EXPLORING PASSWORD-AUTHENTICATED KEY-EXCHANGE ALGORITHMS

A project report submitted to The University of Manchester
for the degree of Bachelors of Science
in the Faculty of Science and Engineering

2023

Sam Leonard
f41751sl
Supervisor: Professor Bernardo Magri

Department of Computer Science

Declaration

No portion of the work referred to in this project report has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Abstract

Exploring Password-Authenticated Key-Exchange Algorithms

Sam Leonard, Supervisor: Professor Bernardo Magri

Password-Authenticated Key-Exchange (PAKE) algorithms are a niche kind of cryptography where parties seek to establish a strong shared key, from a low entropy secret such as a password. This makes it particularly attractive to some domains, such as Industrial Internet of Things (IIOT). However many PAKE algorithms are unsuitable for Internet of Things (IOT) applications, due to their heavy computational requirements. Augmented Composable Password Authenticated Connection Establishment (AuCPace) is a new PAKE protocol which aims to make PAKEs accessible to IIOT by utilising Elliptic Curve Cryptography (ECC), Verifier based PAKEs (V-PAKEs) and a novel augmented approach. This project aims to provide an approachable and developer-focused implementation of AuCPace in Rust and to contribute this implementation back to RustCrypto to promote wider adoption of PAKE algorithms.

Acknowledgements

I would like to thank everyone at absolutely wonderful supervisor, RustCrypto, Crypto Hack and my CTF Team (0rganizers). Without your help none of this would have been possible.

Contents

CHAPTER 1

Context

1.1 Background on PAKEs

1.1.1 What problem do PAKEs solve?

In the modern day when you login to a website a Transport Layer Security (TLS) encrypted channel is established, over which your password is sent in plaintext. The server then computes some hash of your password, sometimes with a salt or some other additional data. This hash is then compared with whatever is stored in the server's database for your account, and access is granted based on whether the hash matched.
This approach is fundamentally flawed, allowing the plaintext password to leave your device gives attackers many more opportunities at which they can steal your password. Be that from compromising the TLS channel via a downgrade attack or by malware on the server intercepting your password as the server processes it.
PAKEs solve this problem in a fundamentally different way, and they have many benefits because of this. Namely with PAKEs your password never leaves your device, it is used to calculate a secret key which is shared with the server. Another property of how PAKEs are constructed is that you and the server are both "authenticated" with each other once you acquire the shared key. This is in contrast with the approach of TLS + certificates, where only the server is authenticated.

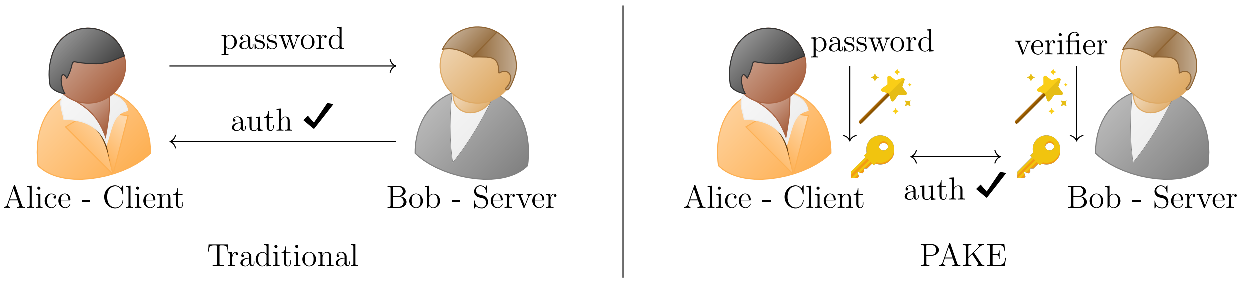


Figure 1.1: An illustration of the difference traditional password based authnetication and PAKEs

1.1.2 What is a PAKE?

PAKEs are interactive, two party cryptographic protocols where each party shares knowledge of a password (a low entropy secret) and seeks to obtain a strong shared key e.g. for use later with a symmetric cipher. Critically an eavesdropper who can listen in two all messages of the key negotiation cannot learn enough information to bruteforce the password. Another way of phrasing this is that brute force attacks on the key must be "online".
There are two main types of PAKE algorithm - Augmented PAKEs and Balanced PAKEs.

- Balanced PAKEs are PAKEs where both parties share knowledge of the same secret password.
- Augmented PAKEs are PAKEs where one party has the password and the other has a "verifier" which is computed via a one-way function from the secret password.

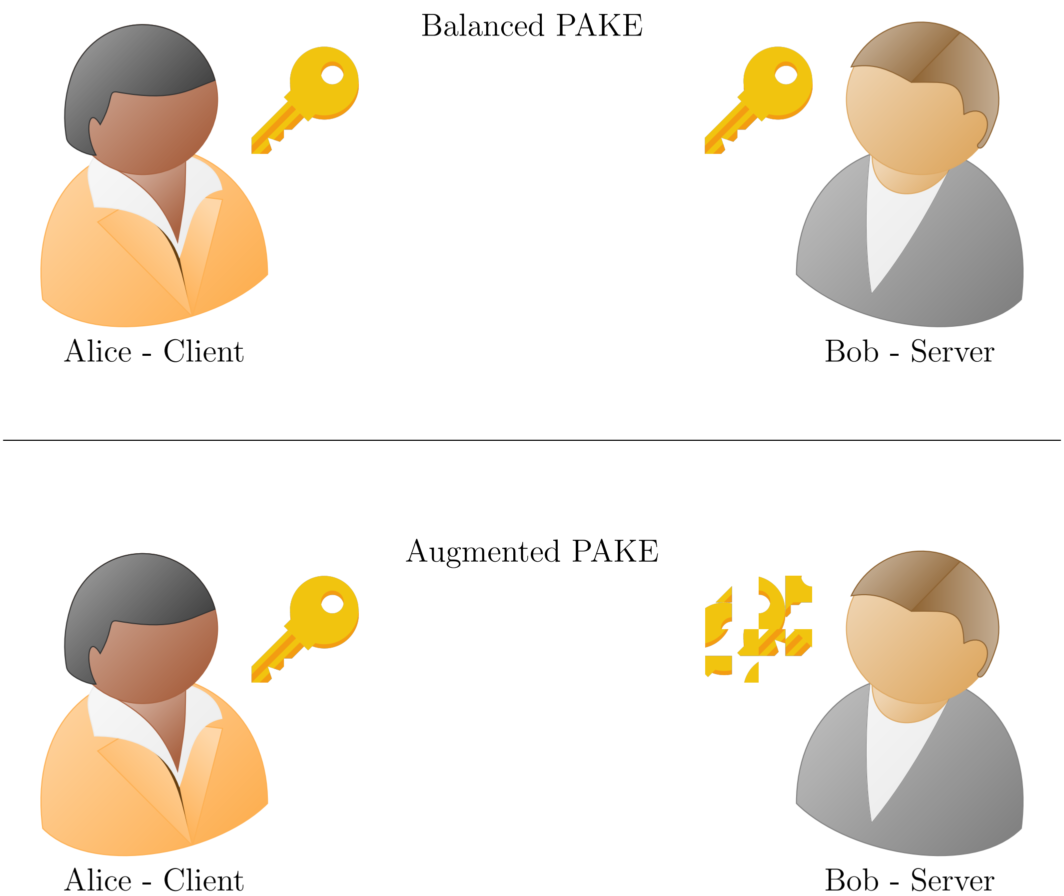


Figure 1.2: An illustration of the difference between PAKEs and tranditional password-based authentication

