously to replace a document in S3, it would take 24 hours without a request to the domain to clear the cache in CloudFront.

Without this inactivity period, our team couldn't view the modifications to our pages and scripts uploaded to S3. Working with lambda became difficult when generating Signed URLs due to AWS transitioning from their old library "Boto" to their new well-documented library "Boto3" a combination of both these libraries was necessary to complete respective actions. Generating Signed URLs required legacy "Boto" whereas Lambda to Lambda communication required the "client" module from the "Boto3" library. Additionally, to the libraries were roles and security groups within AWS; by default, communication between services is sandboxed and restricted, needing either roles or permissions to communicate with each other. Our team needed to configure custom policies and roles to allow access to the MySQL database, Bank Lambda Functions, S3, and CloudFront distributions from proper origins.

Outside of AWS development issues persisted as the team's Flask python server running on the localhost had CORS issues with the communications to the browser. The Team needed to research how to construct CORS headers and accept incoming and outgoing transitions. Flask also caused issues with its processing of POST requests. Flask processes POST requests as a form submission rather than a JSON such as lambda in AWS.

Within our algorithm for generating primes and values, we experienced Lambda timeouts, and memory exceeded errors; this prompted us to reduce the prime number size to a more manageable numeric value. Prime number generation is the fastest part of the initial variable generation process; our team found that finding a Q, and alpha value to fit the strict criteria of the Zero-Knowledge protocol became exceedingly more intensive as the initial prime grew.

D. Protocol Justification

The largest issues our team faced were in the project planning phase. Our team needed to develop a way to maintain C.I.A of the zero-knowledge protocol between two possible malicious parties, while reducing the computational and storage overhead for the prover. This meant that key exchanges and variable requests needed to give what was necessary to the other party to complete the verification process at a given time, while also preparing for multidevice support and data loss possibilities client side.

Solving this issue lead us to strategically break up the verification process into segments that execute at different times in the login process. When registering the P, Q, Alpha values are transmitted to the client, next the client returns an identifying username and their public key derived from the given P, Q, Alpha values, leaving the commitment un-generated until the login process occurs. When the client requests to login the public key isn't important to the client's future proving calculations, meaning that the public key doesn't need to be retransmitted over the network. This allows for the public key to only get transmitted once during registration minimizing the risk for MITM public key gathering attacks.

Not submitting the commitment at registration allows the user to only require the password during login, were as if the commitment was sent during registration, the client would have to store the commitment and the offset used to generate the commitment in order to login in the future. This causes unnecessary risk as only one device has the secret information to login and the secret information needs to be stored and secured by the user. In our application, secret information only exists and is valid for the moment it's being used to compute responses; at no time does a secret or data reducing the security of the protocol need to be stored. Additionally, variables are rotated after every failed and successful login.

A failed login causes the client to generate new offsets and the server to generate new challenges, while in the case of a successful login all variables are deleted, and a new P, Q, Alpha, public key values are generated in preparation for subsequent logins.

E. Protocol Execution

The web application is built using Python 3.8, MySQL, where the client is required to run a local python server to handle number processing. The client will first request the website's domain which will get routed and monitored by CloudFront. CloudFront will then return a cached version of the static website or a version from S3, depending if the site has been requested within the TTL specified within CloudFront. Client data isn't stored in cached versions of the site as every communication is stateless and dynamically generated as needed. A new user would choose to register for the website's services; in doing so, they will request the register HTML page. Upon loading of the register HTML, an API call is made in the background requesting the P, Q, Alpha values from the TA.

Once a response containing these values is received, the user will be allowed to submit their account credentials providing a unique username and password. The submission in registration is a two-step process, first, the P, Q, Alpha, and password is sent in a POST request to the localhost python server function for processing of a public key. This function converts a string into an integer modulo Q, then computes the public key with these variables. The response is returned to the browser, followed by a second POST request to the AuthApi, which will check the database for repeated usernames. If a duplicate entry isn't found AuthApi inserts the user's information into the database, redirecting the client via signed URL to the restricted landing page of the site. This page cannot be accessed without a signed URL, lambda generates this signed URLs using a 2048-bit RSA private key with CloudFront's CLI access keys. Alternatively, a returning user could request the login page, which will request the user's username.

