Opal: Oblivious Programmable Access Limiting

Kaminow, Austin

1 Introduction

As security continues to get stronger, the actual users continue to be a problem. In addition to this, there is not much protection for companies once an administrator's password does get stolen. To demonstrate this issue, consider an example where a doctor, Alice, stores important patient information on an AWS server. Consider an attacker that retrieves Alice's password using a phishing scam. The attacker now logs onto Alice's AWS account and has access to all of the important patient documents. A simple suggestion may be to encrypt the files on the AWS machine, but this would include additional passwords that may be stolen in a similar phishing scam. In addition, more passwords would create a much less convenient environment for Alice.

One idea to limit the amount of damage an adversary could do with a user's password is to enforce rate limiting. Rate limiting is typically used by APIs in order to enforce account limits and require users to pay for the API's service. This could, however, be applied to Alice in the above example as well. In this case, Alice would be permitted to set her own access controls. Let's assume Alice only visits with three patients per day. She could then set the access controls to only allow access to three patients' documents per day. If an adversary were to now steal Alice's credentials using a phishing scam, the adversary would be limited to what he could retrieve. While he could still view documents from three patients, he would not be able to exfiltrate all of Alice's patient documents. Lots of work has been done in the field of cybersecurity to prevent attacks from happening in the first place. In addition, researchers have been finding ways to automate the process of finding security issues after they have happened and automatically correcting the issue. Access control and rate limiting is an intermediary step with significantly less work performed on it. Rate limiting limits the damage that an attacker can do once they have already gotten onto a system, but before the automated processes are able to identify and banish the attacker. While the idea mentioned above could certainly limit the damage in cases such as the one described above, it is quite easy to implement. First, Alice will protect all of her patient documents with a password unique to the patient. She can then send these passwords to a server that implements rate limiting. The server will have a counter associated with Alice's account, and once she queries the server more than three times, it won't return any more passwords to Alice. While this technique would certainly solve the initial problem, it might in fact create new problems. Namely, the server will now have information about Alice. It will even continue to learn more information about her as she queries it more times.

1.1 Our Contributions

In this paper, we present Opal, a programmable OPRF to allow for arbitrary access controls and rate limiting. This work has two major contributions:

Programmable OPRF: Opal combines the use of oblivious PRFs (OPRFs) with zero-knowledge proofs (ZK proofs). An OPRF on its own would not be able to impose any restrictions on the user since all of the inputs would be completely oblivious to the server. On the other hand, with a regular PRF, the server would learn the inputs of the user, and the user's privacy would be breached. By combining the OPRF with a ZK proof, Opal is able to ensure that the user has the proper authorization to receive a key without learning anything about the user's input.

Arbitrary access controls for multi-user systems: Opal is able to enforce access controls even for a system with many users on it at once. In addition to this, each user can even have different access controls from each other. This allows many users who have different needs to use the system at once for their own purposes.

2 Related Work

In our setting, we will rely heavily on PRFs, but unfortunately the typical PRF setting will not be sufficient for this work. In order to generate the decryption key for specific objects in our access control scheme, the user and server must jointly generate the key. While the typical PRF setting could allow for the joint creation of the input, one party would have to actually calculate the PRF call on that input. This would require that that party knows the input from the user and the server. This would be problematic, as either the server would learn which object the user is querying or the user would learn the server's private key. In order to fix this issue, we will use oblivious pseudorandom functions (OPRFs).

An OPRF allows for the user and server to both generate their inputs privately and for only one party, the user in this case, to receive the output. Specifically, we will be using the Hashed Diffie Hellman (HashDH) OPRF [2]. This scheme uses blinded signatures which were first introduced by Chaum [3, 4] and used later in schemes like Privacy Pass [5]. To create this OPRF call, the user will first hash her

input. She then blinds this by raising the output to a random value. We can then assume the Diffie-Hellman assumption which says the server will not be able to unblind the output. The server will then raise the blinded hash output to the power of its secret key and return that value to the user. Finally, the user will unblind the output for the final key.

