EXPLORING PASSWORD-AUTHENTICATED KEY-EXCHANGE ALGORITHMS

A PROJECT REPORT SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF BACHELORS OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2023

Sam Leonard
f41751sl
Supervisor: Professor Bernardo Magri

Department of Computer Science

Declaration

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

Copyright

Abstract

Exploring Password-Authenticated Key-Exchange Algorithms

Sam Leonard, Supervisor: Professor Bernardo Magri

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

Acknowledgements

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

Contents

CHAPTER 2

Design

2.1 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 efficient binaries. Embassy is especially impressive as they have implemented a 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.

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 National Institute of Standards and Technology (NIST)'s list of "Safer Languages" which it recommends for writing safety focussed 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 [ST20].

2.2 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).

2.2.1 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 2.1 and 2.2 are partially reproduced from [HL18] just in significantly fewer words.

Table 2.1: AuCPace Parameters

parameter	explanation
PBKDF_σ	A Password-Based Key Derivation Function (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 Diffie-Hellman (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 2.2: 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

2.2.2 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.

2.2.3 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 implmenting the library. Allowing users of the library to pick from either Argon2, Scrypt or PBKDF2 at their discrection, 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 yciteosswu-map, which is known to be less efficient than the Elligator2 map defined for Montomery curves. 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.

2.3 Initial Proof of Concept design

2.4 Improving the initial design

[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

Glossary

Abelian Group A group whose operator is also commutative. e.g. Addition over $\mathbb{Z}$. . 13, 14

AES Advanced Encryption Scheme. 33

AKE Authenticated Key-Exchange. 16

API Application Programming Interface. 29

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. 10, 11

AuCPace Augmented Composable Password Authenticated Connection Establishment. 4, 16, 17, 18, 19, 22, 25, 26, 27, 29, 41

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. . 9, 12, 15, 16, 32

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. . 9, 11, 15, 16

CFRG Crypto Forum Research Group. 15, 32

CPace Composable Password Authenticated Connection Establishment. 15, 18

DH Diffie-Hellman. 11, 14, 16, 18, 27, 38

EAP Extensible Authentication Protocol. 10

ECC Elliptic Curve Cryptography. 4, 15

ECDLP Elliptic Curve Discrete Logarithm Problem. 15

EKE Encrypted Key Exchange. 10, 11, 12

FFI Foreign Function Interface. 23

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. 14

FIPS Federal Information Processing Standards. 26

HMI human machine interface. 16

IETF Internet Engineering Task Force. 15, 16

IIOT Industrial Internet of Things. 4, 16, 17, 25

IOT Internet of Things. 4, 25

iPAKE identity-binding PAKE. 16

IRTF Internet Research Task Force. 15

KHAPE Key-Hiding Asymmetric PAKE. 16, 17

MCU Microcontroller. 41

NIST National Institute of Standards and Technology. 13, 26, 29

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. . 17

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. 9, 11

OPAQUE An Asymmetric PAKE Protocol Secure Against Pre-Computation Attacks. Augmented PAKE Winner of the Crypto Forum Research Group (CFRG) PAKE selection process. The name is a play on words from OPAKE, where O is Oblivious Pseudo Random Function (OPRF). 15, 16, 17

OPRF Oblivious Pseudo Random Function. 16, 32

OSSWU Optimized Simplified Shallue-van de Woestijne-Ulas. 29

PAKE Password-Authenticated Key-Exchange. 4, 8, 9, 10, 12, 15, 16, 17, 23

PBKDF Password-Based Key Derivation Function. 27, 28

PKI Public-Key-Infrastructure. 16, 17

PRS Password Related String. 18

PSK Pre-Shared Key. 17

RCIG Rust Cryptography Interest Group. 27

RSA Rivest-Shamir-Adleman. 10, 13

RTT Real-Time-Transfer. 25

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. . 11, 12

SHA Secure Hash Algorithm. 27

SPAKE Simple PAKE. 11, 12

SRP Secure Remote Password. 12, 38

SSID Sub-Session ID. 17, 18

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). 10

TLS Transport Layer Security. 8, 12

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 12. 9, 12

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.

[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.

[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.

[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.

[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.

[ST20] 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.