

**Technical University of Košice  
Faculty of Electrical Engineering and Informatics**

**Compact OPAQUE Protocol Implementation  
for Embedded Cryptographic Applications**

**Master thesis**

**2024**

**Bc. Patrik Zelenák**

**Technical University of Košice**  
**Faculty of Electrical Engineering and Informatics**

# **Compact OPAQUE Protocol Implementation for Embedded Cryptographic Applications**

**Master thesis**

Study Programme: Computer networks  
Field of Study: Computer Science  
Department: Department of Electronics and Multimedia Communica-  
tions  
Supervisor: prof. Ing. Miloš Drutarovský, CSc.

**Košice 2024**

**Bc. Patrik Zeleňák**

## Abstract in English

This thesis focuses on a new and modern asymmetric password-authenticated key exchange (aPAKE) protocol called OPAQUE, which holds significant potential to become a widely adopted method for client-server authentication. The OPAQUE protocol works well with prime-order elliptic curves, which is a significant advantage in modern cryptography. However, this thesis describes the usage of non-prime-order elliptic curves, focusing specifically on Bernstein's Curve25519. Security concerns associated with this adaptation are addressed by applying a fast transformation to transition into a more secure abstract group known as the Ristretto255 group, utilizing the Ristretto255 transformation. Due to the absence of OPAQUE implementations targeted for microcontroller units (MCUs), we chose to develop our own implementation of the Ristretto255 transformation and subsequently the entire OPAQUE protocol. This implementation employs various strategies and optimization techniques designed for the ARM Cortex-M4 core. Furthermore, our implementation adopts an endian-agnostic approach, ensuring platform independence. The implementation strategies and optimization techniques heavily focus on accelerating critical arithmetic operations within  $\mathbb{GF}(2^{255} - 19)$ . As a result, we have created compact and fast implementations of Ristretto255 and the OPAQUE protocol in C language, specifically targeted for the ARM Cortex-M4 core. This research contributes to expanding the applicability of OPAQUE authentication to resource-constrained embedded systems, especially those with the ARM Cortex-M4 core, enhancing its potential for widespread adoption in modern cryptographic applications. This thesis provides our optimized implementation of the Ristretto255 transformation as well as the entire OPAQUE protocol. The results of our measurements are available in Chapter ??

## Keywords

OPAQUE protocol, aPAKE, Ristretto255, ARM Cortex-M4, Embedded cryptography, Microcontrollers

---

## Abstrakt v slovenčine

Táto práca sa zameriava na nový a moderný asymetrický protokol pre autentifikáciu heslom a výmenu kľúčov (aPAKE) nazývaný OPAQUE, ktorý má potenciál stať sa široko používanou metódou pre autentifikáciu klienta a servera. OPAQUE protokol poskytuje podporu pre eliptické krivky prvočíselného rádu, čo je značná výhoda v oblasti modernej kryptografie. Avšak táto práca opisuje použitie eliptických kriviek s neprvočíselným rádom, konkrétne so zameraním na Bernsteinovu krivku Curve25519. Obavy týkajúce sa bezpečnosti súvisiace s touto adaptáciou sú riešené aplikovaním rýchlej transformácie, ktorá zabezpečuje prechod do bezpečnejšej abstraktnej grupy známej ako Ristretto255 grupa, prostredníctvom Ristretto255 transformácie. Vzhľadom na nedostatok implementácií OPAQUE protokolu zameraných na mikrokontroléry sme sa rozhodli vyvinúť vlastnú implementáciu Ristretto255 transformácie a následne celého OPAQUE protokolu. Táto implementácia využíva rôzne stratégie a optimalizačné techniky navrhnuté pre procesory s jadrom ARM Cortex-M4. Okrem toho naša implementácia poskytuje kód nezávislý na endianite, čo zabezpečuje platformovú nezávislosť. Implementačné stratégie a optimalizačné techniky sa zameriavajú na zrýchľovanie kritických aritmetických operácií nad  $\mathbb{GF}(2^{255} - 19)$ . V dôsledku toho sme vytvorili kompaktnú a rýchlu implementáciu Ristretto255 transformácie a zároveň aj OPAQUE protokolu v jazyku C, ktoré sú špeciálne optimalizované pre procesory s jadrom ARM Cortex-M4. Táto práca prispieva k rozšíreniu aplikovateľnosti autentifikačného protokolu OPAQUE pre vstavané systémy, ktoré disponujú obmedzenými požiadavkami na hardvér, najmä na tie s jadrom ARM Cortex-M4, čím zvyšuje jeho potenciál pre široké uplatnenie v moderných kryptografických aplikáciách. Táto práca poskytuje našu optimalizovanú implementáciu Ristretto255 transformácie a celého OPAQUE protokolu. Výsledky našich meraní sú k dispozícii v kapitole 7.

