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
10447494
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, the Rust discord community, Crypto Hack and my CTF Team (0rganizers). Without your help none of this would have been possible.

I would also like to thank my wonderful boyfriend for keeping me sane throughout and cheering me up when things weren't going so well.

Contents

CHAPTER 1

Introduction

Today's standards for Encryption and Authentication are fundamentally broken. The age old standard practice of obtaining an encrypted connection to a server, then sending a cleartext password to authenticate users is flawed. PAKEs are a profoundly different way of looking at how encryption and authentication are performed, and could represent the basis for a safer and more secure future even in the face of the ever increasing threat quantum computers represent to encryption.

1.1 Aims

This project aims to understand the nature of how PAKEs work, what they can and cannot do, and how they can be used to improve the security of almost all password based authentication systems. The ”quantum annoying” property of PAKEs will also be explored, to understand how PAKEs can be used to delay the need for post-quantum cryptography by years or even decades [ES21]. Additionally an implementation of a modern PAKE protocol (AuCPace) will be created and and contributed back to the RustCrypto open source project.

1.2 Deliverables

The implementation of the AuCPace protocol in Rust is the core deliverable for this project. Performance of the library will be compared against those for other PAKE protocols. The metrics of execution time, code size and message size will be used.

1.3 Challenges

Working with elliptic curve cryptography proved to be particularly challenging. A lack of understanding of one particular area proved fatal when a catastrophic bug was found in the codebase late in the project. Chapter 5 talks about remediating this bug and ensuring it cannot happen again.

1.4 Structure

The report will follow the following structure:

- Chapter 1 introduces the project and provides an explanation of the objectives.
- Chapter 2 goes into detail on the history of PAKE algorithms and elliptic curve cryptography.
- Chapter 3 explains the design decisions made around the implementation of AuCPace.
- Chapter 4
- Chapter 5
- Chapter 6 wraps up the project, reflecting on the project and giving a conclusion.

CHAPTER 2

Context

2.1 Background On PAKEs

2.1.1 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.

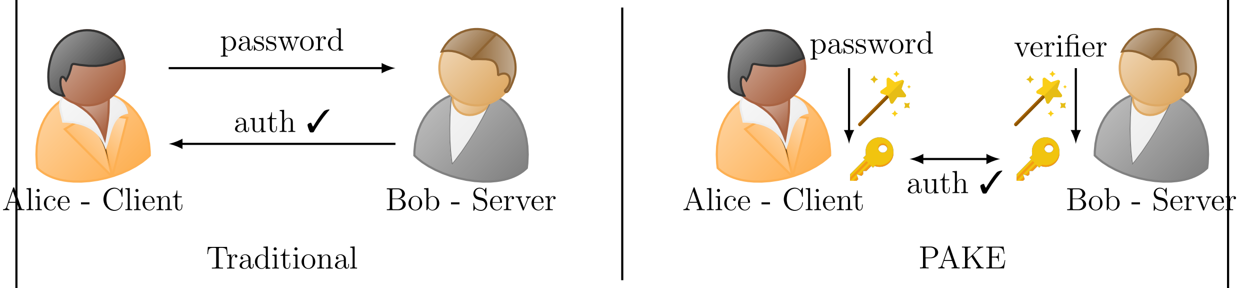


Figure 2.1: An illustration of the difference traditional password based authentication and PAKEs

2.1.2 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.

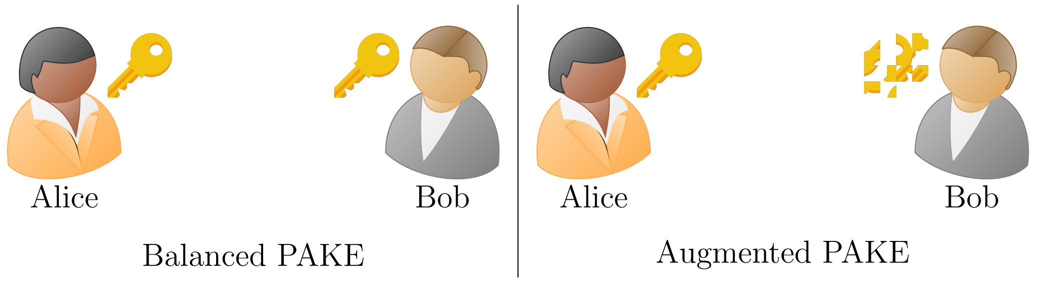


Figure 2.2: An illustration of the difference between Augmented PAKEs and Balanced PAKEs

2.1.3 How Can PAKEs Thwart The Effectiveness Of Quantum Computers

With quantum computing advancing ever faster, classical cryptography systems such as those based on the hardness of the factoring problem (Rivest-Shamir-Adleman (RSA)), or the discrete-logarithm problem (Diffie-Hellman (DH), Elliptic-curve Diffie-Hellman (ECDH)) are increasingly coming into question as quantum computers can solve these problems. During the Crypto Forum Research Group (CFRG)'s recent PAKE standardisation efforts a new property of PAKEs emerged called "quantum annoying". In their paper *The "quantum annoying" property of password-authenticated key exchange protocols*, Eaton and Stebila define the "quantum annoying" property of PAKEs as a scheme where a quantum computer can compromise a scheme but that solving one discrete logarithm problem only gives one guess at the password. This makes PAKEs incredibly attractive as stopgap solution giving researchers a few more vital years to research and test post-quantum cryptography. The need for these years of research was made painfully clear by Castryck and Decru when they recently broke the Supersingular isogeny Diffie-Hellman (SIDH) post-quantum cryptography scheme [CD22].

2.1.4 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 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 2.3 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 2.1: EKE shared parameters

Shared Parameter	Secret	Explanation
P	yes	the shared password

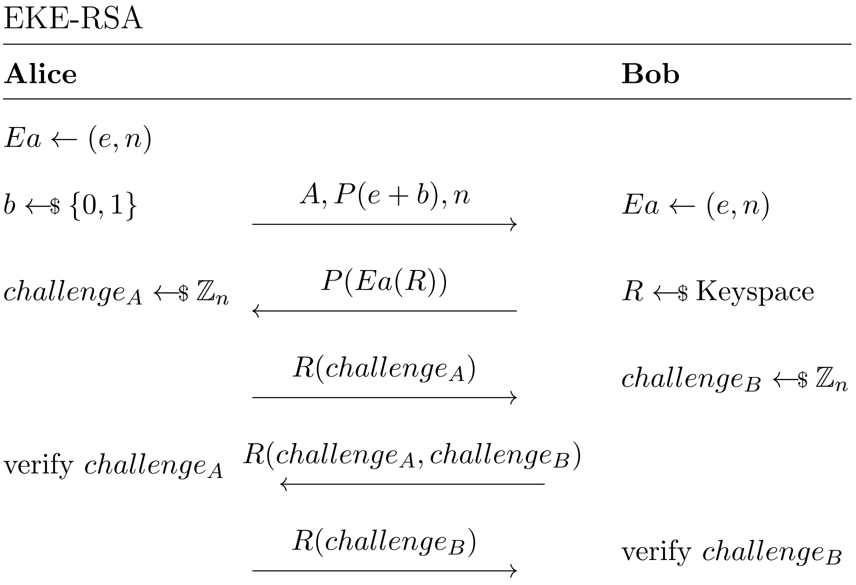

EKE-RSA
Alice
Bob
$Ea \leftarrow (e, n)$
$b \leftarrow\!\$ \{0, 1\}$
$A, P(e + b), n$
$Ea \leftarrow (e, n)$
$challenge_A \leftarrow\!\$ \ \mathbb{Z}_n$
$P(Ea(R))$
$R \leftarrow\!\$$ Keyspace
$R(challenge_A)$
$challenge_B \leftarrow\!\$ \ \mathbb{Z}_n$
verify $challenge_A$
$R(challenge_A, challenge_B)$
$R(challenge_B)$
verify $challenge_B$

Figure 2.3: 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 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 2.2: 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 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

SPAKE2

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

Figure 2.4: 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.

rowcolors0mintbg

Table 2.3: 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

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

Figure 2.5: SRP-6 Protocol

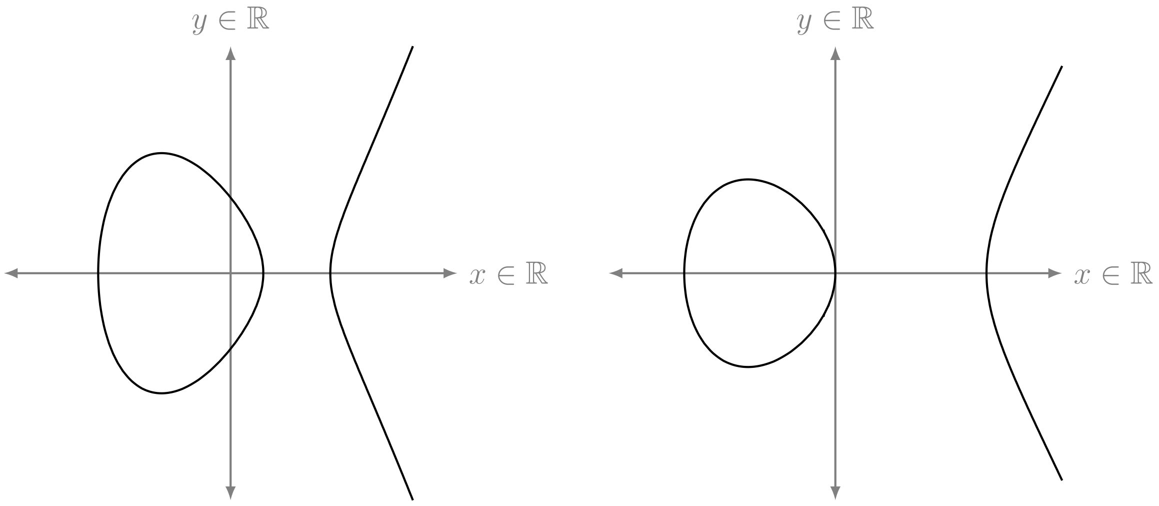
2.2 Elliptic Curve Cryptography

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 [ST20a].

2.2.1 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

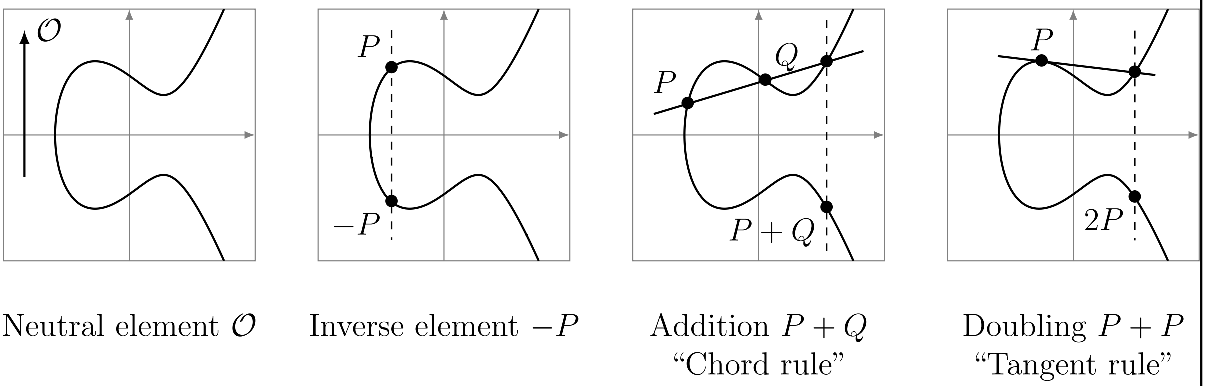
Figure 2.6: Elliptic curves over $\mathbb{R}$, Adapted from TikZ for Cryptographers [Jea16]

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].

2.2.2 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 2.7 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 2.7: 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 - $\mathbb{Z}_{89}$.

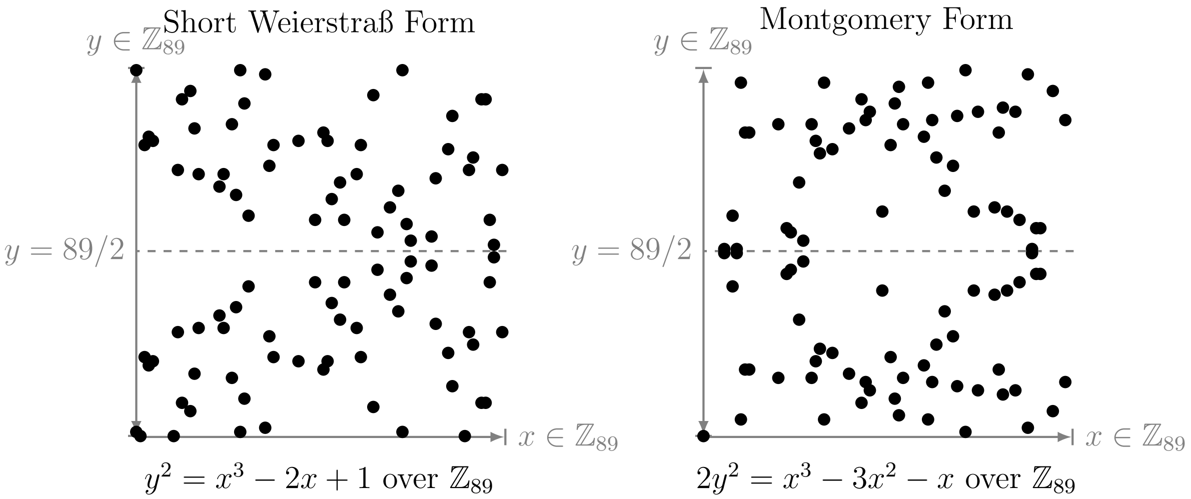


Figure 2.8: Elliptic curves over $\mathbb{Z}_{89}$, Adapted from TikZ for Cryptographers [Jea16]

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.

2.2.3 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.

2.3 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) 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.

2.3.1 CHIP+CRISP

Introduced in 2020 by Cremers et al., CHIP and CRISP are two protocol compilers which instantiate 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.

2.3.2 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.

2.3.3 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.

2.4 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?
- Are there patents covering 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 readily accepted into their collection of PAKEs – `https://github.com/RustCrypto/PAKEs`. RustCrypto will be discussed further in section 2.6. Considerations about open source contribution will be omitted for this reason. For now I will omit considerations about open source contribution.

Table 2.4: Modern PAKE Protocol Comparison

Protocol	Applicability	Implementations	Existing Solutions	Patented
CHIP+CRISP	WiFi	C++ reference impl	Pre-Shared Key (PSK)	Yes (IB-KA)
KHAPE	Replacing PKI	Educational Rust impl	OPAQUE	Yes (HMQV)
AuCPace	IIOT	C reference impl + Go impl	hard coding the key	No

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. Additionally it is the only one which is not under patent of any kind. 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.

2.5 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 2.9 and 2.10 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

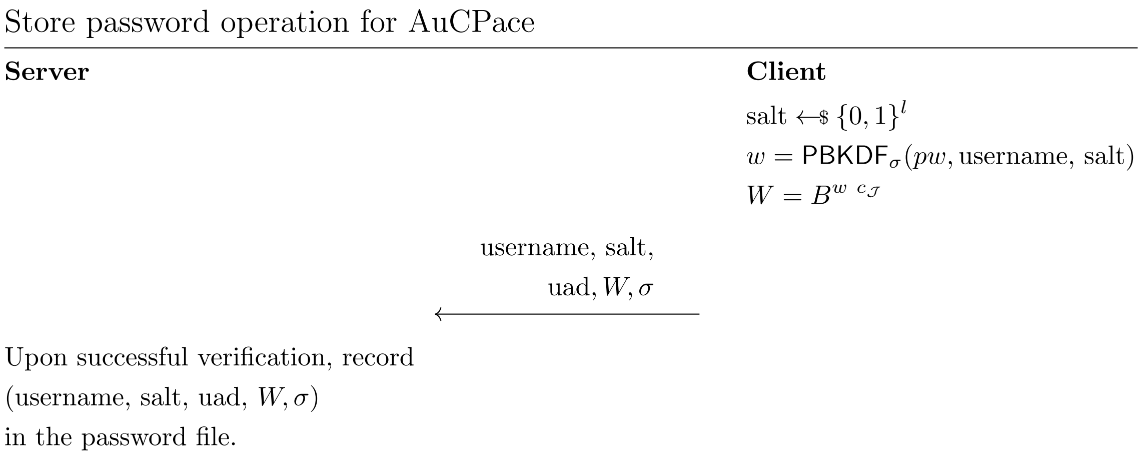
  - the server generates a new ephemeral DH key
  - the server receives the client's username, looks up the verifier and salt.
  - the server then sends across the group $\mathcal{J}$, the public key X, the salt and the parameters for the PBKDF - σ
  - both then compute the Password Related String (PRS), 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

Upon successful verification, record
(username, salt, uad, W, σ)
in the password file.

