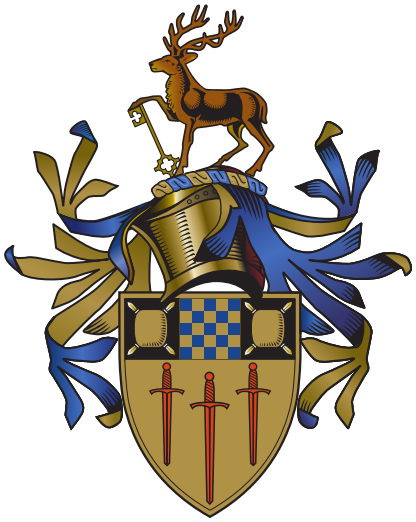


Formal Verification and Implementation of a Post-Quantum Multi Factor and Multi Server Authentication Scheme

A thesis submitted in partial fulfilment of the
requirements for the award of

MASTERS OF SCIENCE IN CYBER SECURITY

by

Aran Simon Jannson

August 31, 2026

School Of Computer Science & Electronic Engineering
University of Surrey
Guildford, England
United Kingdom
GU2 7XH

Supervised By: Dr Daniel Gardham

Copyright 2026 © Aran Simon Jannson — All Rights Reserved

I declare that this dissertation is my own work and that the work of others is acknowledged and indicated by explicit references.

Aran Simon Jannson
November 20, 2026

Copyright 2026 © Aran Simon Jannson

Abstract

This thesis explores the formal verification of Multi-Server Authentication (MSA) schemes, looking at different implementation formats. Including classical-cryptographic implementations and post-quantum cryptographic implementations paired with biometrics. The research tested the protocol design provided by the paper titled *Post-Quantum Secure Multi-Factor Authentication Protocol for Multi-Server Architecture* [48]. Using ProVerif the existence of a DoS attack was discovered. A solution to mitigate this attack was developed using Message Authentication Codes (MAC).
The protocol was also tested for practicality and feasibility in a real world environment. A physical fingerprint scanner and an emulated smart card system was incorporated. This was done using Python and FastAPI, allowing for a Secure Sketch, Fuzzy Extractor, and the multi-Server authentication protocol to be implemented and tested in a a real world environment.

Acknowledgements

I would like to thank my supervisor Dr Daniel Gardham for the assistance and guidance he provided through out this project. He was always available when I needed to ask a question and gave his full attention and effort to answer any concerns I had along the way. His support and feedback was invaluable and I thank him for everything he has done.

Abbreviations

FE	Fuzzy Extractor
FS	Fingerprint Scanner
PQ	Post-Quantum
MFA	Multi Factor Authentication
MSA	Multi Server Authentication
PQC	Post-Quantum Cryptography
ML-KEM	Module-Lattice-Based Key-Encapsulation Mechanism
RSA	Rivest–Shamir–Adleman Algorithm
SHA	Secure Hash Algorithms
LWE	Learning With Errors

Contents

1 Introduction **1**
- 1.1 Overview . . . 1
- 1.2 Project Goals . . . 2

2 Literature Review **3**
- 2.1 Multi Server Authentication (MSA) . . . 3
  - 2.1.1 Biometric Password-Based MSA . . . 3
  - 2.1.2 ID-Based MSA . . . 4
  - 2.1.3 Diffie–Hellman-Based MSA . . . 4
  - 2.1.4 Elliptic-Curve Cryptography Based MSA . . . 4
  - 2.1.5 BAN-Logic Based MSA . . . 5
  - 2.1.6 Quantum-Key-Distribution Based MSA . . . 5
- 2.2 Security Verification Tools . . . 5
  - 2.2.1 Scyther . . . 5
  - 2.2.2 Tamarin Prover . . . 6
  - 2.2.3 ProVerif . . . 7
  - 2.2.4 Conclusion . . . 7
- 2.3 Reusable Fuzzy Extractor from LWE . . . 7
  - 2.3.1 General Algorithm Methodology . . . 8
  - 2.3.2 Gen Algorithm . . . 9
  - 2.3.3 Rep Algorithm . . . 10
  - 2.3.4 Conclusion . . . 11

3 Background **11**
- 3.1 Standard Cryptography . . . 11
  - 3.1.1 General Overview Of Cryptography . . . 11
  - 3.1.2 Symmetric Cryptography . . . 11
  - 3.1.3 Public Key Cryptography . . . 11
  - 3.1.4 Hash Functions . . . 12
  - 3.1.5 Message Authentication Codes (MAC) & Hashed-MAC . . . 13
- 3.2 Problems with Classical Cryptography . . . 13
- 3.3 Learning With Errors (LWE) . . . 14
- 3.4 Post-Quantum Cryptography . . . 14
  - 3.4.1 Lattice-based PQ Cryptography . . . 14
- 3.5 ML-KEM/CRYSTALS-Kyber . . . 15
- 3.6 Biometric Security . . . 16
  - 3.6.1 Biometric Authentication . . . 16
  - 3.6.2 Biometric Accuracy . . . 16
  - 3.6.3 Fuzzy Extractors (FE) . . . 16
- 3.7 Secure Sketches . . . 17
- 3.8 Conclusion . . . 17

4 Post-Quantum Secure Multi-Factor Authentication Protocol for Multi-Server Architecture **18**
4.1 Registration Phase . . . 18
4.1.1 Server Registration . . . 18
4.1.2 User Registration . . . 19
4.2 Login and Mutual Authentication Phase . . . 20
4.3 Password Updating . . . 21
4.4 Smart Card Revocation . . . 21
4.5 RoR Security Model . . . 22
4.6 Critical Evaluation . . . 22
4.7 Conclusion . . . 23

5 Design, Implementation, & Hardware Choices **23**
5.1 General Design Choices . . . 23
5.2 Protocol Design . . . 24
5.2.1 Server Registration . . . 24
5.2.2 User Registration . . . 24
5.2.3 Login . . . 25
5.3 Verification Tools . . . 27
5.4 Biometrics . . . 27
5.5 Secure Sketch . . . 28
5.5.1 Pre Processing . . . 29
5.5.2 Gen . . . 30
5.5.3 Rec . . . 30
5.6 Fuzzy Extractor . . . 31
5.6.1 Gen . . . 31
5.6.2 Rep . . . 32
5.7 Smart Card Emulation . . . 33

6 Formal Security Analysis **35**
6.1 Introduction . . . 35
6.2 ProVerif Base Model . . . 35
6.3 Mutual Authentication . . . 39
6.4 Confidentiality . . . 40
6.5 DoS Security . . . 40

7 Fixed Protocol **41**
7.1 Protocol Description . . . 41
7.2 Possible Solutions . . . 42
7.3 Security Analysis . . . 42
7.3.1 Protocol . . . 42
7.4 Implementation . . . 43
7.5 Security Properties . . . 45

8 Software Testing **45**
8.1 Testing Environment 45
8.2 Registration 45
8.2.1 User Registration 45
8.2.2 Server Registration 46
8.3 Login 47
8.4 Biometric and Secure-Sketch Accuracy Testing 48

9 Project Reflections **49**
9.1 General Project Reflection 49
9.2 Success In The Project 49
9.3 Issues Faced 50
9.4 Future Work 50
9.5 Final Thoughts 51

List of Tables

1 Project Objectives 2
2 Common Hash Functions and Digest Lengths 12
3 Comparison of Hash Functions, MACs, and HMAC 13
4 Core Python Libraries 23
5 Smart Card Structure 34
6 Equivalent values calculated by the user and server. 43
7 BCH Values Used During Testing 49

List of Figures

1 Example Overview of a MSA Scheme Used In This Project 3
2 Scyther On Linux Example [18] 6
3 Tamarin Prover Example Attack Trace [13] 6
4 ProVerif Attack Trace Example [8] 7
5 ML-KEM Encapsulation and Decapsulation Flow [20] 15
6 Secure Sketch Scheme [19] 17
7 Fingerprint Collection Using Joy-It Fingerprint Scanner 33
8 DoS Attack Trace 41
9 MAC verification between the user and the server 43
10 DoS Attack Failed 44
11 Server Registration Times 46
12 Initial and secondary login times based on ML-KEM profile 47
13 Biometric accuracy for based on BCH error-correction values 48

1 Introduction

1.1 Overview

The growth in demand for cloud services, in peoples personal lives and professional careers, has resulted in people interacting with several services and systems. These systems process sensitive information about the user, such as identification data, passwords, and processed biometrics. Due to this a requirement has come to attention in which users should be securely and safely authenticated for these servers and systems. Multi-Server Authentication (MSA), provides a solution to this problem, it allows for a user to be able to authenticate themselves with the same portal and credentials across multiple servers hosted by multiple separate entities without the need to re-register for each server. The usability of this feature strives on the idea that the user does not need to communicate with a Registration Centre (RC) each time they want to log in to a service. This is critical as the number of services a user needs to access scales exponentially.

Standard user authentication requires the user to input a username and password pair, it provides a convention identification mechanism. However, the security of this pair is highly dependent on the length and strength of the credentials entered by the user. A user who uses a low character password with a lack of character diversity would still be susceptible to having their account breached. Weak or exposed passwords can leave authentication systems vulnerable to attacks such as password guessing and dictionary attacks, while compromised credentials can potentially provide an attacker with access to multiple services. Multi-Factor Authentication (MFA) can help reduce this risk and dependence on passwords by requiring additional authentication steps such as passkeys, question sets, and biometrics. Biometrics are especially useful for providing MFA as biometrics provide a secure, unique, yet unforgettable element to user authentication. However, as secure as using biometrics can be, the data provided from biometric scans are noisy, this means that multiple scans of the same biometric element will not necessarily produce identical data. To solve this, techniques such as Fuzzy Extractors (FE) can be used to generate cryptographic data from biometric measurements whilst accounting for variations between scans.

Combining biometric MFA and MSA provides a more secure authentication scheme, which combines standard password based authentication with the addition of a secure biometric component, whilst still allowing for a user to log into multiple servers or systems with the same credential set. Existing frameworks have attempted to accomplish this. However, authentication protocols designed must not only consider existing attacks, but must also consider attacks posed by advances in technology.

A major threat emerging is the expanding development in quantum computing. A quantum computer with enough computational power could break current asymmetric-cryptographic schemes such as RSA and Diffie-Hellman key exchange protocols. These asymmetric-cryptographic methods are standardised to protect digital communications. In an attempt to solve this issue, the National Institute of Standards and Technology (NIST) has developed a standardisation process for Post-Quantum Cryptography (PQC), this includes a deadline for implementing PQC standards as well as PQC schemes that can be used as a replacement for existing asymmetric-cryptographic protocols. One of these standards is the Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM) which incorporates lattices to provide quantum protection and is one of the standards outlined by NIST in the FIPS 203 report [34].

These modern advancements in the field of PQC make the investigation into post-quantum authentication systems particularly important. PQC-MSA schemes have been proposed to provide multi-server authentication whilst replacing quantum-vulnerable cryptographic protocols

Copyright 2026 © Aran Simon Jannson

currently in use with PQC alternatives. Although the introduction of PQC is a significant leap in security for MSA, it does not directly guarantee that the protocol is secure. If the protocol has faults in the design, then it may still be vulnerable to attacks. Considering the importance of MSA protocols, it is important that the proposed protocols are tested to a high degree before being deployed in real-world scenarios.

Therefore, the goal of this thesis is to examine the existing protocol designs and the feasibility of a PQC-MSA scheme that takes advantage of biometrics. The security, practicality are examined and tested for possible vulnerabilities and, if any are detected, a fix will be provided. A practical implementation will also be created to test the practicality and feasibility of the protocol.

1.2 Project Goals

The main goals of the project are to assess the security and feasibility of implementing a proposed PQC-MSA system, including biometric MFA. Biometric scans are collected using an FE. The protocol examined is based on the authentication scheme proposed in [48], alongside the reusable Fuzzy Extractor construction presented in [47].

The project implements the proposed protocol and examines its security, investigates claims made by the paper [48], and the practicality of implementing such a protocol. This includes applying a formal verification on the protocol following the Dolev-Yao model to identify any security vulnerabilities, but also developing the protocol in Python to test its real world application and practicality.

Objective	**Description**
1	Evaluate The Contents in Both Core Papers.
2	Evaluate The Security of The Protocol.
3	Create Dolev-Yao Based Formal Verification Scripts
4	Implement the Secure Sketch and FE Scripts
5	Test FE Script with Fingerprint Scanner
6	Implement MFA-PQC Scripts
7	Implement FE Scripts into MFA-PQC Scripts
8	Create Basic UI & REST-API
9	Fix Vulnerabilities & Improve
10	Test and Compare The Updated Scheme With The Original Scheme

Table 1: Project Objectives

Copyright 2026 © Aran Simon Jannson

2 Literature Review

2.1 Multi Server Authentication (MSA)

Multi-server authentication (MSA) is a feature that allows a user to log into multiple servers with the same credentials after registering once with a registration centre. An MSA usually includes users, a set of servers and a registration centre and provides credentials of the newly registered user to each server automatically [45, 52].

MSA systems provide a secure session that is established between the user, the registration centre, and the server. This allows for reliable authentication to take place, insuring that all parties within a communication are verified [45, 48, 52].

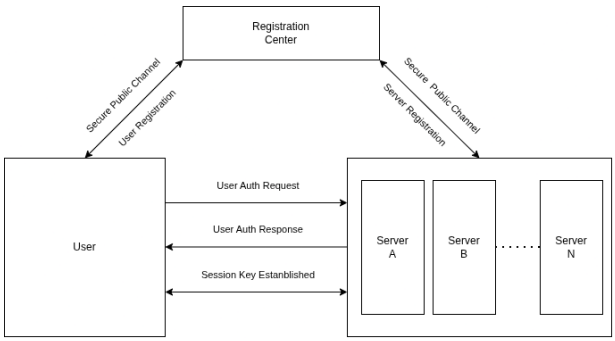


Figure 1: Example Overview of a MSA Scheme Used In This Project

An MSA protocol can provide its RC services online or offline. An online RC service seems to be the most common solution, essentially having the user communicate with an additional server outside of the domain it is trying to access to provide authentication. This allows both the user and the server to mutually verify each other. An offline RC creates long term keys that can be shared between entities, essentially only using the RC during the registration phase and taking it out of the equation during the login phase of the protocol [48].

2.1.1 Biometric Password-Based MSA

The first multi-server authentication scheme presented by Dexin Yang and Bo Yang [49] proposed the implementation of a biometric protocol that combines the use of biometric data and a password within a smart card. The scheme uses a registration centre, they claim that the combination of using biometric data with a password provides security benefits [49].

In their scheme, the user registers by providing an identifier $\mathbf{ID_i}$, a password $\mathbf{PW_i}$ and a biometric scan $\mathbf{w}$. This data is provided to a fuzzy extractor which generates helper values. These helper values can then be used with a secondary fingerprint scan to authenticate the user locally, a authentication request is then sent to the server. The server generates its own random value, and then sends an authentication response to the user. The user verifies this response and the server and the user create a shared session key using the Diffie–Hellman key exchange [49].

The user and the server mutually authenticate each other by confirming pre-registered values are valid, as well as whether the recovered biometric image matches the original value or not. If the user and the server are able to successfully recover these values, provided the timestamps are within a pre-defined range, the user and the server have a shared session key. Allowing communication between the two entities to function [49].

This paper provides an alternative use case to biometric MSA protocols, by using a more simple approach based on standard public key cryptography rather than vector based lattice cryptography. The paper explores a smart card style multi-factor authentication method similar to the PQMSA paper explored by this project. However, the smart card provides a more simplified format, and the values stored do not provide protection against post-quantum attacks that may arise in the near future.

2.1.2 ID-Based MSA

The protocol proposed by Jen-Ho Yang, Ya-Fen Chang, and Chih-Cheng Huang [50] provided a solution to a multi server authentication scheme by taking an ID-based approach. The protocol takes advantage of direct hash values and XOR operations to reduce computational requirements.

The registration phase requires a user to provide an identifier $\mathbf{ID_u}$ and a password $\mathbf{PW_u}$, which is passed on to an identity provider, a secret key $\mathbf{x}$ is then generated using these values. Similarly, the server is registered by generating an identifier $\mathbf{ID_{S_i}}$. The server then receives the value $\mathbf{B}$ that is generated using the user's private key and provides the server with identification values for a user [50].

The scheme provides an efficient solution to the MSA problem. However, the lack of communication stages and formal verification lacks sufficient depth from the protocol. The security analysis presented largely follows informal arguments against individual attacks, such as insider attacks. No direct formal verification is shown on the paper that is tested by a tool such as Scyther [5] or ProVerif [17]. Compared to the core paper PQMSA, it can be seen that the difference between analysis as well as post quantum security differences between the two papers is drastic. The ID-based paper having been designed prior to the standardisation of post-quantum schemes allows it to be vulnerable to attacks not present during its design time frame [50].

2.1.3 Diffie–Hellman-Based MSA

Himanshu Mittal proposes an MSA protocol that takes advantage of smart cards as well as Diffie-Hellman key exchange [30]. Similarly, taking advantage of hash functions and XOR operations, the scheme is based on a user, a server, and an authentication centre. A user registers with the authentication centre by selecting a user identifier $\mathbf{ID}$ and a password $\mathbf{Pw}$, in addition a random value $\mathbf{K}$ is chosen and these are then provided to the authentication centre.

The protocol enables a secure nonce based MSA solution. The use of the authentication centre as a middle man allows for the user and the server to securely communicate with each other without directly communicating, enabling enhanced security. However, similarly to the other schemes, the protocol does not provide post quantum security and relies on cryptographic primitives which can be broken within polynomial time once quantum computing becomes mainstream.

2.1.4 Elliptic-Curve Cryptography Based MSA

Yunru Zhang, Min Luo, Kim-Kwang, Raymond Choo, and Debiao He, [53] proposed an MSA protocol scheme in which users and servers register with a central registration point. The solution provided by this paper differs from others as it uses an Elliptic-Curve Cryptography (ECC) based key agreement protocol. This scheme provides security claims that provide single registration, mutual authentication, anonymity, un-traceability, and forward and backward secrecy, with its authenticated key agreement protocol. The scheme relies on ECC and the hardness of the ECDH

problem, which leads to the scheme not being quantum secure. Leaving it vulnerable to future post-quantum attacks.

2.1.5 BAN-Logic Based MSA

Huawei Wang, Dianli Guo, Hua Zhang, and Qiaoyan Wen proposed an MSA scheme [46] that aimed to address impersonation and server spoofing problems. The design uses a RC that users register with once and are then able to access all servers on the network. During registration, a biometric scan is registered using a fuzzy extractor reproducible secret $\mathbf{R_i}$, which is then combined with the user's identification value and password to create an authentication token $\mathbf{K_ij}$. The paper analyses authentication using Burrows—Abadi—Needham (BAN) logic and claims resistance to impersonation and spoofing attacks. A down side to this scheme is its reliance on RSA based protocols and modular arithmetic, leading to a lack of post quantum security.

2.1.6 Quantum-Key-Distribution Based MSA

Ze Yang, Qin Shi, Teng Cheng, Qiang Zhang, Qiang Liu, Yangyang Liu, and Shenghao Peng explored the idea and proposed a protocol for Quantum-Key-Distribution (QKD) based MSA [51]. This system was designed for blockchain-enabled Internet of Vehicles environments. It allows for a quantum secure cloud usage for registering and authenticating vehicles. However, the protocol should not be considered post-quantum secure at face value as its initial registration and authentication phase still continue to use elliptic-curve cryptography. Providing a base entry for post-quantum security does not provide the complete necessary post-quantum elements necessary. Furthermore the system requires specialised hardware and systems to function, adding additional integration and practicality restrictions.