While using OPRFs is necessary in order to hide the inputs from the server, this does cause one additional problem. The server now has no way to verify that the access control restrictions are satisfied. There is one solution to this problem which is called Pythia [6]. Pythia is a partial oblivious PRF. This, however, as the name suggests, does reveal part of the input. This is certainly an improvement as compared to a traditional PRF, but it still reveals some information about the inputs. Our construction, on the other hand, does not reveal any information about the OPRF input. The user will have to reveal some information such as a random nonce and serial number, but no actual information about the OPRF call.

In order to allow for the server to learn only whether these limits are satisfied, we use zero knowledge proofs, specifically we use zk-SANRKs. By using zk-SNARKs, we are able to allow the user to create a proof that her inputs satisfy all of the access control constraints without revealing what any of her input values are.

3 Security Guarantees

Pseudorandomness: The first security guarantee is pseudorandomness. This means that if the user and server generate a key using the OPRF, the output should be computationally indistinguishable from random. In other words, if there is an adversary who receives an output from the OPRF and a uniformly random number, the adversary could not which was the OPRF output.

Obliviousness: The second guarantee is obliviousness. This guarantee works in both directions. We guarantee that in the evaluation of the OPRF, the server will not learn anything about the input of the user and will not learn anything about the output. In this scheme, we of course allow the server to learn whether the access control constraints are fulfilled, but no additional information about the input. Finally, we guarantee that the user will not learn anything about the server's signing key.

Unlinkability: The third guarantee is unlinkability. This means that multiple activities by the same user are never linkable to each other. This is somewhat similar to obliviousness which says that activity is not linkable to a specific user. This, however, is an even stronger notion which says that even if the server cannot identify the user, this is not enough. Instead, the server must not even be able to identify that previous or future actions are made by the same user.

Confidentiality: The fourth guarantee is confidentiality. This means that only the user that is permitted to view the data is able to. This combined with the above guarantees of obliviousness and unlinkability allows for a very secure system.

Correctness: The final guarantee is correctness. As the name suggests, this means that the OPRF evaluates correctly each time.

4 Access Control Scheme

4.1 Setup Stage

Before any users are able to use the system for rate limiting, the server has to set up two databases to record the necessary data:

Nonce DB: The first database the server will generate is to store all of the nonces that have been used to create OPRF calls in the past. As explained below in section 4.2, the server will ensure that each new key generation has an unused nonce to assist in the enforcement of rate limiting.

Serial Number DB: The second database the server will generate is to store the serial numbers that are used for retrieving keys. As explained in section 4.3, the server must ensure that each OPRF call contains a new serial number. In order to enforce this rule, the server must store all of the serial numbers used and ensure that each new serial number is not already contained in the database.

4.2 Key Signing

For the rate limiting scheme, users will be able to either generate a new key or retrieve an old key. Opal is able to perform either action and only enforce rate limiting when retrieving an old without alerting the server which action the user is performing. In order to accomplish this, the user will have to send the exact same input to the server regardless of whether she is generating a new key or retrieving an old key. For this scheme, the user will send five inputs to the server regardless. The actual generation of these inputs will be explained in the next two sections, as it is different for key generation and key retrieval. A short description of each input is below.

Serial number: As mentioned above in section 4.1, each new query to the server will include a serial number. This prevents a user from forking an old state and abusing the rate limits. Each time the user generates a new key or retrieves an old key, she will reveal the previous serial number, and the server will record it.

New nonce: As mentioned above in section 4.1, each new query will also contain a nonce. The nonce, as opposed to the serial number, will be used for something different depending on whether the user plans to retrieve an old key or generate a new key. If the user is generating a new key, the nonce will be part of the new key generated. Otherwise, the nonce is irrelevant to the actual OPRF call and just there to confuse the server.

OPRF call: This will be the data that the OPRF is actually called on. The user will initiated the OPRF call by computing a hash using a hash-to-group function of the input data. Once this hash is computed, the user will blind it by raising to the power of a randomly generated value r.

State: The user will provide a new state each time they query the server either to generate a new key or retrieve an old key. The state will be the hash of the user's counter values and her serial number. The server will then sign the state using ECDSA.

Proof: Finally, the user will attach a zk-SNARKs proof to her request. The server will verify that the proof is a valid proof either for key generation or for key retrieval. The actual things the user will prove completely depends on whether the user is retrieving an old key or generating a

new key and will be explained in great detail in sections 4.3 and 4.4.