1.1.3 A brief history of PAKE algorithms

The first PAKE algorithm was Bellovin and Merritt's Encrypted Key Exchange (EKE) scheme [BM92]. It works using a mix of Symmetric Cryptography and Asymmetric Cryptography to perform a key exchange. This comes with many challenges and subtle mistakes that are easy to make; primarily for the security of the system whatever is encrypted by the shared secret key(P) must be indistinguishable from random data. Otherwise an attacker can determine whether their guess at a trial decryption is valid. The Rivest-Shamir-Adleman (RSA) variant of EKE has this issue - the RSA parameter e is what is encrypted and sent in the first message. For RSA all valid values of e are odd, so this would prevent it being used. This is solved by adding 1 to e with a 50% chance. Figure 1.3 shows this protocol in full. While many of the initial variants on EKE have been shown to flawed/vulnerable, later variants have made it into real world use, such as in Extensible Authentication Protocol (EAP) [Vol+04] where it is available as EAP-EKE [She+11]. In appendix A you can find a Python implementation of this scheme.

An Aside on Notation

- $\leftarrow$: Assignment - $x \leftarrow\ 5$ means x is assigned a value of 5.
- $\leftarrow\$$: Sampling from a given set - $x \leftarrow\$\ \mathbb{R}$ means to choose x at random from the set of real numbers.

Table 1.1: EKE shared parameters

Shared Parameter	Secret	Explanation
P	yes	the shared password

EKE-RSA

Alice		**Bob**
$Ea \leftarrow (e, n)$		
$b \leftarrow\$ \{0, 1\}$	$\xrightarrow{A, P(e+b), n}$	$Ea \leftarrow (e, n)$
$challenge_A \leftarrow\$ \mathbb{Z}_n$	$\xleftarrow{P(Ea(R))}$	$R \leftarrow\$$ Keyspace
	$\xrightarrow{R(challenge_A)}$	$challenge_B \leftarrow\$ \mathbb{Z}_n$
verify $challenge_A$	$\xleftarrow{R(challenge_A, challenge_B)}$	
	$\xrightarrow{R(challenge_B)}$	verify $challenge_B$

Figure 1.3: Implementing EKE using RSA

SPAKE

SPAKE1 and SPAKE2 are Balanced PAKEs' which were introduced slightly later on by Michel Abdalla and David Pointcheval [AP05] as variations on EKE. They are very similar so we will just explore SPAKE2 as we are more interested in online algorithms. Simple PAKE (SPAKE) differs from EKE in the following ways:

1. The encryption function is replaced by a simple one-time pad.
2. The Asymmetric Cryptography is provided by Diffie-Hellman (DH)
3. There is no explicit mutual authentication phase where challenges are exchanged. This has the advantage of reducing the number of messages that need to be sent.

Table 1.2: SPAKE shared parameters

Shared Parameter	Secret	Explanation
pw	yes	the shared password encoded as an element of $\mathbb{Z}_p$
$\mathbb{G}$	no	the mathematical group in which we will perform all opertions
g	no	the generator of $\mathbb{G}$
p	no	the safe prime which defines the finite field for all operations in $\mathbb{G}$
M	no	an element in $\mathbb{G}$ associated with user A
N	no	an element in $\mathbb{G}$ associated with user B
H	no	a secure hash function

SPAKE2

Alice		**Bob**
$x \xleftarrow{\$} \mathbb{Z}_p$		$y \xleftarrow{\$} \mathbb{Z}_p$
$X \leftarrow g^x$		$Y \leftarrow g^y$
$X^* \leftarrow X \cdot M^{pw}$		$Y^* \leftarrow X \cdot N^{pw}$
	$\xrightarrow{X^*}$	
	$\xleftarrow{Y^*}$	
$K_A \leftarrow (Y^*/N^{pw})^x$		$K_B \leftarrow (X^*/M^{pw})^y$
$SK_A \leftarrow H(A, B, X^*, Y^*, Ka)$		$SK_B \leftarrow H(A, B, X^*, Y^*, Kb)$

Figure 1.4: SPAKE2 Protocol

SRP

Finally we will look at Secure Remote Password (SRP) an Augmented PAKE first published in 1998, unlike SPAKE2 it is not a modification of EKE. SRP has gone through many revisions, at time of writing SRP6a is the latest version. SRP is likely the most used PAKE protocol in the world due to it's use in Apple's iCloud Keychain [Sec21] and it's availability as a TLS ciphersuite [Wu+07]. However it is quite weird for what it does and there is no security proof for it [Gre18]. An implementation of the protocol in Python can be found in appendix B.

Table 1.3: SRP parameters

Parameter	Secret	Explanation
v	yes	the verifier stored by the server: $v = g^{H(s,I,P)}$
P	yes	the user's password
I	no	the user's name
g	no	the generator of $\mathbb{G}$
p	no	the safe prime which defines the finite field for all operations in $\mathbb{G}$
H	no	a secure hash function

SRP

Alice		**Bob**
$a \leftarrow\!\!\$\ \{1 \ldots n-1\}$	$\xrightarrow{I}$	$s, v \leftarrow \text{lookup}(I)$
$x \leftarrow H(s, I, P)$	$\xleftarrow{s}$	$b \leftarrow\!\!\$\ \{1 \ldots n-1\}$
$A \leftarrow g^a$	$\xrightarrow{A}$	$B \leftarrow 3v + g^b$
$u \leftarrow H(A, B)$	$\xleftarrow{B}$	$u \leftarrow H(A, B)$
$S \leftarrow (B - 3g^x)^{a+ux}$		$S \leftarrow (Av^u)^b$
$M_1 \leftarrow H(A, B, S)$	$\xrightarrow{M_1}$	verify M_1
verify M_2	$\xleftarrow{M_2}$	$M_2 \leftarrow H(A, M_1, S)$
$K \leftarrow H(s)$		$K \leftarrow H(S)$