2.2 Security Verification Tools

Formal security verification tools enable a method to analyse the security properties of a provided scheme. This allows a formal representation of the protocol to be generated. The tools can also be useful in finding attack traces that where previously unknown, tools such as Scyther [5], ProVerif [17], and Tamarin Prover [13] provide this ability. However, one tool may model specific security properties better than the other.

2.2.1 Scyther

Scyther is an open source tool that is used to analyse and model cryptographic protocols [5]. It provides its own language structure as well as a coding environment. The tool provides logical modelling for key generation and sharing however, it lacks the ability to provide mathematical operations such as XOR. The tool as seen Figure 2 provides easy adoption to those new to cryptographic modelling for detecting and verifying attacks. Although the tool has some logical limitations, Almuzaini and Ahmad [16] were able to use Scyther to analyse the Signal messaging protocol [12] by dividing the protocol into subsections, rather than modelling the protocol in its entirety.

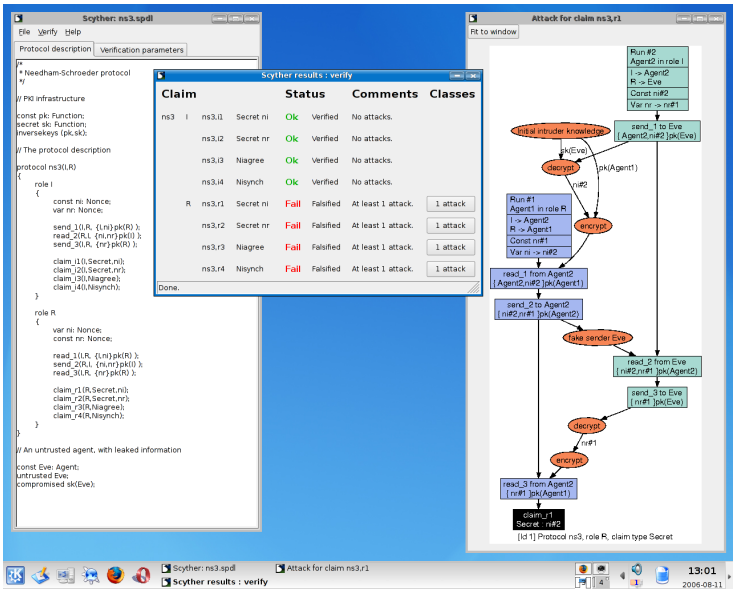


Figure 2: Scyther On Linux Example [18]

2.2.2 Tamarin Prover

Tamarin Prover provides a more complete and complex modelling tool [13], following the Dolev-Yao model to generate attack traces and testing protocols. Enabling automatic verification of attacks. Tamarin models rules that represent the protocol being modelled, the rules can provide information about whether a variable can be captured by an attacker or if it is private. Attack traces are then generated using the defined rules, whilst any required security rules are taken into account.

An example of Tamarin Prover being used can be seen in the paper written by Mieno, Takehiko and Okazaki, Hiroyuki and Arai, Kenichi and Futa, Yuichi [28]. In the paper, they use Tamarin Prover to formally verify encryption schemes by modelling symmetric and authenticated message codes. An example of a Tamarin Prover attack trace can be seen in Figure 3, where an adversary can gain information that can be used to formulate an attack.

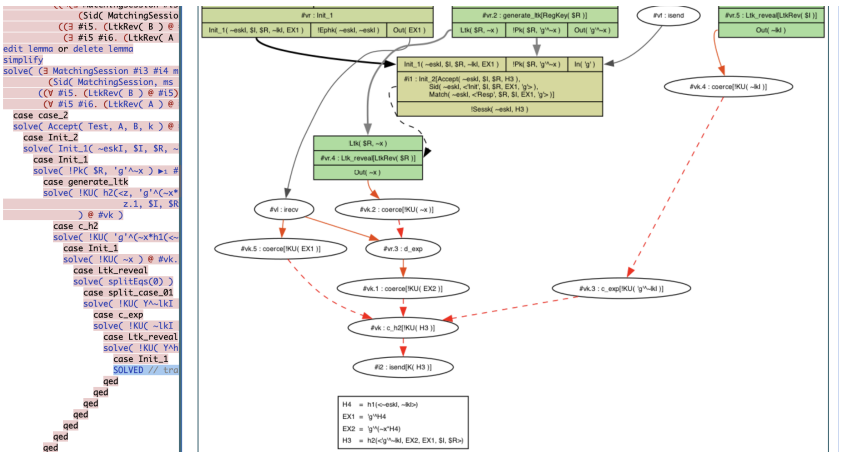


Figure 3: Tamarin Prover Example Attack Trace [13]

Copyright 2026 © Aran Simon Jannson

2.2.3 ProVerif

ProVerif provides another alternative to automatically modelling and analysing cryptographic protocols [17]. The tool allows for cryptographic logic to be represented through defined rule sets, including secrecy and integrity. The tool does not provide a first party graphical interface. However, it supports plug-ins for existing coding environments, allowing for a lightweight baseline usage. ProVerif also supports the generation of attack traces, which get generated and trigged by applying events and attack queries to existing variables and custom functions.

Tran Thi Nga, Nguyen Thanh Tung, Nguyen Duc Cong, Dinh Van Hung, and Duc-Tho Mai [33] used ProVerif to formally model a 5-G blockchain using the Dolev-Yao model. In the developed protocol, the attacker controlled the network and can intercept and modify information on the network. The attacker also had access to public functions. Showing that the tool can be used to model network attacks successfully. An example of a ProVerif attack tree can be seen in Figure 4, where an attack was detected were a secret message was recoverable.

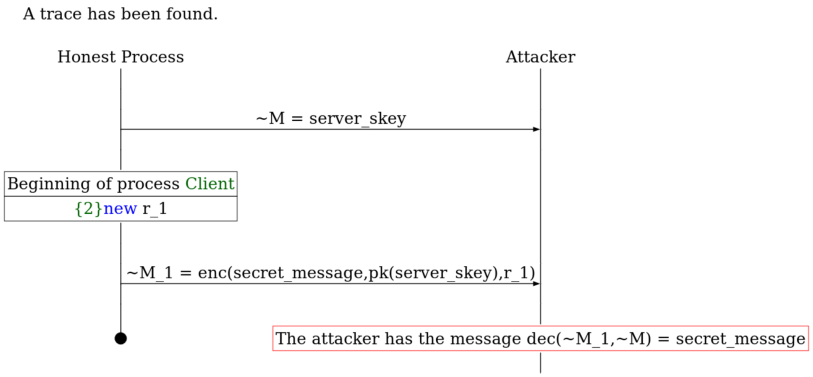


Figure 4: ProVerif Attack Trace Example [8]

2.2.4 Conclusion

In conclusion, ProVerif was chosen for this project due to its ability to be easily learnt and because of its ability to emulate functions as well as partially implement XOR logic in protocols. Although Tamarin can develop more complex logic, ProVerif can develop the necessary protocols without the higher learning curve. Scyther was not considered due to being unable to implement XOR functions in any form, however, it was initially attempted and was later dropped in favour of ProVerif.

2.3 Reusable Fuzzy Extractor from LWE

The paper written by Yunhua Wen and Shengli Liu [47], provides a reusable solution to fuzzy extractors, which bases its security on Learning With Errors (LWE) assumptions. A standard FE generates a cryptographic value from a noisy biometric scan, applying error correction up to a pre-specified hamming distance. This research takes this core principle and creates helper data from an initial registration scan of the user biometrics. This allows the value to be reproduced from a second biometric scan. Standard FE may not allow for repeated scans from the same biometric source because the resulting keys remain related to the original biometric scan. This paper solves this issue by taking advantage of the homomorphic properties of a secure sketch

algorithm as well as a homomorphic hashing algorithm. The usage of a secure sketch allows for the correction between two biometric reads, in this case being the stored pre-registered value and the scan taken during a login attempt. A homomorphic hash output is used as an encryption key instead of the final value. The key is derived from a sample value that is randomly selected and encrypted, this allows for the usage of multiple levels of keys to be generated from a singular value whilst the pseudo randomness remains. This provides a reusable FE that is stated to be the first of its kind by the authors of the paper [47].

2.3.1 General Algorithm Methodology

The proposed algorithm takes a secure input $Init(1^{\lambda})$ that generates a set of public parameters pp [47]. The main objective of the public parameters is to define the set of hash functions to be used for the FE Gen and Rep functions. The algorithm Gen takes the inputs pp and w, where $w \rightarrow BioRead$ and $w \in M$ [47]. Its goal is to generate a helper value P and an extracted value R from an inputted biometric read, P is used in the next step during the reproduction of a secondary scan to provide a comparison of the originally generated scan with the new secondary scan. The value of R is a bit string of length l calculated during the Gen process. Rep is responsible for authenticating the registered biometrics using the values pp, P, and w', where $w' \rightarrow BioRead'$ and $w' \in M$. It outputs a value R which should match the one outputted in the Gen algorithm [47].

Internal Components

1. **Homomorphic Secure Sketch:** A secure sketch (SS) applied error correction to a bit-string. In the case of the paper, the bitstring is the biometric input $\mathbf{w}$. This creates a helper variable which can later be used to reconstruct the biometric input $\mathbf{w}$ from a new input $\mathbf{w}'$, as long as the new scan provided is within a specified hamming distance.

   In the paper a SS generation is calculated from the error correcting LWE code:

   $$s \leftarrow \mathrm{SS.\,Gen}(w) = Hw$$

   This provides a helper value for the secure sketch s which allows for the recovery algorithm to re-create the fingerprint w with a newly taken scan w' when the hamming distance of the two is below a specified threshold.

2. **Homomorphic Universal Hash Function:** The paper defines the hash function as:

   $$H_A(w) \leftarrow \mathbf{A}w$$

   with the homomorphic nature being defined as follows:

   $$H_A(w + w') \leftarrow H_A(w) + H_A(w')$$

   This hash derives a value $S = H_i(w)$, where w is the biometric input. The value S is then used to derive keys using LWE symmetric encryption.

3. **LWE Symmetric Encryption:** The value derived from the homomorphic hash S is used to create a LWE symmetric private key. A random message value m is then encrypted, and that random message becomes the final key R. Rather than using $S = H_i(w)$ as the private key, which could allow for similarity to be detected when the same biometric is scanned multiple times, it is used to encrypt a random value to prevent this. LWE is then applied to hide the similarities between the different keys.

2.3.2 Gen Algorithm

The *Gen* algorithm provides an output of P and R. The user inputs the biometric value w, in which the *Gen* algorithm starts by completing the operations [47]:

$$s \leftarrow \mathrm{SS.\,Gen}(w)$$
$$\mathbf{S} \leftarrow H_i(w) \in \mathbb{Z}_q^{n \times l}$$

The value s is a secure sketch based on the biometric input w, following this a homomorphic hash S is generated that fits within $\mathbb{Z}_q^{n \times l}$ [47].

$$\mathbf{A} \xleftarrow{\$} \mathbb{Z}_q^{m \times n}$$
$$\mathbf{E} \leftarrow \chi^{m \times l}$$

The matrix A belongs to $\mathbb{Z}_q^{m \times n}$ whilst χ is a Centered Binomial Distribution (CBD) with width σ.

$$\mathbf{B} \leftarrow (\mathbf{A},\, \mathbf{AS} + \mathbf{E}) \in \mathbb{Z}_q^{m \times (n+l)}$$

With the value of $\mathbf{AS}$, having a dimension of $m \times l$, it can be paired with the error value $\mathbf{E}$. It can then create a completed matrix value used to provide an LWE assumption as the value $\mathbf{B}$.

$$\mathbf{x} \xleftarrow{\$} \{0,1\}^m$$

A random bit string $\mathbf{x}$ is generated based on the length of $\mathbf{m}$, which will then be used as the random value applied to the final output of the fuzzy extractor function **Gen** [47].

$$\mathbf{m} \xleftarrow{\$} \{0,1\}^l$$

Similar to the previous value of $\mathbf{x}$, the value $\mathbf{m}$ is also a bit string, however, this time its is of length $\mathbf{l}$ and is applied to the final output as a random value.

$$\mathbf{c}^\mathsf{T} \leftarrow \mathbf{x}^\mathsf{T}\mathbf{B} + \left(\mathbf{0}^\mathsf{T}, \mathbf{m}^\mathsf{T} \cdot \left\lfloor \frac{q}{2} \right\rceil\right)$$

Finally, the value of c^T is a combination of the LWE assumption value $\mathbf{B}$ and the random values $\mathbf{x}$ and $\mathbf{m}$ [47], a transposed zero bit is added to enable the reproduction algorithm to differentiate both bit values, if the LWE error values added are reasonably small [47].

$$P \leftarrow (s, \mathbf{c})$$
$$R \leftarrow \mathbf{m}$$
$$\textbf{return } (P, R)$$

The algorithm provides the user with the output $\mathbf{P} \leftarrow \mathbf{s}, \mathbf{c}$ and $\mathbf{R}$. $\mathbf{P}$ contains both the secure sketch $\mathbf{s}$ and the LWE ciphertext $\mathbf{c}$. Whilst, the exported value $\mathbf{R}$ is the randomly sampled bit string $\boldsymbol{m}$.

Following the output of the **Gen** algorithm, the values can then be stored and used in the **Rep** algorithm to reproduce the fingerprint from the helper data and a new scan. This can be used to allow the user access to a system such as a server. The recreation of the fingerprint relies on the values generated by the **Gen** algorithm and any mistakes with its implementation could prevent the fingerprint from being reproduced by the **Rep** algorithm.

2.3.3 Rep Algorithm

The **Rep** algorithm takes the inputs of $\mathbf{pp}$, $\mathbf{P} \leftarrow (\mathbf{s}, \mathbf{c})$ and a new fingerprint scan $\mathbf{w'}$. Where $\mathbf{pp}$ is the public parameters used in the **Gen** algorithm, $\mathbf{P}$ is the public helper value stored from initial registration.

$$R \leftarrow \mathrm{Rep}(pp, P, w')$$
$$\text{Parse } P = (s, c)$$

Using newly scanned fingerprint data $\mathbf{w'}$ and helper data from the **Gen** algorithm $\mathbf{s}$ is used to create a regenerated version of the original scan $\widetilde{\mathbf{w}}$. The length of both $\mathbf{w}$ and $\mathbf{w'}$ must be the same and the error correction applied provides a valid value based on the predefined amount $\mathbf{t}$. In principle, if the same biometric element is scanned, then $\mathbf{w} = \mathbf{w'}$ [47].

$$\widetilde{w} \leftarrow \mathrm{SS.Rec}(s, w')$$

Using the constructed value of $\widetilde{\mathbf{w}}$, the same hash function used during **Gen** is applied to allow the LWE secret key to be reconstructed. If all values and the hash function match, then $\mathbf{H_i}(\widetilde{\mathbf{w}}) = \mathbf{H_i}(\mathbf{w})$ [47]. Creating the same matrix $\mathbf{S}$ as in **Gen** which respects $\mathbf{S} \in \mathbb{Z}_{\mathbf{q}}^{\mathbf{n} \times \mathbf{l}}$.

$$\mathbf{S} \leftarrow H_i(\widetilde{w}) \in \mathbb{Z}_q^{n \times l}$$

The calculation of the value $\mathbf{d}$ produces the same value as $\mathbf{m}$ in the **Gen** function, allowing for the recovery of the encrypted message that contains an LWE value.

$$\mathbf{d} \leftarrow \mathbf{c}^{\mathsf{T}} \begin{pmatrix} -\mathbf{S} \\ \mathbf{I}_l \end{pmatrix} \in \mathbb{Z}_q^l$$

A check is applied to each bit of $\mathbf{d}$ and a decision is made based on if $\mathbf{d[i]} \in \left[\frac{\mathbf{q}}{\mathbf{4}}, \frac{\mathbf{3q}}{\mathbf{4}}\right]$ [47], if the value of $\mathbf{d[i]}$ is an element of the matrix then the current value is marked as a 1 otherwise as a 0. Providing the value of $\mathbf{m}$, this is then outputted to the user as the value $\mathbf{R} := \mathbf{m}$. With the goal that $\mathbf{R_{Rep}} = \mathbf{R_{Gen}}$ [47] should be true. If the values do not match it can be said that the biometric inputs did not match each other.

$$\text{For } i = 1 \text{ to } l,$$
$$\mathbf{m}[i] = \begin{cases} 1, & \text{if } \mathbf{d}[i] \in \left[\frac{q}{4}, \frac{3q}{4}\right], \\ 0, & \text{else} \end{cases}$$
$$R := \mathbf{m}$$
$$\text{Return } R$$

2.3.4 Conclusion

In conclusion, the paper on reusable fuzzy extractors provides a low level algorithmic solution to being able to accurately create and reproduce a cryptographic key from biometric fingerprint data. This can be implemented in schemes such as authentication. The use of homomorphic attributes enables an accurate recreation of a cryptographic key based on the fingerprint scan, using helper data to provide a set of points that can be used to reconstruct the cryptographic key associated with biometric data [47]. Using LWE allows for an increased security by reducing the relation between multiple keys created from different readings of the same fingerprint. Finally, the accuracy of the reconstruction based on the biometric fingerprint scan remains within a predefined error correction hamming distance [47].

3 Background

3.1 Standard Cryptography

3.1.1 General Overview Of Cryptography

Cryptography is a form of mathematics that is used to protect data and information against unauthorised access and modification. These techniques can be used to provide user authentication, authorisation, and accountability. Techniques like these are essential in the modern day world as most of the global infrastructure has gone digital, creating secure services for platforms such as digital banking, cloud services, as well as a wider spectrum of personal information critical.

The security of an algorithm is only secure as long as the private keys remain a secret. In addition to this, the algorithm complexity in generating these keys also plays an equally important role, if the user repeats values or uses values that are predictable, this can compromise the entire system which relies on these keys.

3.1.2 Symmetric Cryptography

In symmetric cryptography both the receiver and the sender rely on a single shared key in order to encrypt and decrypt data sent between each other. The main examples of symmetric encryption are Data Encryption Standard (DES) and the Advanced Encryption Standard (AES) [35, 38], both of which provide block cipher based encryption solutions. The benefits to using symmetric encryption can be seen in the form of efficiency. If a system that has limited resources requires encryption and the data stored is not extremely critical, symmetric encryption is the best solution as it provides a level of security that is greater than having none at all. It can also be computed on low end systems without requiring drastic headroom. The main limitation of symmetric cryptography is the key distribution method. Before two people can communicate securely, the secret key should have already been shared prior to the conversation attempt. This creates a key management problem, especially in large-scale systems.

3.1.3 Public Key Cryptography

Public key cryptography uses a mathematically paired set of public and private keys to enable encryption and key generation on insecure channels. Most public key cryptographic schemes are based on integer factorisation (RSA), or discrete logarithm problems [21] (Diffie-Hellman and elliptic curve cryptography) [42].

RSA encryption computes the ciphertext as $c = m^e \pmod{N}$, while RSA decryption recovers the plaintext as $m = c^d \pmod{N}$ [37]. However, the Diffie-Hellman key exchange allows two users to establish a shared secret over an insecure channel. For example, Alice would select a private value a and publish $A = g^a \pmod{p}$ into a public domain, whilst Bob generates a private value b and publishes $B = g^b \pmod{p}$ to a public domain as well. Alice can then compute $K_A = B^a \pmod{p}$, and Bob can compute $K_B = A^b \pmod{p}$. This then creates the same values for Alice and Bob as $g^{ab} \pmod{p}$, allowing a shared secret key to be available [15].

3.1.4 Hash Functions

A hash function is an algorithm which maps an input of undefined length to a fixed-length output. Hash functions are commonly used to provide data integrity and support authentication protocols. NIST standardises the SHA-1 and SHA-2 family in **FIPS 180-4**, while SHA-3 and SHAKE are standardised separately in **FIPS 202** [40, 41].