This username will then be sent via a POST request to the AuthApi to access and return corresponding P, Q, Alpha values for the given username. The public key value is never returned to the user to eliminate Public Key harvesting of users, nor is the public key ever resubmitted. Once variables are sent to the server, they are saved in the database and queried as needed to prevent a malicious user from changing values after making a commitment. Only upon successful login do variables become rotated, with the exception being the commitment and random challenge variables that rotate at every login attempt to prevent two-timing attacks. The webpage modifies itself upon a successful request to prompt the user for a password. Upon submitting the password, a POST request is made to the localhost server containing the P, Q, Alpha values received by the webserver, and these values are then used along with a locally generated random offset to construct the public commitment of identity.

If the server modified P, Q, Alpha values in a malicious manner, no information could be derived from the user's communication, nor could proper authentication occur, thus it's in the best interest of all parties, to be honest in this communication. The commitment of identity is then sent via a POST request to AuthApi along with the corresponding username, AuthApi then generates a random challenge to the incoming commitment and returns the challenge within the response.

The python server then receives this response and generates a Y value that completes the equation reduction, then submits the Y values via POST back to the AuthApi. The AuthApi then checks to see if the public commitment and the equation equal each other; if true, the user is authenticated receiving a signed URL and redirected to the restricted web page. In the case of a wrong password, an error message is shown, and the process of password conversion and challenge responses needs to occur once more. This is an additional benefit as account lockouts can be implemented with ease to prevent brute force attempts.

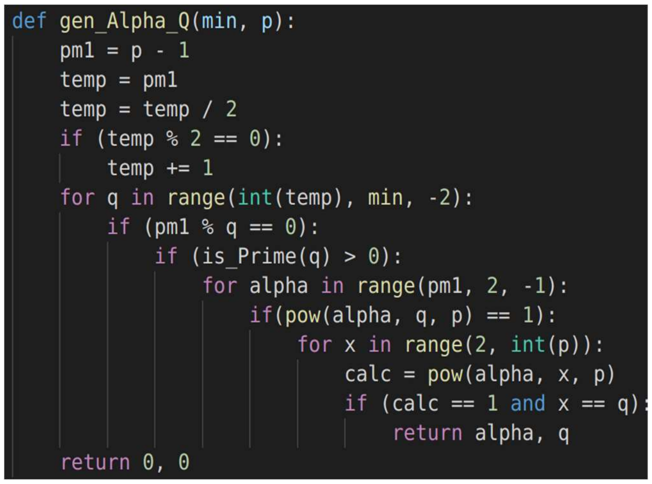
Upon load of the restricted landing page, the user submits a secure JWT token signed with a 2048 character, marked for time expiry, and stored in the cookie storage of the browser to a Lambda function responsible for getting previous transactions. The cookie has protection regarding domain restrictions and secure transport requirements, ensuring that XSS and other methods for harvesting tokens don't occur. This Lambda function only returns information about the user's previous purchase history based on the user inside of the token, although it is still secured with expiring tokens to prevent data leaks and enhance privacy. Users now will be able to place orders with the website to purchase APK files.

A user will select the APK that they wish to buy, then entering their zero-knowledge bank information. Once the information is sent, the same technique for logging into the webstore will occur with the bank to verify the user's identity. The information submitted is the same as regular credit card information except that it also requires the username of the user who registered with the bank instead of the name of the cardholder. This username is not the same as the web store's username, as the bank must support the zero-knowledge protocol as well. A separate registration page and process are used to make an account with the bank. Security isn't compromised if the website observes information transferring between the bank and the user as only public information will be submitted, although in production systems, not exposing public keys unnecessarily is optimal as it increases entropy.

The communication method being through the web store or a side-channel is optional and would depend on implementation. Since this is a fake mockup banking app that doesn't have to the infrastructure of real banks, this implementation is processing payment relayed through the web store to receive a successful payment response from the bank's Lambda function. In production scenarios, a client notifying the web store that a successful side-channel payment has been made while providing the transaction number should trigger the web store to check records of its own to verify that the transaction went through and proceed with the purchase process. After successful completion of payment, the bank and website both update their respective databases, and the website starts the download through a signed URL.