Figure 1.5: SRP-6 Protocol

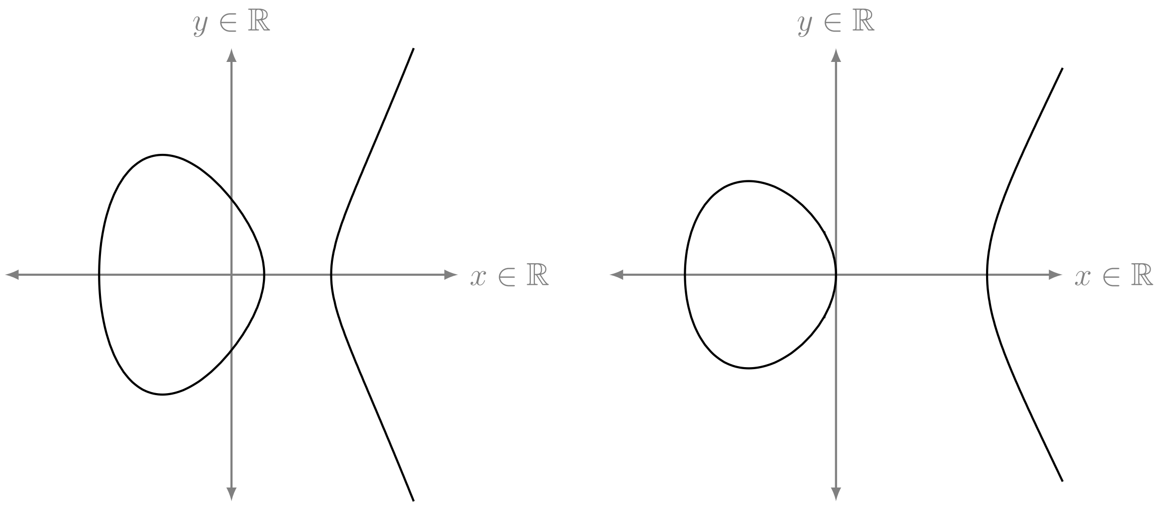
1.2 Elliptic Curve Cryptography

Many modern Cryptograhpic protocols make use of a mathematical object known as an elliptic curve. First proposed in 1985 independently by Neal Koblitz [Kob87] and Victor S. Miller [Mil86]. Elliptic curves are attractive to cryptographers as they maintain a very high level of strength at smaller key sizes, this allows for protocols to consume less bandwidth, less memory and execute faster [KMV00]. To illustrate just how great the size savings are - National Institute of Standards and Technology (NIST) suggests that an elliptic curve key of just 256 bits provides the same level of security as an RSA key of 3072 bits [ST20].

1.2.1 But what actually is an elliptic curve?

With regards to Cryptography elliptic curves tend to come in one of two forms:

- Short Weierstraß Form: $y^2 = x^3 + ax + b$
- Montgomery Form: $by^2 = x^3 + ax^2 + x$



$y^2 = x^3 - 2x - 1$ over $\mathbb{R}$

Short Weierstraß Form

$2y^2 = x^3 - 3x^2 - x$ over $\mathbb{R}$

Montgomery Form

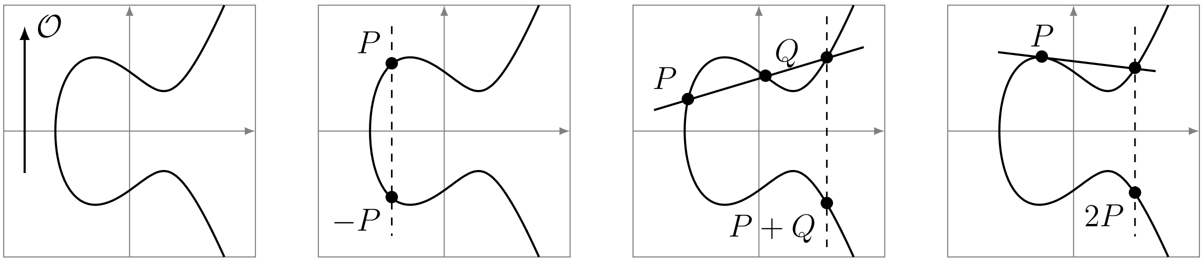
Figure 1.6: Elliptic curves over $\mathbb{R}$, Adapted from TikZ for Cryptographers [Jea16]

Weierstraß form is special as it is the general case for all elliptic curves, meaning all elliptic curves can be expressed as a Weierstraß curve. This property means that it is commonly used for expressing various curves. Montgomery form isn't quite as flexible, however it is favourable because it leads to significantly faster multiplication and addition operations via Montgomery's ladder [BL17].

1.2.2 How do we do Cryptography with curves?

To perform Cryptography with elliptic curves we need to define an "Abelian Group" to work in. An Abelian Group is a group whose group operation is also commutative, for

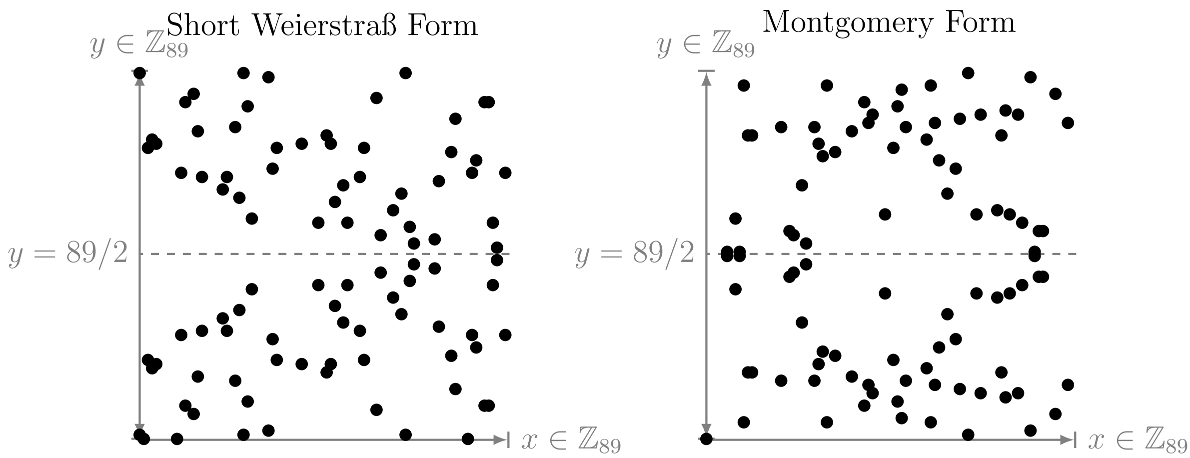
example the addition operator over the integers: $(+, \mathbb{Z})$ is an Abelian Group. Abelian Groups form the basis of many modern Cryptographic algorithms, a DH key exchange can be performed in any Abelian Group for instance.
Our Abelian Group is built on the idea of "adding" points on the curve. To add two points, we find the line which passes through our two points and we continue along that line until we hit our curve again. We then reflect this point in the x-axis to get our result. What if we want to add our point to itself? Now there isn't a unique line through one point, however we are making the rules so in this case we will take the tangent to the curve at that point and then we can treat it the same as before. What if our line doesn't intersect with the curve? In this case we define a new point called the "neutral element" - $\mathcal{O}$. It is also called the point at infinity as it can be considered to be the single point at the end of every vertical line at infinity. Figure 1.7 illustrates all of these rules and edge cases.



