

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

- i. The author of this project report (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the “Copyright”) and they have given The University of Manchester certain rights to use such Copyright, including for administrative purposes.
- ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.
- iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the thesis, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.
- iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see <http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=24420>), in any relevant Thesis restriction declarations deposited in the University Library, The University Library’s regulations (see <http://www.library.manchester.ac.uk/about/regulations/>) and in The University’s policy on presentation of Theses

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 the 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 (Organizers). Without your help none of this would have been possible.

Contents

1	Context	7
1.1	Background on PAKEs	7
1.1.1	What is a PAKE?	7
1.1.2	A brief history of PAKE algorithms	8
1.2	Elliptic Curve Cryptography	11
1.2.1	But what actually is an elliptic curve?	11
1.2.2	How do we do Cryptography with curves?	11
1.3	Modern PAKEs	12
1.4	AuCPace	12
1.5	Who are RustCrypto?	12
2	Design	13
2.1	Why Rust?	13
2.2	Developer Focussed Design	13
3	Implementation	14
3.1	Overview of RustCrypto and Dalek Cryptography	14
4	Testing	15
4.1	Creating Test Vectors	15
5	Reflection and Conclusion	16
5.1	Achievements	16
5.2	Reflection	16
5.3	Future Work	16
	Glossary	17
A	Python implementation of EKE	21
B	Python implementation of SRP	24

Chapter 1

Context

1.1 Background on PAKEs

1.1.1 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.

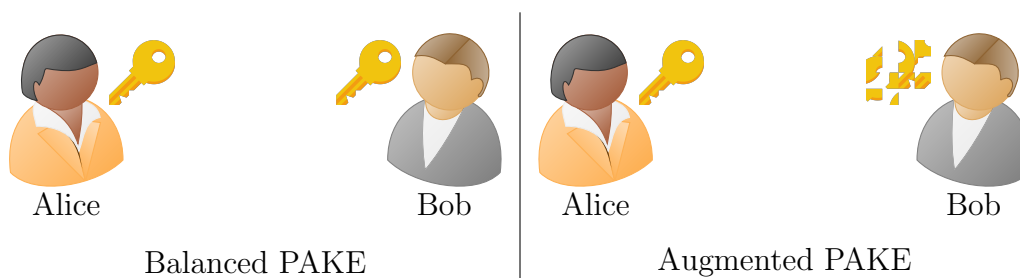


Figure 1.1: An illustration of the difference between Augmented PAKEs and Balanced PAKEs

1.1.2 A brief history of PAKE algorithms

The first **PAKE** algorithm was Bellovin and Merritt's **Encrypted Key Exchange (EKE)** scheme.² 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)**¹⁸ where it is available as **EAP-EKE**.¹⁷ 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.

Shared Parameter	Secret	Explanation
P	yes	the shared password

Figure 1.2: **EKE** shared parameters

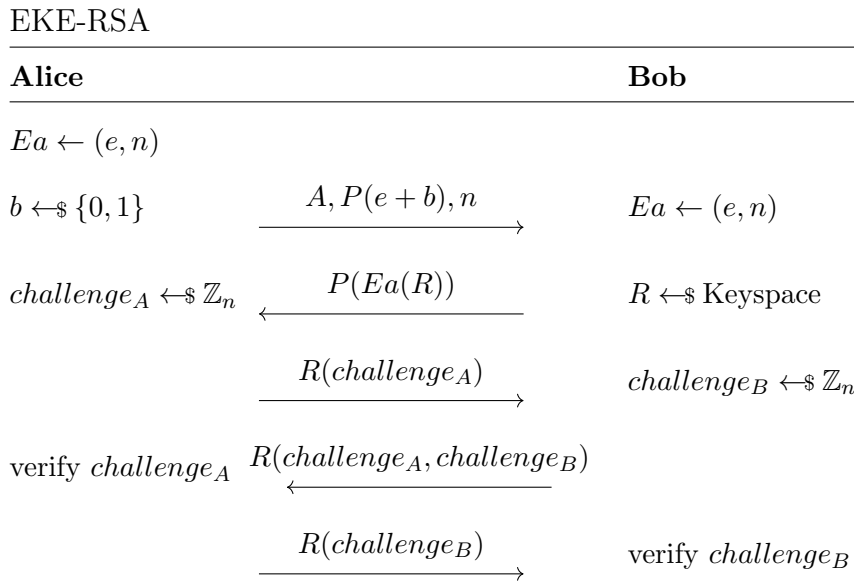


Figure 1.3: Implementing **EKE** using **RSA**

SPAKE

SPAKE1 and **SPAKE2** are **Balanced PAKEs**’ which were introduced slightly later on by Michel Abdalla and David Pointcheval¹ as variations on **EKE**. **SPAKE2** and **SPAKE2** are very similar so we will just explore **SPAKE2** as we are more interested in online algorithms. **SPAKE2** is also **Simple Password-Authenticated Key-Exchange (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.