Figure 2.9: AuCPace protocol for password configuration.

AuCPace

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

Figure 2.10: 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, and a table showing the high level tradeoffs that are made by picking each protocol variant.

Store password operation for AuCPace

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

AuCPace Augmentation Layer

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

Figure 2.11: Differences in Partial Augmentation

Store password operation for strong AuCPace

Server		**Client**
		$\cancel{\text{salt} \leftarrow\!\!\$ \{0,1\}^l}$
		$\text{q} \leftarrow\!\!\$ \{\mathbf{0},\mathbf{1}\}^{\mathbf{l}}$
		$\mathbf{Z} = \mathsf{Map2Point}(\mathsf{H}_{\mathbf{1}}(\text{username}\|\|\mathbf{pw}))$
		$\text{salt} = \mathbf{Z}^{\mathbf{q}\ \mathbf{c}_{\mathcal{J}}}$
		$w = \mathsf{PBKDF}_\sigma(pw, \text{user}, \text{salt})$
		$W = B^{w\ c_{\mathcal{J}}}$
	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 2.12: Differences in Strong AuCPace

implicit auth CPace substep

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

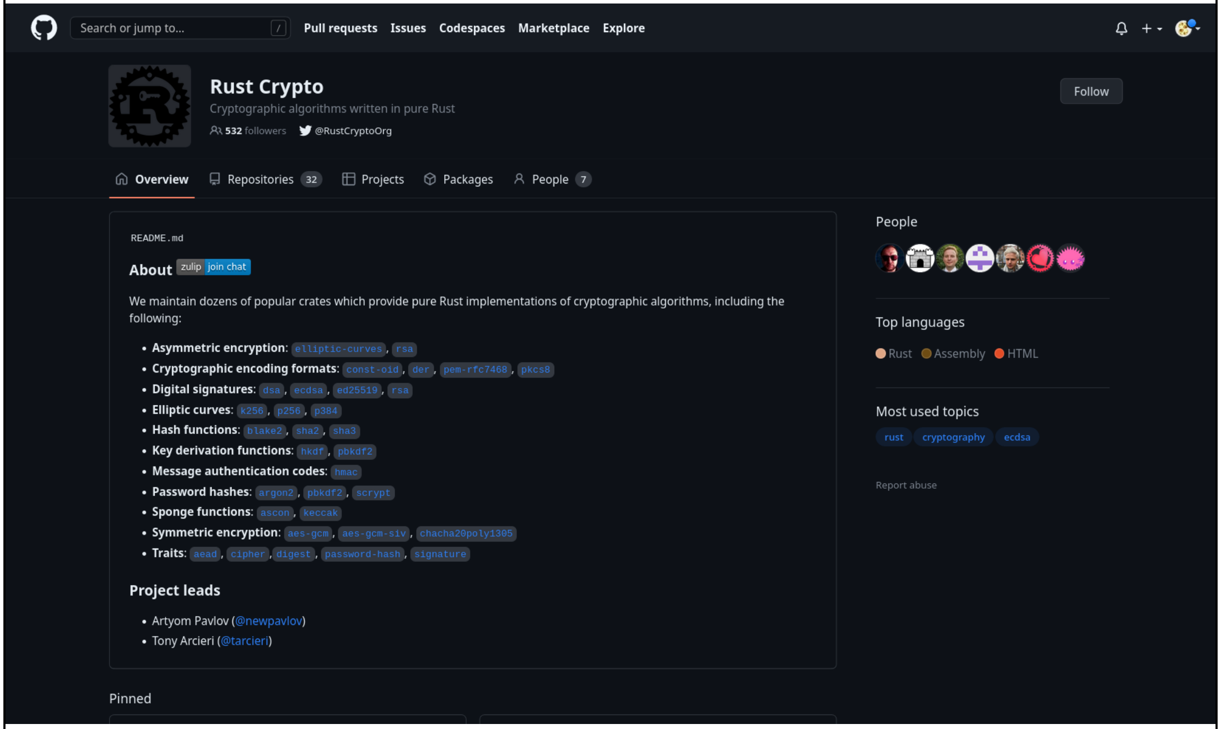
Figure 2.13: Differences in Implicit Authentication

Table 2.5: 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]

2.6 Who Are RustCrypto?

RustCrypto is a GitHub Organisation / online community who are dedicated to implementing fast and secure Cryptography in pure rust. Pure rust in this context means that all of the code is in rust and there are no Foreign Function Interface (FFI) bindings to other (normally C) libraries which implement the functionality (see fig. 2.15). RustCrypto have implemented many Cryptographic primitives and also have an existing repository for PAKE algorithms. Using RustCrypto's libraries provides a good foundation for building any protocol as well as opening up opportunities to open source the implementation back to them. A number of major companies also use RustCrypto's code to secure their applications, e.g. 1Password who use RustCrypto for their Password Manager. The decision to use Rust will be discussed further in section 3.1.



Rust Crypto

Cryptographic algorithms written in pure Rust

532 followers @RustCryptoOrg

README.md

About zulip join chat

We maintain dozens of popular crates which provide pure Rust implementations of cryptographic algorithms, including the following:

- **Asymmetric encryption**: `elliptic-curves`, `rsa`
- **Cryptographic encoding formats**: `const-oid`, `der`, `pem-rfc7468`, `pkcs8`
- **Digital signatures**: `dsa`, `ecdsa`, `ed25519`, `rsa`
- **Elliptic curves**: `k256`, `p256`, `p384`
- **Hash functions**: `blake2`, `sha2`, `sha3`
- **Key derivation functions**: `hkdf`, `pbkdf2`
- **Message authentication codes**: `hmac`
- **Password hashes**: `argon2`, `pbkdf2`, `scrypt`
- **Sponge functions**: `ascon`, `keccak`
- **Symmetric encryption**: `aes-gcm`, `aes-gcm-siv`, `chacha20poly1305`
- **Traits**: `aead`, `cipher`,`digest`, `password-hash`, `signature`

Project leads

- Artyom Pavlov (@newpavlov)
- Tony Arcieri (@tarcieri)

Figure 2.14: RustCrypto's Github Organisation page

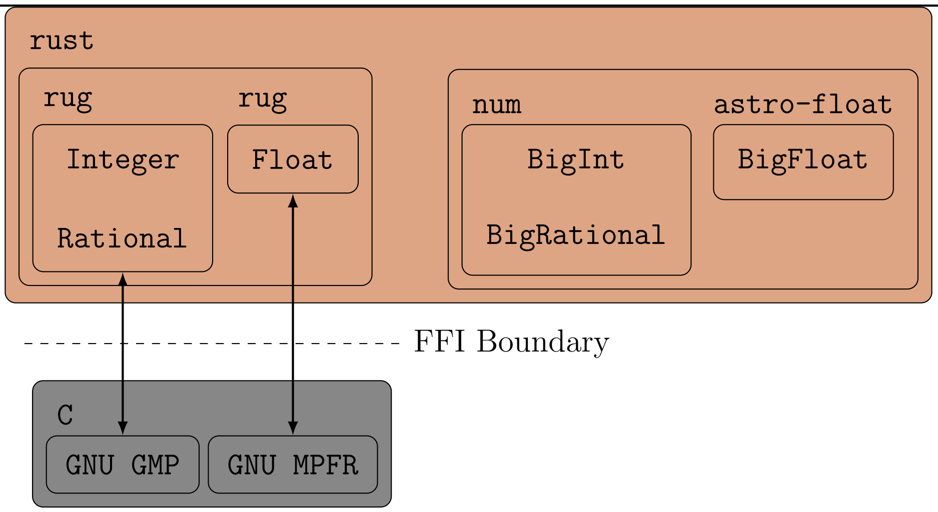


Figure 2.15: Pure Rust library (`num+astro-float`) vs Rust wrapper library (`rug`)

CHAPTER 3

Design

3.1 Why Rust?

AuCPace explicitly targets IIOT in it's design. Rust is rapidly becoming a popular choice for IOT and embedded software applications. This is due to it's focus on memory safety, developer experience and it's strong embedded ecosystem. Libraries like Embassy and RTIC allow the user to program high level logic and use powerful abstractions to interact with the hardware through Rust objects, while still compiling down to small and efficient binaries. Embassy is especially impressive as they have implemented an async executor so that multitasking in embedded applications can be performed with the same async/await framework that programmers are familiar with. A short Embassy examples is shown in listing 1. Tools such as `probe-rs` allow developers to maintain the same workflow they would when working on a normal rust binary, by implementing a `cargo` runner which flashes the binary to the embedded device then using Real-Time-Transfer (RTT) to receive debug messages from the device. Those debug messages can be setup automatically using libraries such as `defmt_rtt` which use RTT to send a compressed representation of the debug message to be formatted later on using a technique called deferred formatting, allowing for debug messages to take up a fraction of the size of the original message. Together this makes rust a compelling choice for writing embedded code.

Rust Listing 1: Embassy async/await example

```rust
use defmt::info;
use embassy::executor::Spawner;
use embassy::time::{Duration, Timer};
use embassy_nrf::gpio::{AnyPin, Input, Level, Output, OutputDrive, Pin,
↪ Pull};
use embassy_nrf::Peripherals;

// Declare async tasks
#[embassy::task]
async fn blink(pin: AnyPin) {
    let mut led = Output::new(pin, Level::Low, OutputDrive::Standard);

    loop {
        // Timekeeping is globally available, no need to mess with hardware timers.
        led.set_high();
        Timer::after(Duration::from_millis(150)).await;
        led.set_low();
        Timer::after(Duration::from_millis(150)).await;
    }
}

// Main is itself an async task as well.
#[embassy::main]
async fn main(spawner: Spawner, p: Peripherals) {
    // Spawned tasks run in the background, concurrently.
    spawner.spawn(blink(p.P0_13.degrade())).unwrap();

    let mut button = Input::new(p.P0_11, Pull::Up);
    loop {
        // Asynchronously wait for GPIO events, allowing other tasks
        // to run, or the core to sleep.
        button.wait_for_low().await;
        info!("Button pressed!");
        button.wait_for_high().await;
        info!("Button released!");
    }
}
```

Rust is also very well suited for implementing cryptographic software. It's lifetimes system and compile time safety guarantees make it ideal for building security focused software. Rust was recently added to NIST's list of "Safer Languages" which it recommends for writing safety focused programs in [ST23].

As well as this many algorithms, formats and primitives are implemented, and freely available as crates for anyone to use. Rust's trait system also lends itself well to this, it is possible to use implement a trait representing an elliptic curve and then an algorithm can be written to be agnostic about the curve that it is using for instance. This allows library writers to easily write generic code to give user's of the libraries as much flexibility and choice around how they implement their program. This is especially important for systems which might need to interact with legacy systems or that need to provide a certain level of security for Federal Information Processing Standards (FIPS) standards like FIPS-140-2 [ST20b].

3.2 Planning The Library

Before implementing AuCPace it was necessary to plan ahead what libraries to use. Without planning it would be easy to end up in a situation where different libraries aren't compatible with each other, or have become superseded by another library as this information is not readily available on crates.io (crates.io is the package repository for all public rust packages).

3.2.1 What Primitives Do We Need To Implement AuCPace?

AuCPace has many parameters which can be changed to drastically change how the protocol works, this is by design to allow customisability for each user's needs, however it can be quite confusing to navigate. As such it is worthwhile to look at the parameters are and thus what primitives we will need. Tables 3.1 and 3.2 are partially reproduced from [HL18] just in significantly fewer words.

Table 3.1: AuCPace Parameters

parameter	explanation
PBKDF_σ	A PBKDF parameterised by σ. The parameters of the PBKDF are algorithm specific, but usually would include settings such as the memory consumption of the algorithm, the hash used or the iteration count (number of times to perform the hash).
$\mathcal{C}, \mathcal{J}, c_\mathcal{J}, B$	A (hyper-)elliptic curve $\mathcal{C}$ with a group $\mathcal{J}$ with co-factor $c_\mathcal{J}$ and a DH protocol operating on both, $\mathcal{C}$ and it's quadratic twist $\mathcal{C}'$. B denotes the DH base point in $\mathcal{J}$.
$\mathsf{Map2Point}$	A function mapping a string s to a point from a cryptographically large subgroup $\mathcal{J}_m$ of $\mathcal{C}$. The inverse map $\mathsf{Map2Point}^{-1}$ is also required.
$\mathsf{H}_0 \ldots \mathsf{H}_5$	A set of 6 distinct hash functions.

Table 3.2: Selected parameters of the reference implementation – AuCPace25519

parameter	explanation
PBKDF_σ	Scrypt [Per16] an optimally memory-hard [Alw+17] PBKDF, parameterised with a memory usage of 32Mb.
$\mathcal{C}, \mathcal{J}, c_\mathcal{J}, B$	Curve25519 [Ber06] a Montgomery form elliptic curve, with excellent speed properties. X25519 an x-coordinate-only DH protocol.
$\mathsf{Map2Point}$	The Elligator2 map introduced by Bernstein et al. in [Ber+13].
$\mathsf{H}_0 \ldots \mathsf{H}_5$	The SHA512 hash function where the index is prepended as a little-endian four-byte word.

So in summary we need the following primitives:

- a PBKDF
- an elliptic curve, a group on the curve, a DH protocol operating on the group
- a mapping from strings to curve points
- a hash function

3.2.2 What Rust Libraries Actually Exist For Cryptography?

There are many sites online which act as collections of rust packages that you can search by topic to find similar or related packages. The Rust Cryptography Interest Group (RCIG) maintain a list of Rust's Cryptographic libraries at `https://cryptography.rs/`, this proved to be a great help while researching libraries.

For the required primitives the following Rust crates were identified as potential candidates:

- The PBKDF:
  - `argon2` - RustCrypto's Argon2 implementation
  - `pbkdf2` - RustCrypto's PBKDF2 implementation
  - `scrypt` - RustCrypto's Scrypt implementation
  - `rust-bcrypt` - a pure Rust Bcrypt implementation
  - `rust-argon2` - a pure Rust Argon2 implementation
  - password-hash - trait to allow implementations to be generic over the password hashing algorithm used
- The elliptic curve:
  - `curve25519-dalek` - Dalek Cryptography's implementation of Curve25519 and Ristretto255 [Val+19]
  - `elliptic-curve` - traits for operating over a generic elliptic curve, part of RustCrypto
  - `elliptic-curves` - RustCrypto's meta-repo holding implementations for the following curves: brainpoolP256r1/t1, brainpoolP384r1/t1, Secp256k1, P-224, P-256, P-384, 1P-52
- The Map2Point function:
  - `curve25519-dalek` - includes `RistrettoPoint::from_uniform_bytes` which implements Ristretto flavoured Elligator2
  - `elliptic-curve` - includes `MapToCurve` which implements the hash-to-curve operation for NIST P-256 and Secp256k1
- The hash function:
  - `digest` - a trait for operating generically over hash functions, from RustCrypto
  - `hashes` - RustCrypto's meta-repo holding implementations for the following hashes: Ascon, BLAKE2. KangarooTwelve, SHA2, SHA3, Tiger, Whirlpool, and several more.

