Al Dente: An Evaluation of PASTA

Bennett Larson
University of Minnesota
lars4652@umn.edu

Michael McLaughlin
University of Minnesota
mclau361@umn.edu

Logan Praneis
University of Minnesota
prane001@umn.edu

Paul Rheinberger
University of Minnesota
rhein055@umn.edu

ABSTRACT

Token-based authentication is widely used across the Internet by companies like Amazon, Facebook, and Google[3, 9, 10] to allow users to share account information with third party services, without sharing their passwords with the third party. The most notable standard for token-based authentication is OAuth.[16] In token-based authentication standards, such as OAuth, the identity authentication server becomes a single point of failure for the system. If these servers are compromised, the adversary can use the secret key to forge arbitrary tokens and perform offline dictionary attacks to recover user passwords. Agrawal et al. present a password-based threshold authentication scheme in [2] that defends against these single point of failure attacks. Our work combines the work of Agrawal et al. with the work Burns et al. on oblivious elliptic curve pseudorandom functions. We demonstrate the security of threshold authentication against dictionary attacks and prove that this security comes at little extra cost. Finally, we explore the differences between our elliptic curve implementation with the one proposed by Agrawal et al. We also suggest further improvements upon this protocol to increase security and improve generation time.

KEYWORDS

PASTA, Password-Based Threshhold Authentication, TO-PRF, Threshhold Token Generation, Threshhold Oblivious Pseudorandom Function, EC-OPRF, Elliptic Curve Oblivious Pseudorandom Function, RSA Threshold Signature

ACM Reference Format:
Bennett Larson, Michael McLaughlin, Logan Praneis, and Paul Rheinberger. 2019. Al Dente: An Evaluation of PASTA. CSCI 5471 Modern Cryptography, Spring 2019, Nicholas Hopper. 9 pages.

1 INTRODUCTION

Session and token-based authentication schemes are used throughout the Internet to provide a sense of state in the otherwise stateless HTTP protocol. In session based authentication, the server stores session data and the client stores the session ID. The client then sends the session ID in every request for the server to verify. In token-based authentication schemes, the user's state is encrypted into a token stored on the client. Similarly, the token is sent with every subsequent request to be verified by the server.

Since no session data is stored by the server in token-based authentication, these tokens can be scaled to provide single sign-on for multiple independent parties. That is, a client may use a service to become authenticated and receive a valid token, then they can use this token as authentication on any other third party that accepts tokens from that specific authentication service. Companies such as Google, Facebook, and Amazon[3, 9, 10] are using open standards, such as OAuth[16], to allow their generated tokens to be accepted as authentication to third parties. This allows the third party to never interact with the user's password.

Token-based authentication schemes work as follows. The client signs on to an identity provider(i.e. Google, Facebook, Amazon, etc) with their username and the hash of their password, typically. The identity provider stores pairs of usernames and hashed passwords. Upon receipt of the clients credentials, the identity provider looks up the username and verifies that the password hashes are the same. After verification the identity provider will issue an authentication token signed by the identity provider's secret key. This token is generally a digital signature or message authentication code on the client's data, token expiration data, and other token access information. When the client sends this token to use as authentication to a third party, the third party uses the identity provider's verification key to verify that the token is legitimate and has not expired.

A major security advantage to these schemes is that the third party is given no information about the user's password, while still being able to verify that the user is legitimate. This makes token-based authentication an ideal scheme for third parties that do not have the resources to develop and maintain secure session based schemes.

However, since all information required to issue a token is stored in one place, the identity provider becomes a single point of failure. Two different attacks can be carried out by an attacker that manages to breach the identity provider. Firstly, the attacker has access to the identity provider's secret key and can therefore forge arbitrary tokens. Secondly, the attacker has access to the hashed passwords of all users. The attacker can use an offline dictionary attack to recover the user's passwords.

Agrawal et al. [2] propose a password-based threshold token-based authentication(PASTA) framework to prevent these sorts of attacks. In their solution, the identity provider is distributed over n servers. Any t of these n servers can collectively authenticate and produce tokens, but no $t-1$ servers are able to forge a valid token or recover user passwords. That is, up to $t-1$ servers can be compromised while not allowing the attacker to forge tokens or recover passwords.

We extend the work of Agrawal et al. by implementing their Password-Based Threshold Authentication(PASTA) scheme using elliptic curves to construct the oblivious pseudorandom function primitive. In their original paper, [2], Agrawal et al. construct an oblivious pseudorandom function based on the decisional Diffie-Hellman assumption. We choose to base our protocol on the security of the elliptic curve discrete logarithm problem using elliptic curves. Elliptic curve implementations have a number of advantages over alternatives, including a smaller required key size and more efficient computation.

The organization for the remainder of the paper is as follows. Section 2 describes the first of two main cryptographic primitives used, threshold token generation. Subsections 2.1, 2.2, and 2.3 give technical details on RSA signatures, threshold signing, and implementation details, respectively. Section 3 describes the second main cryptographic primitive, oblivious pseudorandom functions. Subsections 3.1 and 3.2 give mathematical and implementation details of oblivious pseudorandom functions, respectively. Subsection 3.3 provides a concrete example of oblivious transfer between two parties. Section 4 provides implementation details to our replication of the solution in [2]. Section 5 provides additional evaluation on [2] and evaluation on our solution. Finally, section 6 examines the significance of our contribution and concludes our paper.

2 THRESHOLD TOKEN GENERATION

A Threshold Token Generation (TTG) algorithm distributes the task of generating a secret that can be used as a token for authentication over n servers. The unforgeability property of such an algorithm requires that for a threshold of $t \leq n$ servers, $t-1$ are unable to compute a new token value. The distributed generation of a token protects the master secret key (*msk*), as each of the t servers only receive a share of the *msk*, and the same threshold guarantee that applies to forging a token also applies to the recovery of *msk*. The token generated by a TTG algorithm can also be verified, via public key or MAC methodologies, depending on the cryptographic primitives used in its generation.