Neutral element $\mathcal{O}$ | Inverse element $-P$ | Addition $P+Q$ "Chord rule" | Doubling $P+P$ "Tangent rule"

Figure 1.7: Elliptic Curve Group Operations, reproduced from TikZ for Cryptographers [Jea16]

However it's not quite that simple for us. We cannot use $\mathbb{R}$ as computers only have finite resources we need a finite set to work in. Instead we define our operations over a Finite Field, we will use the Finite Field of the integers mod a prime, denoted $\mathbb{Z}_p$ for some prime p. Lets take a look at what our finite fields look like in a small finite field - $\mathbb{Z}_{89}$.



$y^2 = x^3 - 2x + 1$ over $\mathbb{Z}_{89}$ | $2y^2 = x^3 - 3x^2 - x$ over $\mathbb{Z}_{89}$

Figure 1.8: Elliptic curves over $\mathbb{Z}_{89}$, Adapted from TikZ for Cryptographers [Jea16]

This now looks very different to when we were looking at them in $\mathbb{R}$, however it shows very clearly what the elements of our set look like. They are points in the 2d coordinate plane with a symmetry around $p/2$. This might not feel intuitive but it is actually exactly what we should expect to happen, in our finite field when we negate our point's y coordinate, instead of flipping it around the y-axis, our points get wrapped around $y = p$. Hence our new point is the same distance from $y = p$ as our first point was with $y = 0$, this is where our symmetry arises.

1.2.3 Where can Elliptic Curve Cryptography go wrong?

There are many attacks against various aspects of Elliptic Curves, in general they fall into the following categories:

- Attacks against the Elliptic Curve Discrete Logarithm Problem (ECDLP) security of the curve:
  - The rho method [Pol78]
  - Transfer Security [MVO91; Sem98]
  - CM Field Discriminants [BL]
  - Curve Rigidity [BL13]
- Attacks against the concrete implementation of ECC:
  - Ladders required for safe and fast point-scalar multiplication [BL]
  - Twist Security [LL97; BMM00]
  - Completeness [IT02]
  - Indistinguishability [Ber+13]

All of these attacks individually can weaken or even break the security of a given cryptosystem if not accounted for. However choosing the right curve is a good step in the right direction and can mitigate many of the attacks listed above.

1.3 Modern PAKEs

Recently many novel PAKEs algorithms have been published, this is partly due to a request from the Internet Engineering Task Force (IETF) for the Internet Research Task Force (IRTF) Crypto Forum Research Group (CFRG) to carry out a selection process to choose a PAKE for usage in IETF protocols. That process concluded in 2020 with Composable Password Authenticated Connection Establishment (CPace) and OPAQUE being chosen as the recommended Balanced PAKE and Augmented PAKE respectively [Cha20].
Some time has passed since this process and we now have some new PAKEs with various interesting properties that are worth taking a look at.

1.3.1 CHIP+CRISP

Introduced in 2020 by Cremers et al., CHIP and CRISP are two protocol compilers which instanciate what the authors call an identity-binding PAKE (iPAKE) [Cre+20]. iPAKEs are designed to mitigate the threat of compromising the local storage of a device. While in the case of Augmented PAKEs this is considered for the server side, Balanced PAKEs often require both parties to have knowledge of the plaintext password. Examples of this include SPAKE-2 [AP05] and WPA3's DragonFly/SAE [Har08], both of which require the server and client to have knowledge of the plaintext password. CHIP+CRISP solves this problem by binding the password to an arbitrary bit-string called the identity, this can be anything, e.g. "server", "router", "jonathandata0". CHIP and CRISP are both protocol compilers, this means that they aren't themselves protocols but they sit on top of another protocol in order to give that sub-protocol the aforementioned properties, by protecting the underlying data they exchange.

1.3.2 KHAPE

Key-Hiding Asymmetric PAKE (KHAPE) [GJK21] is an Augmented PAKE from the designers of OPAQUE, introduced in 2021 it is a variant of OPAQUE which doesn't rely on the use of an Oblivious Pseudo Random Function (OPRF) to get Augmented PAKE security. Instead the OPRF is used to add precomputation resistance, this is also known as a "strong" PAKE. The advantage of this is that should the OPRF be compromised the protocol remains secure, and only loses the "strong" part of it's security. Similarly to CHIP+CRISP, KHAPE is not itself a protocol but a compiler from any Authenticated Key-Exchange (AKE) to an Augmented PAKE. In the paper they detail the concrete implementation KHAPE-HMQV, which extends the HMQV Authenticated DH Protocol from an AKE to a Augmented PAKE.

1.3.3 AuCPace

Although AuCPace [HL18] didn't quite make the cut for IETF standardisation it is still a very interesting PAKE and well worth a look. Designed specifically for use in IIOT applications, AuCPace is an Augmented PAKE protocol intended for use in situations where traditional Public-Key-Infrastructure (PKI) simply isn't available. The protocol is modelled around a powerful human machine interface (HMI) client device and a weak server device, as this setup is common in IIOT applications, e.g. one PC being used to configure many sensors/actuators. Efficiency is at the core of this protocol, it is taken into consideration at every level, from the high-level protocol design to the low-level arithmetic. A unique bonus of this protocol as well is it takes into account the real-world issue of patents and aims to provide a practical protocol which is free from patents so as to promote the widest possible adoption of the protocol.

1.4 Choosing a PAKE to implement

There were a number of factors to consider when choosing which PAKE to implement:

- How widely applicable is the protocol?
- How many existing implementations are there?
- How good are existing solutions (solutions that aren't the given protocol)?
- Is there potential to contribute an implementation back to an open source project?

After a conversation with the RustCrypto core team on Zulip, it was agreed that an implementation of any of these protocols would be readibly accepted into their collection of PAKEs – `https://github.com/RustCrypto/PAKEs`. I will talk more about RustCrypto and who they are in section 1.6. For now I will omit considerations about open source contribution.

Table 1.4: Modern PAKE Protocol Comparison

Protocol	Applicability	Implementations	Existing Solutions
CHIP+CRISP	WiFi	C++ reference impl	Pre-Shared Key (PSK)
KHAPE	Replacing PKI	Educational Rust impl	OPAQUE
AuCPace	IIOT	C reference impl + Go impl	hard coding the key

AuCPace is the only PAKE in the list where there is a completely inadequate current solution. AuCPace also targets a rapidly growing area, the combined risk of these factors means that AuCPace is uniquely positioned to make a large difference to the security of the IIOT landscape. It is for these reasons that I have chosen to create an implementation of AuCPace and to contribute it back to the open source community via RustCrypto.