3.2.3 Picking Crates For The Required Primitives

Where possible the implementation should match the reference implementation. These choices are what the designers have determined as secure presets so the are good choices should a suitable crate exist.

Choosing the PBKDF

Instead of picking a PBKDF up front, the `PasswordHasher` trait from password-hash allows us to be generic over the PBKDF when implementing the library. Allowing users of the library to pick from either Argon2, Scrypt or PBKDF2 at their discretion, or to implement their own algorithm and supply an implementation of `PasswordHasher` for it.

Choosing the Curve and Map2Point operation

Although the `elliptic-curves` repo implements many different elliptic curves, it doesn't implement Curve25519[1], and the `hash2curve` Application Programming Interface (API) for NIST P-256 uses the Optimized Simplified Shallue-van de Woestijne-Ulas (OSSWU) map [WB19], which is known to be less efficient than the Elligator2 map defined for Montgomery curves [Ber+13]. There have also been questions about whether the coefficients used in NIST's suite of curves have been deliberately tampered with [BL13].
Another issue to consider when picking a curve and group is the problem of cofactor handling. To avoid mishandling group cofactors AuCPace shows everywhere a cofactor multiplication is necessary, failing to perform one of these multiplications would be a serious bug. However we can eliminate the need for handling cofactors altogether by using a prime order group, that is a group with a prime number of elements in it. Ristretto255 [Val+19] is one such group built on top of Curve25519. The `curve25519-dalek` crate implements Ristretto255 as well as the Ristretto flavoured Elligator2 map [Ber+13] which implements the required Map2Point operation.

Choosing the hash function

The hash function is another parameter that is easy to be generic over, thanks to the `digest` crate. This allows users to pick from the plethora of hashes implemented by `RustCrypto/hashes`, enabling them to choose whichever hash function is best suited for their application.

3.3 Initial Designs For The Structure Of The Library

Rust is very flexible in regards to how you wish to structure a library, there are many patterns that are known to work well in Rust and as such have become Rust idioms. Rust is fairly unique among programming languages as it offers very little in the way of inheritance, unlike classes in Java or C++, Rust's structs cannot inherit from each other. Instead if you want to implement some functionality on top of another type you must in some way store a value of that type. As such wrapper structs are common in Rust, the most common use of these would be the iterator adapters. The `Iterator` trait from the standard library has many methods for providing common operations which are agnostic to what is being iterated over, e.g. `Iterator::filter`, `Iterator::map` and `Iterator::rev`. Each of these methods returns a specialised struct which contains the initial iterator, specifically `std::iter::Filter`, `std::iter::Map` and `std::iter::Rev` for the aforementioned methods. These structs are all *owning*, the concept of Ownership is central to Rust, it forms the basis for how the borrow checker works and is the main mechanism by which Rust can guarantee memory safety. In general owned types are always easier to work with than borrowed ones, you don't have to keep track of lifetimes and in general life is easier. The main use-case for references is for when you have some value that you either cannot copy, (e.g. a Mutex), or really don't want to copy (e.g. 10Gb worth of data).

[1] there is currently a push to have it included in the crate, though it is still early on and the implementation is not fit for use

This preference for Owned values leads to one slightly messy but easy to implement pattern whereby one struct is used to implement all of the functionality and all of the state is bundled in this one struct. While easy to implement this approach does have some drawbacks, some of the state might only be needed for one operation then it is worthless, however having everything in one struct means that the space is still allocated whether you need it or not. Being aware of this is especially pertinent as the amount of state required gets larger.

3.3.1 There Are Other PAKEs Implemented In Rust, How Are They Designed?

RustCrypto have implemented two PAKEs - SRP and SPAKE2. SPAKE2 is the simpler of the two protocols so lets analyse it first.

Exploring RustCrypto's SPAKE2 implementation

A diagram of the SPAKE2 protocol can be found in fig. 2.4. The core of the implementation is the following struct:

Rust Listing 2: SPAKE2 Struct

```rust
/// SPAKE2 algorithm.
#[derive(Eq, PartialEq)]
pub struct Spake2<G: Group> {
    //where &G::Scalar: Neg {
    side: Side,
    xy_scalar: G::Scalar,
    password_vec: Vec<u8>,
    id_a: Vec<u8>,
    id_b: Vec<u8>,
    id_s: Vec<u8>,
    msg1: Vec<u8>,
    password_scalar: G::Scalar,
}
```

It contains an owned copy of every piece of data needed to run the entire protocol, although there are quite a few members here, SPAKE2 effectively requires it as the final key SK_B is calculated as $H(A, B, X^*, Y^*, Kb)$, in addition this is a very simple protocol at only one message each way. This means that there isn't as much overhead for keeping lots of data around.

The API exposed by the struct is also very simple, there are number of `start_*` methods which begin the protocol and generate initial values from the Cryptographically Secure Pseudo Random Number Generator (CSPRNG), all of these methods return a tuple `(<state>, <message>)`. The message can then be sent to the other party and when the response is received there is a single `finish` method which takes in this response and produces a `Result<Vec<u8>>`[2], this contains the shared key if everything went well and an error otherwise.

[2] Rust's Result type is used to return a value or an error, the type system forces handling this value and the code will panic if a value is expected and an error occurs. It is very similar to Haskell's Maybe type.

In summary it is implemented as one large struct with many helper methods for all the different ways to start the protocol.

Exploring RustCrypto's SRP implementation

A diagram of the SRP protocol can be found in fig. 2.5.
As SRP is an Augmented PAKE it is implemented with a separate `Client` and `Server` struct as seen below.

Rust Listing 3: SRP Server Struct

```rust
/// SRP server state
pub struct SrpServer<'a, D: Digest> {
    params: &'a SrpGroup,
    d: PhantomData<D>,
}
```

Rust Listing 4: SRP Client Struct

```rust
/// SRP client state before handshake with the server.
pub struct SrpClient<'a, D: Digest> {
    params: &'a SrpGroup,
    d: PhantomData<D>,
}
```

However it is plain to see these structs hold the same values, the only difference is the methods available on each. This is a completely different approach to the SPAKE2 library. In this design how the state is stored is left entirely up to the library consumer, with these structs simply implementing all of the methods for the computation at each step. This does expose a very flexible API and store the absolute minimum amount of data, however it doesn't do anything to protect from accidental misuse by the programmer.

3.3.2 Initial Design Plan

To support contributing the implementation back to RustCrypto, the library will be implemented as a fork of `https://github.com/RustCrypto/PAKEs`, where the AuCPace implementation will be added as a new crate in the Cargo workspace.
As a prototype for the library functionality a design in the style of RustCrypto's SRP implementation was created to explore how the computations required by AuCPace look in rust and how the different libraries interact together.
After this prototype version several attempts were made at a more user-friendly / intuitive, eventually settling on a design where the User ”moves” between various structs by passing messages between the server and client. Each move returns either just the next state, or a tuple of a message and the next state, it is then the user's job to manage just the communication of messages between the client and server. This approach reduces the cognitive overhead of the developer and allows them to just focus on the core of a protocol - passing messages.

CHAPTER 4

Implementation

4.1 Creating The Initial Prototype

The initial prototype was based around the SRP implementation from RustCrypto.

Rust Listing 5: AuCPace Server Prototype

```rust
pub struct AuCPaceServer<D, CSPRNG, const K1: usize>
where
    D: Digest<OutputSize = U64> + Default,
    CSPRNG: CryptoRng + RngCore,
{
    rng: CSPRNG,
    secret: u64,
    d: PhantomData<D>,
}
```

Rust Listing 6: AuCPace Client Prototype

```rust
pub struct AuCPaceClient<D, CSPRNG, const K1: usize>
where
    D: Digest<OutputSize = U64> + Default,
    CSPRNG: RngCore + CryptoRng,
{
    rng: CSPRNG,
    d: PhantomData<D>,
}
```

This struct then implemented methods for each of the computations needed by the protocol. There wasn't anything especially wrong with an implementation like this, but I felt like a higher level API would be nicer to work with and would reduce the cognitive load on any developer using the library.

4.2 Forming The Struct Based API

The first step to developing the struct-based API was deciding what structs were needed, and at what point structs should be created in the protocol. Two models were considered, one where all inputs to the protocol are done up front and the structs solely represent

different steps in the protocol, and another where a struct is declared at each point there is an input to the protocol diagram.
The two protocols would look like this:

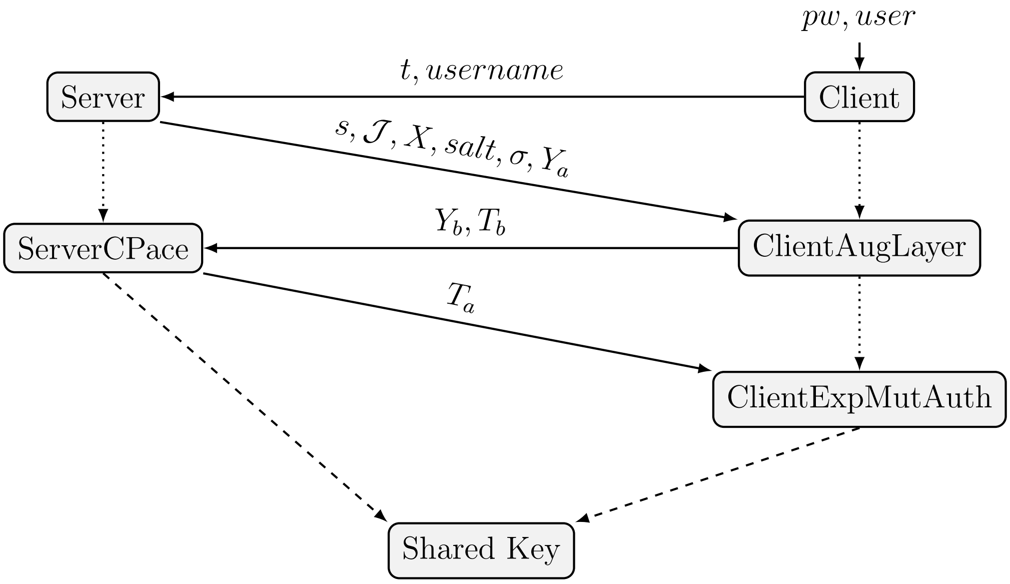


Figure 4.1: Struct based API with all inputs at the start.

At first this layout looks highly desirable, you have the minimum number of messages between each side, and theres only a few structs on each side. However this poses a few problems when it comes to protocol flexibility and message ordering. This layout only allows one message ordering and one protocol variant. There are many reasons that one might want to change the message ordering or protocol variant. Message ordering is often dependent on external factors such as what the your transport layer provides etc, rigidity in this can make implementing a full protocol far more complicated than it needs to be. Only implementing on protocol variant is a viable choice, though it means you are inflicting your choices on the users of your library. If you are making a library which is deliberately opinionated then this might even be desirable, however to support the most number of use cases possible flexibility is highly desirable.
The layout in fig. 4.2 provides the flexibility called for. Although there are many more structs, only one need exist at once, meaning that memory can be reused by allocating each in-place on the stack. While it may also appear that this version requires more individual messages, it is possible to batch the messages together in the same request as there is nothing to stop each side advancing until they need to receive the next message each time. This layout also has the advantage that each individual struct is less complicated and individual functionality can be tested easier. Protocol flexibility is also provided by this layout, implementing (strong) AuCPace is just changing out the aug layer, implementing implicit authentication is just bypassing the last struct. In summary this layout provides a high level of modularity, meaning that both protocol and messaging flexibility can be provided for.

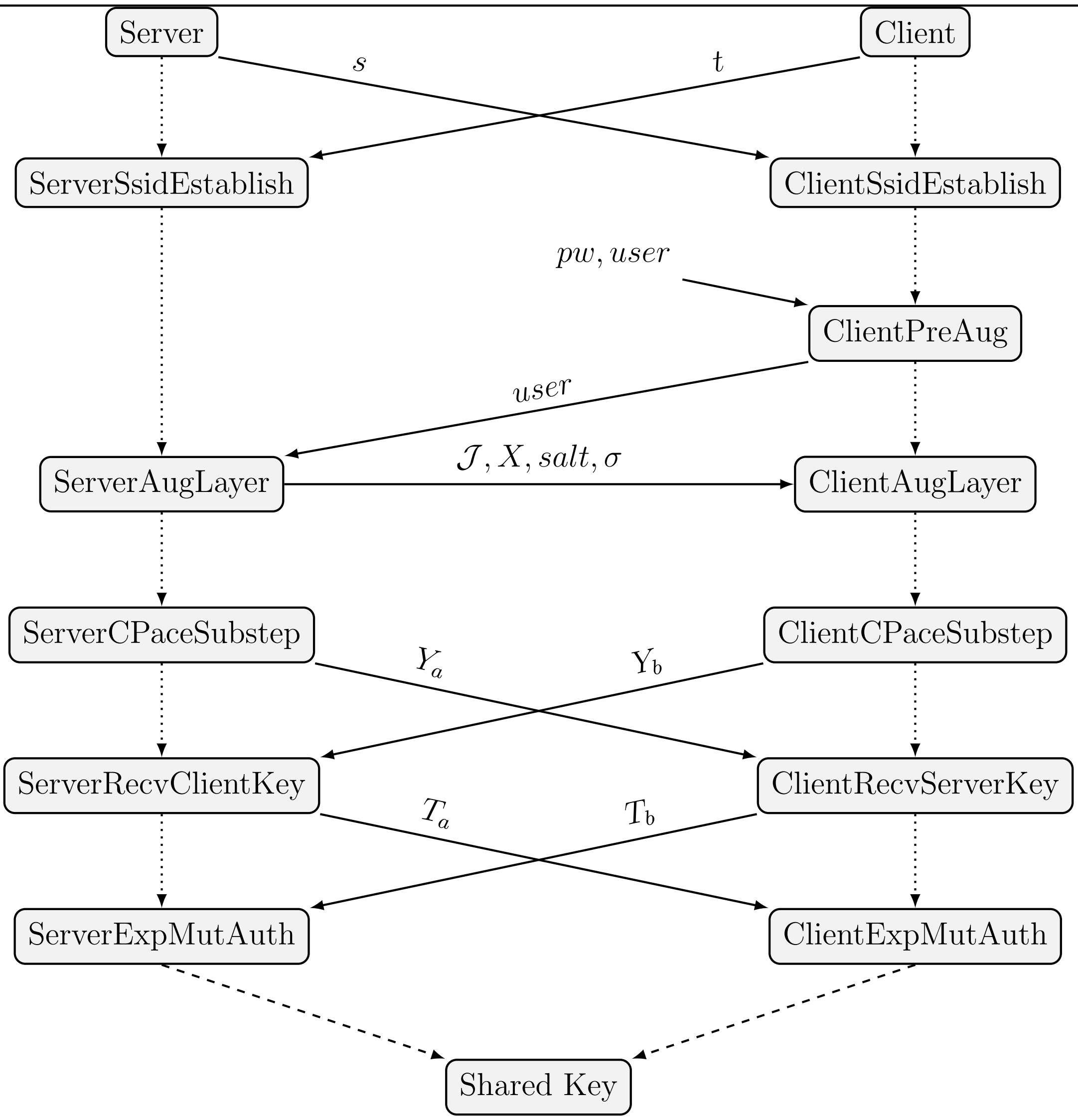


Figure 4.2: Struct based API with inputs as required

4.2.1 Modeling LookupW In Rust's Type System

The *loopupW* operation from the protocol diagram is how the verifier is retrieved from the database. However this proves to be quite from a library implementation point of view, we want to support as many different use cases as possible. But this means that we cannot have some struct which we use as the database as everyone will have different needs for the database. For example a user who wants to store data in their flash storage probably wants a very different implementation to someone who wants to store data in sqlite3. As a single implementation is not suitable for all users, we need to define a Rust trait for users to implement so they can use whatever implementation best fits their needs. We have a few requirements for this trait:

- It should be flexible enough to allow for many different styles of implementation.
- It should be generic over the type of the verifier.
- It should represent both the *lookupW* operation and storing items in the database, for the registration step.