Once these inputs are received by the server, the server will perform a few checks before returning the necessary data. First, the server will ensure that the serial number has not been used before. Next, the server will ensure that the nonce has not been used before. Finally, the server will ensure that the proof is a valid zk-SNARKs proof and verifies with the server's verification key. If all three of these tests pass, the server will perform two operations. First, the server raises the blinded OPRF input to the power of *k*, the server's secret key. Next, the server signs the current state using ECDSA and its secret key *l*. The server returns these values to the user who can then unblind the OPRF input and get their original input raised to the power of *k*.

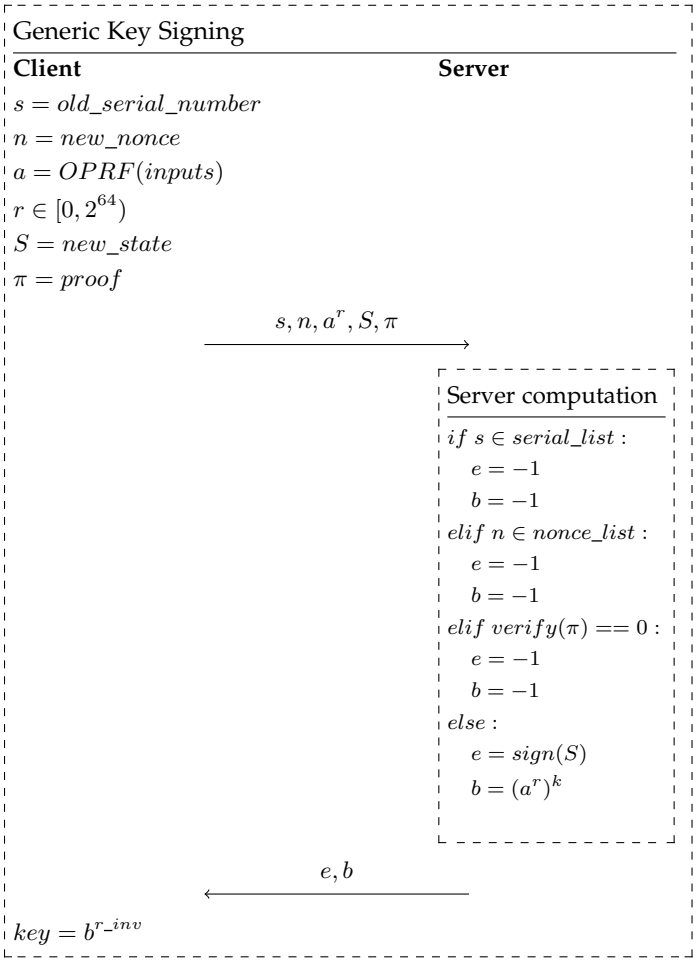

Fig. 1. This shows the key generation stage of the scheme

4.3 Key Generation

In section 4.2, we described each of the input values that the client sends to the server. We now describe how a user would generate these values if she wishes to generate an entirely new key for a new object.

Serial number: The serial number will be generated by uniformly selecting an integer between 0 and 2^{64}.

New nonce: The nonce will also be generated by uniformly selecting an integer between 0 and 2^{64}.

OPRF call: The OPRF call inputs will be the inputs that the user uses for her key. This will include the user's secret key, the bucket and object name that the user wants an encryption key for, and finally, the nonce. In this case, the nonce will be exactly the same as the nonce just generated. The user will hash all four of these values together using some hash-to-group function.

State: Since the user is generating a new key, this will not affect any of her counters. Her new state is identical to her old state with the only difference being a new serial number.

Proof: The proof for key generation will be simple. the user will need to prove four things:

- First, the user will prove that her new state's counter values are the same as her old state's.
- Next, the user will prove that she has a valid signature for her old state.
- Next, the user must prove that the public OPRF call she sends is indeed the hash-to-group function of the required inputs.
- Finally, the user will prove that her public nonce is equal to the nonce used in the OPRF call.