F. Main Methods

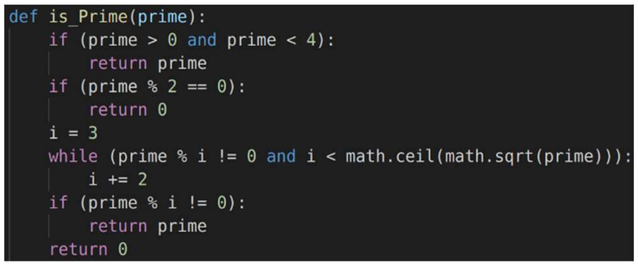
Gen_Alpha_Q is a function that generates and returns an Alpha and Q value for the initialization of the zero-knowledge protocol. This function takes a minimum value and P as parameters where P is the largest prime in the protocol. The optimized time function will first generate the largest Q value that is a prime divisor of P minus 1. Then the function will find an Alpha where alpha is the first number between 2 and P that has the period AlphaQ mod(p). If no Alpha value is found, the second-largest acceptable Q value is chosen to execute the protocol.



```
def gen_Alpha_Q(min, p):
    pm1 = p - 1
    temp = pm1
    temp = temp / 2
    if (temp % 2 == 0):
        temp += 1
    for q in range(int(temp), min, -2):
        if (pm1 % q == 0):
            if (is_Prime(q) > 0):
                for alpha in range(pm1, 2, -1):
                    if(pow(alpha, q, p) == 1):
                        for x in range(2, int(p)):
                            calc = pow(alpha, x, p)
                            if (calc == 1 and x == q):
                                return alpha, q
    return 0, 0
```

Fig: 2.0

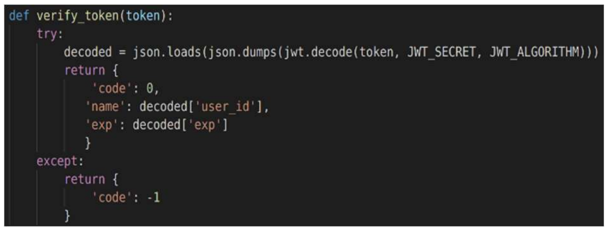
Is_Prime is an optimized time function that determines if a number is truly prime with the time complexity of the O(sqrt(N/2)). Upon determining primality, the prime is returned to the calling function; in the case of failure to determine primality, a 0 is returned.



```
def is_Prime(prime):
    if (prime > 0 and prime < 4):
        return prime
    if (prime % 2 == 0):
        return 0
    i = 3
    while (prime % i != 0 and i < math.ceil(math.sqrt(prime))):
        i += 2
    if (prime % i != 0):
        return prime
    return 0
```

Fig: 2.1

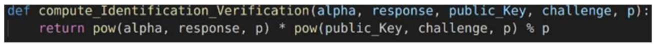
Verify_token is a method that takes a JSON Web Token object as the parameter decodes and returns key attributes of the object. The method first decodes the JWT object; if the token has an invalid expiry or the secret used to create the object is corrupted, then this process will throw an error. Next, the method dumps the JSON object into bytes for python to then load it in a parseable format returning key values.



```
def verify_token(token):
    try:
        decoded = json.loads(json.dumps(jwt.decode(token, JWT_SECRET, JWT_ALGORITHM)))
        return {
            'code': 0,
           'name': decoded['user_id'],
           'exp': decoded['exp']
           }
    except:
        return {
            'code': -1
        }
```

Fig: 2.2

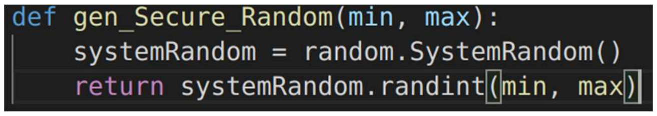
Compute_Identification_Verification is a method that computes the congruency of the user-submitted commitment to the variables given to the server over the zero-knowledge protocol exchange.



```
def compute_Identification_Verification(alpha, response, public_Key, challenge, p):
    return pow(alpha, response, p) * pow(public_Key, challenge, p) % p
```