Rust Listing 7: Database Trait

```rust
pub trait Database {
    type PasswordVerifier;

    fn lookup_verifier(
        &self,
        username: &[u8]
    ) -> Option<(Self::PasswordVerifier, SaltString, ParamsString)>;

    fn store_verifier(
        &mut self,
        username: &[u8],
        salt: SaltString,
        uad: Option<&[u8]>,
        verifier: Self::PasswordVerifier,
        params: ParamsString
    );
}
```

The implementation in listing 7 is the result of several iterations. It meets all of the requirements set out above, as well encoding the optional nature of the `uad` component directly into the type system. The `type PasswordVerifier` part of the listing is what allows implementations to be generic over the type of the verifier, implementors of this trait must specify which type of verifier their database stores. An example implementation of the trait is shown in appendix C.

4.2.2 Adding Implicit Authentication

Implicit authentication was by far the easiest of the protocol variants to add. To implement this variant a new method `implicit_auth(pubkey)` was added to `ClientRecvServerKey` and `ServerRecvClientKey`. This method calculates the shared key *sk* directly without computing any authenticators.

4.2.3 Adding Partial Augmentation

Partial Augmentation was significantly harder to implement as it requires storing and retrieving data similar to the verifier database. To solve this the same approach as the database trait was used. This time storing a public/private keypair instead of a verifier.

4.2.4 Adding Strong AuCPace

Strong AuCPace was by far the hardest to implement, it came with several challenges. As with Partial Augmentation there is a database component, this was solved in the same way by introducing a new trait that models the database from the strong AuCPace protocol. However Strong AuCPace also changes the registration step and the Augmentation Layer, these required new methods to be added to the Client, Server, ClientPreAug and ServerAugLayer structs. On top of this the data which is sent in messages is different for Strong AuCPace, this required adding additional messages for both the client and server. Finally there is the salt unblinding operation:

salt $= UQ^{\frac{1}{r \, (c_{\mathcal{J}})^2} c_{\mathcal{J}}}$, which certainly isn't intuitive. The solution for this is to find the multiplicative inverse of $r(c_{\mathcal{J}})^2$ then multiply that again by $c_{\mathcal{J}}$, and finally we can use it to find the salt from UQ. The implementation of this operation is shown below in listing 8.

Rust Listing 8: Strong AuCPace Salt Unblinding

```rust
// this is a tad funky, in the paper they write (1/(r * cj^2))*cj
// I have interpreted this as the multiplicative inverse of (r * cj^2)
// then multiplied by cj again.
let exponent = (self.blinding_value * cofactor * cofactor).invert() *
↪ cofactor;
let salt = (blinded_salt * exponent).compress().to_bytes();
let salt_string =
↪ SaltString::encode_b64(&salt).map_err(Error::PasswordHashing)?;
```

4.2.5 Adding Feature Flags

Including all of these protocol variants in the library even when a user doesn't need them would cause the library to take up more space than it needs. On Microcontrollers (MCUs) where keeping space used to a minimum is criticially important, this just isn't acceptable. However Rust has a solution designed for exactly this – feature flags. Feature flags are introduced in a Rust project's `Cargo.toml` file, then they can be used inside of code to turn parts on / off. There are a few rules which come with Rust feature flags, for various reasons it is a best practice to keep them as strictly additive, this means that you don't remove functionality with a feature flag, and feature flags should only introduce more APIs to the user.

Adding feature flags to the library turned out to be one of the most time consuming parts of the entire process. Feature flags can go wrong in a myriad of wonderful ways, especially when crates are located in a cargo ”workspace” as is true of the PAKEs repository. This is due to workspaces having special rules for how feature flags are compiled across libraries, however that is beyond the scope of this report.

Notwithstanding the difficulties of adding features across the whole project, individually they are quite simple, as they are just adding an attribute to all functions and structs you want to be feature gated, e.g. `#[cfg(feature = "strong_aucpace")]`.

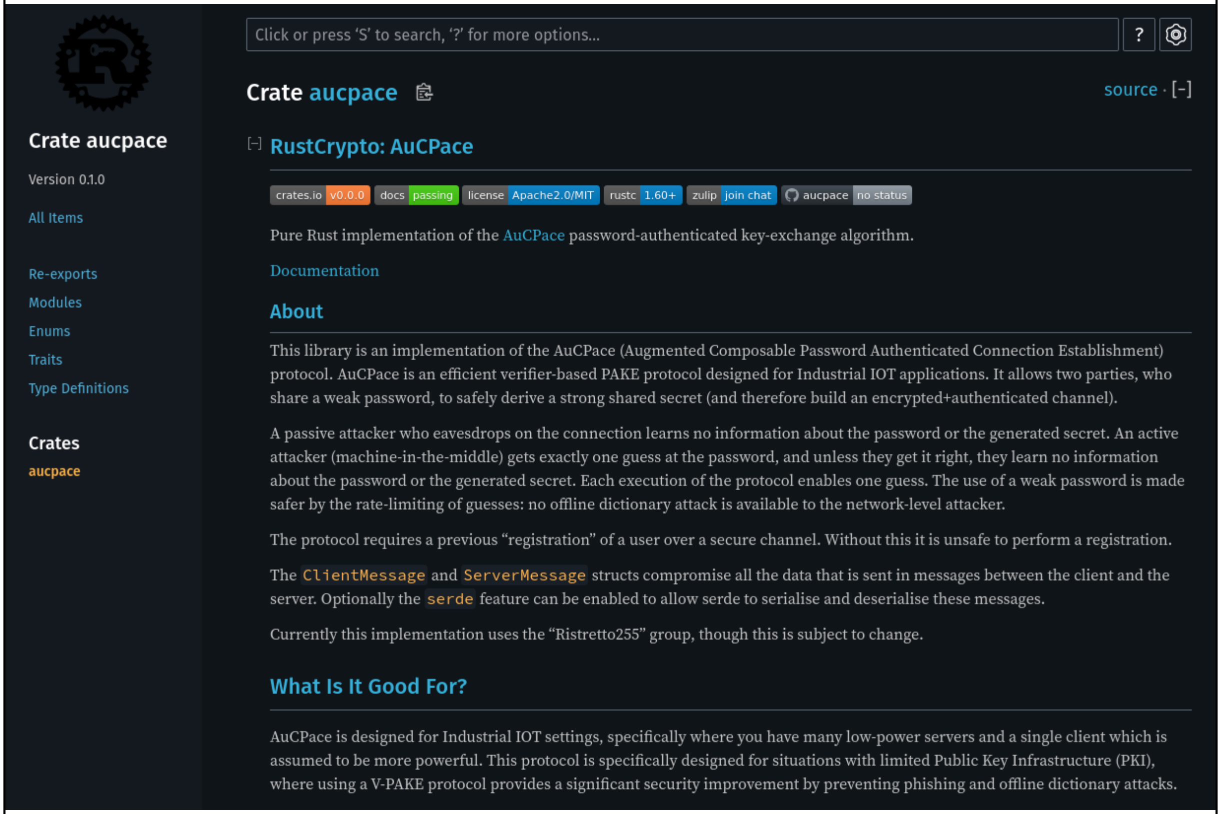
4.2.6 Making It `#[No_std]` Friendly

`#[no_std]` is a Rust attribute that removes the standard library. This is to allow compiling for targets where linking against the standard library is either not appropriate or impossible, e.g. MCUs where there simply isn't the space, or in kernel modules where certain operations aren't allowed. However this comes with a wide range of disadvantages, namely you cannot using anything which allocates memory, and you cannot use anything which interacts with the Operating System (OS). Care was taken to avoid using the standard library during development, however there are a few places where allocating APIs were used to ease development. Namely anywhere that interacts with a user's password. This is due to a conflict between the API of the `password_hash` library, and what the AuCPace paper requires. The `PasswordHasher` trait from `password_hash` allows us to be generic over the PBKDF_σ from the protocol definition,

however it's interface only takes in a password and a salt. This means that to hash together the username, password and salt we need to encode the username and password together in the same piece of data and then pass this to the hasher. The only way to do this for any given username and password is to allocate a buffer big enough to store both items on the heap. But this conflicts with our requirements from `#[no_std]`.
Initially I explored an API which would introduce a new feature flag – `alloc` and then switch it's behaviour based on whether alloc was enabled. Using a fixed size buffer when `alloc` was disabled. However this proved to be very difficult test and ran contrary to the ”strictly additive” guideline for Rust feature flags. After some discussion with RustCrypto in their online Zulip chat (`https://rustcrypto.zulipchat.com`), Tony Arcieri suggested adding a new API which was feature flagged behind `alloc`, then users could use the allocating API if they have an allocator and they can use fixed sized buffer implementation if they didn't. This approach ended up being the best, it even has the benefit that the two methods can be directly tested against each other to ensure they give the same results.

4.2.7 Documentation

As the library is being open sourced back to RustCrypto, great care was taken that everything that needs documentation has it. Rust makes ensuring that everything is documented significantly easier than most languages, it has an attribute – `#![warn(missing_docs)]` that can be used to enable a warning for any item that is missing documentation. This allows guaranteeing a very high level of coverage in docs. In total there are thirty four individual pages of documentation, the main documentation page for the library can be seen in fig. 4.3 and the documentation for one of the structs can be seen in fig. 4.4.



Crate aucpace

[−] RustCrypto: AuCPace

Pure Rust implementation of the AuCPace password-authenticated key-exchange algorithm.

Documentation

About

This library is an implementation of the AuCPace (Augmented Composable Password Authenticated Connection Establishment) protocol. AuCPace is an efficient verifier-based PAKE protocol designed for Industrial IOT applications. It allows two parties, who share a weak password, to safely derive a strong shared secret (and therefore build an encrypted+authenticated channel).

A passive attacker who eavesdrops on the connection learns no information about the password or the generated secret. An active attacker (machine-in-the-middle) gets exactly one guess at the password, and unless they get it right, they learn no information about the password or the generated secret. Each execution of the protocol enables one guess. The use of a weak password is made safer by the rate-limiting of guesses: no offline dictionary attack is available to the network-level attacker.

The protocol requires a previous “registration” of a user over a secure channel. Without this it is unsafe to perform a registration.

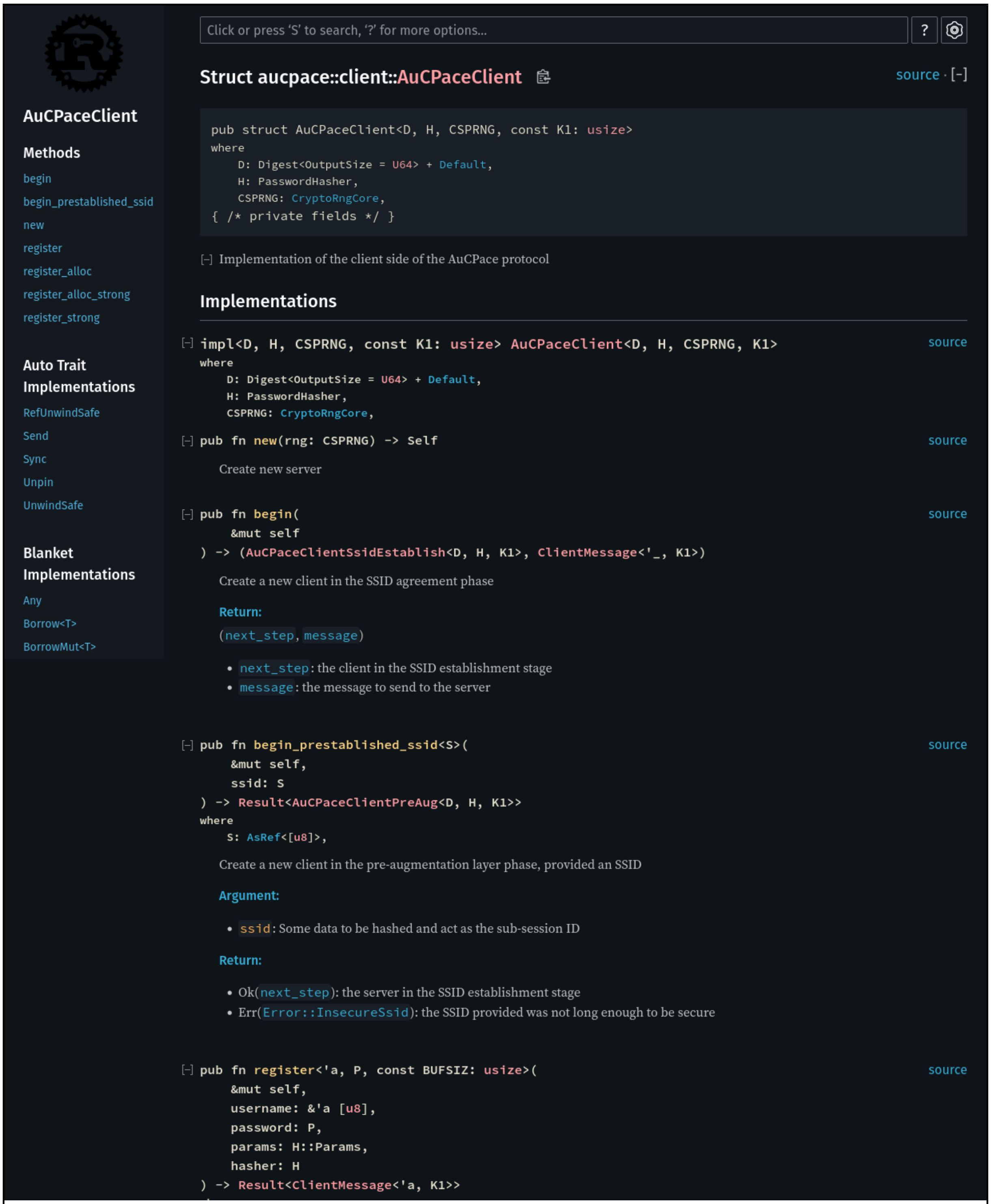
The `ClientMessage` and `ServerMessage` structs compromise all the data that is sent in messages between the client and the server. Optionally the `serde` feature can be enabled to allow serde to serialise and deserialise these messages.

Currently this implementation uses the “Ristretto255” group, though this is subject to change.

What Is It Good For?

AuCPace is designed for Industrial IOT settings, specifically where you have many low-power servers and a single client which is assumed to be more powerful. This protocol is specifically designed for situations with limited Public Key Infrastructure (PKI), where using a V-PAKE protocol provides a significant security improvement by preventing phishing and offline dictionary attacks.

Figure 4.3: Main Documentation Page



Struct aucpace::client::AuCPaceClient

```
pub struct AuCPaceClient<D, H, CSPRNG, const K1: usize>
where
    D: Digest<OutputSize = U64> + Default,
    H: PasswordHasher,
    CSPRNG: CryptoRngCore,
{ /* private fields */ }
```

Implementation of the client side of the AuCPace protocol

Implementations

```
impl<D, H, CSPRNG, const K1: usize> AuCPaceClient<D, H, CSPRNG, K1>
where
    D: Digest<OutputSize = U64> + Default,
    H: PasswordHasher,
    CSPRNG: CryptoRngCore,
```

```
pub fn new(rng: CSPRNG) -> Self
```

Create new server

```
pub fn begin(
    &mut self
) -> (AuCPaceClientSsidEstablish<D, H, K1>, ClientMessage<'_, K1>)
```

Create a new client in the SSID agreement phase

Return:

(`next_step`, `message`)

- `next_step`: the client in the SSID establishment stage
- `message`: the message to send to the server

```
pub fn begin_prestablished_ssid<S>(
    &mut self,
    ssid: S
) -> Result<AuCPaceClientPreAug<D, H, K1>>
where
    S: AsRef<[u8]>,
```

Create a new client in the pre-augmentation layer phase, provided an SSID

Argument:

- `ssid`: Some data to be hashed and act as the sub-session ID

Return:

- Ok(`next_step`): the server in the SSID establishment stage
- Err(`Error::InsecureSsid`): the SSID provided was not long enough to be secure

```
pub fn register<'a, P, const BUFSIZ: usize>(
    &mut self,
    username: &'a [u8],
    password: P,
    params: H::Params,
    hasher: H
) -> Result<ClientMessage<'a, K1>>
```

Figure 4.4: `AuCPaceClient` Documentation

CHAPTER 5

Testing

5.1 Testing For Correctness Of Functionality

As the library is built on other crates, the only tests which can be performed are those on individual components of the protocol, and full run-throughs of the entire protocol.