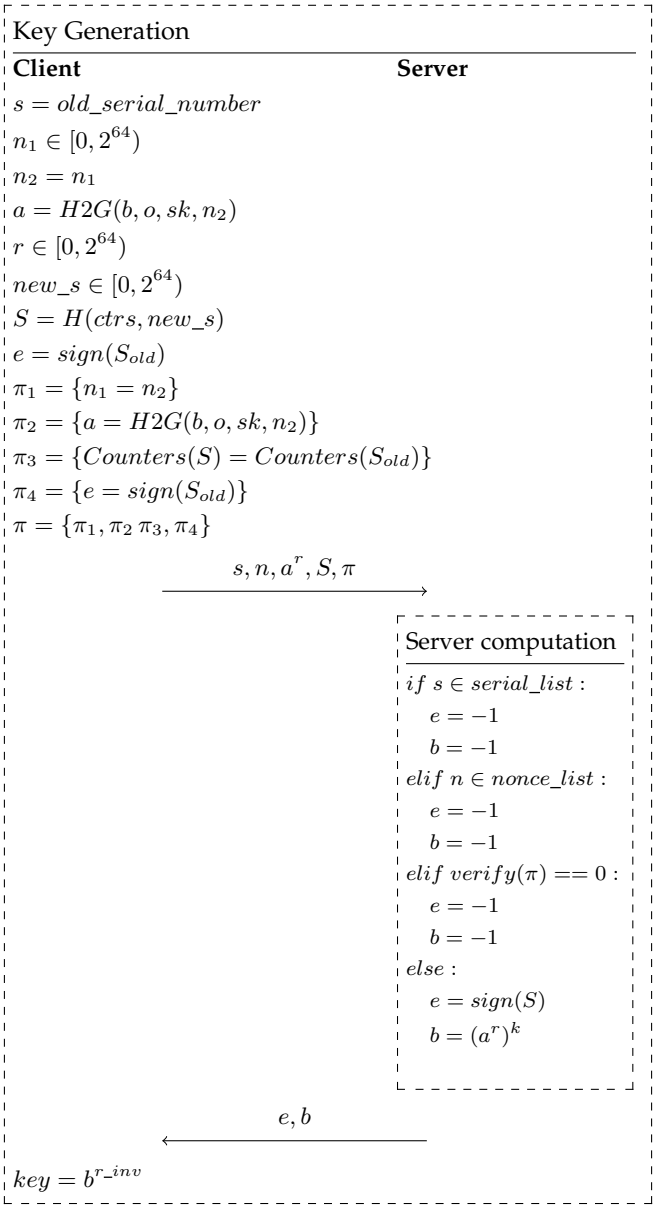

Fig. 2. This shows the key generation stage of the scheme

4.4 Key Retrieval

We now describe how a user would generate the values from section 4.2 if she wishes to retrieve a key that has already been generated.

Serial number: The serial number will be generated by uniformly selecting an integer between 0 and 2^{64}.

New nonce: The nonce will also be generated by uniformly selecting an integer between 0 and 2^{64}.

OPRF call: The OPRF call inputs will be the inputs that the user uses for her key. This will include the user's secret key, the bucket and object name that the user wants an encryption key for, and finally, the nonce. In this case, the nonce will not be the same as the nonce just generated. The user will instead use the nonce from the key when she originally created it. She will then hash all four of these values together using some hash-to-group function.

State: Since the user, in this case, is retrieving an old key, she will have to update her counter values. To illustrate this, consider the following example. Assume Alice has queried three objects from bucket A, four from bucket B, and eight from bucket C. Her state would be {'A: 3, B: 4, C: 8, SERIAL: s_1}. If she now chooses to query a new object, she will add one to the state bucket she accessing. Assume she wishes to access a new object from bucket A. Her new state will become {'A: 4, B: 4, C: 8, SERIAL: s_2}. Notice how Alice also updates her serial number when she updates her state. The state will actually be the hash of all of these values together.