## Kľúčové slová

OPAQUE protokol, aPAKE, Ristretto255, ARM Cortex-M4, Embedded kryptografia, Mikrokontroléry

---

## **Bibliographic Citation**

ZELEŇÁK, Patrik. *Compact OPAQUE Protocol Implementation for Embedded Cryptographic Applications*. Košice: Technical University of Košice, Faculty of Electrical Engineering and Informatics, 2024. 82s. Supervisor: prof. Ing. Miloš Druťarovský, CSc.

**TECHNICAL UNIVERSITY OF KOŠICE**  
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATICS  
Department of Electronics and Multimedia Communications

# **DIPLOMA THESIS ASSIGNMENT**

Field of study: **Computer Science**  
Study programme: **Computer networks**

Thesis title:

## **Compact OPAQUE Protocol Implementation for Embedded Cryptographic Applications**

Kompaktná implementácia OPAQUE protokolu pre vstavané  
kryptografické aplikácie

Student: **Bc. Patrik Zeleňák**

Supervisor: **prof. Ing. Miloš Drutarovský, CSc.**

Supervising department: **Department of Electronics and Multimedia Communications**

Consultant:

Consultant's affiliation:

Thesis preparation instructions:

Analyze methods and algorithms used in the OPAQUE protocol used for secure authentication in a client/server communication. By using a suitable compact library for Curve25519 based elliptic curve cryptography and Ristretto255 group, implement in the C programming language, all of the proposed and required algorithms for complete OPAQUE protocol implementation. Use a PC platform and GCC compiler for implementation and testing of the developed OPAQUE protocol library. Demonstrate basic functionality of the library with a suitable demo application. The developed library must be written in ANSI C and must conform to the relevant RFC documents. Consider a possible future porting to the embedded platforms as the main design criteria. Analyze computational complexity of the main components of developed OPAQUE library and identify the time-critical functions suitable for an optimization in microcontrollers with the ARM Cortex M4 core.

Language of the thesis: English  
Thesis submission deadline: 19.04.2024  
Assigned on: 31.10.2023



*N. E. Pichlerová*  
.....  
prof. Ing. Liberios Vokorokos, PhD.  
Dean of the Faculty

## **Declaration**

I declare that I have independently prepared the master's thesis on the topic: "Compact OPAQUE Protocol Implementation for Embedded Cryptographic Applications" using the literature referenced under the professional guidance of prof. Ing. Miloš Drutarovský, CSc.

Košice, 19.4.2024

.....

*Signature*

## **Acknowledgment**

I would like to thank the supervisor of my masters's thesis, prof. Ing. Miloš Dru-tarovský, CSc., for his methodological guidance, willingness, and expert advice he provided me during the preparation of my master's thesis.