Hash Function	**Hash Family**	**Digest Length**
MD5	Legacy	128 bits
SHA-1	SHA-1	160 bits
SHA-256	SHA-2	256 bits
SHA-512	SHA-2	512 bits
SHA3-256	SHA-3	256 bits
SHA3-384	SHA-3	384 bits
SHA3-512	SHA-3	512 bits
SHAKE128	SHA-3 XOF	Variable
SHAKE256	SHA-3 XOF	Variable

Table 2: Common Hash Functions and Digest Lengths

The table above presents a collection of common hash functions and their digest length variations. A cryptographic hash function can be defined as [2, 40]

$$h : \{0,1\}^* \to \{0,1\}^n,$$

Here the function takes an input bit string of undefined length and maps it to a fixed length output of n bits. The outputted hash value is notated as $h(x)$. An important property of cryptographic hash functions is its ability to be collision resistance, this means that a hashed value x and x' should not produce the same hash such that

$$x \neq x' \quad \text{and} \quad h(x) = h(x').$$

Two different inputs that produce the same hash are known as collisions [2]. Secure hash functions are designed from the group up to prevent such collisions. A hash value can be used to enable identification of a file or image but is most importantly used to secure passwords. For example, when a user wants to log into a system such as a server, they type their password in plaintext into a text box. Once the form is submitted, the plain text password is hashed and compared to a hashed password linked to this account on registration. If the two hashes match, then the password is presumed correct, otherwise it is incorrect.

To prevent hashes from being stored and predicable, a salt value can be added during the hashing process [14]. A salt is a random value that is included at the start or end of the plaintext

which alters the outputted hash value to enable uniqueness for the server provider.

$$s \leftarrow \{0,1\}^l$$

$$d = h(s \parallel x)$$

3.1.5 Message Authentication Codes (MAC) & Hashed-MAC

A Message Authentication Code (MAC) is a symmetric cryptographic method used to provide authenticity within messages sent between two or more parties. Unlike a hash function, a MAC uses a secret key shared between two parties. This key is communicated before the main communication in one form or another. The sender computes a MAC on a message using the secret key, and the receiver verifies the MAC using the same key. A MAC can be represented as [31]

$$mc = \text{MAC}_K(m)$$

Where K is the key that was shared between the parties, m is the message, and mc is the generated authentication MAC. The receiver can then verify the MACs authenticity by checking

$$\text{Verify}_K(m, mc) \in \{\text{accept}, \text{reject}\}.$$

If a message is modified during transit by an attacker, the verification should fail, enabling validation of authenticity. This can only fail if the attacker has access to the original key K used to generate the MAC [31].

A Hash-based Message Authentication Code, is a widely implemented form of MAC based on a hash function. HMAC is a hash based algorithm for message authentication using a reliable hash function [39]. A HMAC takes a secret key K, a message m, and a hash function h. Using these values it produces a fixed-length authentication MAC. It can be represented as [31, 39]

$$\text{MAC(text)}_t = \text{HMAC}(K, \text{text})_t = H\left((K_0 \oplus \text{opad}) \parallel H\left((K_0 \oplus \text{ipad}) \parallel \text{text}\right)\right)_t$$

Where K_0 is the key padded or shortened to the hash function block size, ipad and opad are the inner and outer padding constants applied to the hash to allow the required length [39].

Primitive	**Uses Secret Key?**	**Main Purpose**
Hash Function	No	Produces a fixed-length digest
MAC	Yes	Enables message integrity and authenticity
HMAC	Yes	Enables hash-based message authentication

Table 3: Comparison of Hash Functions, MACs, and HMAC

3.2 Problems with Classical Cryptography

Classical cryptography provided the foundation for security in the modern world. However, most public-key schemes rely on mathematical problems that are hard to brake using classical computers. For example, RSA security is based on factorising a large prime integer. The public key contains N, while the private key depends on the knowledge of $\varphi(N) = (p-1)(q-1)$. If an attacker can recover the factor N, then the private key can be computed, allowing messages to be decrypted.

The assumption of computationally complex to solve encryption schemes becomes vulnerable when quantum computing gets introduced. Shor's algorithm [10, 20] showed that integer factors and discrete logarithm based hard problems can be solved efficiently on a quantum computer, directly threatening RSA, Diffie-Hellman and elliptic curve cryptographic schemes. This means that a quantum computer can threaten these schemes, making them insecure, compromising the confidentiality and integrity of standard cryptography.

3.3 Learning With Errors (LWE)

Learning With Errors (LWE) is a hardness problem that has become an important foundation within lattice-based cryptography [43]. LWE is the problem of recovering a secret from an equation after a small random error value has been added. Without this error, the secret can be easily recovered using standard algebraic mathematics. The addition of this random error allows for the additional complexity which evidence suggests that it can not be broken within polynomial time by quantum computers. This makes it a perfect candidate for securing system against quantum attacks. This is why it is perfect for lattice-based cryptography [43].

The LWE problem is defined using a set of inputted pair values $(\mathbf{a}_i, b_i)$. For each pair, $\mathbf{a}_i \sim \mathbb{Z}_q^n$ is a vector sampled randomly, while b_i is calculated as [43, 44] :

$$b_i \leftarrow \mathbf{a}_i \cdot \mathbf{s} + \text{noise} \pmod{q}.$$

3.4 Post-Quantum Cryptography

Post-quantum cryptography (PQC) refers to cryptographic algorithms designed to remain secure against attacks from conventional computers and future quantum computers [32]. This has become more important as public cryptographic schemes are not quantum secure and rely on cryptography that is complex for a standard classical computer. This can be solved with ease using quantum computing.

3.4.1 Lattice-based PQ Cryptography

Lattice Based PQC, the type used in this project, is one of the most popular post-quantum schemes currently used. In lattice-based cryptography the security is commonly based on hard problems such as Learning With Errors (LWE) which supports key encryption schemes, and digital signatures. However, lattice-based schemes often introduce larger public keys and ciphertexts compared with traditional RSA or ECC. This creates a trade-off between long-term quantum resilience and storage overhead.

A lattice is a set of points in an n-dimensional space created by a combinations of basis vectors. If we have a basis matrix B, the lattice created by B is defined as [27]

$$\mathcal{L}(B) = \{B\mathbf{z} : \mathbf{z} \in \mathbb{Z}^n\}$$

For example, if

$$B = \begin{pmatrix} 2 & 1 \\ 0 & 3 \end{pmatrix}$$

Then the lattice is the set of points

$$\mathcal{L}(B) = \left\{ \begin{pmatrix} 2z_1 + z_2 \\ 3z_2 \end{pmatrix} : z_1, z_2 \in \mathbb{Z} \right\}$$

Each lattice point is formed by choosing an integer value for z_1 and z_2. For example, if we choose $z_1 = 1$ and $z_2 = 1$ and provide the lattice point of $(3, 3)$, and if we set $z_1 = -1$ and $z_2 = 2$ this will result in the values $(0, 6)$. A lattice is the collection of all points such as this.

3.5 ML-KEM/CRYSTALS-Kyber

ML-KEM, also known as CRYSTALS-Kyber, is a post-quantum key encapsulation algorithm [20]. It is designed to be a replacement for current public-key based key exchange algorithms. Due to ML-KEM being a post-quantum scheme, it is safe from future quantum attacks. Schemes such as RSA rely on hard problems such as large prime numbers which can be easily computed by a quantum computer, as proven by Shor's algorithm [20].

The security of ML-KEM is derived from lattice cryptography, taking advantage of LWE. Each operation is based on a set of vectors and matrixes computed using modular arithmetics. Each vector/matrix has a specified error added to them making it difficult for attackers to reverse engineer the public key values. It can be seen as noise or padding added for additional randomisation [20].

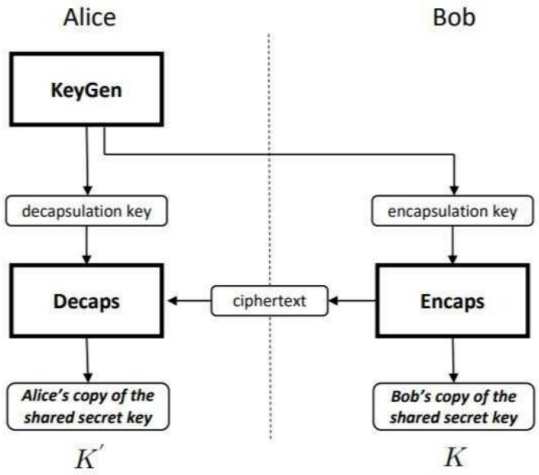


Figure 5: ML-KEM Encapsulation and Decapsulation Flow [20]

ML-KEM includes three core algorithms: *KeyGen*, *Encaps*, *Decaps*. The *KeyGen* algorithm is the core of the system and its main goal is to create a private and public key based on lattices. *Encaps*, encapsulates the secret message shared between two parties, it chooses a hashed new message to be sent to the public key owner, which is decided through a secure channel [20]. Using this message, two ciphertexts are created combined with noisy data. During *Decaps*, the recipient uses their private key and the ciphertext generated in *Encaps* to generate the same shared secret value as the sender.

3.6 Biometric Security

3.6.1 Biometric Authentication

Biometric authentication verifies a user's physical characteristics, such as fingerprints, iris scans, and facial features. In an authentication algorithm, biometrics represent a unique factor, or something the user is, and can be combined with password hashes and identifiers.

Biometric algorithms normally function for either the identification or verification of a user. To successfully identify a user, a one-to-many comparison is applied to a scan of the user's biometric to determine the identity associated with a biometric sample within a set hamming distance. Verification performs a one-to-one comparison between a newly captured scan and the biometric reference stored with a registered identity [29].

3.6.2 Biometric Accuracy

Unlike passwords, the same biometric is unlikely to produce identical bit strings when scanned multiple times. Noise, finger positioning, pressure, moisture, and cuts can cause for variable amounts of differences between the current read and the stored sample. To solve this issue, a hamming distance is set to allow for a set percentile of differences to be accepted between the two bit strings [29].

The hamming distance threshold determines whether a comparison is accepted. A low hamming distance may improve usability but increase the probability of accepting another entities biometrics as correct. A high hamming distance can reduce this risk, but may reject legitimate users. Two measurements for this are the false acceptance rate (FAR) and the false rejection rate (FRR) [29, 36]:

$$FAR = \frac{\text{Number of falsely accepted attempts}}{\text{Total number of fake attempts}},$$

$$FRR = \frac{\text{Number of falsely rejected attempts}}{\text{Total number of real attempts}}.$$

3.6.3 Fuzzy Extractors (FE)

Fuzzy extractors are the core system used to determine the authenticity of a biometric scan, specifically using fingerprints. Each biometric reading may differ from each other due to the noisy nature of the data, a fuzzy extractor converts this noisy, biometric input into a reproducible and almost uniformly distributed string suitable for cryptographic use.

A fuzzy extractor has two algorithms that can be called, these algorithms are: Gen and Rep. When a user registers their biometric data, the generation algorithm takes a biometric input w and produces a random string R and public helper data P [47]:

$$(R, P) \leftarrow \mathsf{Gen}(w)$$

When a user wants to login using a pre registered biometric a secondary input is taken as w'. Provided that the distance between w and w' is within the supported hamming distance t, the Rep process should return the same result [47]:

$$\text{HammingDis}(w, w') \leq t \quad \Longrightarrow \quad \mathsf{Rep}(w', P) = R$$

3.7 Secure Sketches

A secure sketch (SS) is an algorithm designed to protect noisy, non uniform data. Unlike standard cryptographic keys, it does not rely on the concept of being able to be recreated exactly as it was initially generated. An SS is provided with input values that vary between each calculation attempt. An SS allows an original value to be retrieved even if the inputted values are inaccurate by a few bytes by providing a helper string that includes enough information to allow error correction [22, 26].

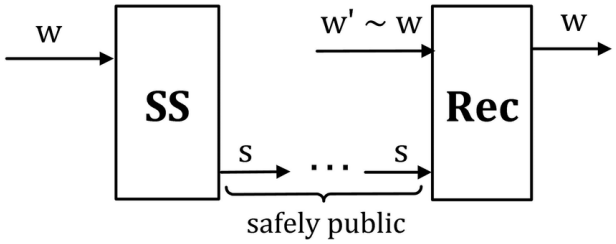


Figure 6: Secure Sketch Scheme [19]

Let $\mathcal{M}$ be a space that has a distance function [26]. A secure sketch consists of two algorithms, an algorithm to generate the sketch $SS \leftarrow Gen(w)$ and an algorithm to regenerate the original value $\widehat{w} \leftarrow \mathsf{Rec}(w', ss)$. The value of $w \in \mathcal{M}$ is the original input, ss is the publicly stored sketch that is used as a helper value that in turn is used in the recovery phase, and $w' \in \mathcal{M}$ is the secondary scan after initial generation which is used in the recovery stage. If the hamming distance between the original read used in Gen and the new noisy input w' is not greater than the error-tolerance threshold t that is predefined, the recovery algorithm should be able to reproduce the original value.

The probability of the value being correct follows the following formula [22]:

$$\mathrm{dis}(w, w') \leq t \quad \Longrightarrow \quad \mathsf{Rec}\left(w', \mathsf{SS}(w)\right) = w$$

A secure sketch provides the core error correction nature within a fuzzy extractor algorithm, while the fuzzy extractor additionally produces a stable value suitable for cryptographic use [22].

3.8 Conclusion

For the functionality of a post-quantum, multi server authentication scheme provides a modern quantum proof secure way to allow users to register once with a registration centre and access multiple services with the same credentials.

Examining the current symmetric and public key cryptographic security algorithms, it can be seen that symmetric encryption provides efficiency when a shared key is agreed upon between two entities prior to attempting to encrypt and decrypt. Whilst public key schemes are less efficient but allow for keys to be established mathematically between the two entities during transmission rather than having the keys pre-agreed upon. They both face the same problem of being vulnerable to attacks introduced by quantum computing.

Hash functions are used to further encapsulate ciphertext outputs, enabling prevention of dictionary attacks as well as compromising another component which may use the same input values by adding a salt value. A message authentication code can then be used to check the integrity of transmitted message by comparing the transmitted data with the message authentication code using a mac key when it arrives on the receivers end.

Post Quantum cryptography such as lattice based cryptography provides the foundations to protect against quantum computing power. Lattice cryptography uses vector mathematics to increase the mathematic complexity of encrypting data. Preventing quantum computers from being able to break encrypted cypher text data within polynomial time.

Biometric security provides a way for each user to have their own key that is never altered and is always unique. This key could be their fingerprint, retinal scan, or face. The challenges of biometric security are centred around the accuracy of determining if a secondary scan is the same as the first scan, as a users fingerprint could be off-centre slightly when compared to the original first scan. To fix this fuzzy extractors and secure sketches are used to calculate the difference in noisy data sets with a pre-specified error acceptance range where the user's biometric scan can not match exactly yet still get accepted.

4 Post-Quantum Secure Multi-Factor Authentication Protocol for Multi-Server Architecture

The paper written by Yunhua Wen, Yandong Su, and Wei Li [48], presents a scheme for a multi-server authentication (MSA) protocol. The standard MFA scheme provides a secure form of authenticating a user on a singular server, if a user wants to use multiple servers, they would have to register on each server individually [48]. This paper proposes a solution to this problem by designing a MSA scheme that uses a registration centre between the user and a set of registered servers, allowing the user to register once and then to be able to use the same credentials on all the servers that were registered with the registration centre at the time [48]. If the registration service is offline, the user will still be able to log on to the servers without the need of communicating with the registration centre. The paper then also provides a quantum secure approach that enables this scheme to be secure once quantum computing becomes mainstream by implementing the ML-KEM lattice-based cryptography approach [48].

4.1 Registration Phase

The registration phase is composed of two stages, server registration and user registration, both phases communicating individually with the registration centre. Once registered, the server and the user can mutually authenticate each other, provided that the server was not registered after the user [48]. If a user has registered before a particular server, they will not be able to gain access until they re-register with the registration centre.

4.1.1 Server Registration

During the server registration phase, an initial server identifier $\mathbf{SID_j}$ is generated. This identifier remains the same and does not change. Using the value $\mathbf{1}^\lambda$ a set of Kyber keys $(\mathbf{pk_{S_j}}, \mathbf{sk_{S_j}})$ is created. $\mathbf{pk_{S_j}}$ is the public key of the server, whilst $\mathbf{sk_{S_j}}$ is the private key and is stored by the server.

$$
\begin{aligned}
&SID_j \leftarrow GenID \\
&(pk_{S_j}, sk_{S_j}) \leftarrow Kyber.KeyGen(1^\lambda) \\
&Store \leftarrow sk_{S_j}
\end{aligned}
$$

The public key $\mathbf{pk_{S_j}}$ is then forwarded along with the server identifier $\mathbf{SID_j}$ to the registration centre. Once received, the registration centre selects $\mathbf{k'}$ unique to the value of $\mathbf{SID_j}$ for servers

that register in the future [48]. Following this, the value of $\mathbf{PSR_j} = \mathbf{h}(\mathbf{SID_j} \parallel \mathbf{K})$ is calculated using a hash of the server identification value and the registration centre private key $\mathbf{K}$. The public key of the server and its identifier is then publicly published $\{\mathbf{SID_j}, \mathbf{pk_{S_j}}\}$ and the value of $\mathbf{PSR_j}$ is then stored locally at the registration centre and also forwarded back alongside $\mathbf{SID_j}$ to the initiating server [48].

$$
\begin{aligned}
&k' \leftarrow \text{unique } SID_j \\
&\text{Check } SID_j \\
&PSR_j \leftarrow h(SID_j \parallel K) \\
&\text{Publish } \{SID_j, pk_{S_j}\} \\
&\text{Store } PSR_j
\end{aligned}
$$

Once the values of $\mathbf{SID_j}$ and $\mathbf{PSR_j}$ are sent back to the initiating server, the value of the known $\mathbf{SID_j}$ is compared with the value sent back by the registration centre. If it matches, the server stores the value $\mathbf{PSR_j}$. The server is then considered registered [48].

$$
\begin{aligned}
&\text{Compare } SID_j \\
&\text{Store } \leftarrow PSR_j
\end{aligned}
$$

4.1.2 User Registration

During the user registration phase, the user is asked to select a user identifier $\mathbf{UID_i}$ and a password $\mathbf{PW_i}$ [48]. Once the user inputs these values, they are asked to provide a biometric scan of their fingerprint $\mathbf{BIO_i}$. Following this a fuzzy extractor generation algorithm is applied to biometric data $(\mathbf{R_i}, \mathbf{P_i}) \leftarrow \mathbf{Gen}(\mathbf{BIO_i})$, the values $(\mathbf{R_i}, \mathbf{P_i})$ represent a reproducible value $\mathbf{R_i}$ and a helper value $\mathbf{P_i}$ [48]. This allows slightly different biometric scans to be reconstructed. Following this a random nonce value α is generated and is hashed with $\mathbf{UID_i}$ and $\mathbf{PW_i}$ to create the value $\mathbf{PWU_i}$. The $\mathbf{UID_i}$ is also hashed with the value $\mathbf{R_i}$ to create $\mathbf{AID_i}$. Finally, before communicating with the registration centre, the value $\mathbf{A_i}$ is generated by applying an exclusive or between $\mathbf{AID_i}$ and the nonce value α [48].

$$
\begin{aligned}
&\text{Select } UID_i, PW_i \\
&\text{Scan } BIO_i \\
&(R_i, P_i) \leftarrow Gen(BIO_i) \\
&\alpha \leftarrow\$ \{0,1\}^{\lambda} \\
&PWU_i \leftarrow h(UID_i \parallel PW_i \parallel \alpha) \\
&AID_i \leftarrow h(UID_i \parallel R_i) \\
&A_i \leftarrow AID_i \oplus \alpha
\end{aligned}
$$

The values $\{\mathbf{AID_i}, \mathbf{PWU_i}\}$ are sent to the registration centre [48]. A check is applied to determine the uniqueness of the value $\mathbf{AID_i}$. Provided the value is unique, it is then stored by the registration centre [48], otherwise it is rejected and the protocol ends at this stage. Following the successful check the values $\mathbf{V_{ij}}$ and $\mathbf{P_{ij}}$ are generated for each server that is registered. The value for $\mathbf{V_{ij}}$ is calculated as $\mathbf{V_{ij}} \leftarrow \mathbf{h}(\mathbf{AID_I} \parallel \mathbf{PSR_j}) \oplus \mathbf{PWU_i}$ and the value of $\mathbf{P_{ij}}$ is calculated as $\mathbf{P_{ij}} \leftarrow \mathbf{h}(\mathbf{SID_j} \parallel \mathbf{PSR_j}) \oplus \mathbf{PWU_i}$ These values are then stored within a smart card $\mathbf{SC_i}$ with there specific server identifier $\mathbf{SID_j}$ [48].