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 operations
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

Figure 1.4: **SPAKE** shared parameters

SPAKE2

Alice

$$x \leftarrow \$ \mathbb{Z}_p$$

$$X \leftarrow g^x$$

$$X^* \leftarrow X \cdot M^{pw}$$

Bob

$$y \leftarrow \$ \mathbb{Z}_p$$

$$Y \leftarrow g^y$$

$$Y^* \leftarrow Y \cdot N^{pw}$$

$$X^*$$

$$\longrightarrow$$

$$Y^*$$

$$\longleftarrow$$

$$K_A \leftarrow (Y^*/N^{pw})^x$$

$$SK_A \leftarrow H(A, B, X^*, Y^*, K_A)$$

$$K_B \leftarrow (X^*/M^{pw})^y$$

$$SK_B \leftarrow H(A, B, X^*, Y^*, K_B)$$

Figure 1.5: 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 Keychain¹⁶ and it's availability as a [Transport Layer Security \(TLS\)](#) ciphersuite.¹⁹ However it is quite weird for what it does and there is no security proof for it.⁶ An implementation of the protocol in Python can be found in [appendix B](#).

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

Figure 1.6: [SRP](#) parameters

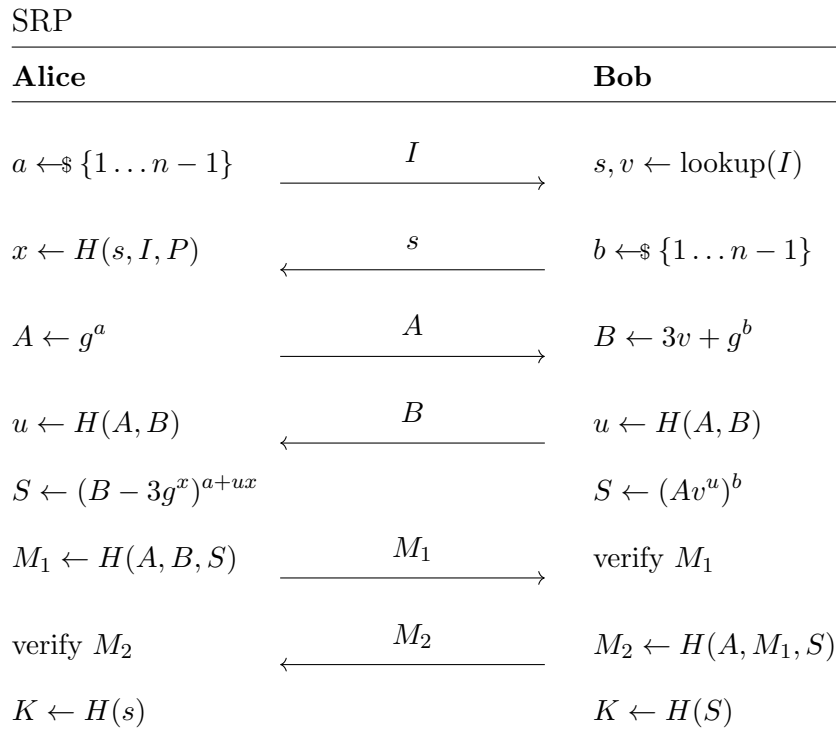


Figure 1.7: SRP-6 Protocol

1.2 Elliptic Curve Cryptography

Many modern Cryptographic protocols make use of a mathematical object known as an elliptic curve. First proposed in 1985 independently by Neal Koblitz¹² and Victor S. Miller.¹⁴ 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.¹³ 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.¹⁵

