Hash Function Implementation and Machine Learning

*

Jose Nocelo
Computer Science Major
Hunter College, CUNY
New York, USA
Jose.Nocelo59@myhunter.cuny.edu

Anthony Jerez
Computer Science Major
Queens College, CUNY
New York, USA
anthony.jereztenecela94@qmail.cuny.edu

***Abstract*—This paper presents a novel implementation of a hash function using machine learning techniques. The original hash function is based on the work by Corentin Le Coz et al. (2022), which uses the special linear group $\mathrm{SL}_n(\mathbb{F}_p)$. We describe the process of generating input-output sequences, preprocessing the data, and training a neural network model to predict the hash function's outputs.**

***Index Terms*—Hash Function, Machine Learning, Neural Networks, SLn(Fp)**

I. Introduction

Hash functions are fundamental components in cryptography and data integrity verification. This work explores the implementation of a hash function inspired by the paper "Post-quantum hash functions using $\mathrm{SL}_n(\mathbb{F}_p)$" [1]. We leverage machine learning techniques to model and predict the hash function's behavior.

II. Related Work

Our implementation is based on the work by Corentin Le Coz et al. (2022) [1], which proposes a post-quantum hash function utilizing the special linear group over finite fields. Previous implementations have focused on classical approaches, while our work integrates machine learning to enhance prediction capabilities.

III. Methodology

A. Hash Function Description

The hash function operates on sequences of matrices derived from parameters a and b and matrix powers. The matrices A and B are constructed as follows:

```python
def create_matrices(n, a, b):
    A = np.eye(n)
    B = np.eye(n)
    for i in range(n - 1):
        A[i, i + 1] = a
        B[i + 1, i] = b
    return A, B
```

After we construct Matrices A and B we create our Python function that will multiply matrices depending on the character found in the sequence that we will explain shortly:

```python
def compute_hash_function(sequence, matrices, p):
    result = np.eye(matrices[1].shape[0])
    for char in sequence:
        number = int(char)
        result = np.dot(result, matrices[number])
    return np.mod(result, p)
```

The next step is to create the dictionary s, which will hold the value to which each key is mapped. Since input sequences can only be made up of 1's, 2's and 3's there are only three keys in our dictionary along with their respective values:

```python
# Create the matrices
A, B = create_matrices(n, a, b)
A_power_l = matrix_power(A, l)
B_power_l = matrix_power(B, l)
A_inv = inverse_matrix(A_power_l)
B_inv = inverse_matrix(B_power_l)

s = {
    1: B_power_l,
    2: A_inv,
    3: B_inv
}
```

B. Data Generation

We generate all possible sequences of length 8 using the characters '1', '2', and '3'. The corresponding hash function outputs are computed and saved.

```python
import itertools
sequences = [''.join(seq) for seq in itertools.product('123', repeat=8)]
inputs = []
outputs = []
for sequence in sequences:
    output_matrix = compute_hash_function(sequence, s, p)
    inputs.append(sequence)
    outputs.append(output_matrix.flatten())
np.save('all_inputs.npy', inputs)
np.save('all_outputs.npy', outputs)
```

C. Example Calculation from Section 2

To validate our implementation, we computed the hash function for a specific sequence mentioned in Section 2 of the reference paper [1]. The sequence provided is '[2, 2, 3, 2, 2, 2, 1]'. The result of the hash function for this sequence is shown below:

```python
# Example sequence
sequence = [2, 2, 3, 2, 2, 2, 1]

# Compute the result of the hash function
result = compute_hash_function(sequence, s, p)

print("Result of the hash function:")
print(result)
```

The output for this sequence, as calculated by our implementation, is:

Result of the hash function:

$$\begin{bmatrix} 6.94190977e+08 & 2.33260720e+08 & 2.92979520e+07 \\ -3.83796480e+07 & -1.28962550e+07 & -1.61979200e+06 \\ 1.19193600e+06 & 4.00512000e+05 & 5.03050000e+04 \end{bmatrix}$$

These numbers are identical to the ones given in the concrete example from Section 2 of the paper, confirming the correctness of our implementation.