Fig: 2.3

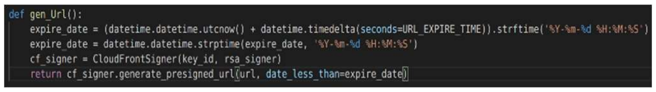
Gen_Secure_Random is a method used to generate random numbers that the program uses for the zero-knowledge primes and offsets; this method is also used to generate challenges in response to the incoming commitments from the client.



```
def gen_Secure_Random(min, max):
    systemRandom = random.SystemRandom()
    return systemRandom.randint(min, max)
```

Fig: 2.4

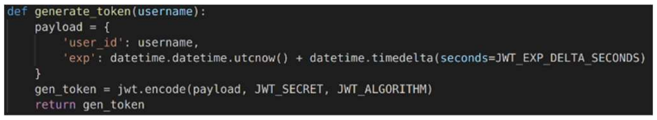
Gen_Url is a method the server uses to generate signed and time-limited URLs for the client to access restricted resources. These URLs are given an expiry and a specific page that can be accessed, this information is then used along with an RSA private key pair to generate a URL that is returned to the client.



```
def gen_Url():
    expire_date = (datetime.datetime.utcnow() + datetime.timedelta(seconds=URL_EXPIRE_TIME)).strftime('%Y-%m-%d %H:%M:%S')
    expire_date = datetime.datetime.strptime(expire_date, '%Y-%m-%d %H:%M:%S')
    cf_signer = CloudFrontSigner(key_id, rsa_signer)
    return cf_signer.generate_presigned_url(url, date_less_than=expire_date)
```

Fig 2.5

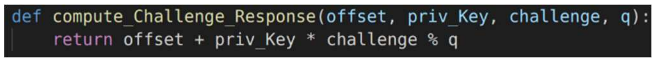
Generate_Token is a method that takes the username of an authenticated user along with a future offset of the current time and date. The method generates a signed token that can't be modified, giving it to the user to access restricted functions that can only be accessed by authenticated users.



```
def generate_token(username):
    payload = {
        'user_id': username,
        'exp': datetime.datetime.utcnow() + datetime.timedelta(seconds=JWT_EXP_DELTA_SECONDS)
    }
    gen_token = jwt.encode(payload, JWT_SECRET, JWT_ALGORITHM)
    return gen_token
```

Fig: 2.6

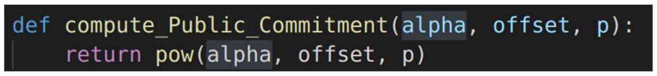
Compute_Challenge_Response is a method that runs on the client's device. This method takes the randomly generated offset, private key, q, and the challenge from the server as parameters. The private key is then multiplied by the challenge, and the random offset is then added modulo q.



```
def compute_Challenge_Response(offset, priv_Key, challenge, q):
    return offset + priv_Key * challenge % q
```

Fig: 2.7

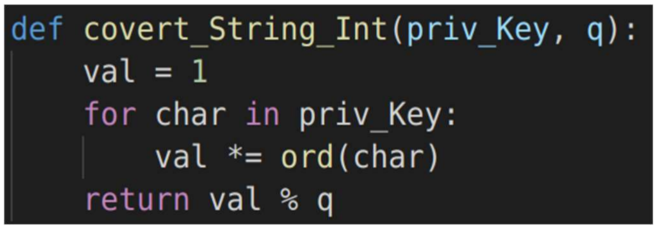
Compute_Public_Commitment is a method that runs on the client's device. This method takes the randomly generated offset, alpha, and p as the parameters. The method then returns alpha to the power of the offset modulo p. This is done to get the client to commit to a random offset to prevent the server from selecting a challenge that will then disclose the secret.



```
def compute_Public_Commitment(alpha, offset, p):
    return pow(alpha, offset, p)
```

Fig: 2.8

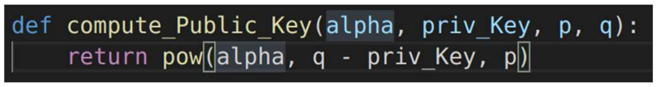
Convert_String_Int is a function that our program uses to convert a string input into a number modulo q; the function multiplies the int values of each of the chars in the password string, returning the value as the secret in the authentication system.



```python
def covert_String_Int(priv_Key, q):
    val = 1
    for char in priv_Key:
        val *= ord(char)
    return val % q
```