The solutions proposed by Agrawal et al.[2] include: the DDH-based and PRF-based DPRFs or Naor et al. [15], the threshold RSA signature based scheme of Shoup [20], and the pairing based signature scheme of Boldyreva [6]. Our solution implements scheme proposed by Shoup [20] based on RSA signatures.

2.1 RSA Signatures

The RSA signature scheme of Rivest et al. [18] involves choosing

$$ed \equiv 1 \pmod{\phi(N)}$$

where $N = pq$ for primes p, q of bit length k and

$$\phi : \mathbb{N} \mapsto \mathbb{N}$$

is the Euler Totient function. The secret key pair is (N, d) and the public key pair is (N, e). Signing a message $m \in \mathbb{M}$ is equivalent to

$$\sigma := m^d \pmod{N}$$

and verifying the signature involves using the public key to calculate

$$\sigma^e \overset{?}{\equiv} m \pmod{N}$$

The RSA signature scheme proposed by Shoup [20] adds restrictions that

$$p = 2p' + 1$$

$$q = 2q' + 1$$

where p', q' are also large primes, in addition to asserting that $e > n$ where n represents the total number of servers involved in the token generation scheme. The threshold construction of the scheme by Shoup [20] will be further discussed in Subsection 2.3.

The RSA scheme described above is vulnerable to existential forgeries if m can be chosen arbitrarily, and so our implementation, in addition to the implementation of Shoup [20], involves a RSA-FDH scheme originally proposed by Bellare et al. [4]. In this extension of the original RSA signing scheme, the message m is first hashed using

$$\mathbf{H} : \{0, 1\} \mapsto \mathbb{Z}_N$$

modelled as a random oracle. Thus the signing algorithm sets $\sigma := \mathbf{H}(m)^e$, allowing the message to be an arbitrary length in addition to having unforgeability, given the assumption that the RSA problem is hard.

2.2 Threshold Signing

The secret sharing scheme originally proposed by Shamir [19] explores how to split a secret, *msk* into shares $sk_1, sk_2, \cdots, sk_n$ where $t \leq n$ shares are required to reconstruct *msk*. In order to split a secret in $msk \in \mathbb{Z}_p$ into n parts, a Lagrange interpolating polynomial is utilized. We let

$$S(x) = c_1 + c_2x^1 + c_3x^2 + \cdots + c_nx^{n-1}$$

be a polynomial, with $c_i \in_R \mathbb{Z}_p$ and evaluate

$$msk = S(0)$$

$$sk_i = S(i) \pmod{p}$$

Given a set $\mathcal{T}$ with $|\mathcal{T}| \geq t$ of shares, we can reconstruct the *msk* by performing a Lagrange polynomial interpolation on the points in $\mathcal{T}$ and then finding $S'(0)$. The Lagrange polynomial interpolation formula gives us that

$$S'(x) = \sum_{j=1}^{n} S'_j(x)$$

$$S'_j(x) = y_i \prod_{k=1, k \neq j}^{n} \frac{x - x_k}{x_j - x_k}$$

for points $\mathcal{T} = (x_1, y_1), \cdots, (x_t, y_t)$. Thus, we can compute *msk* by using this polynomial, given that the threshold requirement is met. Additionally, any amount of shares less than the threshold, t reveals no information about $S(0) = msk$, as this value is uniformly distributed in $\mathbb{Z}_p$.

We adapt this scheme, in addition to adapting the RSA signing scheme mentioned in Subsection 2.1 in order to more efficiently compute threshold signatures.

2.3 Construction

The TTG scheme in our implementation of the PASTA protocol uses the RSA signature signing scheme, modified to work in a threshold situation by Shoup [20]. The protocol uses 2048-bit RSA keys, as recommended by NIST [13]. The scheme below is defined by a threshold t of n total parties.

1. (Generation)
The generation of the parameters needed for the RSA signature scheme involves creating a RSA key pair,

$$p = 2p' + 1, q = 2q' + 1$$

where p, q, p', q' are prime.

$$N = pq, m = p'q'$$

Most of the work will be done in $\mathbb{Z}_m^*$. We then generate $e > n$ and calculate d such that $de \equiv 1 \pmod m$. The set (N, e) makes up the public key and (N, d) makes up the private key. A polynomial is generated of degree $t - 1$ where

$$f(X) = d + a_1x + a_2x^2 + \cdots + a_{t-1}x^{t-1}$$

where $a_i \in \mathbb{Z}_m$. This constitutes the *secret* polynomial to be recreated by the token servers as $f(0) = d$. We then compute $s_i = f(i) \pmod m$ for $1 \leq i \leq n$, the shares of the secret given to each token server, indexed by a public index i. We also assign $\Delta = n!$ for use in later algorithms.

2. (Partial Evaluation)
The partial evaluation of a token share is done by each of the servers. Given t partial evaluations, one can combine the shares to recover the original secret. Given s_i, we can calculate a token share by first hashing the original message, using SHAKE256: an Extensible Output Hash Function proposed by Dworkin [8], referred to as $\mathcal{H}$. This allows us to hash the message into $\mathbb{Z}_N$ in order for the RSA signature parameters to be satisfied. We denote a token share $y_i \in \mathbb{Z}_N$ as:

$$y_i = \mathcal{H}(x)^{4\Delta s_i}$$

3. (Combine)
Having received at least t shares from the token servers, the final token can be constructed. Given a set of index, share pairs: $\mathcal{S} = (i, y_i)_{i \in \{1..n\}}$, we first check that $|\mathcal{S}| \geq t$ and that $\mathcal{S}$ is made up of distinct shares. In the case that either one of these checks fail, the combine algorithm returns $\perp$. Otherwise, the token is generated. The Lagrange coefficients are calculated for interpolation at $y = 0$, scaling by the constant Δ