# Contents

---

<b>List of Abbreviations</b>	<b>1</b>
<b>List of Symbols</b>	<b>3</b>
<b>Introduction</b>	<b>4</b>
<b>Opaque Protocol Overview</b>	<b>7</b>
<b>1 OPAQUE Protocol Overview</b>	<b>7</b>
1.1 Offline Registration Phase . . . . .	8
1.2 Online Authentication Phase . . . . .	9
<b>Basic Cryptographic Building Blocks and Protocols Used in OPAQUE</b>	<b>11</b>
<b>2 Basic Cryptographic Building Blocks and Concepts Used in OPAQUE</b>	<b>11</b>
2.1 Group Theory Overview . . . . .	12
2.2 Modern Elliptic Curve25519 . . . . .	12
2.3 Ristretto255 Group and Transformation . . . . .	15
2.3.1 Encoding from Ristretto255 Group . . . . .	18
2.3.2 Decoding to Ristretto255 Group . . . . .	19
2.3.3 Hash to Ristretto255 Group . . . . .	19
2.4 Password-authenticated Key Exchange . . . . .	21
<b>Detailed Description of OPAQUE Phases</b>	<b>24</b>
<b>3 Detailed Description of OPAQUE Phases</b>	<b>24</b>
3.1 Client to Server Registration phase . . . . .	25
3.1.1 Oblivious Pseudorandom Function . . . . .	25
3.2 Client to Server Authentication Phase . . . . .	28
3.2.1 First AKE Message . . . . .	29
3.2.2 Second AKE Message . . . . .	30

3.2.3	Third AKE Message . . . . .	32
<b>Development Enviroment</b>		<b>36</b>
<b>4</b>	<b>Development Enviroment</b>	<b>36</b>
4.1	Development Platform . . . . .	36
4.1.1	Programming enviroment . . . . .	37
4.1.2	Python Prototype . . . . .	37
4.1.3	Embedded Platform . . . . .	37
4.2	QEMU for Big Endian Code on Little Endian Devices . . . . .	38
<b>Implementation Strategy of Ristretto255 Transformation</b>		<b>40</b>
<b>5</b>	<b>Implementation Strategy</b>	
	<b>of Ristretto255 Transformation</b>	<b>40</b>
5.1	Constant Time Approach . . . . .	40
5.2	Concept of Negative Elements Used $\mathbb{GF}(p)$ . . . . .	42
5.3	Secure Wiping of Local Variable . . . . .	43
5.4	Portable Endian Agnostic Code . . . . .	44
<b>Optimization Techniques for embedded platform</b>		<b>47</b>
<b>6</b>	<b>Optimization Techniques for embedded platform</b>	<b>47</b>
6.1	Available High-level Ristretto255 Libraries . . . . .	47
6.2	State of the Art Embedded C libraries for GF25519 . . . . .	48
6.2.1	TweetNaCl . . . . .	49
6.2.2	MonoCypher . . . . .	50
6.2.3	CycloneCRYPTO . . . . .	50
6.3	Function for Computation of Inverse Square Root . . . . .	51
6.4	Functions for ModL Arithmetic . . . . .	53
6.5	Minimizing Processor Stack Requirements via Shared Local Variables . . . . .	54
6.6	Approach to Using Existing Highly Optimized ASM Routines for $\mathbb{GF}(p)$ Operations . . . . .	56
6.7	Additional Optimalization Approaches . . . . .	57
<b>Experimental Results</b>		<b>58</b>
<b>7</b>	<b>Experimental Results</b>	<b>58</b>
7.1	Testing for Little Endian Platforms . . . . .	58

7.2	Testing Big Endian in QEMU . . . . .	59
7.3	Deep Testing of Ristretto255 . . . . .	61
7.4	Target ARM Cortex M4 platform/board . . . . .	64
	<b>Experimental Results</b>	<b>71</b>
8	<b>Discussion</b>	<b>71</b>
9	<b>Conclusion</b>	<b>74</b>
	<b>Bibliography</b>	<b>76</b>
	<b>List of Appendixes</b>	<b>83</b>
	<b>Appendix A</b>	<b>84</b>
	<b>Appendix B</b>	<b>88</b>
	<b>Appendix C</b>	<b>90</b>