Check uniqueness of AID_i
For $1 \leq j \leq k + k'$, compute
$V_{ij} \leftarrow h(AID_j \parallel PSR_j) \oplus PWU_i$
$P_{ij} \leftarrow h(SID_j \parallel PSR_j) \oplus PWU_i$
Store (SID_j, V_{ij}, P_{ij}) in smart card SC_i

The smart card $\mathbf{SC_i}$ is then sent back to the user and they store the following values inside the smart card $\mathbf{P_i}, \mathbf{PWU_i}, \mathbf{A_i}$. Where $\mathbf{SC_i} \leftarrow \{\mathbf{P_i}, \mathbf{PWU_i}, \mathbf{A_i}, \{(\mathbf{SID_j}, \mathbf{V_{ij}}, \mathbf{P_{ij}}) | \mathbf{1} \leq \mathbf{j} \leq \mathbf{k} + \mathbf{k'}\}\}$ [48]. Once the user has successfully stored the smart card, the user registration is completed [48].

4.2 Login and Mutual Authentication Phase

When a user wants to log on to a server, they locate the $\mathbf{SID_j}$ of the server of their choosing. Once they know this, the process can begin, the user will insert their smart card data $\mathbf{SC_i}$ and input their personal login credentials $\{\mathbf{UID'_i}, \mathbf{PW'_i}, \mathbf{BIO'_i}\}$. These values are presumed to possibly be different from the values used during registration, and if this is the case, the user is refused access to the server. This is checked further along the line of the first communication, the next step in the process computes the value of $\mathbf{R'_i} = \mathbf{Rep}(\mathbf{BIO'_i}, \mathbf{P_i})$, which is the reproduced value based on the biometric input $\mathbf{BIO'_i}$ along with the helper value $\mathbf{P_i}$ stored in the smart card $\mathbf{SC_i}$, these are passed into the fuzzy extractor function $\mathbf{Rep}$ and the reconstruction of $\mathbf{R_i}$ is provided as $\mathbf{R'_i}$. This allows for the presumption that the values inputted might be incorrect. The value of $\mathbf{AID_i}$ is then recreated using the data provided as $\mathbf{AID'_i} = \mathbf{h}(\mathbf{UID'_i} \parallel \mathbf{R'_i})$. This value is then paired with $\mathbf{A_i}$ to create the value $\mathbf{a'}$. Following this $\mathbf{PWU_i}$ is reproduced as $\mathbf{PWU'_i} = (\mathbf{UID'_i} \parallel \mathbf{PW'_i} \parallel \mathbf{a'})$, these values are then compared, and if they do not match, the session is terminated [48].

If the two values match, then the process continues by computing the encapsulation values of the public key of the server, $(\mathbf{c_1}, \mathbf{K_1}) \leftarrow \mathbf{Kyber.Encaps}(\mathbf{pk_{S_j}})$ and the public and private key pair for the user $(\mathbf{pk_{U_i}}, \mathbf{sk_{U_i}}) \leftarrow \mathbf{Kyber.KeyGen}(\mathbf{1}^\lambda)$. At this stage, the protocol recomputes the value $\mathbf{AID'_i}$ and then starts the process of creating a set of messages that the server can recompute to enable mutual authentication. These messages share a mathematical connection between the user's values and the servers $\mathbf{V_{ij}}, \mathbf{P_{ij}}$ values. In addition, it includes a timestamp to prevent replay attacks from being possible [48].

$$M_1 \leftarrow PWU'_i \oplus V_{ij}$$
$$M_2 \leftarrow PWU'_i \oplus P_{ij}$$
$$M_3 \leftarrow AID'_i \oplus M_2 \oplus K_1$$
$$M_4 \leftarrow h(M_1 \parallel K_1 \parallel T_1)$$

Once all four messages have been produced, a combined message is sent to the server as $\mathbf{Msg1} = \{\mathbf{c1}, \mathbf{M_3}, \mathbf{M_4}, \mathbf{T_1}, \mathbf{pk_{U_i}}\}$ [48]. This is sent over a public channel and recovered by the server. Once received by the server, the timestamp is checked and if it is within a set threshold $\mathbf{\Delta T}$ the process continues by computing the decapsulation of $\mathbf{K'_1} = \mathbf{Kyber.Decaps}(\mathbf{sk_{S_j}}, \mathbf{c1})$, which provides the server version of the $\mathbf{K_1}$ value generated by the user. The next set of messages is generated that allows the value of $\mathbf{M_4}$ to equate to the value of $\mathbf{M_8}$, if this holds, then the algorithm continues otherwise the communication between user and server ends [48].

Copyright 2026 © Aran Simon Jannson

$$M_5 \leftarrow h(SID_j \parallel PSR_j)$$
$$M_6 \leftarrow M_5 \oplus M_3 \oplus K_1'$$
$$M_7 \leftarrow h(M_6 \parallel PSR_j)$$
$$M_8 \leftarrow h(M_7 \parallel K_1' \parallel T_1)$$

If $\mathbf{M4} = \mathbf{M8}$ then a new set of encapsulated keys are generated using the user's public key, $(\mathbf{c_2}, \mathbf{K_2}) \leftarrow \mathbf{Kyber.Encaps}(\mathbf{pk_{U_i}})$. The value $\mathbf{K_2}$ is then used to create the session key $\mathbf{SK_{ij}}$ by equating $\mathbf{SK_{ij}} = \mathbf{h}(\mathbf{M_5} \parallel \mathbf{M_7} \parallel \mathbf{K_1'} \parallel \mathbf{K_2} \parallel \mathbf{T_2})$, including messages $\mathbf{M_5}, \mathbf{M_7}$ the encapsulation and decapsulation value $\mathbf{K_2}, \mathbf{K_1'}$ and a new time stamp $\mathbf{T_2}$. The session key is then combined as follows $\mathbf{M_9} = \mathbf{h}(\mathbf{SK_{ij}} \parallel \mathbf{M_7} \parallel \mathbf{T_2})$ to create the new message value $\mathbf{M_9}$ [48].

In the next stage, the message $\mathbf{Msg2} = \{\mathbf{M_9}, \mathbf{T_2}, \mathbf{c_2}\}$ is sent from the server back to the user [48]. Once again, the timestamp is checked to see if it is within the time frame $\mathbf{\Delta T}$ and if it is, then the decapsulation of $\mathbf{K_2}$ is generated as $\mathbf{K_2'} = \mathbf{Kyber.Decaps}(\mathbf{sk_{U_i}}, \mathbf{c1})$ and the user generates the value $\mathbf{SK_{ij}'} = \mathbf{h}(\mathbf{M_2} \parallel \mathbf{M_1} \parallel \mathbf{K_1} \parallel \mathbf{K_2'} \parallel \mathbf{T_2})$ which at the end of the login algorithm can prove $\mathbf{SK_{ij}'} = \mathbf{SK_{ij}}$ will be the session key shared between the user and the server. Once the session key prime is established, the user can then compute the message $\mathbf{M_{10}} = \mathbf{h}(\mathbf{SK_{ij}'} \parallel \mathbf{M_1} \parallel \mathbf{T_2})$ that includes the session key prime, the initial message block, and a new time stamp $\mathbf{T_3}$. This is then compared with $\mathbf{M_{10}} = \mathbf{M_9}$ and if the two message blocks match, then a final message block $\mathbf{M_{11}} = h(SK_{ij}' \parallel M_1 \parallel K_2' \parallel T_3)$ is created and sent to the server as a final transmission $\mathbf{Msg_3} = \{\mathbf{M_{11}}, \mathbf{T_3}\}$ [48] to confirm that the session keys on both sides match. The servers final job will be to insure that this final transmission is accepted.

The final step in the login phase is the server validating that the provided message block $\mathbf{M_{11}}$ equates to the same value of $\mathbf{M_{12}} = \mathbf{h}(\mathbf{SK_{ij}} \parallel \mathbf{M_7} \parallel \mathbf{K_2} \parallel \mathbf{T_3})$, as well as that the timestamp $\mathbf{T_3} \leq \mathbf{\Delta T}$. If these final steps are completed successfully, then the session key $\mathbf{SK_{ij}}$ is established and the user and the server can communicate securely between each other [48].

4.3 Password Updating

To update a user's password, they must first pass the authentication phase as described in the login section. If this stage is successfully completed, the user can select a new password $\mathbf{PWU_i^{new}}$ and compute their new set of authentication keys for each server on the network. Once completed, an updated smart card is issued to the user [48].

$$PWU_i^{\text{new}} \leftarrow h(UID_i \parallel PW_i^{\text{new}} \parallel a')$$
For $1 \leq j \leq k + k'$, compute
$$V_{ij}^{\text{new}} \leftarrow V_{ij} \oplus PWU_i \oplus PWU_i^{\text{new}}$$
$$P_{ij}^{\text{new}} \leftarrow P_{ij} \oplus PWU_i \oplus PWU_i^{\text{new}}$$
$$SC_i \leftarrow \left\{PWU_i^{\text{new}}, P_i, A_i, \left\{SID_j, V_{ij}^{\text{new}}, P_{ij}^{\text{new}} \mid 1 \leq j \leq k + k'\right\}\right\}$$

4.4 Smart Card Revocation

If a smart card becomes compromised or needs to be renewed for any other reason, a smart card revocation function is provided in the paper. To do this, the user would insert their old smart card $\mathbf{SC_i}$ and then select new login credentials $\mathbf{UID_i^{new}}, \mathbf{PWU_i^{new}}, \mathbf{a^{new}}$ and provide a new biometric reading $\mathbf{BIO_i'}$. The biometric value generated previously is then reproduced using the fuzzy extractor [48]:

$$R_i' \leftarrow \text{Rep}(BIO_i', P_i)$$

Using the reproduced value $\mathbf{R_i'}$ recreates their user profile and generates the new values that will be included in the new smart card $\mathbf{SC_i^{new}}$ [48]:

$$
\begin{aligned}
&AID_i' \leftarrow h(UID_i' \parallel R_i') \\
&PWU_i^{\text{new}} \leftarrow h(UID_i^{\text{new}} \parallel PW_i^{\text{new}} \parallel a^{\text{new}}) \\
&AID_i^{\text{new}} \leftarrow h(UID_i^{\text{new}} \parallel R_i') \\
&A_i^{\text{new}} \leftarrow AID_i^{\text{new}} \oplus a^{\text{new}}
\end{aligned}
$$

The revocation request is then sent to the registration centre along with the newly generated credential values, and the newly registered server authentication variables are generated [48]:

for $1 \leq j \leq k + k'$ **do**
$V_{ij}^{\text{new}} \leftarrow h(AID_i^{\text{new}} \parallel PSR_j) \oplus PWU_i^{\text{new}}$
$P_{ij}^{\text{new}} \leftarrow h(SID_j \parallel PSR_j) \oplus PWU_i^{\text{new}}$
endfor

Once these values have been generated, they can be added to the new smart card $\mathbf{SC_i^{new}}$, and the smart card can then be issued to the user [48].

4.5 RoR Security Model

The Real-or-Random (ROR) security model provides a testing style in which an attacker has the best case of fifty percent chance of detecting whether they are using real or random test data. The attacker is given information from an oracle that allows them to know how the protocol functions. When an attack is attempted either a real value or a random value is used to try and break the protocol.

If the attacker cannot figure out whether the value being used is real or random, it has a probability of guessing the value is fifty percent and has a low advantage even with the information provided to it by the oracle.

4.6 Critical Evaluation

The paper claims a post quantum secure multi server authentication scheme that is secure and resistant to attacks. The scheme covers multiple attack surfaces according to its claims and provides a way to invalidate or update credentials as needed. However, whilst examining the paper, a few flaws were discovered, particularly within the login function. Furthermore, a claim is made that the security of the protocol was tested using the Dolev-Yao model, however, was also tested using the RoR model. This is contradictory, as the Dolev-Yao model presumes perfect cryptography. The provided proofs use a cryptographic adversary, however, present the results within a Dolev-Yao claim.

Notation Inconsistencies

During the final stages of the login process, it can be seen that the scheme recomputes the value $\mathbf{K_2}$ as $\mathbf{K_2'}$. The formula used for this is $\mathbf{K_2' = Kyber.Decaps(sk_{U_i}, c_1)}$ [48], however, during the creation of the initial encapsulation that generated the key, it was formatted as $\mathbf{(c_2, K_2) = Kyber.Encaps(pk_{U_i})}$. This then leads to the observation that, during the decapsulation stage the value $\mathbf{c_1}$ should not be used, and rather the value $\mathbf{c_2}$ should take its place. Otherwise, the value $\mathbf{K_2 \neq K_2'}$, if the original formulation is used.

Another notational inconsistency can also be seen in the first stages of the login process, where the user calculates $\mathbf{AID'_i}$ twice [48]. The value is calculated immediately after the biometric input is processed and then again once $\mathbf{PWU_i} = \mathbf{PWU'_i}$ is validated after the ML-KEM key generation function. These two values are mathematically identical and it can be seen that values such as $\mathbf{PWU_i}$ are reused after being initialised before the accuracy check leading to question the double initialisation of $\mathbf{AID_i}$.

4.7 Conclusion

In conclusion, the proposed scheme enables a strong fundamental post quantum secure multi-server authentication system. It takes advantage of standard user credentials as well as multi-factor biometric data using a fuzzy extractor to securely process this. The usage of ML-KEM allows the scheme to be quantum secure, enabling future security against undiscovered attacks. The inclusion of a revocation and password updating feature enables further reusability for when a user wants to update their credentials or if a smart card has been compromised [48].

However, whilst analysing the scheme, a few inaccurate notations were discovered. These were discovered within the login phase of the system in which the value of $\mathbf{K'_2}$ [48] was incorrectly calculated and the value of $\mathbf{AID_i}$ was initialised twice [48].

5 Design, Implementation, & Hardware Choices

5.1 General Design Choices

During the development of this project, the core articles [47, 48], were followed and translated into functional Python scripts, and the formal security was tested using ProVerif. To provide a function demo, the developed scripts where embedded into a FastAPI script to allow for API calls to be made to the developed scripts. Python was chosen due to its rich libraries for cryptographic operations.

Library	Description
kyber_py	Provided a CBD function as well as the ML-KEM PQC schemes
galois	Provided the required BCH functionality for the Secure Sketch
hmac	Provided the Message Authentication Code functionality
pyfingerprint	Allowed the python scripts to communicate with the fingerprint scanner
hashlib	Provided hash functionality
secrets	Provided secure random values
numpy	Provides mathematical operations

Table 4: Core Python Libraries

For a user to register with the system, they are prompted with a sign-up page, at which stage they provide their unique username and secure password. Once they have provided the required inputs, the user will select the sign-up button at which time they will be prompted to scan their finger on the fingerprint scanner. Once a successful scan is taken, a "smart-card" style

JSON output is provided for the user to use when they log in to a server, paired along side their username and password.

5.2 Protocol Design

5.2.1 Server Registration

When setting up the MSA system, each server in the network must be registered with the RC. For the server to be considered registered, an identifier value $\mathbf{SID_j}$ is selected and a set of ML-KEM keys $(\mathbf{pk_{S_j}}, \mathbf{sk_{S_j}}) \leftarrow \mathbf{Kyber.KeyGen}(\mathbf{1}^\lambda)$ are generated.

The server saves the secret key $\mathbf{sk_{S_j}}$ locally and transmits the identifier and the public key to the registration centre.

```
def serverRegister(SID_j):
    pk, sk = ky.ML_KEM_768.keygen()
    return (pk, sk), SID_j
```

Once the RC receives the identifier and public key, the RC checks whether the identifier value $\mathbf{SID_j}$ is unique and only if it is will it continue to the next steps. Provided the check passes the RC will generate the value $\mathbf{PSR_j} = \mathbf{h}(\mathbf{SID_j} \parallel \mathbf{K})$, hashing the server identifier and the RC value $\mathbf{K}$. The RC then publishes the values $\{\mathbf{SID_j}, \mathbf{pk_{S_j}}\}$, allowing users to register on the server. The RC finally stores the value $\mathbf{PSR_j}$ locally and sends the message $\{\mathbf{SID_j}, \mathbf{PSR_j}\}$. The server then compares the $\mathbf{SID_j}$ returned with the one generated at the start and, provided they match, the server saves the value $\mathbf{PSR_j}$ locally, marking the server as registered with the RC.

$\mathbf{SID_j}$ Originality Check

```
if server.SID in rc["servers"]:
    print(f"Server exists already.")
    return False
```

Server Registration

```
    (pk, sk), SID_j = serverRegister(SID)
    with open("server.json", "w") as server_jsn:
        server_data["sk"] = sk.hex()
        json.dump(server_data, server_jsn, indent=4)
    async with httpx.AsyncClient() as client:
        response = await client.post(f"{RC_URL}/servers/register",
            json={"SID": SID_j, "pk": pk.hex()},
        )
        response.raise_for_status()
        rc_data = response.json()
    PSR_j = rc_data["PSR"]
    if rc_data["SID"] != SID_j:
        return {"message": "Server A is not registered. SID_j does not match."}
    with open("server.json", "w") as server_jsn:
        server_data["SID"] = SID
        server_data["PSR"] = PSR_j
        json.dump(server_data, server_jsn, indent=4)
```

5.2.2 User Registration

For a user to be registered, at least one server must be linked to the RC. Once at least one server is registered with the RC, the user can request to be registered. The first step requires the user to provide a user identifier and a password $\{\mathbf{UID_i}, \mathbf{PW_i}\}$. Once these are provided, a biometric scan $\mathbf{w}$ is taken and creates the values $(\mathbf{P}, \mathbf{R}) \leftarrow \mathbf{Gen}(\mathbf{pp}, \mathbf{w})$.

User Credential Inputs

```
UIDi = input("Enter UID: ")
PWi = input("Enter PW: ")

w = register()

P, R = gen(pp, w)
```

From this phase, the user does not need to interact with the protocol in any form. A nonce value **a** is generated and hashed with the user identifier $\mathbf{UID_i}$ and password $\mathbf{PW_i}$. In addition, the user identifier is also hashed with the biometric value **R**, these produce the values of $\{\mathbf{PWU_i}, \mathbf{AID_i}\}$. The $\mathbf{AID_i}$ and nonce value **a** are then XORed together to create the value $\mathbf{A_i}$.