Proof: The proof for key retrieval will be more complicated than the proof for key generation. In this round, the user will need to prove five different things:

- First, the user will prove that her new state is some function of her old state.
- Once she proves that her new state is a function of her old state, she will prove that the function is valid. This can mean one of two things. It could mean that only one bucket was iterated, and the counter for that bucket was iterated once. It could also mean that a new bucket was created, the counter for that bucket is 1, and no other counters were altered.
- Next, the user will prove that she has a valid signature for her old state. This will prove that the old state was signed by the server and is not a fake. This prevents the user from developing fake old states and then performing valid functions on them to get unlimited key retrievals.
- Next, the user must prove that the public OPRF call she is sending is indeed the hash-to-group of the required inputs.
- Finally, the user will prove that the new state is valid. Since Opal is able to support arbitrary rate limits, this will depend on the implementation. The main thing that is required for a valid state, however, is that it meets all of the counter limits. Assume that Alice is implementing Opal and she chooses to allow access to three buckets and eight objects per bucket. Alice would need to prove she is in compliance with both of these rules.

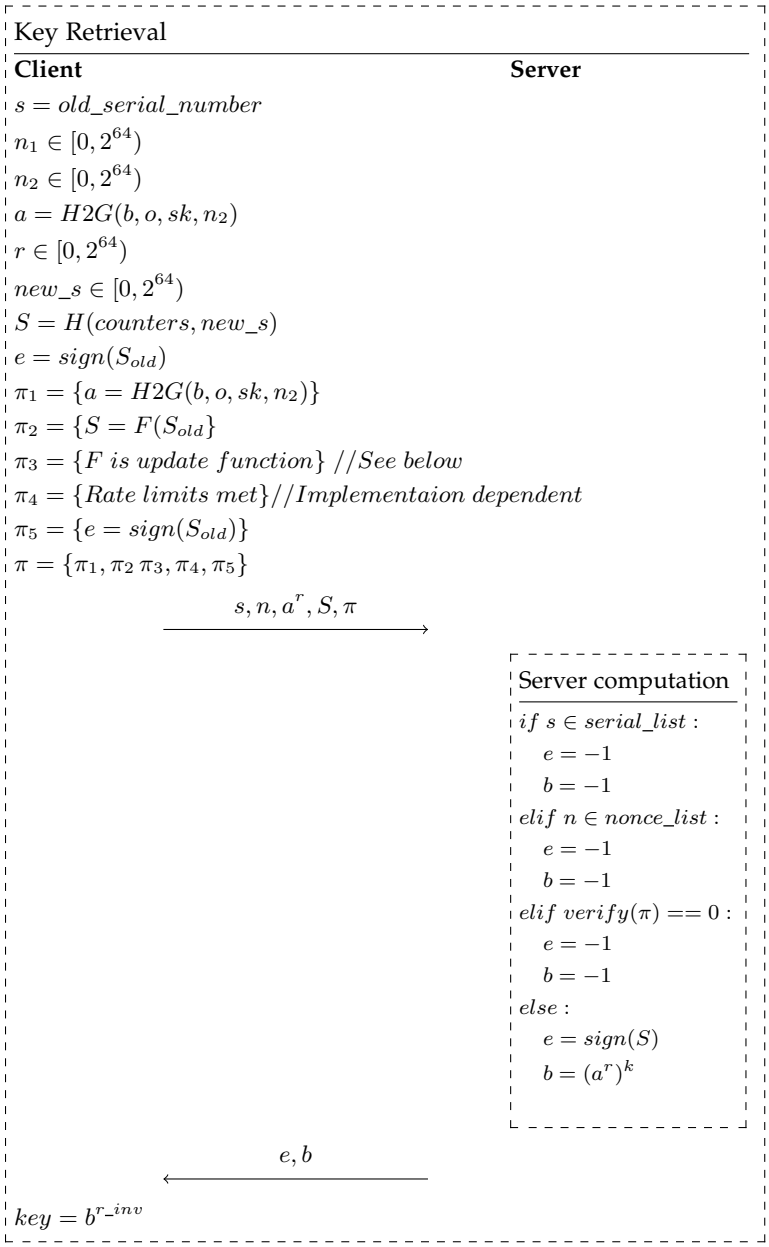

Fig. 3. This shows the key generation stage of the scheme

Algorithm 1 F: State update function

```
counters ← old_state.counters
was_added ← False
for bucket in counters.keys() do
    if new_bucket == bucket then
        counters[bucket] ← counters[bucket] + 1
        was_added ← True
    else
        counters[bucket] ← 1

    end if
end for
```

5 Applications

5.1 A File System

The most obvious application of the rate limiting described above is the example used in the introduction. We assume that a user has a filesystem that they wish to impose rate limiting on. For this example, let's assume Alice works closely with clients in a very sensitive manner. Alice knows how many clients she typically meets with per day. She could then set rate limits to only allow the retrieval of documents from that many clients per day. This would be a very basic type of rate limiting. Alice would simply have a limit on how many documents could be viewed.