1.5 AuCPace in detail

Now that we have chosen to implemented AuCPace it is worth going over the protocol itself to understand better what implementing it will entail. Figures 1.9 and 1.10 contain protocol diagrams reproduced from [HL18]. There are four phases to the protocol some of which can be made optional or can be adjusted based on the sub-variant of the protocol in use.

1. Sub-Session ID (SSID) Agreement
   - server and client each generate a nonce and send it to the other party
   - each receives the other's nonce and calculates the SSID
2. AuCPace Augmentation Layer
   - the server generates a new ephemeral DH key
   - the server receieves the client's username, looks up the verifier and salt.
   - the server then sends across the group $\mathcal{J}$, the public key X, the salt and the parameters for the PBKDF - σ

- both then compute the Password Related String (PRS), the client aborts here if the server's public key X is invalid

3. CPace substep
   - both compute g' and G from the hash of $ssid||PRS||CI$
   - both generate an ephemeral private key y and the compute the public key $Y = G^{y\ c_{\mathcal{J}}}$
   - public keys are then exchanged
   - the shared point K is then computed
   - if either public key is invalid both parties abort here
   - $sk1$ is then computed as the hash of the SSID and K
4. Explicit mutual authentication
   - both compute authenticators T_a, T_b
   - each party sends a different
   - if either authenticator is invalid the protocol is aborted
   - finally the shared key sk is computed

There are three different configurations which can be adjusted to make tradeoffs between security, speed and storage:

- full vs partial augmentation
- implicit authentication vs explicit mutual authentication
- with pre-computation attack resistance ("strong AuCPace") vs without (AuCPace)

Store password operation for AuCPace

Server		**Client**
		salt $\leftarrow\$ \{0,1\}^l$ $w = \mathsf{PBKDF}_\sigma(pw, \text{username, salt})$ $W = B^{w\ c_{\mathcal{J}}}$
	username, salt, uad, W, σ $\longleftarrow$	
Upon successful verification, record (username, salt, uad, W, σ) in the password file.		

Figure 1.9: AuCPace protocol for password configuration.

AuCPace

Server		**Client**
	Agree on $ssid$	
$s \leftarrow\!\!\$ \{0,1\}^{k_1}$		$s \leftarrow\!\!\$ \{0,1\}^{k_1}$
	$\xrightarrow{s}$	
	$\xleftarrow{t}$	
$ssid = \mathsf{H}_0(s\|\|t)$		$ssid = \mathsf{H}_0(s\|\|t)$
	AuCPace Augmentation Layer	
$x \leftarrow\!\!\$ \{1 \ldots m_{\mathcal{J}}\}$		
$X = B^{c_{\mathcal{J}}}$		
	$\xleftarrow{\text{username}}$	
$W, \text{salt} = \text{lookup}W(\text{user})$		
	$\xrightarrow{\mathcal{J}, X, \text{salt}, \sigma}$	
		$w = \mathsf{PBKDF}_{\sigma}(pw, \text{user, salt})$
if lookup failed $PRS \leftarrow\!\!\$ \{0,1\}^{k_2}$		abort if X invalid
else $PRS = W^{x\ c_{\mathcal{J}}}$		$PRS = X^{w\ c_{\mathcal{J}}}$
	CPace substep	
$g' = \mathrm{H}_1(ssid\|\|PRS\|\|CI)$		$g' = \mathrm{H}_1(ssid\|\|PRS\|\|CI)$
$G = \mathsf{Map2Point}(g')$		$G = \mathsf{Map2Point}(g')$
$y_a \leftarrow\!\!\$ \{1 \ldots m_{\mathcal{J}}\}$		$y_b \leftarrow\!\!\$ \{1 \ldots m_{\mathcal{J}}\}$
$Y_a = G^{y_a\ c_{\mathcal{J}}}$		$Y_b = G^{y_b\ c_{\mathcal{J}}}$
	$\xrightarrow{Y_a}$	
	$\xleftarrow{Y_b}$	
$K = Y_b^{y_a\ c_{\mathcal{J}}}$		$K = Y_a^{y_b\ c_{\mathcal{J}}}$
abort if Y_b invalid		abort if Y_b invalid
$sk_1 = \mathsf{H}_2(ssid\|\|K)$		$sk_1 = \mathsf{H}_2(ssid\|\|K)$
	Explicit mutual authentication	
$T_a = \mathsf{H}_3(ssid\|\|sk_1)$		$T_a = \mathsf{H}_3(ssid\|\|sk_1)$
$T_b = \mathsf{H}_4(ssid\|\|sk_1)$		$T_b = \mathsf{H}_4(ssid\|\|sk_1)$
	$\xleftarrow{T_b}$	
	$\xrightarrow{T_a}$	
verify T_b		verify T_a
$sk = \mathsf{H}_5(ssid\|\|sk_1)$		$sk = \mathsf{H}_5(ssid\|\|sk_1)$

Figure 1.10: AuCPace Protocol

All of these options lead to 8 different sub-protocols. What's special about these sub-protocols is how little they change the overall protocol. Below are protocol diagrams illustrating what changes with each configuration change.

Store password operation for AuCPace

Server		**Client**
		salt $\leftarrow\$ \{0,1\}^l$ $w = \mathsf{PBKDF}_\sigma(pw, \text{username, salt})$ $W = B^{w\ c_\mathcal{J}}$
	username, salt, uad, W, σ $\longleftarrow$	
$\mathbf{x} \leftarrow\$ \{\mathbf{1} \ldots \mathbf{m}_\mathcal{J}\}$ $\mathbf{X} = \mathbf{B}^{\mathbf{x}\ \mathbf{c}_\mathcal{J}}$ Upon successful verification, record (username, salt, uad, $W, \sigma, \mathbf{x}, \mathbf{X}$) in the password file.		

AuCPace Augmentation Layer

Server		**Client**
$\cancel{x \leftarrow\$ \{1 \ldots m_\mathcal{J}\}}$ $\cancel{X = B^{c_\mathcal{J}}}$		
	username $\longleftarrow$	
$\mathbf{x}, \mathbf{X}, W$, salt $=$ lookup$W$(user)		
	$\mathcal{J}, X$, salt, σ $\longrightarrow$	
		$w = \mathsf{PBKDF}_\sigma(pw, \text{user, salt})$
if lookup failed $PRS \leftarrow\$ \{0,1\}^{k_2}$ else $PRS = W^{x\ c_\mathcal{J}}$		abort if X invalid $PRS = X^{w\ c_\mathcal{J}}$

Figure 1.11: Differences in Partial Augmentation

Store password operation for strong AuCPace

Server		**Client**
		$\cancel{\text{salt} \leftarrow_{\$} \{0,1\}^{l}}$
		$\mathrm{q} \leftarrow_{\$} \{\mathbf{0},\mathbf{1}\}^{\mathbf{l}}$
		$\mathbf{Z} = \mathsf{Map2Point}(\mathsf{H}_{\mathbf{1}}(\text{username}\|\|\mathbf{pw}))$
		$\text{salt} = \mathbf{Z}^{\mathbf{q}\ \mathbf{c}_{\mathcal{J}}}$
		$w = \mathsf{PBKDF}_{\sigma}(pw, \text{user}, \text{salt})$
		$W = B^{w\ c_{\mathcal{J}}}$
	username,$\mathbf{q}\cancel{salt},$ uad, W, σ $\longleftarrow$	
Upon successful verification, record (username, $\mathbf{q}$ ~~salt~~, uad, W, σ) in the password file.		