5.1.1 Testing Individual Components

Traditionally in cryptographic software, test-vectors have been used to test for the correct functionality of various componenets. However due to the use of Ristretto255, instead of raw Curve25519, there are no test-vectors available. Once the implementation was feature complete, I emailed Björn Haase to discuss test vector creation[1]. He was interested in the project and mentioned they were interesteed in a second implementation of AuCPace at Endress+Hauser. However he did not have the time to create test vectors and by this point in the project I was too busy to do so as well. So while test vectors have not yet been created, it is likely they will be in the near future.

5.1.2 Integration Testing

To perform integration testing, the tests were split into four categories based on which of the two major features they required[2].

Table 5.1: Integration Tests Split

`strong_aucpace`	`partial_aug`	test
✗	✗	`test_key_agreement.rs`
✗	✓	`test_key_agreement_partial_aug.rs`
✓	✗	`test_key_agreement_strong.rs`
✓	✓	`test_key_agreement_strong_partial_aug.rs`

Inside of each test file there are two main scaffolding functions: `init` and `test_core`, the former Perfoms the initial setup of the client, server and database, and the latter performs all the functionality which is common to every test. From there there are four tests in each file, each testing the following:

[1] Björn is the creator of AuCPace

[2] This work was completed after the code freeze, so is only available on the github.

Table 5.2: Per-file Test Splits

static SSID	implicit auth	test
✗	✗	test_key_agreement
✗	✓	test_key_agreement_implicit_auth
✓	✗	test_key_agreement_prestablished_ssid
✓	✓	test_key_agreement_prestablished_ssid_implicit_auth

5.2 The Ultimate Test

As with any piece of software the ultimate test is to get the entire thing running / working on real hardware.

5.2.1 Choosing The Microcontroller / Platform

Because AuCPace is intended for use in the IIOT, it was decided that an ST-Microelectronics (STM) based MCU would be a good fit. STM MCUs are very prevelant in industry, have many dev boards supporting a wide variety of different chips, and they also have excellent rust support. Following this decision the NUCLEO-F401RE dev board was selected. It is a small MCU with a modest 512K of flash memory. While certainly not the least powerful platform around it should be representative of a majority of IOT systems.

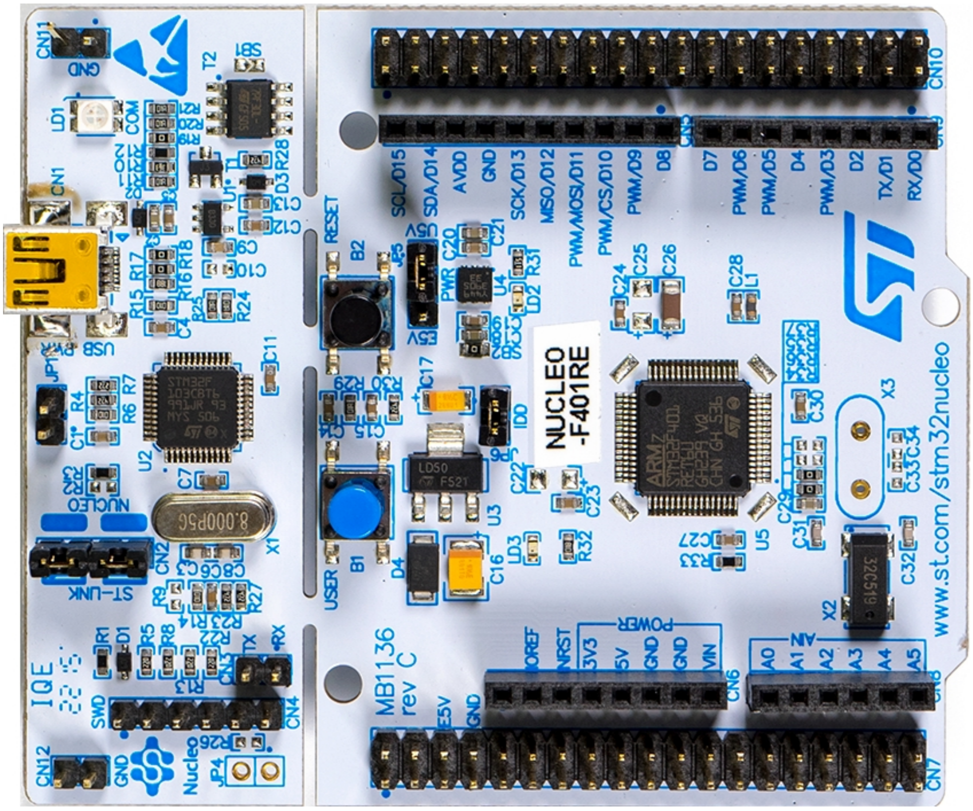


Figure 5.1: NUCLEO-F401RE Dev Board

5.2.2 Choosing An Embedded Rust Platform

Initially the Real-Time Interrupt-drived Concurrency (RTIC) framework was selected as example code for interacting over serial was easy to find online. However getting RTIC to work on the actual board proved challenging as low level details such as clocks and timers confused things considerably.
After a few attempts[3] with RTIC, efforts were switched to using the Embassy project. This proved to greatly alleviate the strain of working in the embedded software world. Embassy performs all low level setup for you and allows the programmer to work with async/await constructs instead of timers etc. There was still some considerable strife in getting the serial connection to work consistently. After a lengthy conversation with `sjm#0205` on the Rust discord server, we came to the conclusion that the board's low quality crystal oscillators were causing the serial connection to get out of sync. Reducing the baud rate to 28800 resulted in a much more reliable connection, this is due to clock accuracy being less important at lower baud rates as there is a bigger margin for error in when lines go high/low[4]. After establishing a reliable serial connection work could now begin on implementing AuCPace.

5.2.3 Implementing The AuCPace Protocol

The server side of `examples/key_agreement.rs` was initially adapted to fit the code structure of the embedded app. However this immediately brought around a problem with the AuCPace implementation, it was refusing to compile. As it happened this was a quirk of how rust works that I wasn't aware of. Care was taken to develop the library using the `#[no_std]` attribute, this tells the compiler that this code isn't allowed to use the Rust standard library. Even the `examples/key_agreement_no_std.rs` example program wasn't enough to weed this out. Because the standard library is still available when linking on x86, the `#[no_std]` example still compiles, despite containing code from the standard library. When it was adapted for the microcontroller, it was targeting `thumbv7em-none-eabihf`, the standard library is simply not available for this target as it doesn't have an operating system. The malignant crate turned out to be `serde-arrays` which, although not containing any code that should require the standard library, it was not marked as `#[no_std]` and thus fails to compile as it implicitly links to the standard library. The solution was very simple thankfully, just weeks prior to attmpting this embedded example, a new crate was released `serde-byte-arrays`, this solved my problem better than `serde-arrays`, was marked `#[no_std]` and was more efficient at the same time! Once these two primitives were reliable, it was quite straightforward to implement the protocol in full.

5.2.4 benchmarking

Benchmarking was performed by switching feature flags on and off on both the client and server. An additional element of the compiler flags was added because Rust's compiler allows you to make tradeoffs between code-size and speed quite easily. The easiest method for doing this is by creating a profile for the Rust compiler to use, there are two profiles normally available – `debug` and `release`; `debug` is intended to be used during development

[3] several days of work
[4] This solution was found after code freeze so is only available on the github

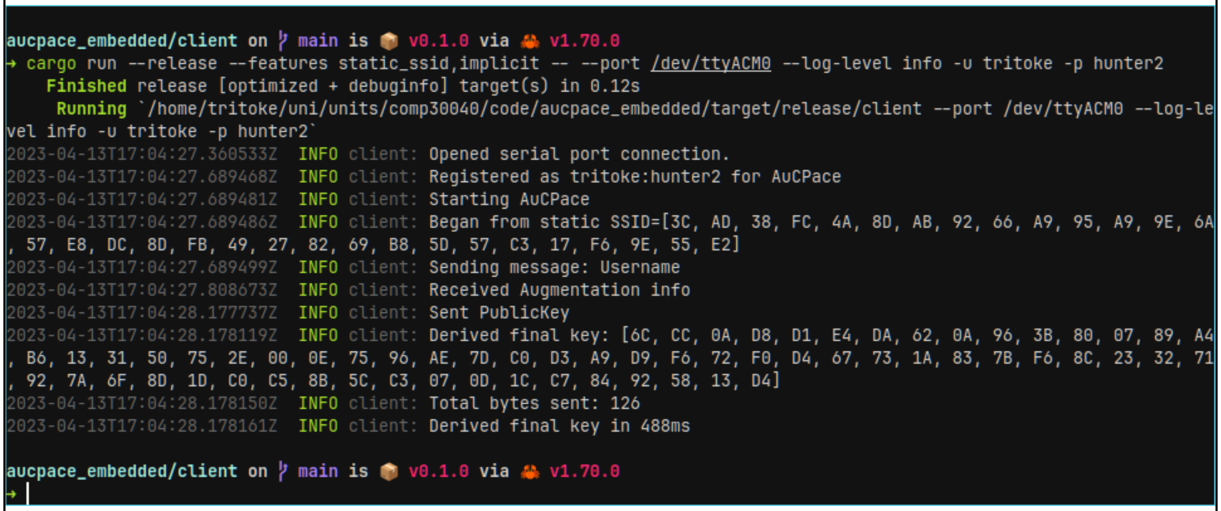
and is tuned for fast compile times and extensive debug info. `release` is tuned for speed and removes all debug info normally. In listing 1 the debug information is re-enabled for `release` mode, this means that the Rust compiler includes DWARF debug in the compiled binary. A `server` profile is also listed, this profile inheirits from `release` meaning that initially it is the same as `release`, `opt-level` is then set to `"s"`, this makes the compiler instead optimise for code size instead of speed. The two remaining options – `lto` and `codegen-units` are less significant and are just there to improve the information available to the compiler to allow it to generate more efficient code.

The ”Compute time” was based on the compute time calculated by the server as it ran through the protocol as can be seen in fig. 5.3. These are based on just one measurement of the time, this is due to the use of constant time algorithms wherever possible, when the number of ticks was measured directly, the variance was on the order of 5 ticks, which is just 0.005ms (5μs), this variance is not big enough to effect the results.

Tables 5.3 to 5.6

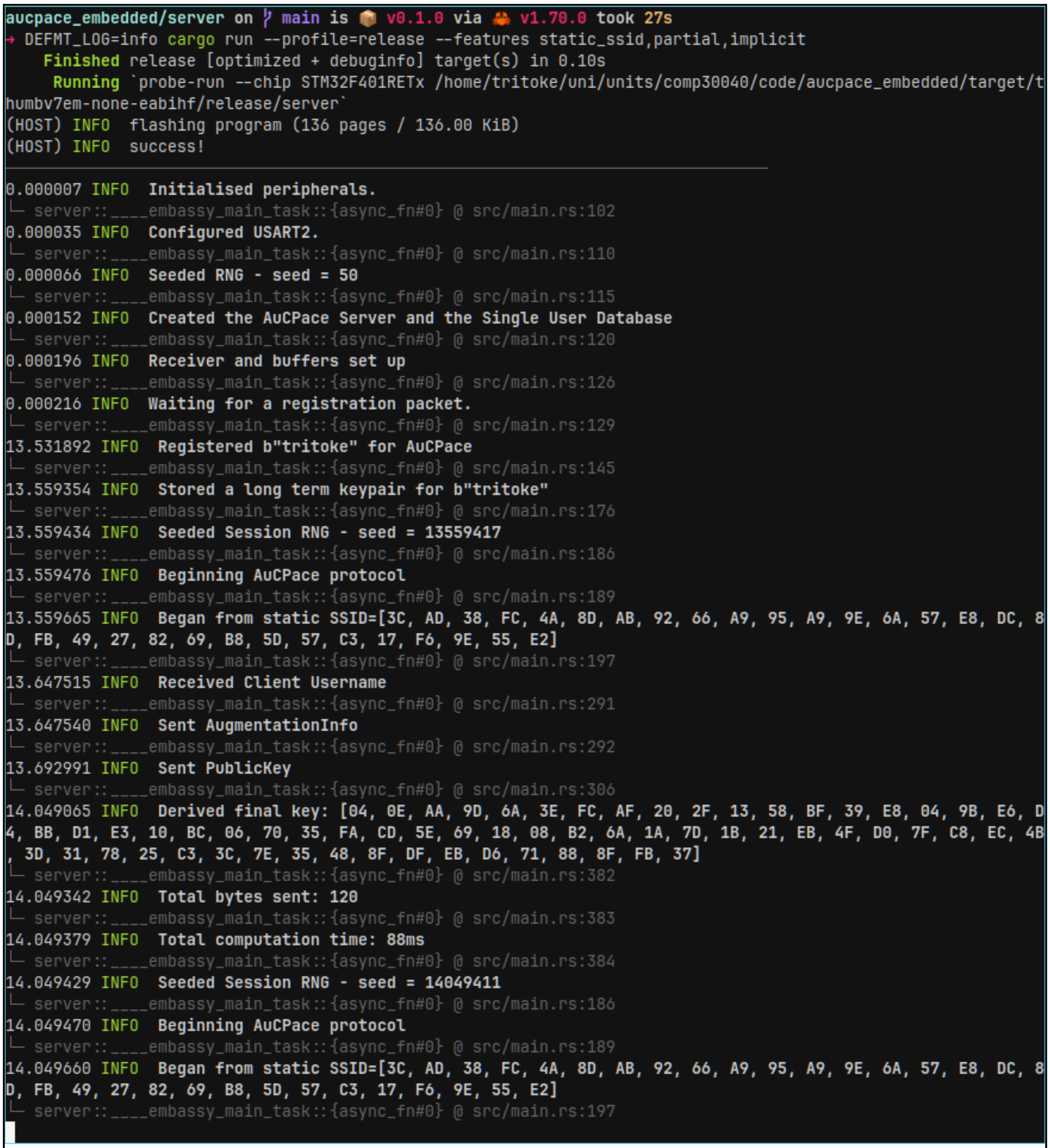
Toml Listing 1: Server Compiler Profile

```toml
[profile.release]
debug = 2

[profile.server]
inherits = "release"
opt-level = "s"
lto = true
codegen-units = 1
```



```
aucpace_embedded/client on  main is 📦 v0.1.0 via 🦀 v1.70.0
→ cargo run --release --features static_ssid,implicit -- --port /dev/ttyACM0 --log-level info -u tritoke -p hunter2
    Finished release [optimized + debuginfo] target(s) in 0.12s
     Running `/home/tritoke/uni/units/comp30040/code/aucpace_embedded/target/release/client --port /dev/ttyACM0 --log-le
vel info -u tritoke -p hunter2`
2023-04-13T17:04:27.360533Z  INFO client: Opened serial port connection.
2023-04-13T17:04:27.689468Z  INFO client: Registered as tritoke:hunter2 for AuCPace
2023-04-13T17:04:27.689481Z  INFO client: Starting AuCPace
2023-04-13T17:04:27.689486Z  INFO client: Began from static SSID=[3C, AD, 38, FC, 4A, 8D, AB, 92, 66, A9, 95, A9, 9E, 6A
, 57, E8, DC, 8D, FB, 49, 27, 82, 69, B8, 5D, 57, C3, 17, F6, 9E, 55, E2]
2023-04-13T17:04:27.689499Z  INFO client: Sending message: Username
2023-04-13T17:04:27.808673Z  INFO client: Received Augmentation info
2023-04-13T17:04:28.177737Z  INFO client: Sent PublicKey
2023-04-13T17:04:28.178119Z  INFO client: Derived final key: [6C, CC, 0A, D8, D1, E4, DA, 62, 0A, 96, 3B, 80, 07, 89, A4
, B6, 13, 31, 50, 75, 2E, 00, 0E, 75, 96, AE, 7D, C0, D3, A9, D9, F6, 72, F0, D4, 67, 73, 1A, 83, 7B, F6, 8C, 23, 32, 71
, 92, 7A, 6F, 8D, 1D, C0, C5, 8B, 5C, C3, 07, 0D, 1C, C7, 84, 92, 58, 13, D4]
2023-04-13T17:04:28.178150Z  INFO client: Total bytes sent: 126
2023-04-13T17:04:28.178161Z  INFO client: Derived final key in 488ms

aucpace_embedded/client on  main is 📦 v0.1.0 via 🦀 v1.70.0
→ |
```