Let's now assume Alice has a slightly more complicated need for rate limiting. Perhaps Alice is a lawyer, and she would not want an adversary to learn everything about even one client if they were to break in. If Alice knows, for example, that she would only need to access documents from one case for each of her clients, she could impose further access control restrictions to only allow for such behavior. This way, if an adversary were to break into Alice's machine, they would not even be able to view all of the documents for one client. The adversary would be limited to viewing only the information from one case for each client.

The above is an example of a slightly more complicated rate limiting scheme. Due to the programmability and customizability of our rate limiting scheme, Alice would actually be able to implement custom rate limiting constraints. Since Alice would know her needs for viewing files, she could construct rate limits that allow her to access all of the documents she would need for typical business activities, but restrict any further access to any documents.

5.2 Social Media

Another application we give for our rate limiting scheme is a social media application. Specifically, assume that Alice wants to limit how many of her posts her friends are able to view. Alice would then be able to impost arbitrary rate limits to limit how many posts her friends could see. This could be quite effective at limiting social media stalking.

To implement this rate limiting, we follow a very similar construction as in section 4 with a slight change. To make a post, Alice will ask the server for an encryption key just as in section 4, and she will encrypt the post. She will then send the encrypted post to the server for storage. Once Bob wishes to retrieve a post, he will ask the server for the password and send a proof that he is within the limits. Next, he will use private information retrieval [7] in order to retrieve the encrypted post from the server without the server learning which post he is retrieving. Finally, he will decrypt the post locally and be able to view it.

In order to actually implement this application of the rate limiting, there will be a slight change. Specifically, if Bob wishes to view one of Alice's posts, he will have to prove that he is actually friends with Alice. In order to accomplish this, Alice will generate a key upon creation of her account. Whenever she adds a new friend, she will share this key with them. In addition, whenever she makes a new post, she will include this key in the OPRF call that she makes. This way, only someone with the key will be able to generate the same OPRF call for decryption.

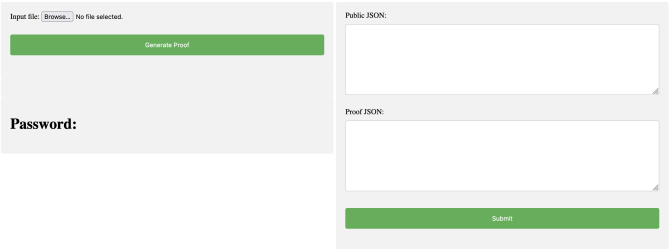

Fig. 4. Flask site starting page

6 Results

We were able to successfully implement a Flask server that ran this rate limiting. You begin by inputting a JSON file with all of your inputs into the website. The website then generates a proof that all of the rate limiting constraints have been met. The proof is then sent to the server along with all of the public values. The server verifies that the proof is valid and that the serial number hasn't been seen before. It will then return the password for that object and the ECDSA signature for the current state.

For the exact constraints, we decided to implement several different rules. We limit the total number of password queries to 5, the maximum number of buckets to 2, and the maximum number of objects per bucket to 3. Finally, we decided to implement a leaky bucket style of adding availability. This means that each day, the limits are not completely reset. For our implementation, we decided to eliminate one previous request. This means that if a user has already used up all of their requests, they will only be allowed one request the following day. If they don't use that query, they will be allowed two the following day and so on.

Figure 4 below shows the Flask site. A user will begin by uploading their inputs to the file input field. Alternatively, the user could generate a proof on their own and copy and paste the results in the respective fields. Once the user inputs his file, he will click the generate button and the public and proof fields will populate. This can be seen in Figure 5 below. The user can then click submit and the information will be sent to the server.