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: $by^2 = x^3 + ax^2 + x$

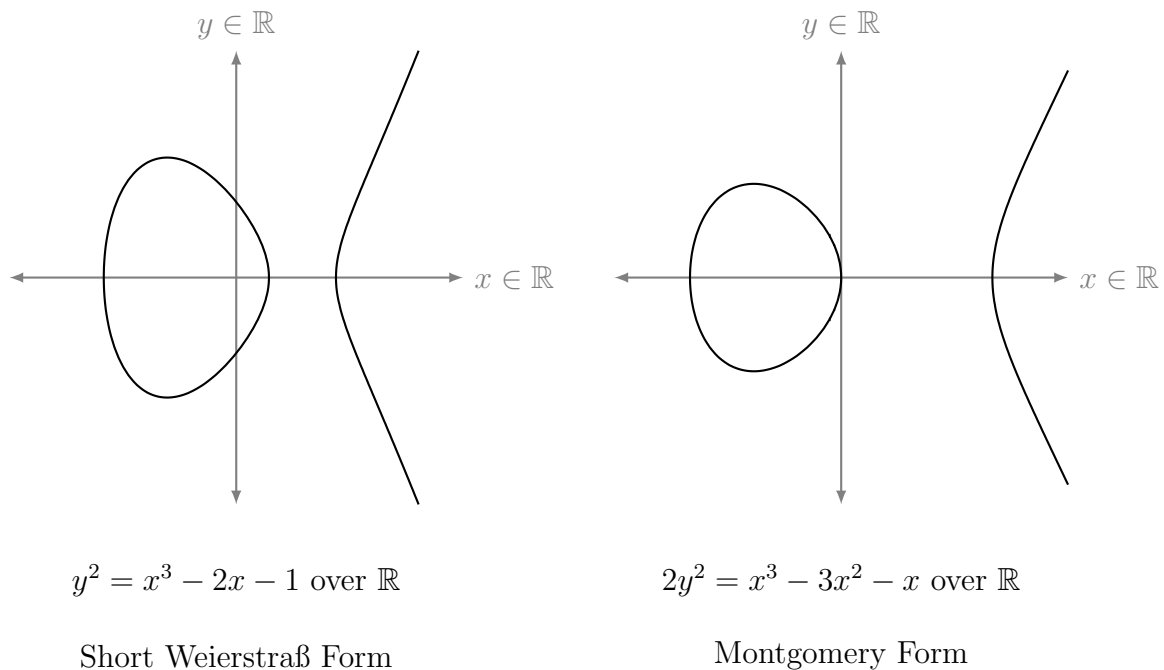


Figure 1.8: Elliptic curves over \mathbb{R} , Adapted from TikZ for Cryptographers¹¹

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.³

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 1.9 illustrates all of these rules and edge cases.

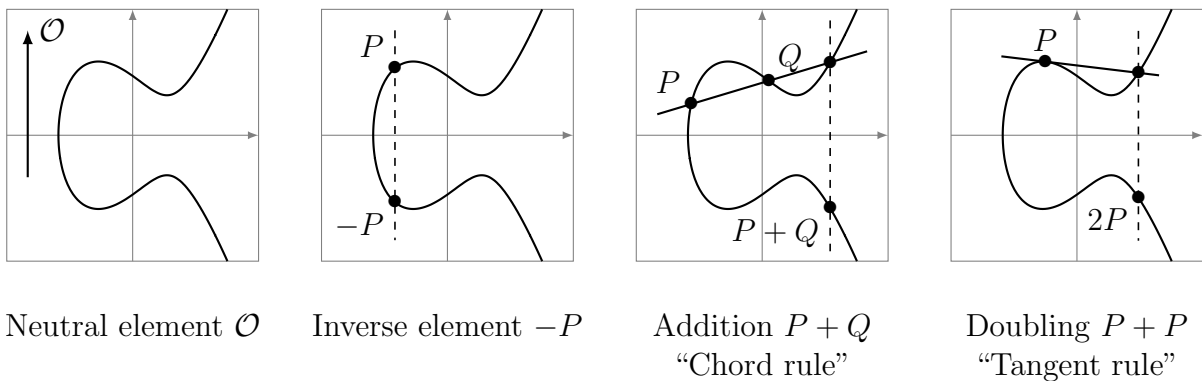


Figure 1.9: Elliptic Curve Group Operations, reproduced from TikZ for Cryptographers¹¹

1.3 Modern PAKEs

1.4 AuCPace

1.5 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} . . 11, 12

AES Advanced Encryption Scheme. 18

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. 8, 9

AuCPace Augmented Composable Password Authenticated Connection Establishment. 4

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. . 7, 10

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. . 7, 9

DH Diffie-Hellman. 9, 12, 24

EAP Extensible Authentication Protocol. 8

ECC Elliptic Curve Cryptography. 4

EKE Encrypted Key Exchange. 8, 9, 10

IIOT Industrial Internet of Things. 4

IOT Internet of Things. 4

NIST National Institute of Standards and Technology. 11

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. 7, 9

PAKE Password-Authenticated Key-Exchange. 4, 7, 8, 10

RSA Rivest-Shamir-Adleman. 8, 11

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. . [9](#), [10](#)