$$\lambda_{0,i}^{\mathcal{S}} = \Delta \prod_{k \in \mathcal{S}, k \neq i} \frac{x_k}{x_k - i}$$

where $x_i = i$ for a given (i, y_i) pair. Let w be defined as

$$w = \prod_{i \in \mathcal{S}} y_i^{\lambda_{0,i}^{\mathcal{S}}}$$

We then calculate constants a, b by using the Extended Euclidean Algorithm such that

$$4\Delta^2 a + eb = 1$$

and generate the token, $tk \in \mathbb{Z}_N$ as:

$$tk = w^a * \mathcal{H}(x)^b$$

4. (Verify)
The verification of the token can be simply computed, given the conditions of the public and private keys computed in (1). The verify algorithm computes

$$v = tk^e \pmod N$$

If $v \equiv \mathcal{H}(x) \pmod N$, the algorithm returns 1, else it returns $\perp$.

The signing scheme used by Shoup [20] differs from the standard scheme in that it works in $\mathbb{Z}_m$ rather than in $Z_{\phi(N)}$, however since $m = \frac{p-1}{2} * \frac{q-1}{2} = \frac{\phi(N)}{4}$, we can consider the calculations to be similarly secure, given a large value of N.

This protocol ensures that t parties can recreate the shared secret, while $t - 1$ have no advantage due to the adoption of Shamir Secret Sharing. Additionally, the computation required by the server is a single exponentiation in $\mathbb{Z}_N$, making the token response time minimal. This scheme also allows for verification of the generated token via a single exponentiation in $\mathbb{Z}_N$, allowing the service provider to accept / reject the token without intensive calculation. More in depth performance of the scheme will be discussed in section 5.

As mentioned above, the RSA threshold signing scheme is publicly verifiable by a single exponentiation. This allows the service provider to verify the token without storing additional information, as required by some of the other schemes proposed in the original paper by Agrawal et al. [2]. Further evaluation of the storage requirements of this protocol will be discussed in section 5.

3 OBLIVIOUS PSEUDORANDOM FUNCTION

A pseudorandom function(PRF) is a keyed, deterministic, and efficiently computable function whose output is indistinguishable from a random oracle by any efficient algorithm with more than a negligible probability. An oblivious pseudorandom function extends a pseudorandom function between two parties, in our case, a client and a server, such that the client holds the input to the PRF and the server holds the key to the PRF. The server computes the PRF value with its key on the client's input. An oblivious PRF has the additional properties that the server does not learn the client's input and the client learns nothing about the server's key throughout the protocol.

The solution proposed by Agrawal et al.[2] use the original work of Jarecki et al.[12] to implement the threshold oblivious pseudorandom function(TO-PRF), based on the decisional Diffie-Hellman inversion assumption. In our solution, we use the work proposed by Burns et al.[7] to implement oblivious pseudorandom functions based on elliptic curves.

3.1 Elliptic Curves

An elliptic curve for a prime p, $E(a, b, p)$, over $GF(p)$ with order r is defined as

$$y^2 = x^3 + ax + b.$$

It is also required that the curve be non-singular, $4a^3 + 27b^2 \neq 0$, so that there are three distinct roots.

The security of our elliptic curve cryptography is based on the assumption that the elliptic curve discrete logarithm problem is hard [14]. The elliptic curve discrete logarithm problem is defined as follows. Let E be an elliptic curve over a finite field K. Let P,Q be points on the elliptic curve. We want to find a scalar value k such that the scalar multiplication of P and k, $k \cdot P$, is equal to the point Q. It is widely assumed in the cryptography field that this is a hard problem.

In order to construct our pseudorandom function, we need use the scalar multiplication operation on elliptic curve points. To compute the scalar multiplication of a scalar and a point, elliptic curve point addition and point doubling are required. Note that point addition is a binary operation defined for 2 unique points on the elliptic curve and point doubling is a unary operation defined for 1 point on the elliptic curve. Point addition of the same point is point doubling.

Figure 1(a) gives a plot of the simple elliptic curve

$$y^2 = x^3 - 2x + 2.$$

For point addition, consider Figure 1(b) that depicts the point addition $P + Q = R$. First, we find the intersection of the elliptic curve and the line going through both P and Q. Then we reflect the point across the x axis to get our solution $R = P + Q$. For point doubling, consider Figure 1(c) that depicts the point doubling $2P = R$. First, we find the intersection of the elliptic curve and the tangent line at point P. Then, as in point addition, we reflect the point across the x axis to get our solution $2P = R$.

With point addition and point doubling, we can compute any scalar multiplication using the 'Double and Add' algorithm. For example, consider scalar multiplying point P by 100, that is $100 \cdot P$. It is clear that $100 = 64 + 32 + 8 = 2^6 + 2^5 + 2^3$, so $100 \cdot P = 2^6 \cdot P + 2^5 \cdot P + 2^3 \cdot P$. Hence, we are left with only point addition and point doubling operations that we know how to compute as given above.

We use scalar multiplications on points on the elliptic curve in the construction of our oblivious pseudorandom function in order to mask the user's input from the server and mask the server's secret key from the client, thus making the function oblivious. Scalar multiplication has two more important properties that are used in our construction. Scalar multiplication is commutative and the modular inverse of a number can be used as the scalar multiplication inverse. Since the elliptic curve discrete logarithm problem is hard [14], our pseudorandom function construction is cryptographically secure.