Figure 5.2: AuCPace Embedded Client Output



```
aucpace_embedded/server on  main is 📦 v0.1.0 via 🦀 v1.70.0 took 27s
→ DEFMT_LOG=info cargo run --profile=release --features static_ssid,partial,implicit
    Finished release [optimized + debuginfo] target(s) in 0.10s
     Running `probe-run --chip STM32F401RETx /home/tritoke/uni/units/comp30040/code/aucpace_embedded/target/t
humbv7em-none-eabihf/release/server`
(HOST) INFO  flashing program (136 pages / 136.00 KiB)
(HOST) INFO  success!
────────────────────────────────────────────────────────────────────────────────
0.000007 INFO  Initialised peripherals.
└─ server::____embassy_main_task::{async_fn#0} @ src/main.rs:102
0.000035 INFO  Configured USART2.
└─ server::____embassy_main_task::{async_fn#0} @ src/main.rs:110
0.000066 INFO  Seeded RNG - seed = 50
└─ server::____embassy_main_task::{async_fn#0} @ src/main.rs:115
0.000152 INFO  Created the AuCPace Server and the Single User Database
└─ server::____embassy_main_task::{async_fn#0} @ src/main.rs:120
0.000196 INFO  Receiver and buffers set up
└─ server::____embassy_main_task::{async_fn#0} @ src/main.rs:126
0.000216 INFO  Waiting for a registration packet.
└─ server::____embassy_main_task::{async_fn#0} @ src/main.rs:129
13.531892 INFO  Registered b"tritoke" for AuCPace
└─ server::____embassy_main_task::{async_fn#0} @ src/main.rs:145
13.559354 INFO  Stored a long term keypair for b"tritoke"
└─ server::____embassy_main_task::{async_fn#0} @ src/main.rs:176
13.559434 INFO  Seeded Session RNG - seed = 13559417
└─ server::____embassy_main_task::{async_fn#0} @ src/main.rs:186
13.559476 INFO  Beginning AuCPace protocol
└─ server::____embassy_main_task::{async_fn#0} @ src/main.rs:189
13.559665 INFO  Began from static SSID=[3C, AD, 38, FC, 4A, 8D, AB, 92, 66, A9, 95, A9, 9E, 6A, 57, E8, DC, 8
D, FB, 49, 27, 82, 69, B8, 5D, 57, C3, 17, F6, 9E, 55, E2]
└─ server::____embassy_main_task::{async_fn#0} @ src/main.rs:197
13.647515 INFO  Received Client Username
└─ server::____embassy_main_task::{async_fn#0} @ src/main.rs:291
13.647540 INFO  Sent AugmentationInfo
└─ server::____embassy_main_task::{async_fn#0} @ src/main.rs:292
13.692991 INFO  Sent PublicKey
└─ server::____embassy_main_task::{async_fn#0} @ src/main.rs:306
14.049065 INFO  Derived final key: [04, 0E, AA, 9D, 6A, 3E, FC, AF, 20, 2F, 13, 58, BF, 39, E8, 04, 9B, E6, D
4, BB, D1, E3, 10, BC, 06, 70, 35, FA, CD, 5E, 69, 18, 08, B2, 6A, 1A, 7D, 1B, 21, EB, 4F, D0, 7F, C8, EC, 4B
, 3D, 31, 78, 25, C3, 3C, 7E, 35, 48, 8F, DF, EB, D6, 71, 88, 8F, FB, 37]
└─ server::____embassy_main_task::{async_fn#0} @ src/main.rs:382
14.049342 INFO  Total bytes sent: 120
└─ server::____embassy_main_task::{async_fn#0} @ src/main.rs:383
14.049379 INFO  Total computation time: 88ms
└─ server::____embassy_main_task::{async_fn#0} @ src/main.rs:384
14.049429 INFO  Seeded Session RNG - seed = 14049411
└─ server::____embassy_main_task::{async_fn#0} @ src/main.rs:186
14.049470 INFO  Beginning AuCPace protocol
└─ server::____embassy_main_task::{async_fn#0} @ src/main.rs:189
14.049660 INFO  Began from static SSID=[3C, AD, 38, FC, 4A, 8D, AB, 92, 66, A9, 95, A9, 9E, 6A, 57, E8, DC, 8
D, FB, 49, 27, 82, 69, B8, 5D, 57, C3, 17, F6, 9E, 55, E2]
└─ server::____embassy_main_task::{async_fn#0} @ src/main.rs:197
```

Figure 5.3: AuCPace Embedded Server Output

Table 5.3: Code size and Compute time by feature flag with `profile=server`

`strong`	`implicit`	`partial`	Code Size (Kib)	Compute Time (ms)	Data Sent (bytes)
✗	✗	✗	82	169	
✗	✗	✓	83	139	
✗	✓	✗	83	135	
✗	✓	✓	83	103	
✓	✗	✗	88	171	
✓	✗	✓	88	138	
✓	✓	✗	87	134	
✓	✓	✓	88	103	

Table 5.4: Code size and Compute time by feature flag with `profile=server` and a static SSID

`strong`	`implicit`	`partial`	Code Size (Kib)	Compute Time (ms)	Data Sent (bytes)
✗	✗	✗	81	169	
✗	✗	✓	82	139	
✗	✓	✗	82	134	
✗	✓	✓	82	103	
✓	✗	✗	86	171	
✓	✗	✓	87	139	
✓	✓	✗	87	135	
✓	✓	✓	87	102	

Table 5.5: Code size and Compute time by feature flag with `profile=release`

`strong`	`implicit`	`partial`	Code Size (Kib)	Compute Time (ms)	Data Sent (bytes)
✗	✗	✗	132	148	342
✗	✗	✓	137	119	342
✗	✓	✗	132	117	206
✗	✓	✓	138	90	206
✓	✗	✗	139	148	383
✓	✗	✓	142	120	383
✓	✓	✗	139	116	247
✓	✓	✓	142	89	247

Table 5.6: Code size and Compute time by feature flag with `profile=release` and a static SSID

`strong`	`implicit`	`partial`	Code Size (Kib)	Compute Time (ms)	Data Sent (bytes)
✗	✗	✗	131	145	302
✗	✗	✓	136	120	302
✗	✓	✗	132	115	166
✗	✓	✓	137	88	166
✓	✗	✗	138	148	343
✓	✗	✓	141	120	343
✓	✓	✗	138	116	207
✓	✓	✓	142	89	207

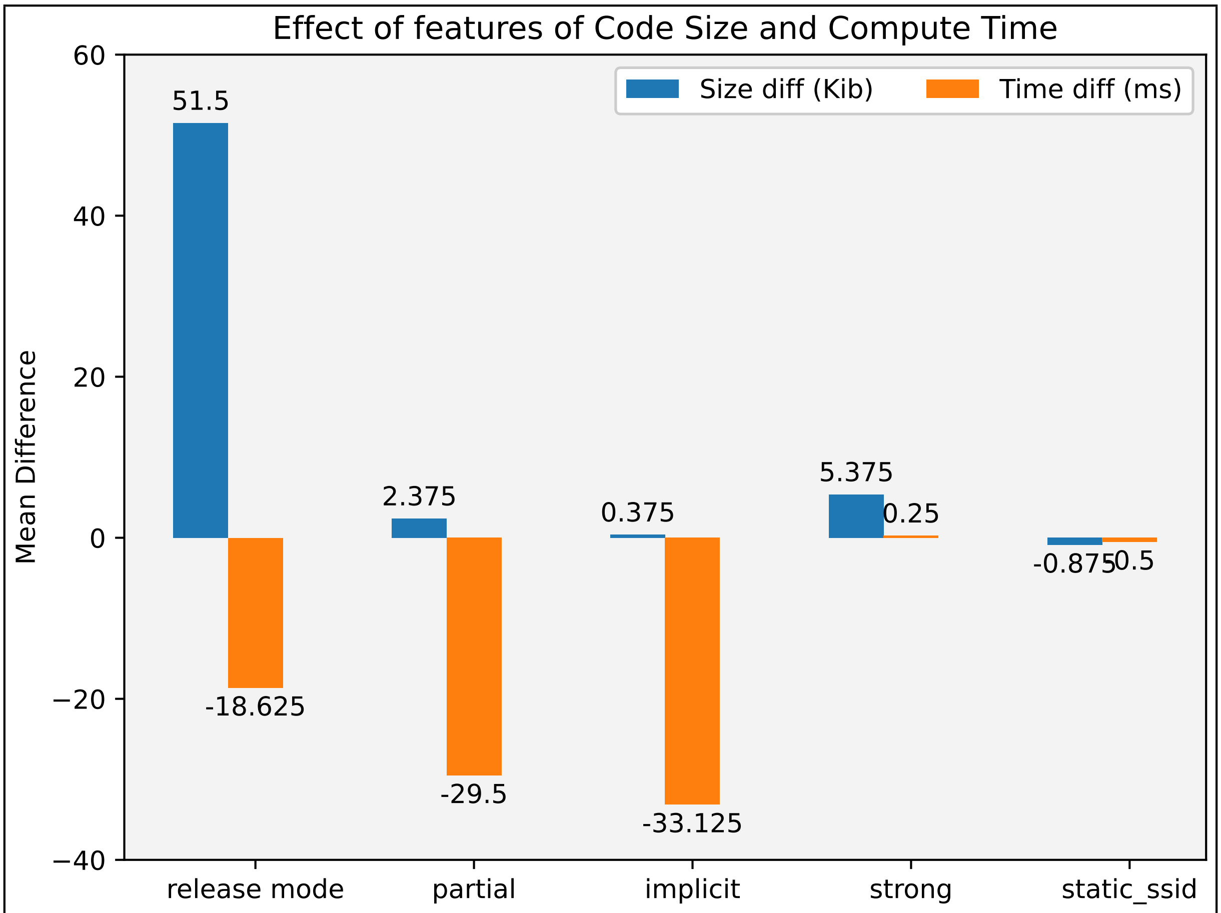
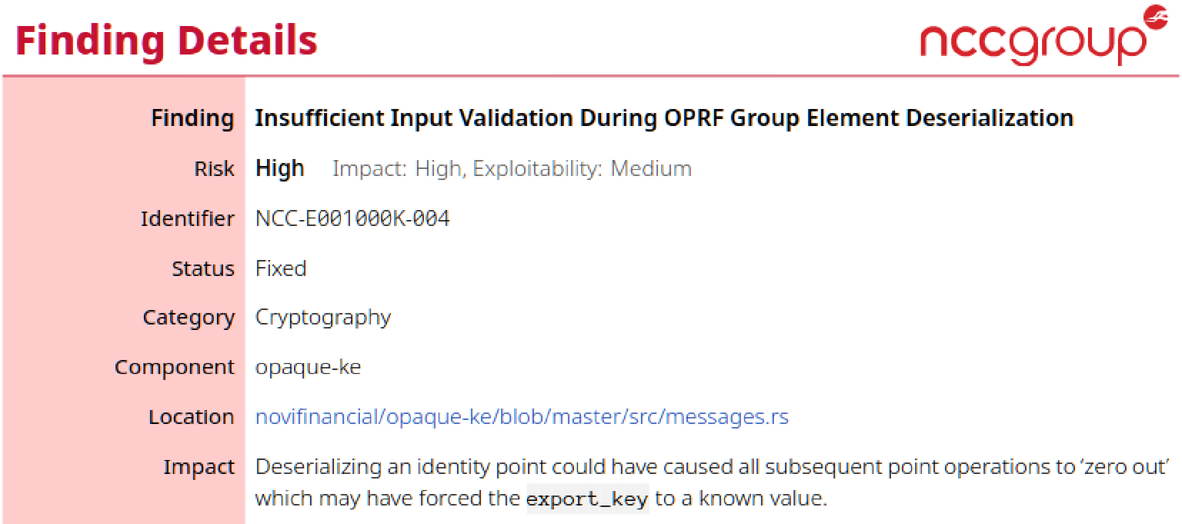


Figure 5.4: Plot of the effect of different feature flags and compiler modes

5.3 Breaking Everything

By happenstance I was reading NCC Group's recent review of Whatsapp's `opaque-ke` crate [Fer21; HH21]. While reading the report the following finding caught my eye:



Finding Details nccgroup

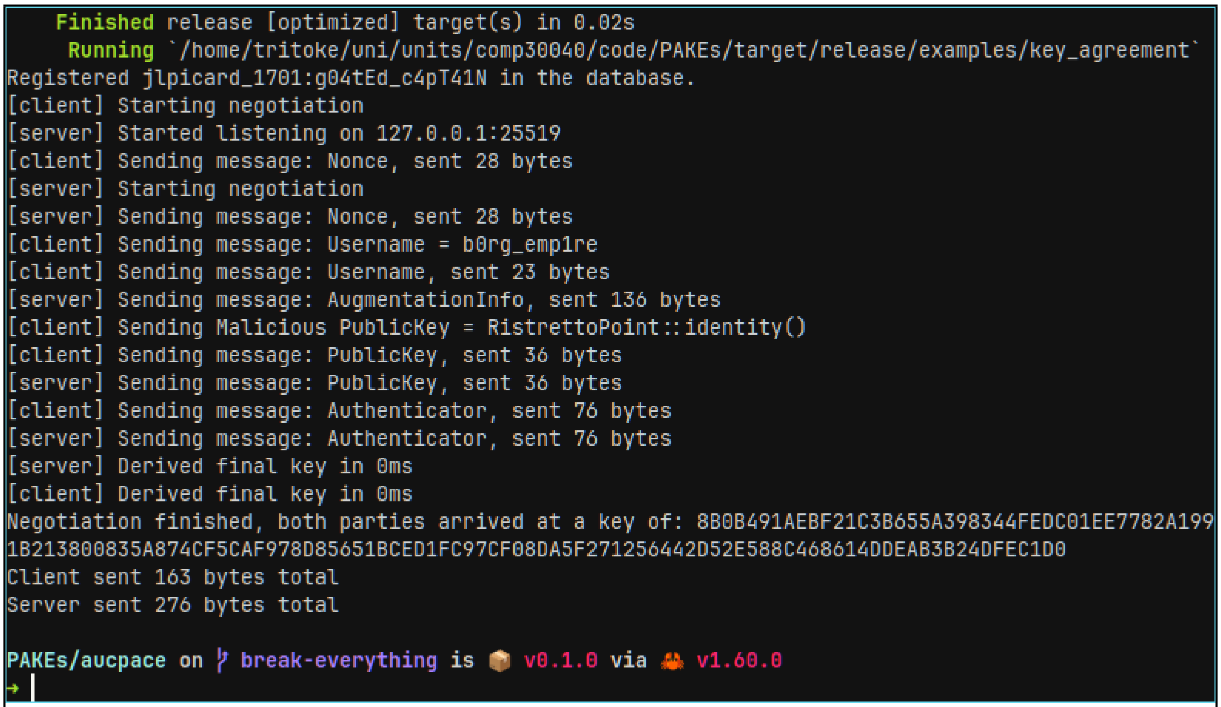
Finding	**Insufficient Input Validation During OPRF Group Element Deserialization**
Risk	**High** Impact: High, Exploitability: Medium
Identifier	NCC-E001000K-004
Status	Fixed
Category	Cryptography
Component	opaque-ke
Location	novifinancial/opaque-ke/blob/master/src/messages.rs
Impact	Deserializing an identity point could have caused all subsequent point operations to 'zero out' which may have forced the `export_key` to a known value.