Hashed User Credentials

```
nonce_a = secrets.token_bytes(32)

PWUi = hashlib.sha3_256(UIDi.encode() + PWi.encode() + nonce_a).digest()
AIDi = hashlib.sha3_256(UIDi.encode() + R.tobytes()).digest()

Ai = strxor(AIDi, nonce_a)
```

A check is applied to ensure that the user has not registered before by querying if the value $\mathbf{AID_i}$ has been registered before. If the value is unique, the user is then registered with each server connecting to the RC, this is done by using the server identifier and registration values $\{\mathbf{SID_j}, \mathbf{PSR_j}\}$. This generates a pair value of $\{\mathbf{Vi_j}, \mathbf{Pi_j}\}$ for each server and is added to the user's smart card that is being created.

Registering The User With The Servers

```
    for SID_j, PSR in rc["servers"].items():
        PSR_j = bytes.fromhex(PSR)
        Vij = strxor(hashlib.sha3_256(AID_i+PSR_j).digest(), PWU_i)
        Pij = strxor(hashlib.sha3_256(SID_j.encode() + PSR_j).digest(), PWU_i)
        SCi[SID_j] = {"Vij": Vij, "Pij": Pij}

    with open("registration_center.json", "w") as rc_jsn:
        json.dump(rc, rc_jsn, indent=4)
    return SCi
```

Once this has been completed, the user's smart card can be put together, which would include the helper value from the biometric scan and their hashed credentials.

5.2.3 Login

The login algorithm begins by computing the user's credentials locally, prior to any public channel data transmission. This ensures that the user's credentials are never seen by an attacker in their raw form [48]. The authentication of the user begins by requiring a new scan of their fingerprint. This newly captured scan is then compared to a stored helper value and if the difference fits within a predefined hamming distance **t** the user can be authenticated.

```
try:
    R_prime = rep(pp, P, BIO)
except ValueError as error:
    print(f"Fingerprint verification failed: {error}")
    return {"Status": False}
```

Using this reconstructed scan data, further values are able to be recomputed with shared elements that both the user and the server calculated during user registration. A check is also applied to check whether the input password matches that of the one stored on the smart card as a hash string.

```
AID_prime  = hashlib.sha3_256(UID.encode() + R_prime.tobytes()).digest()
a_prime    = strxor(bytes.fromhex(user_details["Ai"]), AID_prime)
PWU_prime  = hashlib.sha3_256(UID.encode() + PW.encode() + a_prime).digest()
```

Once the user is locally authenticated, a ML-KEM key pair is established which enables the first shared value between the user and the server, the user uses the public key to encapsulate and produce a ciphertext $\mathbf{c_1}$ and the shared value of $\mathbf{K_1}$. The authentication values $\mathbf{M_1}$, $\mathbf{M_2}$, $\mathbf{M_3}$, and $\mathbf{M_4}$ are then computed using information recovered from the smart card, the shared value $\mathbf{K_1}$, and locally authenticated user credentials. A timestamp $\mathbf{T_1}$ is also used to prevent replay attacks from being possible. If the timestamp does not match the current time within $\mathbf{\Delta}$ time, the request is automatically rejected by the server.

```
if PWU_stored == PWU_prime:
    k1, c1 = ky.ML_KEM_1024.encaps(pk_Sj)
    pk_ui, sk_ui = ky.ML_KEM_1024.keygen()
    AIDi_prime = hashlib.sha3_256(UID.encode() + R_prime.tobytes()).digest()
    Vij = bytes.fromhex(user_details["SC"][SIDj]["Vij"])
    Pij = bytes.fromhex(user_details["SC"][SIDj]["Pij"])

    T1 = int(time.time())
    M1 = strxor(PWU_prime, Vij)
    M2 = strxor(PWU_prime, Pij)

    M2K1 = strxor(M2, k1)

    M3 = strxor(AIDi_prime, M2K1)
    M4 = hashlib.sha3_256(M1 + k1 +  str(T1).encode()).digest()

    Msg1 = {"c1": c1, "M3": M3, "M4": M4, "T1": T1, "pk_ui": pk_ui}
```

Once $\mathbf{Msg_1}$ is sent to the server it is verified by the server by recalculating the value of the message block $\mathbf{M_4}$. This is done by the server calculating its own pair of pre shared values with the user. A check is also applied to ensure the message is within $\mathbf{\Delta}$ time.

```
if abs(int(time.time())-T1) < 10:
    k1_prime = ky.ML_KEM_1024.decaps(sk_sj,c1)
    M5 = hashlib.sha3_256(SID_j.encode() + PSRj).digest()
    M6 = strxor(M5, strxor(M3,k1_prime))
    M7 = hashlib.sha3_256(M6+PSRj).digest()
    M8 = hashlib.sha3_256(M7 + k1_prime + str(T1).encode()).digest()
```

If $\mathbf{M_4} = \mathbf{M_8}$ then the server starts to compute the session key that will connect the user and the server. A new set of encapsulated are generated by the server using the users public key, along side a new time stamp for the user to verify. The session key is implemented into a new message block that is sent to the user to recompute. A new sessions variable is created to emulate the server storing the values processed in this stage for later use.

```
if M8 == M4:
    k2,c2 = ky.ML_KEM_1024.encaps(pk_ui)
    T2 = int(time.time())
    SK_ij= hashlib.sha3_256(M5 + M7 + k1_prime + k2 + str(T2).encode()).digest()
    M9 = hashlib.sha3_256(SK_ij + M7 + str(T2).encode()).digest()

    session = {"SKij": SK_ij, "M7": M7, "k2": k2, "macKeyPrime": macKey_prime}
```

```
    Msg2 = {"M9": M9, "T2": T2, "c2": c2}

    return Msg2, session
```

Once the user receives $\mathbf{Msg_2}$, they extract the contents and create the session key $\mathbf{SK'_{ij}}$ which should equate to the same value generated by the server in the previous communication stage.

```
M9 = bytes.fromhex(Msg2["M9"])
T2 = int(Msg2["T2"])
c2 = bytes.fromhex(Msg2["c2"])

K2_prime = ky.ML_KEM_1024.decaps(sk_ui, c2)
SK_prime_ij = hashlib.sha3_256(M2 + M1 + k1 + K2_prime + str(T2).encode()).digest()
M10 = hashlib.sha3_256(SK_prime_ij + M1 + str(T2).encode()).digest()
T3 = int(time.time())

if M10 == M9:
    M11 = hashlib.sha3_256(SK_prime_ij + M1 + K2_prime + str(T3).encode()).digest()
else:
    print("M10 does not match M9")
```

The value of $\mathbf{M_{11}}$ and a new timestamp is sent to the server. This allows the server to confirm whether or not the derived session key $\mathbf{SK_ij' = SK_{ij}}$.

```
Msg3 = {"M11": M11, "T3": T3}

M11 = msg3["M11"]
T3 = msg3["T3"]

if abs(int(time.time()) - T3) < 10:
    M12 = hashlib.sha3_256(session["SKij"] + session["M7"] + session["k2"] + str(T3).encode()).digest()
    if M11 == M12:
        return True
    return False
```

5.3 Verification Tools

To verify the current and improved security of the scheme, Scyther was used at first to model the algorithm and check any generated attack traces, specifically to show evidence of the DoS attack and its fix. However, after attempting to model the PQMFA Login algorithm [48], it was safe to conclude that the tool was not capable of providing the required functionality effectively, this was specifically noticeable when trying to implement an XOR operation as scyther is unable to compute algebraic expressions. Following this discovery, a switch was made from Scyther to ProVerif. Following this change, the PQMFA Login scheme was more realistically and accurately modelled, providing evidence of the DoS attack, and proving the solution suggested by this paper was successful.

5.4 Biometrics

To implement the biometric requirements of the paper, a physical fingerprint scanner was chosen instead of simulated biometric values to allow the fuzzy extractor to be properly tested to the highest possible standard. Providing accurate examples rather than predetermined simulated data with slight variation.

The core papers choice for biometric data was fingerprint scans [47, 48], to enable the use of the fuzzy extractor the fingerprint scanner chosen for this project was the **Joy-It R301T Fingerprint sensor**. This specific scanner was chosen due to its ability to directly interact

with the scripts developed using the Python library **pyfingerprint**. The library allowed for direct communication between hardware and software with predefined examples to demonstrate the functionality of the library [4, 11].

The script used to read the user's biometric fingerprint data is based on the example scripts for **pyfingerprint** [9] with modifications made to allow the data to be processed.

```
def register():
    try:
        f = PyFingerprint("/dev/ttyUSB0", 57600, 0xFFFFFFFF, 0x00000000)
    except Exception as exc:
        print("Error initialising scanner")
        return False
    try:
        print("Scan your finger")
        while not f.readImage():
            pass
        print("Scanned")
        f.convertImage()
        characteristics = f.downloadCharacteristics()

        return characteristics
    except:
        print("Error reading image")
        return False
```

The hardware of the fingerprint scanner functions by providing an open port for the library to communicate with. The scanner can provide a scanned image of a fingerprint that the sensor initially stores within its internal image buffer. This image is then used within the script developed to provide a computable output for the fuzzy extractor to process [25].

In addition, instead of using the built in fingerprint matching algorithm, the one specified in the main papers [47, 48] was implemented. This was done to more accurately express the expected results from the completed system. In addition, this allowed for repeated fingerprint readings to be taken by the Secure Sketch and fuzzy extractor algorithms, instead of relying on the fingerprint scanner's built in algorithm to provide an authentication decision.

5.5 Secure Sketch

Due to the nature of biometric fingerprint scans varying each time they are created, a secure sketch function is required to enable a functional fuzzy extractor. The paper implements a Bose–Chaudhuri–Hocquenghem (BCH) secure sketch algorithm written for MATLAB [1]. The scheme is rewritten in Python with the necessary changes to allow it to function with the available libraries.

To implement the two functions, a BCH error correction code was selected based on the requirements of the papers [1, 47, 48]. The BCH code was implemented using the **galois** Python library which provided the functionality [24]. Using this, a preset error tolerance value was set to detect and correct a specified amount of bit errors, making the concept perfect for detecting if a fingerprint scan is the same as the one it is being compared against.

$$\mathrm{BCH}(1023, 828, d = 41)$$

The example BCH values represented as $\mathbf{BCH(n, k, d)}$, represents the length of the encoded codeword $\mathbf{n}$, the number of data bits $\mathbf{k}$ and the minimum required distance of value $\mathbf{d}$. This allows a correction amount of $\mathbf{t = 20}$ to be applied to a fingerprint scan passed through the secure sketch algorithm, and any error bits within the tolerance are corrected.

Generation Function

```
def secure_sketch_gen(w):
    w = np.asarray(w).flatten()
    w_pad = []

    for i in range(len(w)):
        if w[i] > 128:
            w_pad.append(1)
        else:
            w_pad.append(0)
    w_pad = np.pad(w_pad, (0, 1023 - len(w_pad)))
    bch = galois.BCH(n,k,d)
    GF2 = bch.field

    xArr = []
    for _ in range(n):
        xArr.append(secrets.randbits(1))

    x = np.asarray(xArr, dtype=np.uint8)
    R = np.bitwise_xor(x,w_pad)

    RNG1Arr=[]
    for _ in range(k):
        RNG1Arr.append(secrets.randbits(1))
    RNG1 = np.asarray(RNG1Arr, dtype=np.uint8)

    codeword = bch.encode(GF2(RNG1))
    code = np.asarray(codeword, dtype=np.uint8)
    s = np.bitwise_xor(w_pad,code)

    return s,x,R
```

Reproduction Function

```
def secure_sketch_rep(w, s, x):
w = np.asarray(w).flatten()
w_pad = []
for i in range(len(w)):
    if w[i] > 128:
        w_pad.append(1)
    else:
        w_pad.append(0)
w_pad = np.pad(w_pad, (0, 1023 - len(w_pad)))
bch = galois.BCH(n, k, d=d)
GF2 = bch.field
r1 = np.bitwise_xor(w_pad,s)
r2 = bch.decode(GF2(r1))
r3 = bch.encode(r2)
wClean = np.bitwise_xor(r3, s)
R = np.bitwise_xor(wClean, x)

return R
```

5.5.1 Pre Processing

The input fingerprint scanner data is converted into a one-dimensional array. Each value is then converted into a single binary value, and is done by applying a rounding function. If the value is greater than 128 it is rounded up to a positive bit, otherwise it is rounded down to a negative bit value:

```
    for i in range(len(w)):
        if w[i] > 128:
            w_pad.append(1)
        else:
            w_pad.append(0)
```

The values are turned into a single dimensional bit array to allow it to fit within the required dimensional variance of the BCH algorithm. The **galois** BCH algorithm only accepts a binary bitstring input.

To calculate the value of **d** or re-calculate the value of **t** the following formulas are applied:

$$d = 2t + 1 = 41$$

$$t = \left\lfloor \frac{d-1}{2} \right\rfloor = 20$$

5.5.2 Gen

The generation algorithm uses the input fingerprint scan **w** and converts the input into a one dimensional array of bits as defined above. If the scan is shorter than the required length of the BCH function, padding is applied to this end to match this length.

The BCH algorithm is applied to the value of **k** and a code word **c** is generated. An XOR operation is then applied between this value and the value $\bar{\mathbf{w}}$. The output from this creates the helper value **s**.

```
RNG1Arr = []
for _ in range(k):
    RNG1Arr.append(secrets.randbits(1))

RNG1 = np.asarray(RNG1Arr, dtype=np.uint8)

codeword = bch.encode(GF2(RNG1))
code = np.asarray(codeword, dtype=np.uint8)
s = np.bitwise_xor(w_pad, code)
```

A separate random value $x \in \{0,1\}^{1023}$ is also generated and XORed with the previously processed fingerprint value $\bar{\mathbf{w}}$ to produce the reproducible output R:

```
R = np.bitwise_xor(x, w_pad)
```

Once these values are generated the values are returned to the user as $\{\mathbf{s}, \mathbf{x}, \mathbf{R}\}$ and can be used to reconstruct the fingerprint from a new scan in the recovery algorithm.

5.5.3 Rec

The recovery algorithm takes a new fingerprint scan input of $\mathbf{w}'$, along with the helper values **s** and **x** which was generated when the initial generation function were executed. Similarly to the **Gen** function the input biometric scan is converted into a one-dimensional bitstring, with any necessary padding applied, producing the value $\bar{\mathbf{w}}'$.

The value $\bar{\mathbf{w}}'$ is XORed with the helper value s:

```
w_pad = np.pad(w_pad, (0, n - len(w_pad)))

bch = galois.BCH(n, k, d=d)
GF2 = bch.field

r1 = np.bitwise_xor(w_pad, s)
r2 = bch.decode(GF2(r1))
r3 = bch.encode(r2)
```

Due to the helper value being generated based on calculation $s = \bar{w} \oplus c$, the value of $\mathbf{r_1}$ can be represented as:

$$r_1 \leftarrow c \oplus (\bar{w} \oplus \bar{w}')$$

The difference between $\bar{w}$ and $\bar{w}'$ is calculated as the remaining error between the BCH codeword value $\mathbf{c}$. This allows for the BCH decoder function to be applied to correct the errors to the hamming distance of $\mathbf{t}$. If the error length fits within the value $\mathbf{t}$, the original encoded value can be recovered.

If the error correction applied to the value was successful then the recovered value $\bar{\mathbf{w}}$ can be XORed with the helper value $\mathbf{x}$ to produce a recovered value $\mathbf{R}'$. If the fingerprint data $\mathbf{w}'$ was within error range of $\mathbf{t}$ then $\mathbf{R}' = \mathbf{R}$, confirming that the original fingerprint scan was recovered successfully.

5.6 Fuzzy Extractor

The fuzzy extractor implantation follows the protocol described in Section 2.3, the cryptographic implementation is converted into a practical Python implementation. The script calls the secure sketch script and processes the provided scan accordingly.

5.6.1 Gen

FE starts by generating the values $\mathbf{s}, \mathbf{x}, \mathbf{R}$ using the secure sketch **Gen** function. A biometric input $\mathbf{w}$. Following this, the values **SS_key** and **SS_helper** are initialised using the data provided by the secure sketch. The hashing algorithm to be used by the FE is also collected from the value set **pp** and initialised.

```
s,x,R = ss.secure_sketch_gen(w)
SS_key = numpy.asarray(R, dtype=numpy.uint8).tobytes()
SS_helper = {"s": numpy.asarray(s, dtype=numpy.uint8).flatten(), "x": numpy.asarray(x,
↪  dtype=numpy.uint8).flatten(), "k": ss_k}

hashType, hashFunc = pp["H_i"]
hashDigest = int.from_bytes(hashFunc(SS_key).digest(), byteorder="big")
```

The matrix $\mathbf{S}$ is then calculated using the lengths $\mathbf{n}$ and $\mathbf{l}$. This is done to comply with the space $\mathbb{Z}_q^{n \times l}$. To be able to process the data it is then converted into a numpy array. Similarly, the values $\mathbf{A}$ is generated to fit within $\mathbb{Z}_q^{m \times n}$. These operations are performed according to the required dimensions and size of the fingerprint scan.

```
S = []
for i in range(n):
    row = []
    for j in range(l):
        row.append(hashDigest % q)
    S.append(row)
S = numpy.array(S)

A= []
for _ in range(m):
    row = []
    for _ in range(n):
        row.append(secrets.randbelow(q))
    A.append(row)
```

The value of **B** is generated using a CBD of a secret value of size 64 by 3 along with the previously created values of **S** and **A**. The value is an element of the space $\mathbb{Z}_q^{m\times(n+l)}$.

```
E_cbd = PolynomialRing().cbd(secrets.token_bytes(64 * 3), 3)
E = numpy.array(E_cbd.coeffs[:m * l], dtype=int).reshape(m, l)

A_array,S_array,E_array = numpy.array(A), numpy.array(S), numpy.array(E)

B = ((A_array @ S_array)  + E_array) % q
B_concat = numpy.concatenate([A_array, B], axis=1)
```

Following the generation of the matrixes and keys, the FE encrypts a random value. This is done by sampling a bitstring of length **l** and **m**. Leading to the generation of the cyphertext value **c**. This is then paired with the helper value generated earlier and is provided as the output of the function as a pair.

```
x_bits, m_bits= [], []

for _ in range(m):
    x_bits.append(secrets.randbits(1))
for _ in range(l):
    m_bits.append(secrets.randbits(1))
x_bitstream, m_bitstream = numpy.array(x_bits), numpy.array(m_bits)

x_TB = numpy.transpose(x_bitstream) @ B_concat
zero_T = numpy.zeros(n, dtype=int)
m_T = m_bitstream * int((numpy.round(q / 2)))
q_rounded = numpy.concatenate([zero_T, m_T])
c_T = (x_TB + q_rounded) % q

P = {"SS_helper": SS_helper, "c": c_T}

return P, m_bitstream
```

5.6.2 Rep

The reproduction function recreates the original biometric scan taken by the generation algorithm, this is done using a fresh scan of a fingerprint biometric data. The function takes the input of the helper value and cypher text created by the generation algorithm along side the same hash type.

```
SS_helper = P["SS_helper"]
c = P["c"]
s = numpy.asarray(SS_helper["s"], dtype=numpy.uint8).flatten()
x = numpy.asarray(SS_helper["x"], dtype=numpy.uint8).flatten()
ss_k = SS_helper["k"]
```

Similarly to the generation function it calls the secure sketch, however, this time it uses the **Rec** function. The script provides the function with the newly taken biometric scan data as well as the two helper values $\mathbf{s}, \mathbf{x}$. If the recovery function does not provide an output, the biometric scan provided was too different from the original scan used to create the helper values.

```
R = ss.secure_sketch_rep(w_prime, s, x)

if R is None:
    print("w_prime is too different from the w")
    return False
```

If the value **R** is returned successfully a key **SS_key** is created using the value. Following this the matrix **S** is computed similarly to the generation algorithm, following same dimensions as before.

```
SS_key = numpy.asarray(R, dtype=numpy.uint8).tobytes()
S=[]
for i in range(n):
    row = []
    for j in range(l):
        row.append(hashDigest % q)
    S.append(row)
S= numpy.array(S)
```

Finally, the cyphertext provided at the start is decrypted using the inverse value of **S** and the identity matrix $\mathbf{I_l}$. Each point in the array is then decoded to recover the original message. In return, confirming whether the scan is authentic and the same biometric as the original input. A preview of a fingerprint scan being collected using the Joy-It Fingerprint scanner can be seen in Figure 7.

```
d = (c @ numpy.vstack([(-S), numpy.eye(l, dtype=int)])) % q

m = []
for i in range(l):
    if q/4 <= d[i] <= (3*q)/4:
        m.append(1)
    else:
        m.append(0)
return numpy.array(m)
```

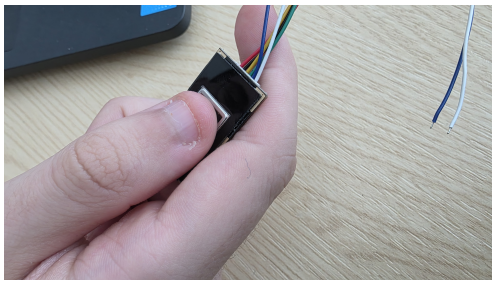


Figure 7: Fingerprint Collection Using Joy-It Fingerprint Scanner

5.7 Smart Card Emulation

The original scheme presented by the PQMFA core paper [48] requires each registered user to have their own smart card. This smart card would include necessary information that is used to authenticate and log the user into their selected server. The smart card stores $\mathbf{PWU_i}$, the fuzzy extractor helper data $\mathbf{P_i}$, the value $\mathbf{A_i}$, and server-specific authentication values ($\mathbf{SID_j}, \mathbf{V_{ij}}, \mathbf{P_{ij}}$) [48].