D. Preprocessing

The input sequences are tokenized and converted to PyTorch tensors for training.

```python
import torch
inputs = np.load('all_inputs.npy',
    allow_pickle=True)
outputs = np.load('all_outputs.npy')
tokenized_inputs = [[int(char) for char in
    sequence] for sequence in inputs]
X = torch.tensor(tokenized_inputs, dtype=torch
    .long)
y = torch.tensor(outputs, dtype=torch.float32)
```

E. Model Training

We train a neural network model to predict the output matrices given the input sequences.

```python
class HashFunctionPredictor(nn.Module):
  def __init__(self, input_size, output_size)
    :
    super(HashFunctionPredictor, self).
      __init__()
    self.embedding = nn.Embedding(4, 10)
    self.lstm = nn.LSTM(10, 50, batch_first=
      True)
    self.fc = nn.Linear(50 * input_size,
      output_size)
  def forward(self, x):
    x = self.embedding(x)
    x, _ = self.lstm(x)
    x = x.reshape(x.size(0), -1)
    x = self.fc(x)
    return x
```

IV. Data Characteristics

When the input sequence is "11111111", the output of the hash function is:

```
Result of the hash function (pre-modulo):
[[1.000e+00 0.000e+00 0.000e+00]
 [6.400e+01 1.000e+00 0.000e+00]
 [1.984e+03 6.400e+01 1.000e+00]]
Result of the hash function (post-modulo):
[[1. 0. 0.]
 [4. 1. 0.]
 [4. 4. 1.]]
```

When searching for this matrix in our output data using the script `search_by_matrix.py`:

```python
import numpy as np

# Load the output dataset
outputs = np.load('all_outputs.npy')

# Function to search for a matrix in the
    outputs
def search_matrix(matrix, outputs):
   matrix_flat = matrix.flatten()
   indices = []
   for i, output in enumerate(outputs):
      if np.array_equal(output, matrix_flat):
         indices.append(i)
   return indices, len(indices)

input_matrix = np.array(input().split(), dtype
    =float).reshape((3, 3))
indices, count = search_matrix(input_matrix,
    outputs)

if count > 0:
   print(f"The matrix appears {count} time(s)
       at indices: {indices}")
else:
   print("The matrix does not appear in the
       output data.")
```

When inputting `1.0 0.0 0.0 4.0 1.0 0.0 4.0 4.0 1.0`, the output is:

> The matrix appears 61 time(s) at indices: [0, 121, 242, 363, 566, 674, 710, 722, 726, 1089, 1538, 1646, 1682, 1694, 1698, 1970, 2006, 2018, 2022, 2114, 2126, 2130, 2162, 2166, 2178, 3267, 4454, 4562, 4598, 4610, 4614, 4886, 4922, 4934, 4938, 5030, 5042, 5046, 5078, 5082, 5094, 5858, 5894, 5906, 5910, 6002, 6014, 6018, 6050, 6054, 6066, 6326, 6338, 6342, 6374, 6378, 6390, 6482, 6486, 6498, 6534]

This result shows that the same output matrix is present multiple times in the output data, indicating that multiple input sequences lead to the same output. This highlights a limitation in our machine learning attack: while our model can predict a correct sequence, it may not always be the exact sequence we are looking for. This also demonstrates an additional layer of security in the hash function, making it challenging to predict the correct sequence.

V. Experiments and Results

The model is trained and evaluated on the generated dataset. We use early stopping to prevent overfitting and save the best model based on test loss.

```
# Check for improvement
if test_loss < best_test_loss:
   best_test_loss = test_loss
   patience_counter = 0
   torch.save(model.state_dict(), '
       best_model.pth')
   print("Model improved and saved.")
else:
   patience_counter += 1
   if patience_counter >= patience:
      print("Early stopping triggered.
          Training halted.")
      break
```

VI. Conclusion

This work demonstrates the potential of integrating machine learning techniques with cryptographic hash functions. It also reveals flaws in our machine learning attack, as using raw input and output data was problematic. The complexity of the hash function made it difficult to predict input sequences accurately. Future work could explore more complex models and larger datasets to improve prediction accuracy further.

VII. References

References

[1] C. Le Coz, C. Battarbee, R. Flores, T. Koberda, and D. Kahrobaei, "Post-quantum hash functions using $\mathrm{SL}_n(\mathbb{F}_p)$", *Cryptology ePrint Archive*, Paper 2022/896, 2022. [Online]. Available: https://ia.cr/2022/896

[2] J. Nocelo, "Hash Function Implementation," GitHub repository, 2024. [Online]. Available: https://github.com/nocelojose/HashFunctionImplementation.git