The chosen elliptic curve will affect the security and efficiency of the oblivious pseudorandom function.[7] An elliptic curve will a smaller order is inherently less secure, as the probability of collisions increases and the elliptic curve discrete logarithm problem becomes much simpler. However, an elliptic curve operations are more computationally expensive on elliptic curves with larger orders. While, there an infinite number of elliptic curve candidates to use, Burns et al. recommend using secure and peer-reviewed elliptic curves when implementing cryptographic protocols. In our construction, we use the NIST elliptic curve P-256 defined in [13]. Alternatives to these NIST approved elliptic curves include Edward's curves and Inverted Edward's curves. [5] Edward's curves allow for more time and space efficient elliptic curve operations,

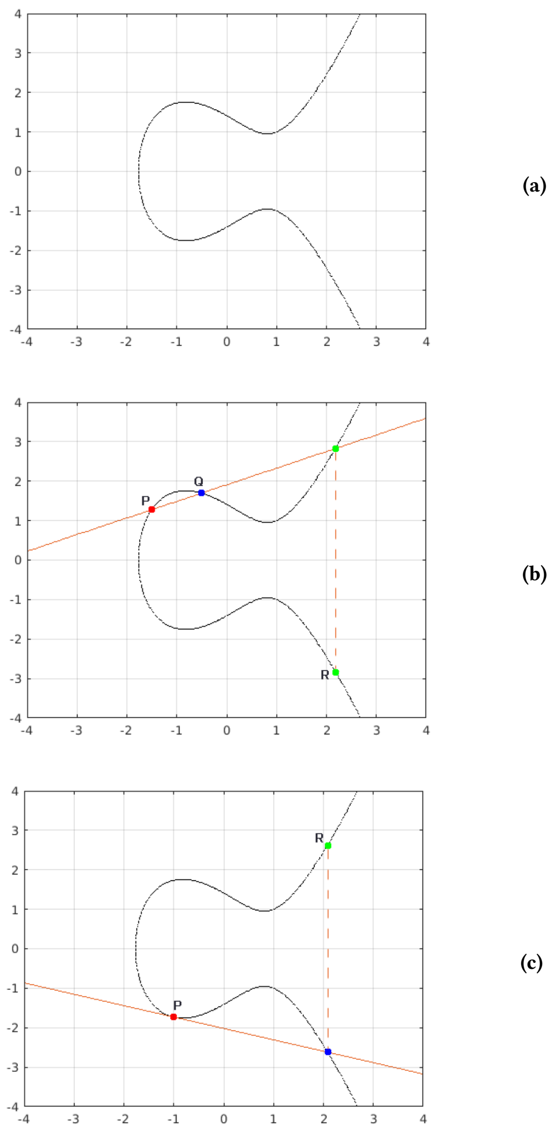

Figure 1: Elliptic curve operation visuals.

such as scalar multiplication. These special forms of elliptic curves can be used when space and time are a major priority of the system.

3.2 Construction

In our construction of the PASTA protocol, we use elliptic curve NIST-256 for our elliptic curve pseudorandom function. The parameters for NIST-256 [13] can be found in Figure 2. Following the work of Burns et al., we construct our EC-OPRF protocol with four steps below. Note that these four steps are used in various functions of the PASTA protocol. A full description of these functions is given in section 4 of this paper.

Name	Description	Value
a	Coefficient	`-0x00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000003`
b	Coefficient	`0x5ac635d8 aa3a93e7 b3ebbd55 769886bc 651d06b0 cc53b0f6 3bce3c3e 27d2604b`
p	Prime Modulus	`0xFFFFFFFF 00000001 00000000 00000000 00000000 FFFFFFFF FFFFFFFF FFFFFFFF`
n	Order	`0xFFFFFFFF 00000000 FFFFFFFF FFFFFFFF BCE6FAAD A7179E84 F3B9CAC2 FC632551`
G_x	Base Point x	`0x6b17d1f2 e12c4247 f8bce6e5 63a440f2 77037d81 2deb33a0 f4a13945 d898c296`
G_y	Base Point y	`0x4fe342e2 fe1a7f9b 8ee7eb4a 7c0f9e16 2bce3357 6b315ece cbb64068 37bf51f5`

Figure 2: NIST P-256 Elliptic Curve $y^2 = x^3 + ax + b$ over $GF(p)$.[13]

1. (Hash) When the user enters their password, the client hashes their input using the SHA-256 hash function, H.

$$g = H(\texttt{password})$$

After hashing the password, the client maps the hash of the password, g, onto a point on the elliptic curve using the "Try-and-Increment" function [11], E, developed by Icart.

$$pwd = E(g)$$

The client sends pwd to the authentication server. This step takes place when the user enters their password for the **SignUp** and the **Request** functions of the PASTA protocol. During the **SignUp** phase, the server generates a random key k and computes the scalar multiplication of k on pwd after receipt of pwd from the client.

$$h = k \cdot pwd$$

The server then stores a tuple containing k and h in the **Store** function corresponding to the user.

2. (Mask) In the **Request** function, after the user's inputted password has been hash and mapped as described in step 1 above, the client generates a random masking value ρ. The client then computes the scalar multiplication of ρ on pwd.

$$c = \rho \cdot pwd$$

The client sends c to the authenticating server.

3. (Salt) When a client attempts to authenticate themselves, they send their user ID and c value, as constructed in step 2 above. The server looks up the k that corresponds to the user and computes the scalar multiplication of k on c.

$$z = k \cdot c$$

The server returns z back to the client. Note that this step takes part in the **Respond** function of the PASTA protocol.

4. (Unmask) After receiving z back from the authenticating server, the client computes the modular inverse of the randomly generated value ρ in step 2. The modulus is the order of the elliptic curve. Then the client computes the scalar multiplication of the inverse of ρ on z.

$$h' = \rho^{-1} \cdot z = \rho^{-1} \cdot k \cdot \rho \cdot pwd = k \cdot pwd = h$$

This step takes part in the **Finalize** step.