Fig: 2.9

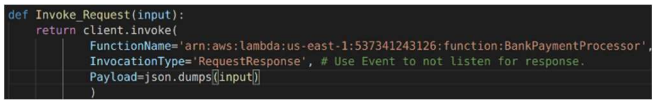
Compute_Public_Key is a function executed by the client to commit to using a set of variables for the duration of the authentication process. This function computes alpha to the inverse of the secret returned from the Convert_String_Int function modulo p. This method doesn't disclose the secret but commits the user to use the same set of variables in future communications.



```python
def compute_Public_Key(alpha, priv_Key, p, q):
    return pow(alpha, q - priv_Key, p)
```

Fig: 2.10

Invoke_Request is a function in our payment processing lambda on the web-store; this function allows the interaction between the web store lambda function and the bank's lambda function.



```python
def Invoke_Request(input):
    return client.invoke(
            FunctionName='arn:aws:lambda:us-east-1:537341243126:function:BankPaymentProcessor',
            InvocationType='RequestResponse', # Use Event to not listen for response.
            Payload=json.dumps(input)
            )
```

Fig: 2.11

VI. CONCLUSION & FUTURE WORKS

Our team has executed an approach to zero-knowledge that is both secure and scalable, while also retaining the benefit of being compatible with all devices regardless of computational power. Through the separation of responsibilities, our implementation has allowed each party to execute the level of security necessary per computation and use case. This use of asymmetric cryptography in zero-knowledge protocol has been acknowledged as being critical to the user's advantage, as smartphones and other low power devices may need to be compatible with such a system. The protocol that our team has executed assumes that both parties are malicious; this unique execution of the zero-knowledge protocol provides versatility and confidence to both parties involved in the communications. Finally, our team has validated the feasibility of our concept through our prototype web application implementation. We plan to continue our research into integrating larger primes and reducing the time complexity of deriving the Q, and Alpha values to increase the timeliness of processing.

REFERENCES

[1] Chain, K., Chang, K.-H., Kuo, W.-C., and Yang, J.-F. (2017) Enhancement authentication protocol using zero-knowledge proofs and chaotic maps, Int. J. Commun. Syst., 30: e2945. doi: 10.1002/dac.2945.

[2] Fiege, U., Fiat, A., & Shamir, A. (1987). Zero knowledge proofs of identity. Proceedings of the Nineteenth Annual ACM Conference on Theory of Computing - STOC 87. doi:10.1145/28395.28419

[3] V. Mainanwal, M. Gupta and S. K. Upadhayay, "Zero Knowledge Protocol with RSA Cryptography Algorithm for Authentication in Web Browser Login System (Z-RSA)," 2015 Fifth International Conference on Communication Systems and Network Technologies, Gwalior, 2015, pp. 776-780, doi: 10.1109/CSNT.2015.90.

[4] Grzonkowski, Slawomir & Corcoran, Peter. (2014). A Practical Zero-Knowledge Proof Protocol for Web Applications. Journal of Information Assurance & Security. 9. 329-343.

[5] Grzonkowski, S., Zaremba, W., Zaremba, M., & Mcdaniel, B. (2008). Extending web applications with a lightweight zero knowledge proof authentication. Proceedings of the 5th International Conference on Soft Computing as Transdisciplinary Science and Technology - CSTST 08. doi:10.1145/1456223.1456241

[6] Wu, H., & Wang, F. (2014). A survey of noninteractive zero knowledge proof system and its applications. The Scientific World Journal, 14.

[7] Peng, K., Boyd, C., & Dawson, E. (2007). Batch zero-knowledge proof and verification and its applications. ACM Transactions on Information and System Security, 10(2), 6. doi:10.1145/1237500.1237502

[8] S. Grzonkowski and P. M. Corcoran, "A secure and efficient micropayment solution for online gaming," 2009 International IEEE Consumer Electronics Society's Games Innovations Conference, London, 2009, pp. 118-125, doi: 10.1109/ICEGIC.2009.5293609.

[9] Y. Matsubara, *PyMySQL*. 2020/

[10] K. Yoshida, *JWT*. 2020.

[11] S. Stuvel, *RSA*. 2020.

[12] M. Garnaat, *Boto*. 2018.

[13] *Boto3*. Amazon Web Services, 2020.

[14] T. Peters, *DateTime*. 2020.