strong AuCPace Augmentation Layer

Server		**Client**
$x \leftarrow_{\$} \{1 \dots m_{\mathcal{J}}\}$		$\mathbf{r} \leftarrow_{\$} \{\mathbf{1} \dots \mathbf{m}_{\mathcal{J}}\}$
$X = B^{c_{\mathcal{J}}}$		$\mathbf{Z} = \mathsf{Map2Point}(\mathsf{H}_{\mathbf{1}}(\text{username}\|\|\mathbf{pw}))$
		$\mathbf{U} = \mathbf{Z}^{\mathbf{r}\ \mathbf{c}_{\mathcal{J}}}$
	username, $\mathbf{U}$ $\longleftarrow$	
$W, \mathbf{q}\cancel{\text{salt}} = \text{lookup}W(\text{user})$		
$\mathbf{UQ} = \mathbf{U}^{\mathbf{q}\ \mathbf{c}_{\mathcal{J}}}$		
abort if UQ invalid		
	$\mathcal{J}, X, \mathbf{UQ}\cancel{\text{salt},} \sigma$ $\longrightarrow$	
		$\text{salt} = \mathbf{UQ}^{\frac{\mathbf{1}}{\mathbf{r}\ (\mathbf{c}_{\mathcal{J}})^{\mathbf{2}}}\mathbf{c}_{\mathcal{J}}}$
		abort if salt invalid
		$w = \mathsf{PBKDF}_{\sigma}(pw, \text{user}, \text{salt})$
if lookup failed $PRS \leftarrow_{\$} \{0,1\}^{k_2}$		abort if X invalid
else $PRS = W^{x\ c_{\mathcal{J}}}$		$PRS = X^{w\ c_{\mathcal{J}}}$

Figure 1.12: Differences in Strong AuCPace

implicit auth CPace substep

Server		**Client**
$g' = \mathrm{H}_1(ssid\|\|PRS\|\|CI)$		$g' = \mathrm{H}_1(ssid\|\|PRS\|\|CI)$
$G = \mathsf{Map2Point}(g')$		$G = \mathsf{Map2Point}(g')$
$y_a \leftarrow\$ \{1 \dots m_{\mathcal{J}}\}$		$y_b \leftarrow\$ \{1 \dots m_{\mathcal{J}}\}$
$Y_a = G^{y_a\ c_{\mathcal{J}}}$		$Y_b = G^{y_b\ c_{\mathcal{J}}}$
	$\xrightarrow{\quad Y_a \quad}$	
	$\xleftarrow{\quad Y_b \quad}$	
$K = Y_b^{y_a\ c_{\mathcal{J}}}$		$K = Y_a^{y_b\ c_{\mathcal{J}}}$
abort if Y_b invalid		abort if Y_b invalid
$sk_1 = \mathsf{H}_2(ssid\|\|K)$		$sk_1 = \mathsf{H}_2(ssid\|\|K)$
$\mathbf{sk} = \mathsf{H}_5(\mathbf{ssid}\|\|\mathbf{sk_1})$		$\mathbf{sk} = \mathsf{H}_5(\mathbf{ssid}\|\|\mathbf{sk_1})$

Figure 1.13: Differences in Implicit Authentication

Table 1.5: Configuration tradeoffs

Configuration	Advantage	Disadvantage
partial augmentation	removes an expensive exponentiation operation for the server, halving the computational complexity for the server	an attacker can impersonate the client by compromising a server
strong AuCPace	pre-computation attack resistance	increases the computational requirements for both the client and server
implicit authentication	saves a round of messages and 2 hash computations.	the protocol downgrades to weak perfect forward secrecy [Kra05]

1.6 Who are RustCrypto?

CHAPTER 2

Design

2.1 Why Rust?

2.2 Developer Focussed Design

CHAPTER 3

Implementation

3.1 Overview of RustCrypto and Dalek Cryptography

CHAPTER 4

TESTING

4.1 Creating Test Vectors

CHAPTER 5

Reflection and Conclusion

5.1 Achievements

5.2 Reflection

5.3 Future Work

Glossary

Abelian Group A group whose operator is also commutative. e.g. Addition over $\mathbb{Z}$. . 12, 13

AES Advanced Encryption Scheme. 27

AKE Authenticated Key-Exchange. 15

Asymmetric Cryptography Asymmetric Cryptography is where the the sender and receiver each have two keys - a public key which can be freely shared, and a private key which must be kept secret. Common examples of this are the RSA scheme and the various DH flavours. 9, 10

AuCPace Augmented Composable Password Authenticated Connection Establishment. 4, 15, 16, 17, 18, 21

Augmented PAKE A Balanced PAKE is one in which both parties share knowledge the same secret. This is in contrast to other schemes such as Verifier-based/Augmented PAKEs. . 8, 11, 14, 15, 27

Balanced PAKE A Balanced PAKE is one in which both parties share knowledge the same secret. This is in contrast to other schemes such as Verifier-based/Augmented PAKEs. . 8, 10, 14, 15

CFRG Crypto Forum Research Group. 14, 27

CPace Composable Password Authenticated Connection Establishment. 14, 17

DH Diffie-Hellman. 10, 13, 15, 16, 35

EAP Extensible Authentication Protocol. 9

ECC Elliptic Curve Cryptography. 4, 14

ECDLP Elliptic Curve Discrete Logarithm Problem. 14

EKE Encrypted Key Exchange. 9, 10, 11

Finite Field A Finite Field is a finite set with an associated addition and multiplication operator, where the operators satisfy the field axioms. Namely they are: Associative, Commutative, Distributive, they have inverses and identity elements. 13

HMI human machine interface. 15

IETF Internet Engineering Task Force. 14, 15

IIOT Industrial Internet of Things. 4, 15, 16

IOT Internet of Things. 4

iPAKE identity-binding PAKE. 15

IRTF Internet Research Task Force. 14

KHAPE Key-Hiding Asymmetric PAKE. 15, 16

NIST National Institute of Standards and Technology. 12

nonce number used only once – A cryptographic term which relates to an ephemeral secret value, an example would be an Initialisation Vector for AES-CBC mode encryption. . 16

Online Cryptography Online cryptography is where interactions with the cryptosystem are only possible via real-time interactions with the server. Primarily this is to prevent offline computation. 8, 10

OPAQUE An Asymmetric PAKE Protocol Secure Against Pre-Computation Attacks. Augmented PAKE Winner of the CFRG PAKE selection process. The name is a play on words from OPAKE, where O is OPRF. 14, 15, 16