Once the user sends these fields to the server, the server will verify that the proof is in fact valid. The server then verifies that the serial number has not been used before. Once the server completes this process, it will send one of three responses. The first response, meaning that everything was successful is the password to decrypt the object, as well as the ECDSA signature. This can be seen below in Figure 6. The second response would be an alert that the rate limiting constraints have not been fulfilled and the proof did not verify. This can be seen below in Figure 7. Finally, the server could respond with an alert that the serial number or nonce has been used before and the request is therefore invalid. This can be seen below in Figures 8 and 9 below.

6.1 Evaluation

In addition to creating a the Flask site presented above, we timed some of the important operations that would be done by the server and user. Specifically, we timed the

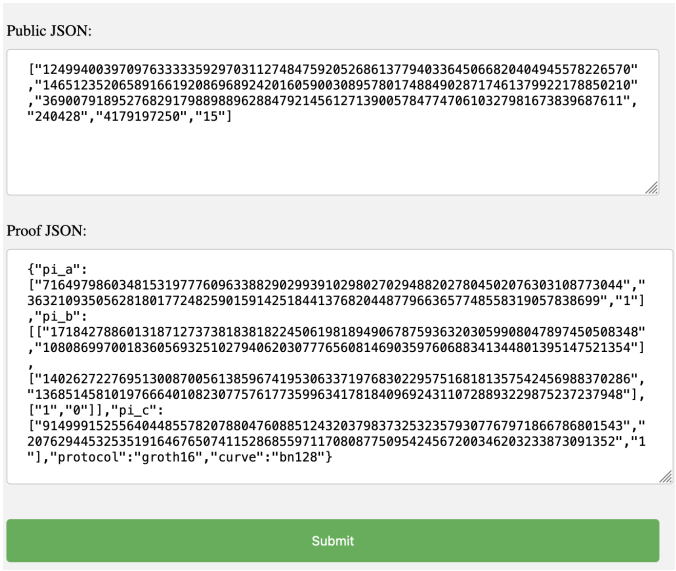

Fig. 5. Flask site with public and proof fields populated

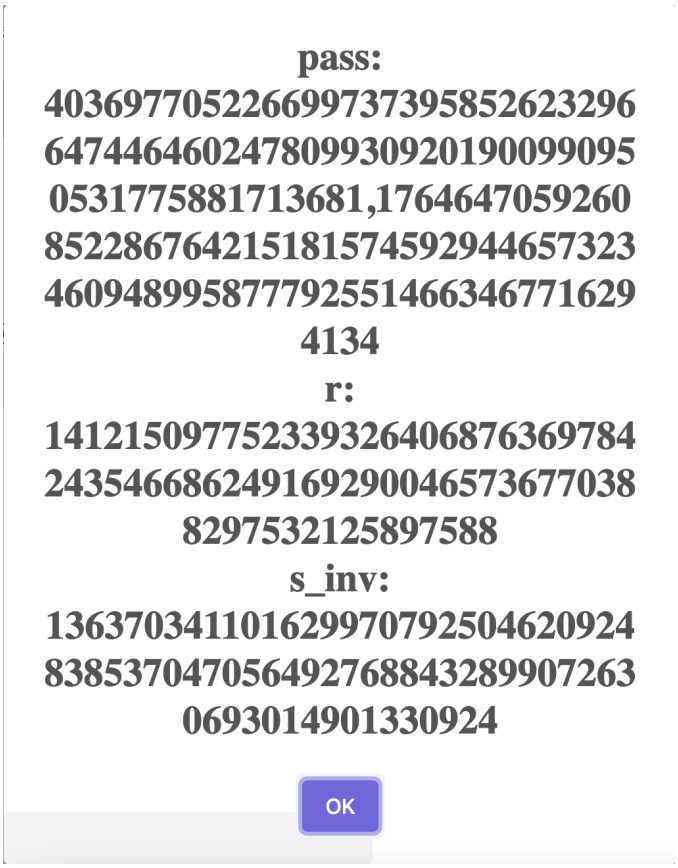

Fig. 6. Successful response from the server including password and ECDSA signature