A decision was made to use a digital representation of a smart card rather than a physical card and reader. The digital smart card was implemented using a JSON file structure which emulated smart card architecture and formatting.

If a user is registered successfully a JSON file representing the smart card is provided to the user. This included the values **PWU** and **Ai** representing the values PWU_i and A_i. Whilst the value **SC** stores the $\mathbf{V_{ij}}$ and $\mathbf{P_{ij}}$ values for each server in an object state. This allows for the single smart card to be verified and log the user in on multiple servers.

The value **P** includes the helper values used by the fuzzy extractor for biometric reconstruction from a secondary scan. It includes the SS values used by the BCH function during recovery

with the ciphertext **c** generated by the fuzzy extractor. The helper values **s** and **x** are also stored as a sub-variable of **P**.

JSON Value	Purpose
PWU	Stores the authenticated value of PWU_i.
Ai	Stores the authentication identity value A_i.
P	Stores the biometric helper data required by the fuzzy extractor, to authenticate the user.
SS_helper.s	Stores the SS helper value s.
SS_helper.x	Stores the random helper value x used during the SS reproduction function.
SS_helper.k	Stores the BCH length used by the SS.
c	Stores the helper component created by the FE.
SC	Stores the authentication information for each registered server.
Vij	Stores the server value V_{ij}.
Pij	Stores the server value P_{ij}.

Table 5: Smart Card Structure

The choice of using JSON for storing this data was decide due to its structured format. This allowed for an easier data extraction by the developed scripts. Further to this, YAML was also considered, however, the libraries used in the project scripts provided an easier implementation base line with JSON in comparison to YAML.

Smart Card Format Example


```
    {
    "a6c7b2abcbaa4667f175909e45d5e88c3b3a6b5b4fcb52f3d0bd3a1c090e982f": {
        "P": {
            "SS_helper": {
                "s": [
                   0,1,1,0,1,0,1,1,1,0,0
                ],
                "x": [
                    0,1,1,0,1,0,1,1,1,0,0
                ],
                "k": 256
            },
            "c": [
               523, 234, 876
            ]
        },
        "PWU": "1b01384ad97eb3747db19c450eb44d51",
        "Ai": "30b6e497423579ad7b7130b24e341b8c",
        "SC": {
            "45c80eac-8e5c-423e-92b1-d53a7f5615e7": {
                "Vij": "7b1211234961139b2919b7e5222d17ac",
                "Pij": "08253127493 69e6aa296892a1c05c6d9"
            }
        }
    }
}
```

6 Formal Security Analysis

6.1 Introduction

The security of the authentication protocol has been modelled in ProVerif with the defined adversarial assumptions. This includes the cryptographic logic for the implemented function within the allowed domain provided by ProVerif, security events have also been marked out. This allows for the detection of modified values that were used to prove the existence of a DoS attack on the system. This is done by tracking the value of the message block $\mathbf{M_3}$ through the protocol execution and the original value is then compared with the value available after transmission. In addition, mutual authentication claims are also tested as well as the secrecy of chosen values, to help prove that the protocol succeeds with these claims.

6.2 ProVerif Base Model

Global Declarations

The model requires the use of three channels, one core channel used for communication between the user and the server, a channel to emulate the users memory to reserve reused values, and a channel for the server to reserve calculated values for the duration of the communication. The core channel is a public channel, whilst the remaining two channels are private as they emulate locally stored memory.

```
free c:channel.
free session: channel [private].
free server: channel [private].
```

The first three channels are initialised:

- Channel **c**: The main channel where all communication between the user and the server takes place.
- Channel **session**: This channel is an internal private channel used to store pre-generated values by the user in a previous transmission.
- Channel **server**: Similarly to the **session** channel, this serves the purpose of storing pre-computed values by the server to be used in later communication stage locally.

Following channel generation, required data type representations are created with the values **skey** and **pkey**, representing secret and public keys created during the algorithm. The type **bio** represents a biometric fingerprint scan input from the user.

The core variables then get set, ready to be used during the algorithms functions. The values for **BIOi**, **PWi** are marked as private variables, as the attacker would not have direct access to this data prior to it being used during computation.

```
free UIDi: bitstring.
free PWi:  bitstring [private].
free BIOi: bio [private].
free SIDj: bitstring.
free PSRj: bitstring.
free AIDi: bitstring.
```

Following this, decisions were made on how to apply computation logic for hashes, XOR, ML-KEM Key Generation, encapsulation, and the reproduction function from the fuzzy extractor. These functions are defined below according to the documentation for ProVerif, as well as example implementations present on GitHub [6, 17].

- Function **Rep**: Provides the recomputed value $\mathbf{R}'_i$, taking an input of the type **bio** and a bitstring that represents the helper value and returns an output of type bitstring. It is initialised as follows: `fun Rep(bio, bitstring): bitstring.`
- Function **h**: Represents a hash function, taking an input bitstring and providing an output bitstring and is defined as follows: `fun h(bitstring): bitstring.`
- Function **kemkey**: A public key value and a bitstring value provided allow the ML-KEM key to be generated, separating the key type from others. It is defined as follows: `fun kemkey(pkey, bitstring): bitstring.`
- Function **xor**: A standard XOR function that takes two inputs of bitstrings and provides an output that applies the bitwise operation outputting a singular bitstring value. It is defined as a function and an equation to ensure that the XOR relations apply:

```
fun xor(bitstring, bitstring): bitstring.
equation forall x: bitstring, y: bitstring; xor(xor(x, y), y) = x.
```

- Function **pk**: It takes an input of a secret key and provides a variant of the public key related to itself: `fun pk(skey): pkey.`
- Function **encap**: The encapsulation function represents the ML-KEM encapsulation protocol and was designed using documentation and examples [6, 17], to ensure accuracy. It takes the inputs of a public key and bitstring and the rules for how it functions are set out in the **reduc** function attached:

```
fun encap(pkey, bitstring): bitstring.
reduc forall s: skey, r: bitstring;
      decap(encap(pk(s), r), s) = encap(pk(s), r).
```

- Function **modify**: A simple function used to modify a bitstring, it takes a singular bitstring input and provides a bitstring output: `fun modify(bitstring): bitstring.`

A set of events are declared with the goal of detecting when a task is completed and if it was successful. The first event declared is the **UserSends** event, which states whether the user has successfully completed the first transmission by successfully generating all required values from the provided inputs. The second event is **ServerAccepts**, which shows if the data sent by the user has been received successfully, the recomputed values are valid and matches the set values generated by the server. Finally, the third event **UserAccepts** shows if a user has accepted the response sent back from the server, verifying its legitimacy.

```
event UserSends(bitstring, bitstring).
event ServerAccepts(bitstring, bitstring).
event UserAccepts(bitstring, bitstring).
```

User → Server

When modelling the communication between the user and the server, the approach taken split each communication stage into individual partitions. The initial partition labelled as **userUi_A** establishes the core values for the user and sends $\mathbf{Msg_1}$ from the user to the server. It takes the inputs provided by the smart card and triggers the **UserSends** event, saving values required in later stages within the session channel and transmitting $\mathbf{Msg_1}$ to the server on channel c.

```
let userUi_A(Ai: bitstring, PWUi: bitstring, Pi: bitstring, pkSj: pkey, Vij: bitstring, Pij: bitstring)
↪  =
let Ri' = Rep(BIOi, Pi) in
let AIDi = h((UIDi, Ri')) in
let a' = xor(Ai, AIDi) in
let PWUi' = h((UIDi, PWi, a')) in

new x: bitstring;
let c1 = encap(pkSj, x) in
let K1 = kemkey(pkSj, x) in


if PWUi' = PWUi then
    new skU: skey;
    let pkU = pk(skU) in
    let M1 = xor(Vij, PWUi') in
    let M2 = xor(Pij, PWUi') in
    let M3 = xor(xor(AIDi, M2), K1) in

    new T1: bitstring;
    let M4 = h((M1, K1, T1)) in

    event UserSends(AIDi, K1);

    out(c, (c1, M3, M4, T1, pkU));
    out(session, (skU, M1, M2, AIDi, K1))
else
    0.
```

Server → User

Once $\mathbf{Msg_1}$ is received from the server by listening to channel c. Using the input of the secret key for the server **skSj** and the user's **AIDi** value. The process of computing the message blocks that should match the values provided by the user is executed. In this implementation, it is presumed that $\mathbf{T_1} = \mathbf{T_1'}$ does not stand because ProVerif does not know the concept of time [17]. This causes the two timestamps that should match to never equal each other as they are considered two separate constants. The server initialises a session key that is then included with the message block $\mathbf{M_9}$.

Once the server has completed the computation of all its values and has generated a new time stamp $\mathbf{T_2}$, the values which make up $\mathbf{Msg_2}$ are packaged and sent on the **c** channel. Further to this the server saves all required values for the next stages of the communication and confirms the server accepted the values provided by the user by triggering the **ServerAccepts** event.

```
let serverSj_1(skSj: skey, AIDi:bitstring) =
in(c, (c1: bitstring, M3: bitstring, M4: bitstring, T1: bitstring, pkU: pkey));

let T1' = T1 in

if T1' = T1 then
    let K1' = decap(c1, skSj) in

    let M5 = h((SIDj, PSRj)) in
```

```
    let M6 = xor(xor(M3, K1'), M5) in
    let M7 = h((M6, PSRj)) in
    let M8 = h((M7, K1', T1)) in

    if M4 = M8 then
        new x: bitstring;
        let c2 = encap(pkU, x) in
        let K2 = kemkey(pkU, x) in

        new T2: bitstring;

        let SKij = h((M5, M7, K1', K2, T2)) in
        let M9 = h((SKij, M7, T2)) in

        out(c, (M9, T2, c2));
        out(server, (SKij, M7, K2, T2, M6));
        event ServerAccepts(AIDi, K1')
    else
        0
else
    0.
```

User → Server

The final communication sent from the user to the server retrieves the values stored previously in the **session** channel, as well as the data sent by the server in the **c** channel. The core goal of this transmission is to create the session key which should equate to the same session key generated by the server following this step. Once again the time stamp is presumed that $\mathbf{T_2} \neq \mathbf{T_2'}$ does not stand by default due to the nature of ProVerif, the value of $\mathbf{T_2'}$ is declared as the same value as $\mathbf{T_2}$. Once the session key has been generated it is passed on to create the message block $\mathbf{M_{10}}$ which is the core element to mutually authenticate the user and the server. If $\mathbf{M_9} = \mathbf{M_{10}}$ then the user has verified the server.

The final step for the user is to calculate a value to be transmitted to the server along with a new timestamp $\mathbf{T_3}$ to allow for the server to verify the user. The event **UserAccepts** is triggered to confirm that the values received by the user from the server are accurate and have been validated.

```
let userUi_B() =
in(session, (skU: skey, M1: bitstring, M2: bitstring, AIDi: bitstring, K1: bitstring));
in(c, (M9: bitstring, T2: bitstring, c2: bitstring));

let T2' = T2 in

if T2' = T2 then
    let K2' = decap(c2, skU) in
    let SKij' = h((M2, M1, K1, K2', T2)) in
    let M10 = h((SKij', M1, T2)) in

    if M9 = M10 then
        new T3: bitstring;
        let M11 = h((SKij', M1, K2', T3)) in

        event UserAccepts(AIDi, K2');
        out (c, (M11, T3))
    else
        0
else
    0.
```

Server → User

The final stage in the transmission allows the server to verify that the session key generated by the user equates to the one stored by the server. A check is applied using the timestamp $\mathbf{T_3}$ and following its validation a new message block $\mathbf{M_{12}}$ is computed and is compared with the value $\mathbf{M_{11}}$ sent by the server which includes the users session key $\mathbf{SK'_{ij}}$. If $\mathbf{M_{11}} = \mathbf{M_{12}}$ then the server can confirm the two sessions keys generated individually by the user and the server match. Confirming that a session is now established between the user and the server.

```
let serverSj_2(skSj: skey, AIDi:bitstring) =
    in(server, (SKij: bitstring, M7: bitstring, K2: bitstring, T2: bitstring, M6: bitstring));
    in(c, (M11: bitstring, T3: bitstring));

    let T3' = T3 in

    if T3' = T3 then
        let M12 = h((SKij, M7, K2, T3)) in
        if M11 = M12 then
            event ServerAccepts(AIDi, K2)
        else
            0
    else
        0.
```

Process Initialisation

The required dummy data placements are set and inputted into each stage of the transmission as required. The values **M5i** and **M7i** allow for a relation to exist between the user and the servers message block values and are initialised within the users server specific login variables. Following these declarations each stage of the algorithm is executed in order.

```
process
    new skSj: skey;
    new Pi: bitstring;
    new a_AI: bitstring;
    let pkSj = pk(skSj) in
    let Ri = Rep(BIOi, Pi) in
    let AIDi = h((UIDi, Ri)) in
    let Ai = xor(a_AI, AIDi) in
    let PWUi = h((UIDi, PWi, a_AI)) in
    let M5i = h((SIDj, PSRj)) in
    let M7i = h((AIDi, PSRj)) in
    let Vij = xor(M7i, PWUi) in
    let Pij = xor(M5i, PWUi) in

    userUi_A(Ai, PWUi, Pi, pkSj, Vij, Pij) | serverSj_1(skSj, AIDi) | userUi_B() | serverSj_2(skSj,
    ↪  AIDi)
```

6.3 Mutual Authentication

Mutual authentication is also tested using ProVerif, and a set of authentication events was placed and queried at the user side as well as the server side. Since the session key $\mathbf{SK_{ij}}$ is independently calculated by both sides, this value can be used to verify mutual authenticity between the two entities. Both $\mathbf{SK_{ij}}$ and $\mathbf{SK'_{ij}}$ are saved within an event and compared using a query.

```
event UserAuth(bitstring).
event ServerAuth(bitstring).

event UserKeyConf(bitstring).
event ServerAuthUsr(bitstring).
```

```
query serverUAuthKey: bitstring;
    event(ServerAuthUsr(serverUAuthKey)) ==> event(UserKeyConf(serverUAuthKey)).


query MutualAuth: bitstring;
    event(UserAuth(MutualAuth)) ==> event(ServerAuth(MutualAuth)).
```

ProVerif checks mutual authenticity by running the **MutualAuth** query at the end of the process. If the events **ServerAuthUsr** and **UserKeyConf** are verified. This is because ProVerif expects **ServerAccepts** to have run prior to **UserKeyConf**. Both are essentially markers that record the same session key on both sides of the communication. However most importantly the same key must be used on both. If the case holds true then it can be said that mutual authentication has succeeded.

6.4 Confidentiality

Data confidentiality in the model is demonstrated by checking for attacks on pre defined input variables as well as proceed values. These values include a check if an attacker can retrieve the password and biometric input $\mathbf{PW_i}, \mathbf{BIO_i}$ and whether the attacker can retrieve the session key $\mathbf{SK_{ij}}$. In addition a check was made to see whether the server key pair $\mathbf{SID_j}, \mathbf{PSR_j}$ was retrievable by the attacker as well. The tests ran show that the papers claims for confidentiality hold for the test cases below run using ProVerif.

```
query attacker(PWi).

query attacker(SKij).

query attacker(BIOi).

query attacker(SIDj).
query attacker(PSRj).
```

Confidentiality is checked in ProVerif using the attacker query. The attacker query checks wether the attacker can compute the value provided from information it is able to retrieve from the protocol. If the data it collects is unable to derive the target value, it can be said that the attacker can not get the value and that it is confidential.

6.5 DoS Security

A test was run in which a chosen message block value $\mathbf{M_3}$ was modified during transmission. Checks were applied to see if the variables integrity was kept throughout the protocols transmission stages. In which it can be seen that if an attacker has modified this value or any other value sent on a public channel. The user and the server can not authenticate each other, and a denial of service based on the logic that both entities presume that one of them has provided incorrect information, rather than an attacker being present during communication has taken place.

In the ProVerif script the following events were placed within the original model, they provided a save state between each communication stage in which the message block $\mathbf{M_3}$ was interacted with. Furthermore, an attacker channel was specified to model a man in the middle environment, in which the attacker captures the inbound communication, modifies it and then sends the modified value to the receiving party. The value received by the attacker and the value sent by the attacker is stored in a secondary event for further comparison possibilities.

```
free  c_mod: channel [private].
event UserSentM3(bitstring).
event ServerReceivedM3(bitstring).
event FakeM3Sent(bitstring).

event AttackGets(bitstring).
event AttackSends(bitstring, bitstring).
```

The attacker is defined as a function in which the communication on channel **c** is received and the events **AttackerGets**, **AttackerSends** and **FakeM3Sent** are executed and the new value of $\mathbf{M_3'}$ is sent to the receiving entity.

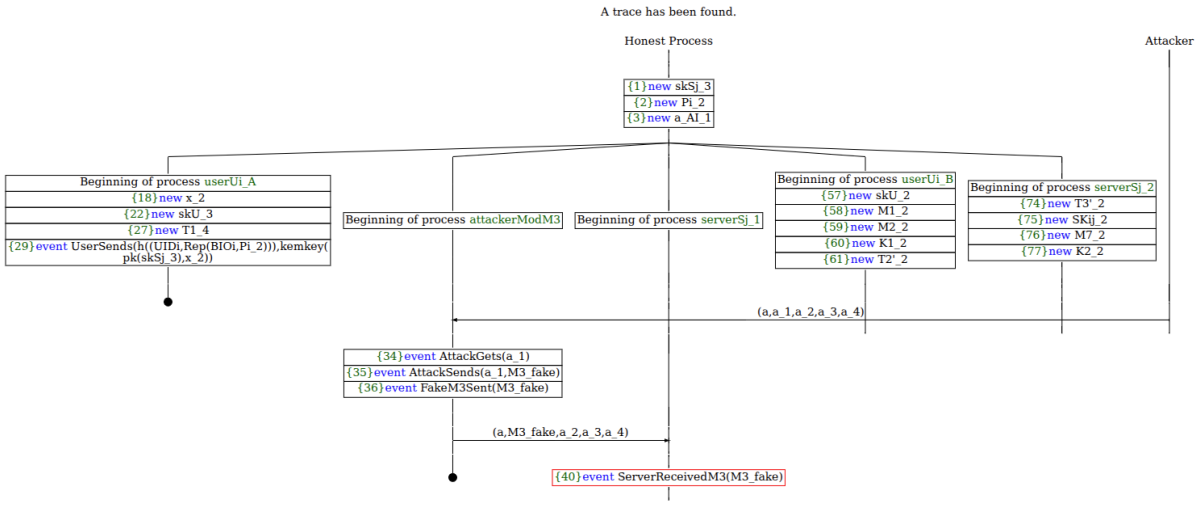


Figure 8: DoS Attack Trace

The DoS attack can be seen on the attack trace, in which ProVerif was able to detect that the original value of $\mathbf{M_3}$ did not match the save state from the start of the protocol. In this specific example, the attacker modified the value during the initial **User** $\rightarrow$ **Server** communication, in which the value was used in the construction of the message block $\mathbf{M_6}$ which led to the conclusion of $\mathbf{M_8} \neq \mathbf{M_4}$. This caused the protocol to end at this stage and not continue the communication due to the two message block values not equating to each other.