Figure 5.5: High severity finding from [HH21].

In Whatsapp's implementation of OPAQUE [JKX18], they used the `RistrettoPoint` type from `curve25519-dalek`, and while deserialising this type they didn't have any checks to see if this point was the identity point. Howell and Henry point out that this leads all subsequent point operations to "zero out" and thus cause the shared key to have a known value. Unsure of whether this would also break my AuCPace implementation I modified `key_agreement.rs` to have the client act as a malicious adversary and to send this identity point.

Rust Listing 9: Malicious AuCPace Client

```
let neutral_element = RistrettoPoint::identity();
let message: ClientMessage<'_, K1> =
 ↪ ClientMessage::PublicKey(neutral_element.clone());
let bytes_sent = send!(stream, message);
CLIENT_BYTES_SENT.fetch_add(bytes_sent, Ordering::SeqCst);
println!(⌋
 ↪ "[client] Sending Malicious PublicKey = RistrettoPoint::identity()");
println!(
    "[client] Sending message: PublicKey, sent {} bytes",
    bytes_sent
);
```



```
    Finished release [optimized] target(s) in 0.02s
     Running `/home/tritoke/uni/units/comp30040/code/PAKEs/target/release/examples/key_agreement`
Registered jlpicard_1701:g04tEd_c4pT41N in the database.
[client] Starting negotiation
[server] Started listening on 127.0.0.1:25519
[client] Sending message: Nonce, sent 28 bytes
[server] Starting negotiation
[server] Sending message: Nonce, sent 28 bytes
[client] Sending message: Username = b0rg_emp1re
[client] Sending message: Username, sent 23 bytes
[server] Sending message: AugmentationInfo, sent 136 bytes
[client] Sending Malicious PublicKey = RistrettoPoint::identity()
[client] Sending message: PublicKey, sent 36 bytes
[server] Sending message: PublicKey, sent 36 bytes
[client] Sending message: Authenticator, sent 76 bytes
[server] Sending message: Authenticator, sent 76 bytes
[server] Derived final key in 0ms
[client] Derived final key in 0ms
Negotiation finished, both parties arrived at a key of: 8B0B491AEBF21C3B655A398344FEDC01EE7782A199
1B213800835A874CF5CAF978D85651BCED1FC97CF08DA5F271256442D52E588C468614DDEAB3B24DFEC1D0
Client sent 163 bytes total
Server sent 276 bytes total

PAKEs/aucpace on  break-everything is 📦 v0.1.0 via 🦀 v1.60.0
→ 
```

Figure 5.6: Malicious AuCPace Client

To my dismay it worked. I immediately imformed RustCrypto of the problem and started working on a fix.

5.3.1 Identifying The Extent Of The Damage

So what could a malicious attacker do with this bug?

- Impersonate any user, even a user who has never registered, even when no user has ever registered.
- Impersonate any server, to any user, regardless of whether they've registered with the server.

Safe to say this is about as bad as it gets.

5.3.2 Fixing The Problem

Thankfully the fix for this issue is incredibly simple. The AuCPace protocol specifies points at which to abort the protocol should an invalid point is encountered. Thus everywhere AuCPace says to abort if the point is invalid, we put in a check for the identity point.

Diff 1: Patch for checking the identity element.

```
@@ -589,9 +598,14 @@ where
     pub fn receive_client_pubkey(
         self,
         client_pubkey: RistrettoPoint,
-    ) -> AuCPaceServerExpMutAuth<D, K1> {
```

```
+     ) -> Result<AuCPaceServerExpMutAuth<D, K1>> {
+         // check for the neutral point
+         if client_pubkey.is_identity() {
+             return Err(Error::IllegalPointError);
+         }
+
          let sk1 = compute_first_session_key::<D>(self.ssid,
 ↪   self.priv_key, client_pubkey);
-         AuCPaceServerExpMutAuth::new(self.ssid, sk1)
+         Ok(AuCPaceServerExpMutAuth::new(self.ssid, sk1))
      }
```

It is clear to see that this is a very simple patch the main issue is ensuring it is caught everywhere.

5.3.3 Why Was This Not Caught Earlier?

There are several reasons why this wasn't caught earlier:

1. My lack of familiarity with ECC – this was my first time ever using ECC and it was simply not something I knew to look out for.

2. My decisiion to implement based on the paper not the IETF document – the paper mentions only to abort if a point is invalid. However there are two ways a point can be invalid, it can be off the curve, and it can be the identity point. `curve25519-dalek` makes the former unrepresentable using Rust's type system, hence my belief that this check was unnecessary. However the IETF draft of AuCPace mentions explicitly to check for the identity element [Haa23].

3. This is quite a subtle bug – it is hard to spot when you are unfamiliar with ECC, case and point Whatsapp made this mistake as well, and so did the the core developers of Java. Java's CVE-2022-21449 ”Psychic Signatures” [MIT22], had the same bug in their implementation of Elliptic-curve Digital Signature Algorithm (ECDSA). Introduced in commit `3c12c4b0f35` Dec 2018 fixed in `e2f8ce9c3ff` Jan 2022, all in it took 3 years for this same bug to get found and patched in Java.

5.3.4 How To Prevent This Bug From Ever Happening Again

Two changes have been implemented to prevent this bug from ever happening again:

1. Every method that handles a RistrettoPoint from the network checks it to make sure it is not the identity point.

2. There are now tests for every method of both the Client and Server to ensure that providing an invalid point returns an `IllegalPointError`.

There is a better way to fix this however, RustCrypto's `elliptic-curve` module solves this problem using the power of Rust's type system. They have a `NonIdentity` type which is guaranteed to never be the identity element. When Curve25519 is introduced to `elliptic-curve` the library will be refactored to move over to this type.

CHAPTER 6

Summary and Conclusion

Through the course of this project a fully-featured and developer focused, implementation of AuCPace was created in Rust. It was then open sourced through the RustCrypto organisation. A severe bug in the code was remediated effictively and swiftly. Contributing to the IIOT security ecosystem in a practical and direct way.

6.1 Achievements

Beginning this project my understanding of ECC was effectively zero, I had heard of it before, and knew that it was somehow better than traditional primitives, but actually getting to understand and effectively leverage ECC took many hours of research and was genuinely something I struggled with.
The AuCPace I have created is both fast and has a tiny code footprint, this was no small engineering feat and took great care over the entire length of the project to ensure that the implementation was truly viable for real world use.
Open sourcing the implementation through RustCrypto will allow for the greatest number of people to make use of the library, and proves that the implementation is of high enough quality to be suitable for real world applications.
Realising that several months of your work are completely and fundamentally broken is completely crushing. Working through and understanding the vulnerability I had unknowingly let into my code proved to be a very valuable experience. Understanding how to responsibly handle disclosing a vulnerability and rapidly rolling out a fix taught me a lot about the pressures of working in cybersecurity and how no matter how careful you are, there is always something lurking that you hadn't considered.

6.2 Reflection

I have two main regrets from this project as a whole. Firstly I wish I had implemented from the IETF draft of the AuCPace protocol instead of from the paper itself. IETF drafts and standards are intended to be used as reference for those implementing the protocol. If I had followed their draft standard I would not have had anything like the trouble I did implementing Strong AuCPace and I wouldn't have had my catastrophic ”neutral point” bug. I consider that bug to be a direct consequence of my decision to follow the paper instead of the draft.

Secondly I wish I had contacted Björn Haase earlier on, when I did contact him late in the project there was limited scope left for collaboration simply due to the lack of remaining time. I believe his input especially around test vector generation could have been very helpful.

That said, I am immensely proud of what I have achieved over the course of this project. I have greatly enjoyed researching PAKEs and working in Rust.

This project has undeniably changed the course of my career, at the start of the year I had planned to continue with my bursary sponsor on to a job in pentesting, however I now intended to pursue a PhD in cryptography.

6.3 Future Work

There are so many directions future work in this area could go. PAKEs are a fascinating topic with a bright future in many areas of cybersecurity.

6.3.1 CHIP+CRISP

If I had more time I would have liked to have also implemented the CHIP+CRISP iPAKE compilers from Cremers et al., I believe these two protocol compilers are fascinating pieces of technology that ara truly innovative.

6.3.2 CPace

On a more practical level, AuCPace is an Augmented PAKE built on top of the CPace Balanced PAKE, future work could include breaking this underlying implementation out into it's own crate.

6.3.3 Post Quantum PAKE

With the rapid emergence of quantum technologies, stopgap solutions such as the ”quantum annoying” property of AuCPace and similar PAKEs simply won't be enough, an implementation of a truly ”post quantum” PAKE would make for an excellent project. An interesting candidate for this would be LATKE [Ros23], an iPAKE instantiated from lattice assumptions.

6.4 Conclusion

PAKE algorithms can be used to improve the security of a wide variety of domains. The IIOT stands to gain particular benefit from improving PAKE algorithms and support, especially as lower power algorithms and those directly targetting embedded devices emerge. The Rust implementation of AuCPace directly contributes to the ever growing IIOT security landscape, by contributing a PAKE with a low code-footprint, efficient computation and no patent barriers.

Glossary

Abelian Group A group whose operator is also commutative. e.g. Addition over $\mathbb{Z}$. . 15, 16

AES Advanced Encryption Scheme. 55

AKE Authenticated Key-Exchange. 18

API Application Programming Interface. 31, 32, 34, 38, 39

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. 12, 13

AuCPace Augmented Composable Password Authenticated Connection Establishment. 4, 8, 9, 18, 19, 20, 21, 24, 27, 28, 29, 31, 33, 35, 37, 38, 41, 42, 43, 48, 49, 50, 51, 52, 68

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. . 11, 14, 17, 18, 33, 52, 54

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. . 11, 13, 17, 18, 52

CFRG Crypto Forum Research Group. 11, 17, 54

CPace Composable Password Authenticated Connection Establishment. 17, 20, 52

CSPRNG Cryptographically Secure Pseudo Random Number Generator. 32

DH Diffie-Hellman. 11, 13, 16, 18, 20, 29, 63

DWARF The debug information format for Executable and Linkable Format (ELF) files. . 44

EAP Extensible Authentication Protocol. 12

ECC Elliptic Curve Cryptography. 4, 17, 50, 51

ECDH Elliptic-curve Diffie-Hellman. 11

ECDLP Elliptic Curve Discrete Logarithm Problem. 17

ECDSA Elliptic-curve Digital Signature Algorithm. 50

EKE Encrypted Key Exchange. 12, 13, 14

ELF Executable and Linkable Format. 53

FFI Foreign Function Interface. 25

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. 16

FIPS Federal Information Processing Standards. 28

HMI human machine interface. 18

IETF Internet Engineering Task Force. 17, 18, 50, 51

IIOT Industrial Internet of Things. 4, 18, 19, 27, 42, 51, 52

IOT Internet of Things. 4, 27, 42

iPAKE identity-binding PAKE. 17, 52

IRTF Internet Research Task Force. 17

KHAPE Key-Hiding Asymmetric PAKE. 18, 19

MCU Microcontroller. 38, 42, 68

NIST National Institute of Standards and Technology. 15, 28, 31

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. . 19

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. 11, 13

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. 17, 18, 19

OPRF Oblivious Pseudo Random Function. 18, 54

OS Operating System. 38

OSSWU Optimized Simplified Shallue-van de Woestijne-Ulas. 31

PAKE Password-Authenticated Key-Exchange. 4, 8, 9, 10, 11, 12, 14, 17, 18, 19, 25, 32, 38, 52

PBKDF Password-Based Key Derivation Function. 29, 30

PKI Public-Key-Infrastructure. 18, 19

PRS Password Related String. 20

PSK Pre-Shared Key. 19

RCIG Rust Cryptography Interest Group. 29

RSA Rivest-Shamir-Adleman. 11, 12, 15

RTIC Real-Time Interrupt-drived Concurrency. 43

RTT Real-Time-Transfer. 27

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. . 13, 14

SHA Secure Hash Algorithm. 29

SIDH Supersingular isogeny Diffie-Hellman. 11

SPAKE Simple PAKE. 13, 14, 32, 33

SRP Secure Remote Password. 14, 32, 33, 34, 63

SSID Sub-Session ID. 19, 20

STM ST-Microelectronics. 42

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). 12

TLS Transport Layer Security. 10, 14

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 14. 11, 14, 20, 36, 37

V-PAKE Verifier based PAKE. 4

Bibliography

[Alw+17] Joël Alwen et al. “Scrypt is maximally memory-hard”. In: *Advances in Cryptology–EUROCRYPT 2017: 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30–May 4, 2017, Proceedings, Part III*. Springer. 2017, pp. 33–62.

[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.

[Ber06] Daniel J Bernstein. “Curve25519: new Diffie-Hellman speed records”. In: *Public Key Cryptography-PKC 2006: 9th International Conference on Theory and Practice in Public-Key Cryptography, New York, NY, USA, April 24-26, 2006. Proceedings 9*. Springer. 2006, pp. 207–228.

[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.

[CD22] Wouter Castryck and Thomas Decru. *An efficient key recovery attack on SIDH*. Cryptology ePrint Archive, Paper 2022/975. 2022. URL: https://eprint.iacr.org/2022/975.

[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.

[ES21] Edward Eaton and Douglas Stebila. *The "quantum annoying" property of password-authenticated key exchange protocols.* Cryptology ePrint Archive, Paper 2021/696. 2021. URL: https://eprint.iacr.org/2021/696.

[Fer21] Jennifer Fernick. *Public Report – WhatsApp opaque-ke Cryptographic Implementation Review.* Accessed 11th April 2023. Dec. 2021. URL: https://research.nccgroup.com/2021/12/13/public-report-whatsapp-opaque-ke-cryptographic-implementation-review/.

[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/.

[Haa23] Björn Haase. *(strong) AuCPace, an augmented PAKE.* Jan. 2023. URL: https://datatracker.ietf.org/doc/draft-haase-aucpace/07/.

[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.

[HH21] Ava Howell and Kevin Henry. *Security Assessment - opaque-ke.* Accessed 11th April 2023. Dec. 2021. URL: https://research.nccgroup.com/wp-content/uploads/2021/12/NCC%5C_Group%5C_WhatsAppLLC%5C_OPAQUE%5C_Report%5C_2021-12-10%5C_v1.3.pdf.

[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.

[JKX18] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. *OPAQUE: An Asymmetric PAKE Protocol Secure Against Pre-Computation Attacks.* Cryptology ePrint Archive, Paper 2018/163. 2018. URL: https://eprint.iacr.org/2018/163.

[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.

[MIT22] MITRE. *CVE-2022-21449*. May 2022. URL: https://nvd.nist.gov/vuln/detail/CVE-2022-21449.

[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.

[Per16] Colin Percival. *The scrypt Password-Based Key Derivation Function*. RFC 7914. RFC Editor, Aug. 2016. URL: https://www.rfc-editor.org/rfc/rfc7914.txt.

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

[Ros23] Michael Rosenberg. *LATKE: An identity-binding PAKE from lattice assumptions*. Cryptology ePrint Archive, Paper 2023/324. 2023. URL: https://eprint.iacr.org/2023/324.

[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.