SPAKE Simple Password-Authenticated Key-Exchange. [9](#), [10](#)

SRP Secure Remote Password. [10](#), [24](#)

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 \(AES\)](#). [8](#)

TLS Transport Layer Security. [10](#)

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 [10](#). [7](#), [10](#)

V-PAKE Verifier based PAKE. [4](#)

Bibliography

- ¹ Michel Abdalla and David Pointcheval. Simple password-based encrypted key exchange protocols. In Alfred Menezes, editor, *Topics in Cryptology – CT-RSA 2005*, pages 191–208, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
- ² Steven Michael Bellare and Michael Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks, May 1992.
- ³ Daniel J. Bernstein and Tanja Lange. Montgomery curves and the montgomery ladder. Cryptology ePrint Archive, Paper 2017/293, 2017. <https://eprint.iacr.org/2017/293>.
- ⁴ Cas Cremers, Moni Naor, Shahar Paz, and Eyal Ronen. Chip and crisp: Protecting all parties against compromise through identity-binding pakes. Cryptology ePrint Archive, Paper 2020/529, 2020. <https://eprint.iacr.org/2020/529>.
- ⁵ Matthew Green. Let’s talk about pake, Oct 2018. <https://blog.cryptographyengineering.com/2018/10/19/lets-talk-about-pake/>.
- ⁶ Matthew Green. Should you use srp?, 2018. <https://blog.cryptographyengineering.com/should-you-use-srp/>.
- ⁷ Yanqi Gu, Stanislaw Jarecki, and Hugo Krawczyk. Khape: Asymmetric pake from key-hiding key exchange. Cryptology ePrint Archive, Paper 2021/873, 2021. <https://eprint.iacr.org/2021/873>.
- ⁸ Björn Haase and Benoît Labrique. Aucpace: Efficient verifier-based pake protocol tailored for the iiot. Cryptology ePrint Archive, Paper 2018/286, 2018. <https://eprint.iacr.org/2018/286>.
- ⁹ Mike Hamburg. Decaf: Eliminating cofactors through point compression. Cryptology ePrint Archive, Paper 2015/673, 2015. <https://eprint.iacr.org/2015/673>.
- ¹⁰ Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. Opaque: An asymmetric pake protocol secure against pre-computation attacks. Cryptology ePrint Archive, Paper 2018/163, 2018. <https://eprint.iacr.org/2018/163>.
- ¹¹ Jérémy Jean. TikZ for Cryptographers. <https://www.iacr.org/authors/tikz/>, 2016.
- ¹² Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.
- ¹³ Neal Koblitz, Alfred Menezes, and Scott Vanstone. The state of elliptic curve cryptography. *Designs, codes and cryptography*, 19:173–193, 2000.

- ¹⁴ Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *Advances in Cryptology — CRYPTO '85 Proceedings*, pages 417–426, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.
- ¹⁵ National Institute of Standards and Technology. Recommendation for key management: Part 1 – general. Technical Report Federal Information Processing Standards Publications (FIPS PUBS) 140-2, Change Notice 2 December 03, 2002, U.S. Department of Commerce, Washington, D.C., 2020.
- ¹⁶ Apple Platform Security. Escrow security for icloud keychain, may 2021. <https://support.apple.com/en-gb/guide/security/sec3e341e75d/web>.
- ¹⁷ Y. Sheffer, G. Zorn, H. Tschofenig, and S. Fluhrer. An eap authentication method based on the encrypted key exchange (eke) protocol. rfc 6124, RFC Editor, February 2011. <https://www.rfc-editor.org/rfc/rfc6124.txt>.
- ¹⁸ John Vollbrecht, James D. Carlson, Larry Blunk, Dr. Bernard D. Aboba, and Henrik Levkowetz. Extensible authentication protocol (eap). rfc 3748, RFC Editor, June 2004. <https://www.rfc-editor.org/rfc/rfc3748.txt>.
- ¹⁹ T. Wu, D. Taylor, N. Mavrogiannopoulos, and T. Perrin. Using the secure remote password (srp) protocol for tls authentication. RFC 5054, RFC Editor, November 2007. <https://www.rfc-editor.org/rfc/rfc5054.txt>.
- ²⁰ Tom Wu. The srp authentication and key exchange system. RFC 2945, RFC Editor, September 2000. <https://www.rfc-editor.org/rfc/rfc2945.txt>.

Appendix A

Python implementation of EKE

While researching Bellovin and Merritt's EKE scheme,² 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

```
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 =
        ↪ 12b(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](#).²⁰ 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

```
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"]

# 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))

```