OPRF Oblivious Pseudo Random Function. 15, 27

PAKE Password-Authenticated Key-Exchange. 4, 7, 8, 9, 11, 14, 15, 16

PKI Public-Key-Infrastructure. 15, 16

PRS Password Related String. 17

PSK Pre-Shared Key. 16

RSA Rivest-Shamir-Adleman. 9, 12

Safe Prime A number $2n + 1$ is a Safe Prime if n is prime, it is the effectively the other part of a Sophie Germain prime. . 10, 11

SPAKE Simple PAKE. 10, 11

SRP Secure Remote Password. 11, 35

SSID Sub-Session ID. 16, 17

Symmetric Cryptography Symmetric Cryptography is where the both the sender and receiver share the same secret key. It is normally computationally more efficient, the most common such scheme is Advanced Encryption Scheme (AES). 9

TLS Transport Layer Security. 7, 11

Verifier A representation of the user's password put through some one-way function. This could be as simple as just storing a hash of the password, though for most PAKEs the verifier is an element of whatever group we are working in. An example can be seen on page 11. 8, 11

V-PAKE Verifier based PAKE. 4

Bibliography

[AP05] Michel Abdalla and David Pointcheval. "Simple Password-Based Encrypted Key Exchange Protocols". In: *Topics in Cryptology – CT-RSA 2005*. Ed. by Alfred Menezes. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 191–208. ISBN: 978-3-540-30574-3.

[Ber+13] Daniel J Bernstein et al. "Elligator: elliptic-curve points indistinguishable from uniform random strings". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 2013, pp. 967–980.

[BL] Daniel J. Bernstein and Tanja Lange. *SafeCurves: choosing safe curves for elliptic-curve cryptography*. Accessed 3rd April 2023. URL: https://safecurves.cr.yp.to.

[BL13] Daniel J Bernstein and Tanja Lange. "Security dangers of the NIST curves". In: *Invited talk, International State of the Art Cryptography Workshop, Athens, Greece*. 2013.

[BL17] Daniel J. Bernstein and Tanja Lange. *Montgomery curves and the Montgomery ladder*. Cryptology ePrint Archive, Paper 2017/293. 2017. URL: https://eprint.iacr.org/2017/293.

[BM92] Steven Michael Bellovin and Michael Merritt. *Encrypted key exchange: Password-based protocols secure against dictionary attacks*. May 1992.

[BMM00] Ingrid Biehl, Bernd Meyer, and Volker Müller. "Differential fault attacks on elliptic curve cryptosystems". In: *Advances in Cryptology—CRYPTO 2000: 20th Annual International Cryptology Conference Santa Barbara, California, USA, August 20–24, 2000 Proceedings 20*. Springer. 2000, pp. 131–146.

[Cha20] CFRG Chairs. *Results of the PAKE selection process*. Apr. 2020. URL: https://datatracker.ietf.org/meeting/interim-2020-cfrg-01/materials/slides-interim-2020-cfrg-01-sessa-results-of-the-pake-selection-process-00.

[Cre+20] Cas Cremers et al. *CHIP and CRISP: Protecting All Parties Against Compromise through Identity-Binding PAKEs*. Cryptology ePrint Archive, Paper 2020/529. 2020. URL: https://eprint.iacr.org/2020/529.

[GJK21] Yanqi Gu, Stanislaw Jarecki, and Hugo Krawczyk. *KHAPE: Asymmetric PAKE from Key-Hiding Key Exchange*. Cryptology ePrint Archive, Paper 2021/873. 2021. URL: https://eprint.iacr.org/2021/873.

[Gre18] Matthew Green. *Should you use SRP?* Accessed 3rd April, 2023. 2018. URL: https://blog.cryptographyengineering.com/should-you-use-srp/.

[Har08] Dan Harkins. "Simultaneous authentication of equals: A secure, password-based key exchange for mesh networks". In: *2008 Second International Conference on Sensor Technologies and Applications (sensorcomm 2008)*. IEEE. 2008, pp. 839–844.

[HL18] Björn Haase and Benoît Labrique. *AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT*. Cryptology ePrint Archive, Paper 2018/286. 2018. URL: https://eprint.iacr.org/2018/286.

[IT02] Tetsuya Izu and Tsuyoshi Takagi. "Exceptional procedure attack on elliptic curve cryptosystems". In: *Public Key Cryptography—PKC 2003: 6th International Workshop on Practice and Theory in Public Key Cryptography Miami, FL, USA, January 6–8, 2003 Proceedings 6*. Springer. 2002, pp. 224–239.

[Jea16] Jérémy Jean. *TikZ for Cryptographers*. https://www.iacr.org/authors/tikz/. 2016.

[KMV00] Neal Koblitz, Alfred Menezes, and Scott Vanstone. "The state of elliptic curve cryptography". In: *Designs, codes and cryptography* 19 (2000), pp. 173–193.

[Kob87] Neal Koblitz. "Elliptic curve cryptosystems". In: *Mathematics of computation* 48.177 (1987), pp. 203–209.

[Kra05] Hugo Krawczyk. "HMQV: A high-performance secure Diffie-Hellman protocol". In: *Crypto*. Vol. 3621. Springer. 2005, pp. 546–566.

[LL97] Chae Hoon Lim and Pil Joong Lee. "A key recovery attack on discrete log-based schemes using a prime order subgroup". In: *Advances in Cryptology–CRYPTO'97: 17th Annual International Cryptology Conference Santa Barbara, California, USA August 17–21, 1997 Proceedings 17*. Springer. 1997, pp. 249–263.

[Mil86] Victor S. Miller. "Use of Elliptic Curves in Cryptography". In: *Advances in Cryptology — CRYPTO '85 Proceedings*. Ed. by Hugh C. Williams. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 417–426. ISBN: 978-3-540-39799-1.

[MVO91] Alfred Menezes, Scott Vanstone, and Tatsuaki Okamoto. "Reducing elliptic curve logarithms to logarithms in a finite field". In: *Proceedings of the twenty-third annual ACM symposium on Theory of computing*. 1991, pp. 80–89.

[Pol78] John M Pollard. "Monte Carlo methods for index computation ($\bmod p$)". In: *Mathematics of computation* 32.143 (1978), pp. 918–924.

[Sec21] Apple Platform Security. *Escrow security for iCloud Keychain*. Accessed 3rd April, 2023. May 2021. URL: https://support.apple.com/en-gb/guide/security/sec3e341e75d/web.

[Sem98] Igor Semaev. "Evaluation of discrete logarithms in a group of ρ-torsion points of an elliptic curve in characteristic ρ". In: *Mathematics of computation* 67.221 (1998), pp. 353–356.

[She+11] Y. Sheffer et al. *An EAP Authentication Method Based on the Encrypted Key Exchange (EKE) Protocol*. rfc 6124. RFC Editor, Feb. 2011. URL: https://www.rfc-editor.org/rfc/rfc6124.txt.