Our protocol gives the client and server a shared secret value h, while the client never revealed the user's password(except during the **SignUp** step) and the server never revealed its secret value k. Thus, our protocol achieves the obliviousness property.

3.3 Example

For concreteness, let us consider an example in which the user's password is `Password123!`. As by design, both the client and the server have agreed to use elliptic curve NIST P-256. Let us assume that the user has already registered with the authentication server and the credentials they give are legitimate. The protocol proceeds as follows.

1. (Hash) We have assumed that the user has already registered with the authentication server. Note that the user sends the hash of their password to the server during the registration phase. Using the SHA-256 hash function, the client computes

$$\begin{aligned} &H('\texttt{Password123!}') = \\ &= \texttt{0xA109E369 47AD56DE 1DCA1CC4 9F0EF8AC} \\ &\qquad \texttt{9AD9A7B1 AA0DF41F B3C4CB73 C1FF01EA}. \end{aligned}$$

Next, the client maps the hash of their password computed above onto the elliptic curve. Using the "Try-and-Increment" algorithm, we map the hashed password to the following point.

$$\begin{aligned} pwd &= E(H('\texttt{Password123!}')) \\ &= (\texttt{0xA109E369 47AD56DE 1DCA1CC4 9F0EF8AC} \\ &\qquad \texttt{9AD9A7B1 AA0DF41F B3C4CB73 C1FF01EA}, \\ &\quad \texttt{0x1E358B0D C4FD8D50 53C5AD9D 1A894495} \\ &\qquad \texttt{E73A6ECA 76F22EF5 99FFB198 0073F604}). \end{aligned}$$

The client sends this elliptic curve point corresponding to their password to the server. Upon receipt of the hashed password, the server generates the random value k to be

$$\begin{aligned} k &= \texttt{0xB509A006 8DDB35EC BA0370FA A74618C8} \\ &\qquad \texttt{71F26B07 5249D4F2 F7DDD23C B2A34F5A}. \end{aligned}$$

Finally, the server computes the scalar multiplication of k on pwd as follows.

$$\begin{aligned} h &= k \cdot pwd \\ &= (\texttt{0x99F790F1 06A1AD22 02DF460A DA4C1F82} \\ &\qquad \texttt{9AEA00A4 51048784 D0B2F24E 4E186045}, \\ &\quad \texttt{0xB9A84FF7 36E75372 6EE76418 221C1631} \\ &\qquad \texttt{8C3685FB 035AD360 CC752BBC A39922E0}). \end{aligned}$$

2. (Mask) When the client wants to authenticate themselves after registering, they hash and map their password as done in step 1 above. However, before sending pwd to the server,

they randomly generate ρ

$$\rho = \texttt{0x7BBF35CE 4A4AACE2 AEB32B98 2E691D38 8A8933EB 71A7EAB1 2128EC52 0DEFCB35}$$

and compute the scalar multiplication of ρ on pwd.

$$\begin{aligned} c &= \rho \cdot pwd \\ &= (\texttt{0xA073D5E6 C6B55AC9 3126B8F3 51916346 A69493A8 FD9ECC0F 5FE1035A C3F7FCA7}, \\ &\quad \texttt{0x0F838E66 766B85A7 34856046 9639A13D 8C4D281B 2DC2BC0F A6AD65B2 228E7234}). \end{aligned}$$

As per the protocol, the client sends their username and c to the authentication server.

3. (Salt) Upon receipt of the client's authentication request, the server retrieves the value k, randomly generated in step 1, corresponding to the username. The server then computes the scalar multiplication of k on c and returns the result to the client.

$$\begin{aligned} z &= k \cdot c \\ &= (\texttt{0x5CE87382 F20B59F7 03022EA6 BCAB6746 0D0F8EF5 C0E28A8E AD143EA4 1A85CF54}, \\ &\quad \texttt{0x2539A4D9 153CD18D 8ADFEE22 1A3473F7 3333B5EC 28032EFB 7172254B CCFE9735}) \end{aligned}$$

4. (Unmask) After receiving z back from the server, the client computes the inverse of ρ, randomly generated in step 2, mod the order of the elliptic curve.

$$\rho^{-1} = \texttt{0x3AFACF8 A05AD534 6B60A746 92E9BD9F 1DE4AC03 A792996C3 B673DD12 A57BAE2B}$$

Finally, the client computes the scalar multiplication of ρ^{-1} on z.

$$\begin{aligned} h &= \rho^{-1} \cdot z \\ &= (\texttt{0x99F790F1 06A1AD22 02DF460A DA4C1F82 9AEA00A4 51048784 D0B2F24E 4E186045}, \\ &\quad \texttt{0xB9A84FF7 36E75372 6EE76418 221C1631 8C3685FB 035AD360 CC752BBC A39922E0}) \end{aligned}$$

Note that the h computed by the server in step 1 and the h computed by the client in step 4 are the same, even though the client never revealed the password(after registering) and the server never revealed their secret value k. Hence, the client and the server have a shared secret.

4 PROTOCOL CONSTRUCTION

4.1 Formal Protocol

Protocol 1 PASTA with Elliptic Curve TOP

Procedures. TTG, TOP as discussed above in addition to SKE, an AES EAX symmetric key encryption scheme and H, a SHA256 hash function.