7 Fixed Protocol

7.1 Protocol Description

Now that the protocol modelled by the core paper [48] has been tested and a proof of a DoS attack has been found. The original paper assumes that the Dolev-Yao adversary is capable of intercepting and modifying a message sent over a public channel. However, the original authentication message is not capable of providing confirmation of integrity over the message sent on the public channel prior to values being processed by either the user or the server. This depends on the direction of communication [48].

7.2 Possible Solutions

When evaluating possible solutions to prevent this DoS attack, two ideas were explored. The initial idea was to apply a digital signature algorithm (DSA) [3] to the messages. The idea was explored, however, it did not solve the DoS attack. If a signature was used an attacker could still modify the data in transmission and the receiving end will still not know whether an attacker has modified the message or if the sender used an invalid or incorrect signing key on the message. Due to this a decision was made to use MAC as even if the attacker were to modify the data in transmission the receiving entity can independently confirm whether the data was modified during transmission as the MAC sent would not match the MAC calculated on the receiving end.

7.3 Security Analysis

7.3.1 Protocol

In the original protocol, the user transmits:

$$Msg_1 = \{c_1, M_3, M_4, T_1, pk_{U_i}\}.$$

Whilst testing the protocol using ProVerif it was seen that a man in the middle attack could take place on the protocol by the attacker intercepting the communication on the public channel. This allows the attacker to modify one or more of the transmitted values, such as $\mathbf{M_3}$ without the knowledge that this was provided to the user or the server. This leads to a failure to authenticate, which essentially locks the user out of the server.

$$Msg_1 = \{(c_1, M_3, M_4, T_1, pk_{U_i}), Msg_{1_{mac}}\}.$$

To prevent this from happening, message authentication codes (MACs) are added to the messages sent between the user and the server. The purpose of this is to provide a point of integrity verification, allowing for detection of modified values during transmission. If a value is modified during transmission, the MAC verification fails, and the modified message block or variable is rejected.

The key generation for the MAC is created using a hash of a timestamp **TMac** and a set of re-computable message blocks:

$$macKey \leftarrow h(TMac \parallel M_1 \parallel M_2 \parallel AID_i \parallel K_1)$$

This key can then be applied to each message sent in transmission between the user and the server. This can then be compared with a recomputed version of the message using the key derived by the server and the message blocks sent in transmission. If the message with the mac key applied matches that of the server-derived value, it can be said that the message was not modified in transmission.

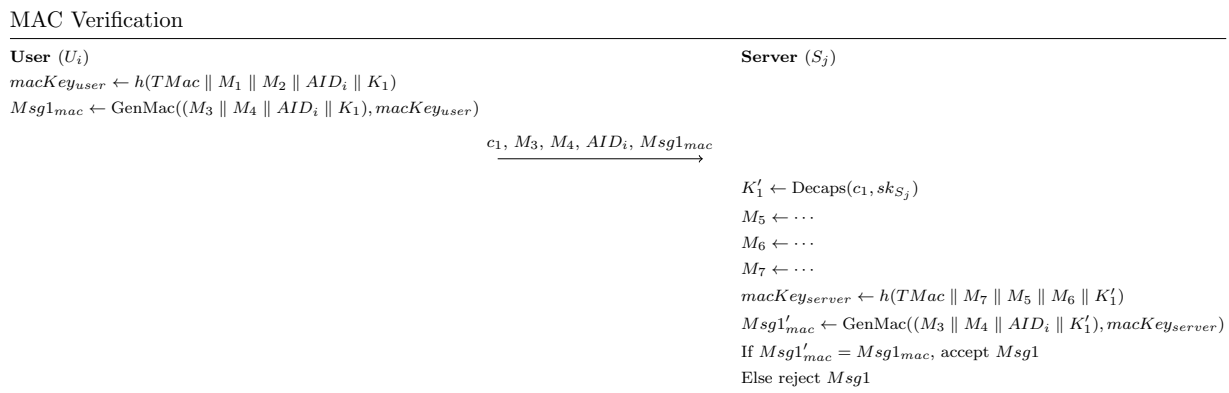


Figure 9: MAC verification between the user and the server

The above is a representation of the MAC fix being applied to **Msg1** that is transmitted between the user and the server. The key generation and recreation is done independently and then the outputted MAC is compared

7.4 Implementation

To model the DoS attack fix in ProVerif, additional functions and variables where set. The first variable established was a private timestamp that will be used during the mac transmission, `free TMac: bitstring [private].` A new function for generating a mac was defined, `fun GenMac(bitstring, bitstring): bitstring.` This was based on the mac generation function in the ProVerif documentation on Page 63 [17].

Solution Evaluation

The ability to use an MAC to solve the attack is linked to the relational values between the user and the server. When a user and a server register with the RC values relating to each other are linked, particularly when a user registered with a server the two values $\mathbf{P_{ij}}$ and $\mathbf{V_{ij}}$ allow the user to have a shared key pair with the server with symmetric properties. Since these values are pre-agreed between the user and the server, the attacker cannot compute them. These are then used in the calculation of the message blocks sent between the user and the server. The values that are claimed identical between the server and the user are stated in the table below:

User Value	Server Value
M_1	M_7
M_2	M_5
AID_i	M_6
K_1	K_1'

Table 6: Equivalent values calculated by the user and server.

During the initial communication request sent by the user, a mac key is generated based on the pre-defined **TMac** timestamp along side values that can be recomputed by the server.

Due to the relational nature of these values the same mac key can be generated on both sides without transmitting the key on the public channel. This key is then used to apply a mac to

$\mathbf{Msg_1}$ and is sent along side the contents of $\mathbf{Msg_1}$ and the mac key is saved by the user in the session channel.

```
if PWUi' = PWUi then

    ...

    let macKey = h((TMac, M1, M2, AIDi, K1)) in
    let Msg1Mac = GenMac((M3, M4, AIDi, K1), macKey) in

    ...

    out(c, (c1, M3, M4, T1, pkU, Msg1Mac));
    out(session, (skU, M1, M2, AIDi, K1, macKey))
else
    0.
```

Once $\mathbf{Msg_1}$ is received by the server, the server generates its own mac key and then generates it own version of $\mathbf{Msg_1}$ using the values provided and if the value of **ExpectedMsg1Mac** is the same as the value of **Msg1Mac** which was sent along side the message body the server can confirm that the message was not altered during transmission.

```
    let macKey = h((TMac, M7, M5, M6, K1')) in
    let ExpectedMsg1Mac = GenMac((M3, M4, M6, K1'), macKey) in

     if ExpectedMsg1Mac = Msg1Mac then
        ...
```

Similarly with the following communication stages, a mac of the message is created and added to the payload sent between the user and the server.

```
    ...
    let Msg3Mac = GenMac((M11, T3), macKey) in
    ...
    let ExpectedMsg3Mac = GenMac((M11, T3), macKey) in
```

A note should be made that the mac key can vary between communications, however, for this implementation a single mac key was used for all transmissions. This can still provide the required security in this example, as the user and the server calculate the key individually without transmitting the key to a public channel.

Once the proposed mac check has been applied to the scheme, the same attack environment is set out. This time it can be seen that the modified message block value of $\mathbf{M_3'}$ was detected and prevented from causing the user to be denied access to the service.

```
--------------------------------------------------------------
Verification summary:

Query event(ServerVerifiedM3(m)) ==> event(UserSentM3(m)) is true.

--------------------------------------------------------------
```

Figure 10: DoS Attack Failed

7.5 Security Properties

Once the MAC fix is applied to the protocol, the same tests that were applied to the original scheme are tested on this new version as well. This is done to provide confirmation that the addition of the MAC fix did not cause other vulnerabilities and secondary tests to fail. To test integrity, a server should not allow a modified request to be accepted, even if it is supposedly from a legitimate user.

Mutual authentication was verified by checking if $\mathbf{SK_{ij}} = \mathbf{SK'_{ij}}$, if the two session keys still matched on both the user and server side then the protocol has successfully accomplished mutual authentication.

Further tests were applied to ensure that values that were never transmitted in their base format were not accessible by the attacker. This included checking that the secrecy of the user's credentials $\{\mathbf{PW_i}, \mathbf{UID_i}, \mathbf{BIO_i}\}$ remained secret before and after the MAC fix was applied. Preventing the attacker from directly posing as the user.

8 Software Testing

A Python based implementation of the Fixed Protocol was designed to test practicality and implementation. An important part of practicality is the functionality and efficiency of the protocol. During testing the registration and login process were tested. This includes functionality and how quickly a user can register or log in on a server of their choosing. To test these, the Rest-API written using FastAPI was communicated with, sending POST request of login and registration attempts using Postman [7] to send these requests to the API.

8.1 Testing Environment

The test environment used is as follows:

- **CPU:** Ryzen 7 9800x3D
- **RAM:** 64GB
- **OS:** Mint Linux (Debian)

8.2 Registration

8.2.1 User Registration

Testing user registration involved sending a new set of user credentials to the RC and confirming that the user was successfully registered and how long it took to register them. During testing, it could be seen that it took ∼**900ms** to register a user with the RC with a **10%** variation between attempts.

Within the ∼**900ms** time frame, the protocol completes a two way communication between the user and the RC. Establishing a shared key pair with the RC and the servers registered with the RC at the time of communication. These keys along with user credentials are compiled into the smart card. This leads to the registration phase for the user being practical in real world environments.

8.2.2 Server Registration

During server registration testing, three variants of ML-KEM were tested, **ML-KEM-512**, **ML-KEM-768**, and **ML-KEM-1024**, the performance of each one applied to the protocol was tested to check wether it was computational feasible in a real world environment. Server registration requires a public and private key pair of ML-KEM keys to be generated. This is a core element of the server registration stage which would impact the performance the highest.

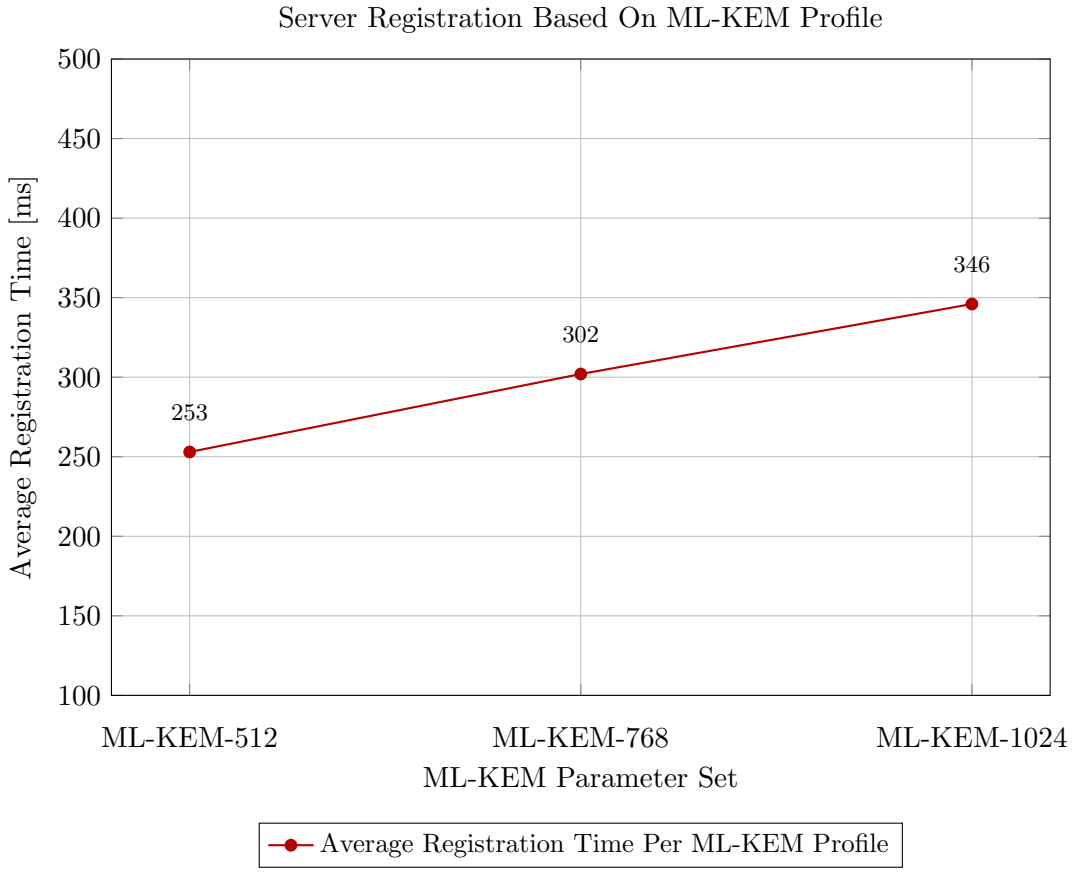


Figure 11: Server Registration Times

As seen in Figure 11, all three profiles provide a similar level performance in terms of time taken to process the server registration. From **ML-KEM-512** at **253ms** to **ML-KEM-768** at **302ms** it can be seen that on average the difference in computation time was **49ms** whilst the difference between **ML-KEM-768** and **ML-KEM-1024** at **346ms** was **44ms**. This represents a marginally small difference at a glance, however, depending on the hardware processing this request the impact could be reasonably impactful. As the difference between **ML-KEM-512** and **ML-KEM-768** can also be viewed as **19.4%** longer processing time, this can expensively increase computation time on less powerful systems. This can be seen at an even larger scale if looking at the difference between **ML-KEM-512** and **ML-KEM-1024** as the difference between these two protocol computation times is **36.8%**, which demonstrates a possible large impact factor.

8.3 Login

When testing practicality and feasibility of the login function of the protocol, all three ML-KEM profiles were tested. This allowed for a understanding of how the key generation, encapsulation, and decapsulation effect performance of the login function. The login function triggers multiple stages of the protocol, including the FE, SS, FastAPI, RC, and Server of choice.

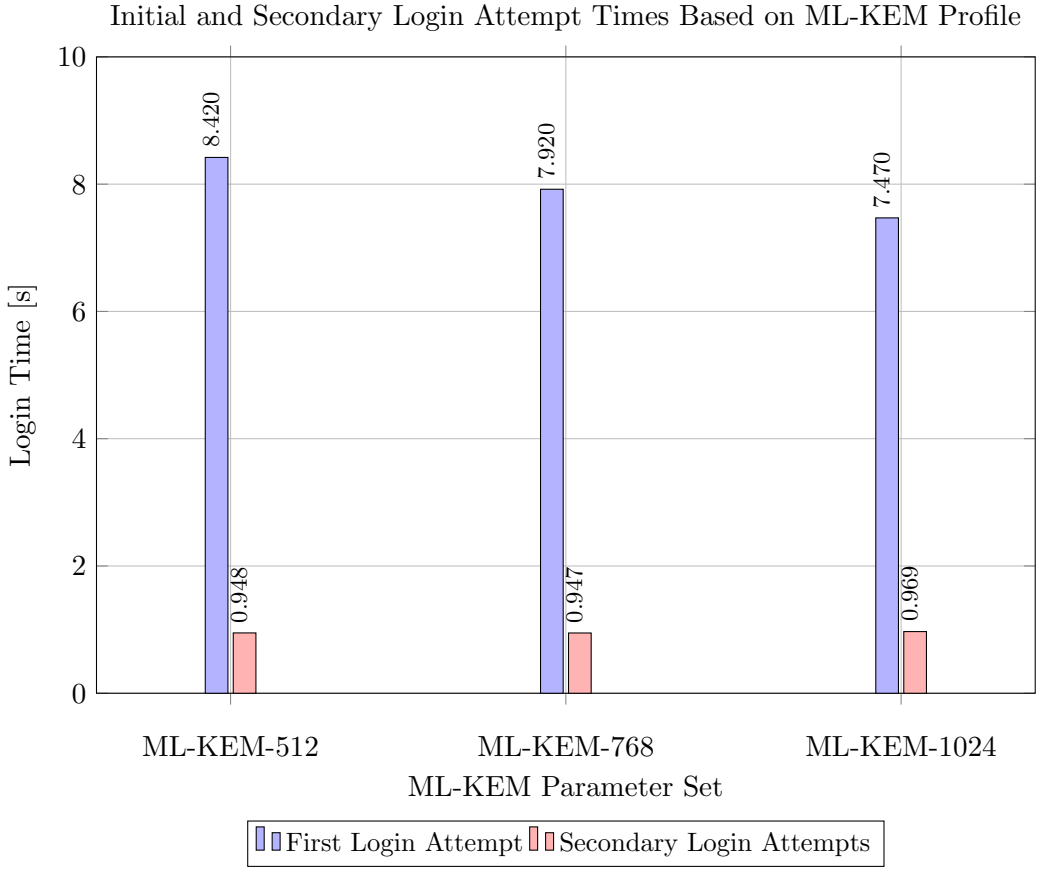


Figure 12: Initial and secondary login times based on ML-KEM profile

As seen in Figure 12, the login function successfully authenticated users on each profile. Two sets of timing were recorded as an inconstancy was noted between a users initial login when the service was newly booted, compared to any secondary login requests. The difference between the initial login attempt when the service is only newly started compared to subsequent login requests shows that additional processes may be running. This could possibly be related to the FastAPI gateway having to initialise and load the scripts on the initial request, alternatively it could also be related to the fingerprint scanner needing to initialise on the first run.

The results show that the choice in ML-KEM profile provides minor recourse constraints as the key size increases. The impact of ML-KEM on the login phase is much lower than during server registration, as the login phases additional components use more system resources than ML-KEM. Therefore, the practicality of the login phase is viable with minimal impact from key generation, however, is rather restricted during initialisation.

Copyright 2026 © Aran Simon Jannson

8.4 Biometric and Secure-Sketch Accuracy Testing

The implementing and testing the SS was one of the most challenging aspects of this project. Due to the natural noisiness of biometric data, creating a reproduced scan of a pre-registered scan is challenging. Due to this, repeated scans of the same biometric fingerprint input may not produce the same identical input value. This can be caused by something as simple as moving the finger slightly on the scanner or if the finger has more dirt on it compared to when it was registered. Therefore, the BCH error-correction implemented in the SS is crucial to providing a releasable level of accuracy. This then led to testing of different BCH values to narrow down on the most optimal value that can be used for the SS. The BCH error-correction code is represented by the set of parameters $[\mathbf{n}, \mathbf{k}, \mathbf{d}]$, where $\mathbf{n}$ is the length of the codeword, $\mathbf{k}$ is the length of the message and $\mathbf{d}$ is the minimum distance. The value of $\mathbf{t}$ is then adjusted to specify the number of bits that can be corrected before the scan is considered invalid.