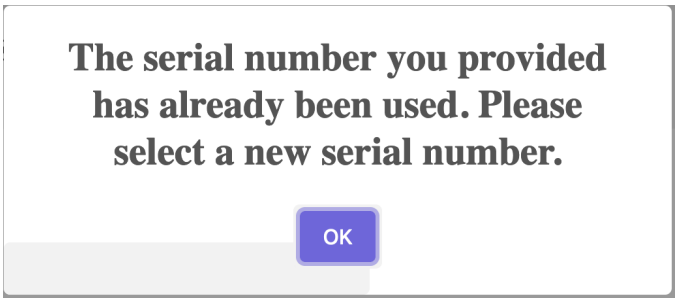

Fig. 7. Alert from server that the proof is invalid

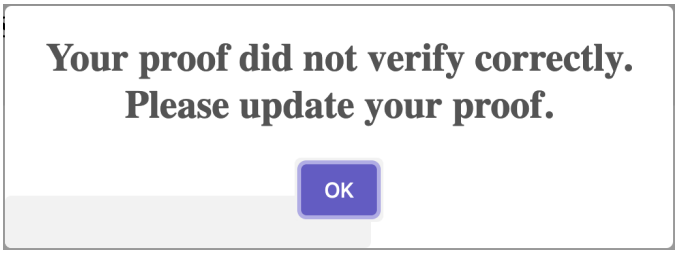

Fig. 8. Alert from server that the serial number has been used before

proof generation and verification time for both generating a new key and retrieving an old key. The data is shown in the table below.

Process Completed	Time Taken
Proof generation for new key	7.72 seconds
Proof verification for new key	0.44 seconds
Proof generation for old key	7.69 seconds
Proof verification for old key	0.44 seconds

7 Conclusion

In this paper, we presented the design and implementation of an oblivious rate limiting scheme. This scheme allows for customizability in the specific rate limiting constraints that are applied. The user will begin by generating a password using an OPRF with the server to generate a password to encrypt a file. When trying to decrypt the file, the user computes the same OPRF and sends a proof that the user is within the given rate limiting constraints. This allows for

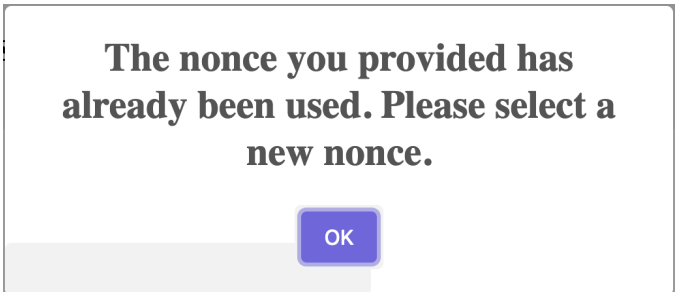

Fig. 9. Alert from server that the nonce has been used before

arbitrary rate limiting constraints for the user to fulfill. In addition, this allows the user's actions to be oblivious to the server and allows for the server to correctly perform the necessary rate limiting.

In this case, the input data is the *bucket name, object name, password, and old nonce*. This is, again, different depending on whether the user is generating a new key or retrieving an old key. If the user is generating a new key, this will be the same as the new nonce and will be a randomly generated number. If the user is retrieving an old key, the old nonce will be the nonce from the key she is retrieving and entirely different from the new nonce.

8 References

[1] Casacuberta, S., Hesse, J., & Lehmann, A. (2022, June). SoK: oblivious pseudorandom functions. In 2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P) (pp. 625-646). IEEE.

[2] David Chaum. Blind signatures for untraceable payments. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, CRYPTO'82, pages 199–203. Plenum Press, New York, USA, 1982.

[3] David Chaum. Blind signature system. In David Chaum, editor, CRYPTO'83, page 153. Plenum Press, New York, USA, 1983

[4] Davidson, A., Goldberg, I., Sullivan, N., Tankersley, G., & Valsorda, F. (2018). Privacy pass: Bypassing internet challenges anonymously. Proceedings on Privacy Enhancing Technologies.

[5] Everspaugh, A., Chaterjee, R., Scott, S., Juels, A., & Ristenpart, T. (2015). The pythia PRF service. In 24th USENIX Security Symposium (USENIX Security 15) (pp. 547-562).

[6] Chor, B., Kushilevitz, E., Goldreich, O., & Sudan, M. (1998). Private information retrieval. Journal of the ACM (JACM), 45(6), 965-981.