The protocol:

```
(1) Setup $(k, \ell, n, t) \mapsto ([sk], vk, pp)$
    – $([sk], vk, tpp) \leftarrow$ TTG.Setup$(k, n, t)$
    – $\Delta := n!$
    – $curve \leftarrow$ TOP.Setup()
    – $pp \leftarrow (k, n, t, \Delta, tpp)$
    – $x \in_R \{0,1\}^\ell$
(2) SignUp (username, pwd, $pp) \mapsto msg_i$
    – $C \leftarrow$ H(username)
    – $P \leftarrow$ H(pwd)
    – $gp \leftarrow$ TOP.GetPoint$(P)$
    – $h \leftarrow$ TOP.Encode$(k_i, gp)$
    – $msg_i \leftarrow$ H$(h_x||h_y||i)$
(3) Store$(C, i, msg_i) =: \textsc{Rec}_{i,C}$
    – $\textsc{Rec}_{i,C} := msg_i$
(4) Request $(C, pwd, \mathcal{T}, pp) \mapsto$ st, $(C, x, y)_{i\in\mathcal{T}}$
    – If $|\mathcal{T}| < t$, output $\textsc{False}$
    – $\rho \in_R \{1, \cdots, curve.n\}$
    – st $\leftarrow (C, pwd, \mathcal{T}, \rho)$
    – $P \leftarrow$ H(pwd)
    – $gp \leftarrow$ TOP.GetPoint$(P)$
    – $req_i \leftarrow (C,$ TOP.Encode$(\rho, gp))$
(5) Respond $(i, sk_i, C, x, req_i, vk, pp) \mapsto (z_i, ct)$
    – Parse $C, req$ from $req_i$
    – $h_i \leftarrow \textsc{Rec}_{i,C}$
    – $z_i \leftarrow$ TOP.Encode$(k_i, req)$
    – $y_i \leftarrow$ TTG.PartEval$(sk_i, vk, C||x, tpp)$
    – $ct \leftarrow$ SKE.Encrypt$(h_i, y_i)$
    – $res_i \leftarrow (z_i, ct)$
(6) Finalize $(st, \{res_i\}_{i\in\mathcal{T}}, pp, x) \mapsto tk$
    – Parse $(C, pwd, \mathcal{T}, \rho)$ from st
    – Parse $(z_i, ct)$ from $res_i$
    – $h \leftarrow$ TOP.Encode$(\rho^{-1}, z_i)$
    – For $i \in \mathcal{T} : h_i :=$ H$(h_x||h_y||i), y_i :=$ SKE.Decrypt$(h_i, ct)$
    – $tk \leftarrow$ TTG.Combine$(\{i, y_i\})_{i\in\mathcal{T}}$
(7) Verify $(vk, C, x, tk, pp) \mapsto 1/0$
    – return TTG.Verify$(vk, C||x, tk, tpp)$
```

4.2 Protocol Description

1. Setup generates the public parameters for future algorithms in addition to choosing the nonce x and creating the elliptic curve.

2. SignUp hashes the username and password supplied by the user and encodes the points on to the elliptic curve. It then generates a message to send to each server by hashing the point on the curve concatenated with the server index.

3. **Store** adds a client message to a database on the server.
4. **Request** takes place when a client wants to construct a token for the service provider. A client sends their client ID and password to a set of servers. The servers then generate a set of requests for use in the next step of the protocol.
5. **Respond** takes the requests created in (4) and creates token shares using the TTG algorithm. It then encrypts these token shares by using the hash created in (2) as the symmetric key. The servers then send the token shares back to the client.
6. **Finalize**: The client receives t token shares from the server and decrypts them by calculating the same hash from (2). Once decrypted, the client combines these shares into a token to send to the service provider.
7. **Verify**: The service provider can verify the token using the public key provided by the client.

5 EVALUATION

As mentioned in the Introduction, the PASTA protocol thwarts offline dictionary attacks, improving the security of a user's password, in the case an Identity Provider server is compromised. This is especially important given the number of users that these identity providers accommodate. If an attacker were to gain access to the server, the user passwords would not be easily cracked with a password cracker, such as John the Ripper [17]. Instead, the hashes stored directly on the server represent a hash of the user password, encoded onto the elliptic curve with a private key value, as shown below:

$$\mathsf{H}(h_x||h_y||i) \tag{1}$$

In this case, h_x, h_y represent the coordinates of the point on the elliptic curve that represents the hash of the user password, blinded with k_i, the server's private key. This ensures that each server only receives the already hashed value shown above, never directly interacting with the client's password. Traditionally, the client's password would be checked on the server side, with the server returning either success or failure based on the validity of a given password. In our protocol, the token share is encrypted using the hash from (1) as the key and then returned to the side of the client for verification. There are two significant benefits obtained by doing the check on the client side. The first, is that it reduces the number of rounds of interaction required during the sign on phase of the protocol which helps circumvent any bottlenecks caused by network latency. The second and most important is that it ensures that an impersonator will not be able to recreate the key from 1 as the password is unknown and thus be unable to recover the encrypted token share.

The metrics we collected demonstrate the effect of a careful choice of threshold. The graph shown in Figure 3 demonstrates the drastic difference in time from a threshold of $(9, 10)$ and one of $(99, 100)$. The threshold of $(99, 100)$ would never be realistically implemented, but it demonstrates how the protocol scales. The timing shown in Figure 3 also does not necessarily represent the actual run time of the protocol, as our implementation runs on a single machine. The more accurate, broken down timing for a single $(5, 10)$ threshold run is portrayed in Figure 5. The threshold level is important to consider for security due to the fact discussed earlier that in a (t, n) system, at most $t - 1$ servers can be compromised before the tokens can be forged. However, as (t, n) gets larger or as t approaches n, the number of computations needed on both the server and client side increases. However, our code was implemented in a single-threaded situation so distributing the task of the token generation in a large (t, n) situation can mitigate some of the effects on the servers, leaving their total computation more or less the same as with a smaller (t, n). The client will always need to assemble t token shares into a valid token, but in future implementations, this token assembly can be parallelized. The computations required by the client are independent Hash and Decryption functions, which are trivially parallelized, and the TTG.Combine function proposed by Shoup [20] could also be made parallel due to the fact that the bulk of the calculation involves finding a product over a explicit range. This improvement could mitigate the effects of increasing the (t, n) specifications, allowing for greater security with comparable computation time.