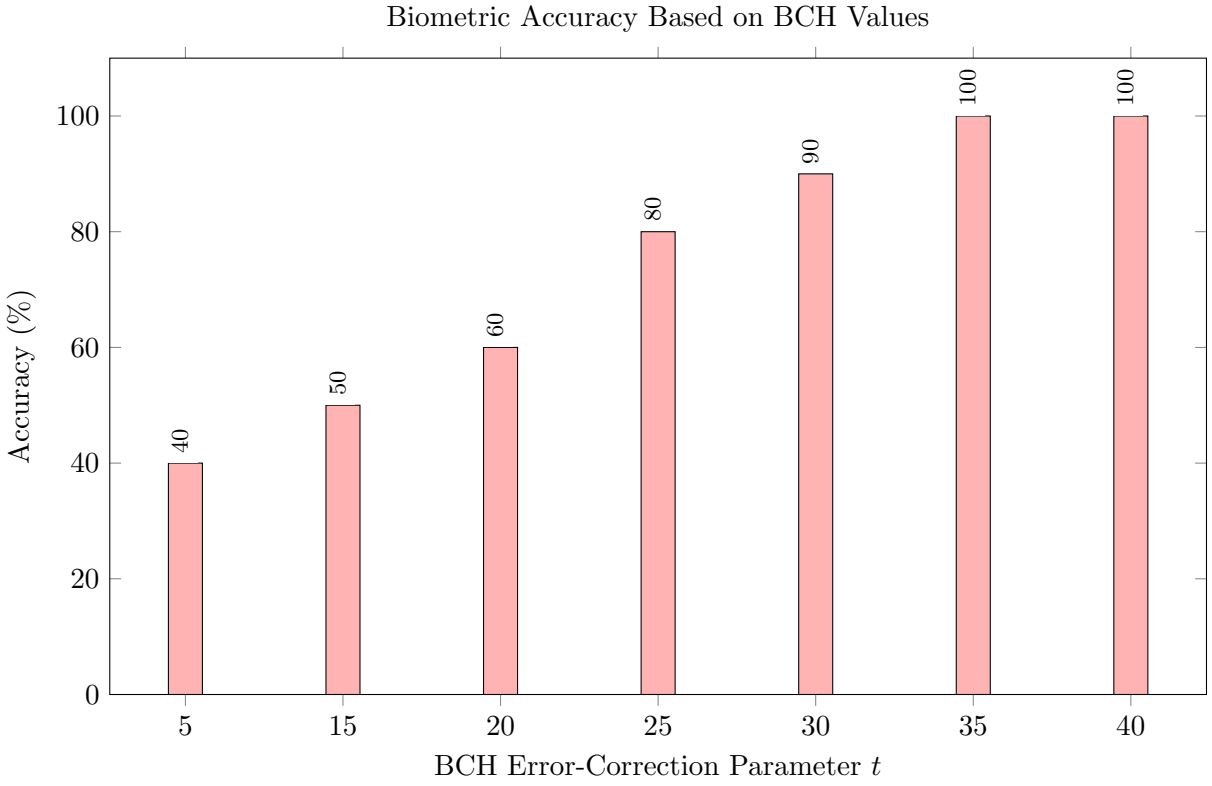


Figure 13: Biometric accuracy for based on BCH error-correction values

During testing, the value $\mathbf{n}$ remained at 1023, was the value of $\mathbf{t}$ varied between tests. Each error correction range was tested 20 times each, which included different biometric scans and the original biometric fingerprint scan as well. The value of $\mathbf{t}$ was increased by a value of 5 each attempt, which allowed a varied range of 5 to 40 to be tested and observed. The value of $\mathbf{k}$ was also increased each time as BCH error-correction using galois [24] requires an increased value of $\mathbf{k}$ for it to function successfully. To calculate these values, the paper *BCH Codes by Yunghsiang S. Han* [23] were used.

Copyright 2026 © Aran Simon Jannson

n	**t**	**d**	**k**
1023	5	11	973
1023	15	31	873
1023	20	41	828
1023	25	51	778
1023	30	61	728
1023	35	71	698
1023	40	81	648

Table 7: BCH Values Used During Testing

The results of the test as seen in Figure 13 show that the set value of **t** affected precision, in the case of biometric data inputted. The tests showed that the ideal value was when $\mathbf{t} \leftarrow \mathbf{20}$, as this provided a 60% highest success rate compared to the false-acceptance rate. The higher the value increased, the more false positives were accepted as a valid scan, this is due to the number of bits that are allowed to be modified to recreate the original scan. This can be seen when $\mathbf{t} \leftarrow \mathbf{35}$ and $\mathbf{t} \leftarrow \mathbf{40}$, where all scans taken were accepted. This data set included false and positive data, however, the system accepted all inputs. The opposite effect can be seen at the other end of the spectrum, where $\mathbf{t} \leftarrow \mathbf{5}$ accepts the least number of scans, provided that the biometric scan is within a small margin. Correcting a limited number of error-bits.

9 Project Reflections

9.1 General Project Reflection

The final result of the project demonstrated the proposed PQC-MSA scheme in a formal and practical environment. Formally, the protocol was tested to analyse its security properties, which in return allowed for the discovery of the DoS attack and the implementation to mitigate the vulnerability by modifying the protocol. The protocol was implemented successfully practically using Python and FastAPI and tested with the Joy-It fingerprint scanner. Both the formal and the practical showed that the protocol was feasible to be implement and proven to be secure based on the tests ran on the protocol.

The overall formal verification of the system was one of the most important outcomes of this project as it demonstrated a serious vulnerability in the original protocol that was proposed. The analysis was done under the Dolev-Yao model that detected the vulnerability. Whilst the practical implementation allowed for feasibility to be tested, where problems were found in places that where unexpected. This includes the implementation of the SS and FE, where the accuracy of the biometric scan was more complex to get correct than initially expected.

9.2 Success In The Project

One of the most important successes was the practical implementation of the protocol, providing a proof of concept for the protocol. Converting the cryptographic formation of the protocol and the DoS fix provided a challenge that allowed for a deeper understanding of the protocol and PQC as a whole. In addition the implantation of a physical fingerprint scanner allowed for more accurate testing and performance evaluations to be done. It showed how real biometric hardware is implemented into a FE rather than relying on test data that does not provide the same level of uniqueness and detail as real fresh biometric data providing variation between scans.

The formal verification was also a major success due discovering the DoS attack. Modelling the protocol using ProVerif allowed for a greater understanding on how ProVerif models and tests cryptographic protocols using Dolev-Yao. This lead to the testing and implementation of a solution to prevent the DoS attack from taking place. Exploring the possibility of digital signatures being used as a fix, however, later testing and implementing a MAC system instead. The formal security of the fix was demonstrated using the same attack that discovered the vulnerability, where the attack failed to succeed once the fix was applied.

9.3 Issues Faced

During formal verification and practical implementations of the protocol, the hardest task was going from cryptographic notation to code. In particular, implementing the protocol in Python was a challenge, as developing an SS and FE that functioned as described by the paper had to be done manually, as no direct library was available that provided the required output and accuracy tune-ability. To solve this issue, creating a first party solution was necessary, however, the accuracy on the SS did vary. Furthermore, the core papers [47,48] described the FE abstractly as $\mathbf{R'_i} = \text{Rep}(\mathbf{BIO'_i}, \mathbf{P_i})$, this is cryptographically accurate but did not present an example data type, causing an appropriate binary format to be used to process the data. Due to this, the generated binary value did not match the required length, leading to an additional padding value of 0 to be added to the end of the produced scan to match the required bit string lengths of BCH and ML-KEM. Furthermore, scans taken where rounded to the value of 0 or 1 based on whether it was greater than 128 or not. This was necessary to be able to process the inputted scan, but may have affected the accuracy of the inputted data.

9.4 Future Work

The future of this thesis could explore the protocol further, testing additional paradigms, as well as exploring and formally verifying other proposed protocols and comparing them with the protocol chosen in this paper.

In addition, further work can be done to increase the accuracy of the SS and FE by expanding the protocol itself to allow for further possibilities. This can be done by testing alternative post-quantum secure cryptographic schemes. Alternative solutions for the SS can also be explored, testing alternatives to the implemented BCH solution. This could be an alternative error-correction mechanism which may provide improved accuracy. However, it is important to balance accuracy and efficiency, as a more accurate solution may require drastically more computational headroom. The benefits of this must be taken in to account to see whether improved accuracy is large enough to merit more computational power. Furthermore, biometric data can also be collected from a large pool of users, to allow for more variations in the test scenarios. This would provide for a more complex tuning and adjustability of false positives and negatives by examining the differences in more detail between different biometric inputs from multiple users.

A practical user interface could also be created to allow for a GUI based demo. This could then be presented to the pool of users that would be used for biometric variety and accuracy tests, allowing them to easily interact with the created software. This feature was originally considered during the initial design phase of the project, however, was later abandoned to focus on security and accuracy improvements as the time frame was limited. Due to this, the designed Rest API can be easily implemented into a frontend solution with minor changes. Allow for an interactive demo to be created.

Copyright 2026 © Aran Simon Jannson

Finally, the server implementation was in the form of a proof of concept. Any future work should consider testing the system with external servers on different networks to provide insight on how the protocol would function in different environments. This would add additional processing considerations to the equation such as network latency and the possibility of package loss effecting the authentication process. However, it is important to test real world scenarios that a test where all systems share a network can not provide.

9.5 Final Thoughts

The project was a successful implementation and verification of a PQC-MSA protocol. This provided valuable insight formally and practically, whilst fixing a major vulnerability in the proposed scheme. The testing showcased that a protocol such as the one in this core paper can be placed into a real world scenario with reasonable computational expectations. The combined implementations provided by the formal and the practical side added to the literature provided by the core paper and protocol.

The goals of project set out in Table 1 were accomplished. With the exception of the user interface for the practical implementation that was unfortunately not implemented due to time restraints. This should be considered for any future work that continues from this research.

Copyright 2026 © Aran Simon Jannson

References

[1] biokey-secure-sketch/biokey at master · biometricsecurity/biokey-secure-sketch — github.com. https://github.com/biometricsecurity/biokey-secure-sketch/tree/master/biokey.

[2] Cryptographic hash functions — ibm quantum learning — quantum.cloud.ibm.com. https://quantum.cloud.ibm.com/learning/en/courses/quantum-safe-cryptography/cryptographic-hash-functions.

[3] Digital signatures — practical cryptography for developers — cryptobook.nakov.com. https://cryptobook.nakov.com/digital-signatures.

[4] Github - bastianraschke/pyfingerprint: Python library for zhiantec fingerprint sensors (e.g. zfm-20, zfm-60) — github.com. https://github.com/bastianraschke/pyfingerprint.

[5] GitHub - cascremers/scyther: The scyther tool for the symbolic analysis of security protocols — github.com. https://github.com/cascremers/scyther.

[6] GitHub - secure-6g/proverif-aka-6g: Formal verification code for an aka protocol for 6g — github.com. https://github.com/Secure-6G/ProVerif-AKA-6G.

[7] Postman: The world's leading api platform — sign up for free — postman.com. https://www.postman.com/.

[8] Proverif — bblanche.gitlabpages.inria.fr. https://bblanche.gitlabpages.inria.fr/proverif/usage.html#modeling.

[9] pyfingerprint/src/files/examples/example_enroll.py at Development · bastianraschke/pyfingerprint — github.com. https://github.com/bastianraschke/pyfingerprint/blob/Development/src/files/examples/example_enroll.py.

[10] Shor's algorithm — ibm quantum documentation — quantum.cloud.ibm.com. https://quantum.cloud.ibm.com/docs/en/tutorials/shors-algorithm.

[11] Sicherheitskritisch — Fingerprint sensor for Raspberry Pi and Debian Linux (Update 8) — sicherheitskritisch.de. https://sicherheitskritisch.de/2015/03/fingerprint-sensor-fuer-den-raspberry-pi-und-debian-linux-en/.

[12] Signal protocol and post-quantum ratchets — signal.org. https://signal.org/blog/spqr/.

[13] Tamarin prover manual — tamarin-prover.com. https://tamarin-prover.com/manual/master/book/003_example.html.

[14] What is salting in security? password hashing and salting explained — authgear.com. https://www.authgear.com/post/password-hashing-salting-function-and-algorithm-explained/.

[15] Maryam Ahmed, Habeeb Omotunde, Baharan Sanjabi, Amirhossein, and Difo Aldiaiz. Diffie helman and its use in secure internet protocols. *International Journal of Engineering Science and Innovative Technology*, 1:69–73, 11 2012.

[16] Nawal Zaied Almuzaini and Iftikhar Ahmad. Formal analysis of the signal protocol using the scyther tool. In *2019 2nd International Conference on Computer Applications Information Security (ICCAIS)*, pages 1–6, 2019.

[17] Vincent Cheval Bruno Blanchet, Ben Smyth and Marc Sylvestre. Automatic cryptographic protocol verifier, user manual and tutorial. https://bblanche.gitlabpages.inria.fr/proverif/manual.pdf.

[18] Cas Cremers. Scyther on linux. https://people.cispa.io/cas.cremers/scyther/scyther-linux.png.

[19] Tran Dang and Thao Nguyen. Thi ai thao nguyen, tran khanh dang: Protecting biometrics using fuzzy extractor and non-invertible transformation methods in kerberos authentication protocol. transactions on large-scale data- and knowledge-centered systems, lncs, springer verlag, 2016. *Transactions on Large-Scale Data- and Knowledge-Centered Systems (TLDKS)*, 01 2016.

[20] Nikola Davidović and Marko Mojsilović. Survey of post-quantum transition from rsa to ml-kem algorithm. In *2026 25th International Symposium INFOTEH-JAHORINA (INFOTEH)*, pages 1–5, 2026.

[21] Richard Davis. Nist special publication (sp) 800-56a rev. 3, recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography — csrc.nist.gov. https://csrc.nist.gov/pubs/sp/800/56/a/r3/final.

[22] Yevgeniy Dodis, Rafail Ostrovsky, Leonid Reyzin, and Adam Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. *SIAM Journal on Computing*, 38(1):97–139, 2008.

[23] Yunghsiang S. Han. Bch codes. https://i4ai.org/hanyunghsiang/BCH_code.pdf.

[24] Matt Hostetter. Galois: A performant NumPy extension for Galois fields, 11 2020.

[25] Joy-IT. *COM-FP-R301T Fingerprint Sensor Manual*. Joy-IT, 2022.

[26] Yen-Lung Lai and Zhe Jin. Input-dependent error sketching model enabled information theoretical secure sketch. *IEEE Access*, 8:134681–134694, 2020.

[27] Daniele Micciancio and Oded Regev. Lattice-based cryptography. In Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen, editors, *Post-Quantum Cryptography*, pages 147–191. Springer, 2009.

[28] Takehiko Mieno, Hiroyuki Okazaki, Kenichi Arai, and Yuichi Futa. Formal verification of authenticated encryption with associated data with tamarin prover. In *2023 Congress in Computer Science, Computer Engineering, Applied Computing (CSCE)*, pages 2465–2471, 2023.

[29] Hassan Mistareehi, Jason Owen, and Abdellatif Aboualy. Biometric authentication in identity and access management. In *2025 Sixteenth International Conference on Ubiquitous and Future Networks (ICUFN)*, pages 650–652, 2025.

[30] Himanshu Mittal. Diffie-hellman based smart-card multi-server authentication scheme. In *2014 International Conference on Computational Intelligence and Communication Networks*, pages 808–812, 2014.

[31] NIST Special Publication 800-38B Morris Dworkin. Recommendation for block cipher modes of operation: The cmac mode for authentication. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-38b.pdf.

[32] National Institute of Standards and Technology. What Is Post-Quantum Cryptography?, 2026. Created August 13, 2024, updated February 27, 2026.

[33] Tran Thi Nga, Nguyen Thanh Tung, Nguyen Duc Cong, Dinh Van Hung, and Duc-Tho Mai. Blockchain-integrated 5g authentication: A proverif formal analysis approach. In *2025 2nd International Conference On Cryptography And Information Security (VCRIS)*, pages 1–6, 2025.

[34] National Institute of Standards and Technology. Federal Information Processing Standard (FIPS) 203, Module-Lattice-Based Key-Encapsulation Mechanism Standard — csrc.nist.gov. https://csrc.nist.gov/pubs/fips/203/final.

[35] National Institute of Standards and Technology. Federal information processing standard (fips) 46-3, data encryption standard (des) — csrc.nist.gov. https://csrc.nist.gov/pubs/fips/46-3/final.

[36] Ryan Ercel O. Paderes. A comparative review of biometric security systems. In *2015 8th International Conference on Bio-Science and Bio-Technology (BSBT)*, pages 8–11, 2015.

[37] JOSHUA PITE and YIYING ZHONG. The rsa cryptosystem. https://math.mit.edu/research/highschool/primes/circle/documents/2024/Honglin.pdf.

[38] Federal Information Processing Standards Publication. Advanced encryption standard (aes). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197-upd1.pdf.

[39] FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION. The keyed-hash message authentication code (hmac). https://csrc.nist.gov/files/pubs/fips/198/final/docs/fips-198a.pdf.

[40] FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION. Secure hash standard (shs) — fips pub 180-4. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf.

[41] FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION. Sha-3 standard: Permutation-based hash and extendable-output functions — fips pub 202. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[42] Karen Randall. Nist special Publication (sp) 800-186, recommendations for discrete logarithm-based cryptography: Elliptic curve domain parameters — csrc.nist.gov. https://csrc.nist.gov/pubs/sp/800/186/final.

[43] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, pages 84–93. Association for Computing Machinery, 2005.

[44] Aviad Rubinstein. Lecture 14: Learning with errors. CS 354: Unfulfilled Algorithmic Fantasies, May 2019. Scribed by Nick Guo. Accessed: 13 July 2026.

[45] Bin Wang and Maode Ma. An efficient and secured multi-server authentication scheme. In *ICICS 2011*, 2011.

[46] Huawei Wang, Dianli Guo, Hua Zhang, and Qiaoyan Wen. Robust multiple servers architecture based authentication scheme preserving anonymity. *Sensors*, 19(14):3144, 2019.

[47] Yunhua Wen and Shengli Liu. Reusable fuzzy extractor from lwe. In *Information Security and Privacy: 23rd Australasian Conference, ACISP 2018, Wollongong, NSW, Australia, July 11-13, 2018, Proceedings*, page 13–27, Berlin, Heidelberg, 2018. Springer-Verlag.

[48] Yunhua Wen, Yandong Su, and Wei Li. Post-quantum secure multi-factor authentication protocol for multi-server architecture. *Entropy*, 27(7), 2025.

[49] Dexin Yang and Bo Yang. A biometric password-based multi-server authentication scheme with smart card. In *2010 International Conference On Computer Design and Applications*, volume 5, pages V5–554–V5–559, 2010.

[50] Jen-Ho Yang, Ya-Fen Chang, and Chih-Cheng Huang. A user authentication scheme on multi-server environments for cloud computing. In *2013 9th International Conference on Information, Communications Signal Processing*, pages 1–4, 2013.

[51] Ze Yang, Qin Shi, Teng Cheng, Qiang Zhang, Qiang Liu, Yangyang Liu, and Shenghao Peng. Qbma-biv: Quantum-key-distribution (qkd)-based multi-server authentication scheme for blockchain-enabled internet of vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 25(11):18433–18448, 2024.

[52] Jie Yu and Qingqi Pei. A multi-server architecture authentication protocol using smart card. In *2012 Eighth International Conference on Computational Intelligence and Security*, pages 511–515, 2012.

[53] Yunru Zhang, Min Luo, Kim-Kwang Raymond Choo, and Debiao He. A general architecture for multiserver authentication key agreement with provable security. *Security and Communication Networks*, 2018(1):2462901, 2018.