[ST20] National Institute of Standards and Technology. *Recommendation for Key Management: Part 1 – General*. Tech. rep. Federal Information Processing Standards Publications (FIPS PUBS) 140-2, Change Notice 2 December 03, 2002. Washington, D.C.: U.S. Department of Commerce, 2020. DOI: 10.6028/10.6028/NIST.SP.800-57pt1r5.

[Vol+04] John Vollbrecht et al. *Extensible Authentication Protocol (EAP)*. rfc 3748. RFC Editor, June 2004. URL: https://www.rfc-editor.org/rfc/rfc3748.txt.

[Wu+07] T. Wu et al. *Using the Secure Remote Password (SRP) Protocol for TLS Authentication*. RFC 5054. RFC Editor, Nov. 2007. URL: https://www.rfc-editor.org/rfc/rfc5054.txt.

[Wu00] Tom Wu. *The SRP Authentication and Key Exchange System*. RFC 2945. RFC Editor, Sept. 2000. URL: https://www.rfc-editor.org/rfc/rfc2945.txt.

APPENDIX A

Python implementation of EKE

While researching Bellovin and Merritt's EKE scheme[BM92], I created a full implementation of the scheme in Python. The full code can be found at https://github.com/tritoke/eke_python. The core negotiation functions for the client and server have been included below:

Listing 1: Client Negotiate

```python
negotiate(self):
# generate random public key Ea
Ea = RSA.gen()

# instantiate AES with the password
P = AES.new(self.password.ljust(16).encode(), AES.MODE_ECB)

# send a negotiate command
self.send_json(
    action="negotiate",
    username=self.username,
    enc_pub_key=b64e(P.encrypt(Ea.encode_public_key())),
    modulus=Ea.n
)

# receive and decrypt R
self.recv_json()
key =
↪   l2b(Ea.decrypt(b2l(P.decrypt(b64d(self.data["enc_secret_key"])))))
R = AES.new(key, AES.MODE_ECB)

# send first challenge
challengeA = randbytes(16)
self.send_json(challenge_a=b64e(R.encrypt(challengeA)))

# receive challenge response
self.recv_json()
challenge_response = R.decrypt(b64d(self.data["challenge_response"]))
```

```
assert challenge_response[:16] == challengeA, "Challenge A failed."

challengeB = challenge_response[16:]

# response with challengeB
self.send_json(challenge_b=b64e(R.encrypt(challengeB)))

# receive success message
self.recv_json()
assert self.data["success"], self.data.get("message",
↪  "ChallengeB failed.")

# store the shared key
self.R = R
```

Listing 2: Server Negotiate

```
handle_eke_negotiate_key(self):
# decrypt Ea using P
P = AES.new(self.database[self.data["username"]].ljust(16).encode(),
↪  AES.MODE_ECB)
e = b2l(P.decrypt(b64d(self.data["enc_pub_key"])))

# e is always odd, but we add 1 with 50% probability
if e % 2 == 0:
    e -= 1

# generate secret key R
R = randbytes(16)
Ea = RSA.from_pub_key(e, self.data["modulus"])
self.send_json(enc_secret_key=b64e(P.encrypt(l2b(Ea.encrypt(b2l(R))))))
x = b64e(P.encrypt(l2b(Ea.encrypt(b2l(R)))))

# transform R into a cipher instance
R = AES.new(R, AES.MODE_ECB)

# receive encrypted challengeA and generate challengeB
self.recv_json()
challengeA = R.decrypt(b64d(self.data["challenge_a"]))
challengeB = randbytes(16)

# send challengeA + challengeB
self.send_json(challenge_response=b64e(R.encrypt(challengeA+challengeB
↪  )))

# receive challengeB back again
self.recv_json()
```

```
success = R.decrypt(b64d(self.data["challenge_b"])) == challengeB

self.send_json(success=success)
self.R = R
```

APPENDIX B

PYTHON IMPLEMENTATION OF SRP

While conducting my initial research on PAKEs I came across SRP[Wu00]. SRP is the first protocol I looked at which took the approach of encoding values as DH group elements. To understand this approach better I chose to create a toy implementation. The full code can be found at `https://github.com/tritoke/srp_python`. The core negotiation functions for the client and server have been included below:

Listing 3: Client Negotiate

```python
negotiate(self):
# send a negotiate command
self.send_json(action="negotiate", username=self.username)

# receive the salt back from the server
self.recv_json()
s = int(self.data["salt"])
x = H(s, H(f"{self.username}:{self.password}"))

# generate an ephemeral key pair and send the public key to the server
a = strong_rand(KEYSIZE_BITS)
A = pow(g, a, N)
self.send_json(user_public_ephemeral_key=A)

# receive the servers public ephemeral key back
self.recv_json()
B = self.data["server_public_ephemeral_key"]

# calculate u and S
u = H(A, B)
S = pow((B - 3 * pow(g, x, N)), a + u * x, N)

# calculate M1
M1 = H(A, B, S)
self.send_json(verification_message=M1)

# receive M2
```

```
self.recv_json()
M2 = self.data["verification_message"]

if M2 != H(A, M1, S):
    print("Failed to agree on shared key.")

K = H(S)

return K
```

Listing 4: Server Negotiate

```
handle_srp_negotiate_key(self):
# receive the username I from the client
# lookup data in database
user = self.data["username"]
I = b2l(user.encode())

if (db_record := self.database.get(user)) is None:
    self.send_json(success=False,
     ↪  message=f"Failed to find user in DB.")
    return

s = db_record["salt"]
v = db_record["verifier"]

# send s to the client
self.send_json(salt=s)

# receive A from the user
self.recv_json()
A = self.data["user_public_ephemeral_key"]

# calculate B
b = strong_rand(KEYSIZE_BITS)
B = 3 * v + pow(g, b, N)

# send B to the client
self.send_json(server_public_ephemeral_key=B)

# calculate u and S
u = H(A, B)
S = pow(A * pow(v, u, N), b, N)

# receive M1 from the client
self.recv_json()
M1 = self.data["verification_message"]
```

```python
# verify M1
if M1 != H(A, B, S):
    self.send_json(success=False,
     ↪  message=f"Failed to agree shared key.")
    return

# calculate M2
M2 = H(A, M1, S)
self.send_json(verification_message=M2)

# calculate key
K = H(S)

# log the derived key - not part of the protocol
print(f"Derived K={K:X}")

# encrypt our final message to the client using our shared key
key = l2b(K)
nonce = get_random_bytes(16)

cipher = AES.new(key, AES.MODE_GCM, nonce=nonce)
ct, mac = cipher.encrypt_and_digest(f⌋
↪  "Successfully agreed shared key for {user}.".encode())

# notify the client of the success
self.send_json(success=True, nonce=b64e(nonce), enc_message=b64e(ct),
↪  tag=b64e(mac))
```