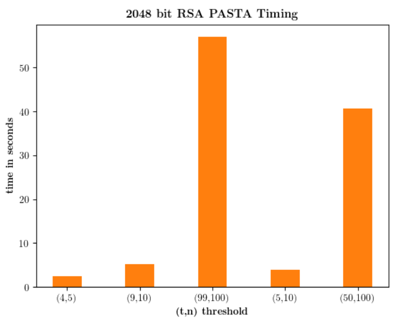

Figure 3: (t,n) Full PASTA Protocol Timing

Tot. Time	Dist. Time	Protocol	Location
2.010	2.010	**Setup**	client
0.271	0.027	**SignUp**	server
0.000	0.000	**Store**	server
0.136	0.027	**Request**	server
1.148	0.230	**Respond**	server
0.156	0.156	**Finalize**	client
0.001	0.001	**Verify**	client

Figure 4: Function Run Time with 2048 bit $(5, 10)$ Threshold

When considering computational intensity, it is important to break down the timing into the client computations and the server computations. The servers will likely be serving numerous clients at once, leading to a need for their required computations to be small. In the PASTA protocol we implemented, this requirement holds, as the work done by the server is distributed over the t servers. Figure 4 demonstrates the breakdown of which parts of the protocol are computed by the client and server. The *Dist. Time* column represents the time required 'per unit', e.g. the client is a single unit and thus does all of the work needed on the client end, but the servers in this implementation require either $\frac{1}{t}$ or $\frac{1}{n}$ of the computation, depending on the current stage of the protocol.

Additionally, much of the computationally intensive parts of the

Total Distributed Time	Location
0.284	server
2.167	client
2.451	**total**

Figure 5: Time Distributed over $(5, 10)$ **Threshold**

protocol, such as **Setup** are only required to establish the client, and will only be required to be re-computed when the token expires. Figure 5 demonstrates the total computation time needed for a (5, 10) threshold implementation with 2048 bit RSA keys. The distributed time refers to the average time needed on each of the t servers. These metrics also represent a *from scratch* generation time, including all of the initial setup for the client.

In addition to low computation requirements for each server, the data storage required per client in the PASTA protocol is very low. The server must store a single $\textsc{Rec}_{i,C}$ in the **Store** part of the protocol. This client record is simply the hash mentioned above, which is only 32 bytes due to the hash function used. This can be stored in a server database, indexed by the client hash, C, which comprises of another 32 byte quantity. Additionally, each server must store its secret key k_i, which is just a value between 1 and n: the order of the curve defined in Figure 2.

Another important aspect of our implementation to examine is that of development language(s). We chose to write the main PASTA protocol and testing suite in Python3 and our front end UI in JavaScript/HTML. Python3 and JavaScript are interpreted languages so at runtime the code must be reduced to machine instructions. Both languages also have automatic garbage collection, which can result in memory being unavailable even when it is no longer being used, since the programmer cannot free the memory at will. Furthermore since we wrote our solution in two languages, we had to first convert the user input to JSON then via AJAX send that JSON to our Flask app where it is read as input to our Python3 code, the process is reversed after the respective computations are made. Had we followed what Agrawal, et al. did in their implementation of PASTA and used a single language, C++, which is compiled and doesn't have automatic memory management, our implementation surely would have been more efficient.

Our main contribution extending the Agrawal et al.'s original work is the use of elliptic curves to implement oblivious pseudorandom functions. The elliptic curve based approach was used due to the smaller key size and computational efficiency advantage over Diffie-Hellman RSA base approaches. Kristan Lauter of Microsoft gives the following results in [1].

> At the 163-bit ECC/1024-bit RSA security level, an elliptic curve exponentiation for general curves over arbitrary prime fields is roughly 5 to 15 times as fast as an RSA private key operation, depending on the platform and optimizations. At the 256-bit ECC/3072-bit RSA security level the ratio has already increased to between 20 and 60, depending on optimizations. To secure a 256-bit AES key, ECC-521 can be expected to be on average 400 times faster than 15,360-bit RSA.

As analytically measured by Lauter, elliptic curve cryptography has tremendous efficiency advantages over RSA based approaches. Hence, using an elliptic curve based approach could increase the security and reduce the overhead of adding the PASTA scheme to an authentication system.

6 SUMMARY AND CONCLUSION

Although we were unable to produce comparable protocol running times, as compared to the original solution developed Agrawal et al., we are confident that these discrepancies are due to the differences in implementation. The biggest factor is that we developed out system using interpreted languages, such as JavaScript and Python 3, where as Agrawal et al. developed their solution in C++. Our implementation is inherently slower due to the nature of these different approaches. Another important factor in the timing discrepancies between our solutions is the fact that we used the RSA based digital signature to construct the threshold token generation scheme. As supported by the evaluation of Agrawal et al. in their original paper, the RSA-based approach was slower than the other three alternatives by an order of magnitude.

Implementation details aside, we believe that our approach using elliptic curve oblivious pseudorandom functions is more efficient computationally than when using the Diffie-Hellman based approach used in [2]. We are interested in further testing our elliptic curve based approach using a more computationally efficient framework. Then a better comparison can be made between our elliptic curve based approach and the Diffie-Hellman base approach.

A major hole in [2] is the assumption that an adversary cannot interfere with the registration phase of the PASTA protocol. If an adversary is allowed to read messages sent between clients and servers during the registration phase, they would receive a hash of the user's password. They could then perform an offline dictionary attack. However, the secret keys would still be protected. No defense against this type of attack is mentioned in [2].