[ST20a] National Institute of Standards and Technology. *Recommendation for Key Management: Part 1 – General*. Tech. rep. NIST Special Publication (SP) 800-57, Part 1, Rev. 5. Washington, D.C.: U.S. Department of Commerce, 2020. DOI: 10.6028/NIST.SP.800-57pt1r5.

[ST20b] National Institute of Standards and Technology. *Security Requirements for Cryptographic Modules*. 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/NIST.FIPS.140-2.

[ST23] National Institute of Standards and Technology. *Safer Languages*. Tech. rep. (TR) 24772 Guidance to avoiding vulnerabilities in programming languages. Washington, D.C.: U.S. Department of Commerce, 2023. URL: https://www.nist.gov/itl/ssd/software-quality-group/safer-languages.

[Val+19] Henry de Valence et al. *The ristretto255 group*. Tech. rep. IETF CFRG Internet Draft, 2019.

[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.

[WB19] Riad S. Wahby and Dan Boneh. *Fast and simple constant-time hashing to the BLS12-381 elliptic curve*. Cryptology ePrint Archive, Paper 2019/403. 2019. DOI: 10.13154/tches.v2019.i4.154-179. URL: https://eprint.iacr.org/2019/403.

[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:

Python 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 =
↪   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
```

Python 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:

Python 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
```

Python 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))
```

APPENDIX C

Example implementation of the Database trait

Rust Listing 10: Example Database Implementation

```rust
/// Password Verifier database which can store the info for one user
#[derive(Debug, Default)]
struct SingleUserDatabase {
    user: Option<Vec<u8>>,
    data: Option<(RistrettoPoint, SaltString, ParamsString)>,
}

impl Database for SingleUserDatabase {
    type PasswordVerifier = RistrettoPoint;

    fn lookup_verifier(
        &self,
        username: &[u8],
    ) -> Option<(Self::PasswordVerifier, SaltString, ParamsString)> {
        match &self.user {
            Some(stored_username) if stored_username == username =>
↪   self.data.clone(),
            _ => None,
        }
    }

    fn store_verifier(
        &mut self,
        username: &[u8],
        salt: SaltString,
        _uad: Option<&[u8]>,
        verifier: Self::PasswordVerifier,
        params: ParamsString,
    ) {
        self.user = Some(username.to_vec());
        self.data = Some((verifier, salt, params));
    }
}
```

```
}
```

Appendix D

Embedded Rust application implementing AuCPace

The client code uses my Rust library to implement the client side of AuCPace. The server code uses my Rust library to implement the server side of AuCPace for an Nucleo-64 (STM32F401RE) MCU.

Rust Listing 11: Client Negotiate

```
fn main() -> Result<()> {
    let args = Args::try_parse()?;

    // setup the logger
    tracing_subscriber::fmt()
        .with_ansi(true)
        .with_max_level(args.log_level)
        .with_writer(io::stderr)
        .init();

    debug!("args={args:?}");

    // list the ports if the user asks for it
    if args.list_ports {
        let mut ports = serialport::available_ports()?;
        ports.retain(|port| matches!(port.port_type,
 ↪  SerialPortType::UsbPort(_)));
        println!("Found the following USB ports:");
        for port in ports {
            println!("{}", port.port_name);
        }

        return Ok(());
    }

    // open the serial port connection
    let port_name = args
```

```
        .port
        .ok_or_else(|| anyhow!("Must supply a USB port."))?;
    let serial = Mutex::new({
        serialport::new(port_name, USART_BAUD)
            .timeout(Duration::from_millis(500))
            .open()?
    });
    let mut receiver = MsgReceiver::new(&serial);
    info!("Opened serial port connection.");

    // start the client
    let mut base_client = Client::new(rand_core::OsRng);
    let mut bytes_sent = 0;

    let user = args.username.as_str();
    let pass = args.password.as_str();
    if !args.skip_register {
        #[cfg(not(feature = "strong"))]
        let message = base_client
            .register_alloc(user.as_bytes(), pass,
↪   Params::recommended(), Scrypt)
            .map_err(|e| anyhow!(e))?;

        #[cfg(feature = "strong")]
        let message = base_client
            .register_alloc_strong(user.as_bytes(), pass,
↪   Params::recommended(), Scrypt)
            .map_err(|e| anyhow!(e))?;

        bytes_sent += send!(serial, message);
        info!(
            "Registered as {user}:{pass} for {}",
            if cfg!(feature = "strong") {
                "Strong AuCPace"
            } else {
                "AuCPace"
            }
        );
    }

    info!("Starting AuCPace");
    let start = Instant::now();
    // ===== SSID Establishment =====
    #[cfg(feature = "static_ssid")]
    let client = {
        let client =
↪   base_client.begin_prestablished_ssid(SSID).unwrap();
```

```
        info!("Began from static SSID={:02X?}", SSID);
        client
    };

    #[cfg(not(feature = "static_ssid"))]
    let client = {
        let (client, message) = base_client.begin();
        bytes_sent += send!(serial, message);

        let server_message = recv!(receiver);
        let client = if let ServerMessage::Nonce(server_nonce) =
 ↪  server_message {
            client.agree_ssid(server_nonce)
        } else {
            panic!("Received invalid server message {:?}",
 ↪  server_message);
        };
        info!("Agreed on SSID");
        client
    };

    // ===== Augmentation Layer =====
    #[cfg(not(feature = "strong"))]
    let (client, message) = {
        info!("Sending message: Username");
        client.start_augmentation(user.as_bytes(), pass.as_bytes())
    };
    #[cfg(feature = "strong")]
    let (client, message) = {
        info!("Sending message: Strong Username");
        client.start_augmentation_strong(user.as_bytes(),
 ↪  pass.as_bytes(), &mut rand_core::OsRng)
    };
    bytes_sent += send!(serial, message);

    let mut server_message = recv!(receiver);
    #[cfg(not(feature = "strong"))]
    let client = if let ServerMessage::AugmentationInfo {
        x_pub,
        salt,
        pbkdf_params,
        ..
    } = server_message
    {
        info!("Received Augmentation info");
        let params = parse_params(pbkdf_params)?;
        client
```

```
            .generate_cpace_alloc(x_pub, &salt, params, Scrypt)
            .expect("Failed to generate CPace step data")
    } else {
        panic!("Received invalid server message {:?}", server_message);
    };

    #[cfg(feature = "strong")]
    let client = if let ServerMessage::StrongAugmentationInfo {
        x_pub,
        blinded_salt,
        pbkdf_params,
        ..
    } = server_message
    {
        info!("Received Strong Augmentation info");
        let params = parse_params(pbkdf_params)?;
        client
            .generate_cpace_alloc(x_pub, blinded_salt, params, Scrypt)
            .expect("Failed to generate CPace step data")
    } else {
        panic!("Received invalid server message {:?}", server_message);
    };

    // ===== CPace substep =====
    let ci = "Server-USART2-Client-SerialPort";
    let (client, message) = client.generate_public_key(ci, &mut
↪   rand_core::OsRng);
    bytes_sent += send!(serial, message);
    info!("Sent PublicKey");

    server_message = recv!(receiver);
    let ServerMessage::PublicKey(server_pubkey) = server_message else {
        panic!("Received invalid server message {:?}", server_message);
    };
    let key = if cfg!(feature = "implicit") {
        client.implicit_auth(server_pubkey)
    } else {
        let (client, message) =
↪   client.receive_server_pubkey(server_pubkey);

        // ===== Explicit Mutual Auth =====
        bytes_sent += send!(serial, message);
        info!("Sent Authenticator");

        server_message = recv!(receiver);
        if let ServerMessage::Authenticator(server_authenticator) =
↪   server_message {
```

```
            client
                .receive_server_authenticator(server_authenticator)
                .expect("Failed Explicit Mutual Authentication")
        } else {
            panic!("Received invalid server message {:?}",
↪   server_message);
        }
    };

    info!("Derived final key: {:02X?}", key.as_slice());
    info!("Total bytes sent: {}", bytes_sent);
    info!(
        "Derived final key in {}ms",
        Instant::now().duration_since(start).as_millis()
    );

    Ok(())
}
```

Rust Listing 12: Server Negotiate

```
#[embassy_executor::main]
async fn main(_spawner: Spawner) -> ! {
    let mut rcc_config: embassy_stm32::rcc::Config =
↪   Default::default();
    rcc_config.sys_ck = Some(Hertz::mhz(32));
    let mut board_config: embassy_stm32::Config = Default::default();
    board_config.rcc = rcc_config;
    let p = embassy_stm32::init(board_config);
    info!("Initialised peripherals.");

    // configure USART2 which goes over the USB port on this board
    let config = Config::default();
    let irq = interrupt::take!(USART2);
    let (mut tx, rx) =
        Uart::new(p.USART2, p.PA3, p.PA2, irq, p.DMA1_CH6, p.DMA1_CH5,
↪   config).split();
    info!("Configured USART2.");

    // configure the RNG, kind of insecure but this is just a demo and
↪   I don't have real entropy
    let now = Instant::now().as_micros();
    let server_rng = ChaCha8Rng::seed_from_u64(now);
    info!("Seeded RNG - seed = {}", now);

    // create our AuCPace server
    let mut base_server: AuCPaceServer<sha2::Sha512, _, K1> =
↪   AuCPaceServer::new(server_rng);
```

```
    let mut database: SingleUserDatabase<100> =
↪   SingleUserDatabase::default();
    info!("Created the AuCPace Server and the Single User Database");

    // create something to receive messages
    let mut buf = [0u8; 1024];
    let mut receiver = MsgReceiver::new(rx);
    let mut s: String<1024> = String::new();
    info!("Receiver and buffers set up");

    // wait for a user to register themselves
    info!("Waiting for a registration packet.");
    #[cfg_attr(not(feature = "partial"), allow(unused))]
    let user = loop {
        let msg = recv!(receiver, s);
        #[cfg(not(feature = "strong"))]
        if let ClientMessage::Registration {
            username,
            salt,
            params,
            verifier,
        } = msg
        {
            if username.len() > 100 {

↪   error!("Attempted to register with a username thats too long.");
            } else {
                database.store_verifier(username, salt, None,
↪   verifier, params);
                info!("Registered {:a} for AuCPace", username);
                break username;
            }
        }

        #[cfg(feature = "strong")]
        if let ClientMessage::StrongRegistration {
            username,
            secret_exponent,
            params,
            verifier,
        } = msg
        {
            if username.len() > 100 {

↪   error!("Attempted to register with a username thats too long.");
            } else {
                database.store_verifier_strong(username, None,
↪   verifier, secret_exponent, params);
                info!("Registered {:a} for Strong AuCPace", username);
                break username;
            }
        }
    };

    #[cfg(feature = "partial")]
    {
        let (priv_key, pub_key) =
↪   base_server.generate_long_term_keypair();
        // it is fine to unwrap here because we have already registered
        // a verifier for the user with store_verifier
        database
            .store_long_term_keypair(user, priv_key, pub_key)
            .unwrap();
        info!("Stored a long term keypair for {:a}", user);
    }

    loop {
        let mut time_taken = Duration::default();

        let start = Instant::now();
        let mut session_rng =
↪   ChaCha8Rng::seed_from_u64(start.as_micros());
        let mut bytes_sent = 0;
        time_taken += Instant::now().duration_since(start);
        info!("Seeded Session RNG - seed = {}", start.as_micros());

        // now do a key-exchange
        info!("Beginning AuCPace protocol");

        // ===== SSID Establishment =====
        #[cfg(feature = "static_ssid")]
        let server = {
            let t0 = Instant::now();
            let server =
↪   base_server.begin_prestablished_ssid(SSID).unwrap();
            time_taken += Instant::now().duration_since(t0);
            info!("Began from static SSID={:02X}", SSID);
            server
        };

        #[cfg(not(feature = "static_ssid"))]
        let server = {
            let t0 = Instant::now();
            let (server, message) = base_server.begin();
            time_taken += Instant::now().duration_since(t0);

            let client_message: ClientMessage<K1> = recv!(receiver, s);
            let t0 = Instant::now();
            let server = if let ClientMessage::Nonce(client_nonce) =
↪   client_message {
                server.agree_ssid(client_nonce)
            } else {
                fmt_log!(
                    ERROR,
                    s,

↪   "Received invalid client message {:?} - restarting negotiation",
                    client_message
                );
                continue;
            };
            time_taken += Instant::now().duration_since(t0);
            info!("Received Client Nonce");

            // now that we have received the client nonce, send our
↪   nonce back
            bytes_sent = send!(tx, buf, message);
            info!("Sent Nonce");

            server
        };

        // ===== Augmentation Layer =====
        let mut client_message = recv!(receiver, s);
        let t0 = Instant::now();
        #[cfg(not(feature = "strong"))]
        let (server, message) = if let
↪   ClientMessage::Username(username) = client_message {
            #[cfg(not(feature = "partial"))]
            let ret = server.generate_client_info(username, &database,
↪   &mut session_rng);
            #[cfg(feature = "partial")]
            let ret =
                server.generate_client_info_partial_aug(username,
↪   &database, &mut session_rng);
            ret
        } else {
            fmt_log!(
                ERROR,
                s,
↪ "Received invalid client message {:?} - restarting negotiation",
                client_message
            );
            continue;
        };

        #[cfg(feature = "strong")]
        let (server, message) = if let ClientMessage::StrongUsername {
↪ username, blinded } =
            client_message
        {
            #[cfg(not(feature = "partial"))]
            let ret =
                server.generate_client_info_strong(username, blinded,
↪ &database, &mut session_rng);
            #[cfg(feature = "partial")]
            let ret = server.generate_client_info_partial_strong(
                username,
                blinded,
                &database,
                &mut session_rng,
            );
            ret
        } else {
            fmt_log!(
                ERROR,
                s,

↪ "Received invalid client message {:?} - restarting negotiation",
                client_message
            );
            continue;
        };
        time_taken += Instant::now().duration_since(t0);

        bytes_sent += send!(tx, buf, message);
        #[cfg(not(feature = "strong"))]
        {
            info!("Received Client Username");
            info!("Sent AugmentationInfo");
        }
        #[cfg(feature = "strong")]
        {
            info!("Received Client Username and Blinded Point");
            info!("Sent Strong Augmentation Info");
        }
```

```
        // ===== CPace substep =====
        let t0 = Instant::now();
        let ci = "Server-USART2-Client-SerialPort";
        let (server, message) = server.generate_public_key(ci);
        time_taken += Instant::now().duration_since(t0);
        bytes_sent += send!(tx, buf, message);
        info!("Sent PublicKey");

        client_message = recv!(receiver, s);
        let ClientMessage::PublicKey(client_pubkey) = client_message
↪   else {
            fmt_log!(
                    ERROR,
                    s,
                    "Received invalid client message {:?}",
                    client_message
                );
            continue;
        };

        let key = if cfg!(feature = "implicit") {
            server.implicit_auth(client_pubkey)
        } else {
            let t0 = Instant::now();
            let server = server.receive_client_pubkey(client_pubkey);
            time_taken += Instant::now().duration_since(t0);
            info!("Received Client PublicKey");

            // ===== Explicit Mutual Authentication =====
            client_message = recv!(receiver, s);
            let t0 = Instant::now();
            let (key, message) = if let
↪   ClientMessage::Authenticator(ca) = client_message {
                match server.receive_client_authenticator(ca) {
                    Ok(inner) => inner,
                    Err(e) => {
                        fmt_log!(
                            ERROR,
                            s,

↪   "Client failed the Explicit Mutual Authentication check - {e:?}"
                        );
                        continue;
                    }
                }
            } else {
                fmt_log!(
                    ERROR,
                    s,
                    "Received invalid client message {:?}",
                    client_message
                );
                continue;
            };

            time_taken += Instant::now().duration_since(t0);
            bytes_sent += send!(tx, buf, message);

            info!("Sent Authenticator");

            key
        };

        info!("Derived final key: {:02X}", key.as_slice());
        info!("Total bytes sent: {}", bytes_sent);
        info!("Total computation time: {}ms", time_taken.as_millis());
    }
}
```