As the PASTA protocol attempts to solve the issue of an adversary forging arbitrary tokens and performing offline dictionary attacks on the hash of user passwords, further research on this protocol should be focused on developing a protocol to allow an adversary to have control of a server during the authentication phase. If an adversary is able to compromise a server, it is highly likely that a new user will attempt to register themselves with the server is under the control of the adversary. It is foolish to think that no users will ever register with a server that has been compromised. Hence, this assumption must be taken into consideration in future research.

Regardless of the attack an adversary can carryout during the registration phase, we recommend that token-based authentication frameworks transition from single-server authentication to the multi-server password-based threshold authentication scheme proposed in Agrawal et al.'s work. The overhead in implementing this protocol is minimal. As stated in [2],

> In the WAN network, since the most time-consuming component is the network latency in our protocol as well as the nae solutions, the overhead of our solution compared with the naïve solutions is fairly small. As

shown in Figure 10, the overhead is less than 5% in all the settings and all token types.

Adding a 5% overhead to the authentication computation is a minuscule amount, while security is greatly enhanced. Even smaller overheads could be achieved with an elliptic curve based approach to the oblivious pseudorandom function. It is important to note that this overhead will only be incurred when registering and requesting a token. After a token has been requested and issued, there is no additional overhead. With these factors taken into account, the authors of this paper recommend using the PASTA framework to improve the security of token-based systems.

REFERENCES

[1] [n. d.]. ([n. d.]).

[2] Shashank Agrawal, Peihan Miao, Payman Mohassel, and Pratyay Mukherjee. 2018. PASTA: PASsword-based Threshold Authentication. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 2042–2059. https://doi.org/10.1145/3243734.3243839

[3] Amazon. 2019. Open ID Connect Providers (Identity Pools). (2019). https://docs.aws.amazon.com/cognito/latest/developerguide/open-id.html

[4] Mihir Bellare and Phillip Rogaway. 1996. The Exact Security of Digital Signatures-how to Sign with RSA and Rabin. In *Proceedings of the 15th Annual International Conference on Theory and Application of Cryptographic Techniques (EUROCRYPT'96)*. Springer-Verlag, Berlin, Heidelberg, 399–416. http://dl.acm.org/citation.cfm?id=1754495.1754541

[5] Daniel J. Bernstein and Tanja Lange. 2007. Faster Addition and Doubling on Elliptic Curves. In *Proceedings of the Advances in Crypotology 13th International Conference on Theory and Application of Cryptology and Information Security (ASIACRYPT'07)*. Springer-Verlag, Berlin, Heidelberg, 29–50. http://dl.acm.org/citation.cfm?id=1781454.1781458

[6] Alexandra Boldyreva. 2003. Threshold Signatures, Multisignatures and Blind Signatures Based on the Gap-Diffie-Hellman-Group Signature Scheme. In *Proceedings of the 6th International Workshop on Theory and Practice in Public Key Cryptography: Public Key Cryptography (PKC '03)*. Springer-Verlag, London, UK, UK, 31–46. http://dl.acm.org/citation.cfm?id=648120.747061

[7] Jonathan Burns, Daniel Moore, Katrina Ray, Ryan Speers, and Brian Vohaska. 2017. EC-OPRF: Oblivious Pseudorandom Functions using Elliptic Curves. *IACR Cryptology ePrint Archive* 2017 (2017), 111. http://eprint.iacr.org/2017/111

[8] Morris J. Dworkin. 2015. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. *Federal Inf. Process. Stds. (NIST FIPS)* 202 (aug 2015).

[9] Facebook. 2019. Facebook Login - Documentation. (2019). https://developers.facebook.com/docs/facebook-login

[10] Google. 2019. OpenID Connect | Google Identity Platform | Google Developers. (2019). https://developers.google.com/identity/protocols/OpenIDConnect

[11] Thomas Icart. 2009. How to Hash into Elliptic Curves. In *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*. 303–316. https://doi.org/10.1007/978-3-642-03356-8_18

[12] Stanis Jarecki and Xiaomin Liu. 2009. Efficient Oblivious Pseudorandom Function with Applications to Adaptive OT and Secure Computation of Set Intersection. In *Proceedings of the 6th Theory of Cryptography Conference on Theory of Cryptography (TCC '09)*. Springer-Verlag, Berlin, Heidelberg, 577–594. https://doi.org/10.1007/978-3-642-00457-5_34

[13] Cameron F. Kerry, Acting Secretary, and Charles Romine Director. 2013. FIPS PUB 186-4 FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION Digital Signature Standard (DSS). (2013).

[14] Kristin E. Lauter and Katherine E. Stange. 2009. The Elliptic Curve Discrete Logarithm Problem and Equivalent Hard Problems for Elliptic Divisibility Sequences. In *Selected Areas in Cryptography*, Roberto Maria Avanzi, Liam Keliher, and Francesco Sica (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 309–327.

[15] Moni Naor, Benny Pinkas, and Omer Reingold. 1999. Distributed Pseudo-random Functions and KDCs. In *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques (EUROCRYPT'99)*. Springer-Verlag, Berlin, Heidelberg, 327–346. http://dl.acm.org/citation.cfm?id=1756123.1756155

[16] OAuth. 2019. OAuth Community Site. (2019). https://oauth.net/

[17] Openwall. 2019. John the Ripper password cracker. (2019). https://www.openwall.com/john/

[18] R. L. Rivest, A. Shamir, and L. Adleman. 1978. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Commun. ACM* 21, 2 (Feb. 1978), 120–126. https://doi.org/10.1145/359340.359342

[19] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* 22, 11 (Nov. 1979), 612–613. https://doi.org/10.1145/359168.359176

[20] Victor Shoup. 2000. Practical Threshold Signatures. In *Proceedings of the 19th International Conference on Theory and Application of Cryptographic Techniques (EUROCRYPT'00)*. Springer-Verlag, Berlin, Heidelberg, 207–220. http://dl.acm.org/citation.cfm?id=1756169.1756190