# List of Figures

---

1.1	OPAQUE registration phase overview . . . . .	9
1.2	OPAQUE authentication phase overview . . . . .	10
2.1	Fundamental cryptographic building blocks used in OPAQUE . . .	11
2.2	Typical usage of RG255, where Curve25519 point in byte form is transformed into the Ristretto255 group. Subsequently, a curve point operation such as scalar multiplication of the curve point is performed, and then transformed back into a Curve25519 point represented in byte form within the prime subgroup $L$ . . . . .	17
3.1	OPAQUE offline registration: sequence of messages sent between client and server . . . . .	26
3.2	OPAQUE authentication: sequence of messages sent between client and server . . . . .	29
3.3	First AKE message generation: client-side flowchart. . . . .	29
3.4	Second AKE Message Generation: Server-Side Flowchart. . . . .	32
5.1	An example pseudocode showing usage of our endian-agnostic RG255 implementation . . . . .	46
6.1	Hierarchy of operations in the Ristretto255 Group supporting higher-level protocols, such as the OPAQUE protocol. . . . .	49
7.1	Terminal output of successful Ristretto255 tests performed on a PC platform running Windows 10 Pro 64-bit OS, operating on a little-endian architecture . . . . .	59
7.2	Terminal output of successful Ristretto255 tests performed on big-Endian platform running on Power-PC64 in QEMU emulator . . . .	60
1	Creating a new virtual machine in VirtualBox. . . . .	91
2	Selection of Debian 11 disk. . . . .	92

3	Adding new disk in VirtualBox. . . . .	92
4	Setting up a shared folder in VirtualBox. . . . .	93
5	Showcase of Tmux terminal. . . . .	96
6	Showcase of debugger for PowerPc64. . . . .	98
7	First terminal window after debugging is finished on second window. . . . .	99

# List of Tables

---

5.1	Comparison of String with Reference and Time Measurement . . .	41
6.1	Comparison of the implementation of $x^{(p-5)/8}$ across various cryptographic libraries . . . . .	52
7.1	Speed measurements for the most critical functions operating over $\mathbb{GF}(2^{255} - 19)$ in various libraries. These libraries employ distinct $\mathbb{GF}$ element representations, and the results are presented in cycles. Measurements were performed on the ARM Cortex-M4, with 168 MHz clock frequency, using the GCC ARM compiler with the -Os flag set. . . . .	65
7.2	Continuation of speed measurements from Table 7.1. for the most critical functions operating over $\mathbb{GF}(2^{255} - 19)$ in various libraries. Measurements were performed on the ARM Cortex-M4, with 168 MHz clock frequency, using the GCC ARM compiler with the -Os flag set. . . . .	65
7.3	Memory usage for functions operating within the modulo $(2^{256} - 36)$ domain in highly-efficient ASM implementation. The memory requirements of each function are presented in bytes. . . . .	66
7.4	Continuation of Table 7.3 for memory usage of functions operating within the modulo $(2^{256} - 36)$ domain in highly-efficient ASM implementation. The memory requirements of each function are presented in bytes. . . . .	66
7.5	Speed for InvModL, including supporting functions and various optimization techniques. Measurements were performed on the ARM Cortex-M4, with 168 MHz clock frequency, using the GCC ARM compiler with the -Os and -O3 flag set. . . . .	66

7.6	Speed for RG255 core functions and InvModL. The results are presented in cycles. The measurements were performed on the ARM Cortex-M4, with 168 MHz clock frequency, using the GCC ARM compiler, with the -Os and -O3 flag set. . . . .	67
7.7	Memory usage for RG255 functions implemented in pure C. The memory requirements of each function are presented in bytes. . . .	67
7.8	Speed of the OPAQUE registration and authentication phases were performed on the client-side, using our optimization techniques, including a combination of fast C and efficient ASM GF operations. The results are presented in cycles. Measurements were performed on the ARM Cortex-M4, with 168 MHz clock frequency, using the GCC ARM compiler with the -Os flag. . . . .	68
7.9	Speed of the OPAQUE registration and authentication phases were performed on the client-side, using our optimization techniques, including a combination of fast C and efficient ASM GF operations. The results are presented in cycles. Measurements were performed on the ARM Cortex-M4, with 168 MHz clock frequency, using the GCC ARM compiler with the -O3 flag. . . . .	68
7.10	Extension of Table 7.8 shows speed measurements for each OPAQUE message generated on the client-side during both registration and authentication phases, utilizing our optimization techniques including a combination of fast C and highly efficient ASM GF operations. The results are presented in cycles. Measurements were performed on the ARM Cortex-M4, with 168 MHz clock frequency, using the GCC ARM compiler with the -Os flag set. . . . .	69
7.11	Extension of Table 7.8 shows speed measurements for each OPAQUE message generated on the client-side during both registration and authentication phases, utilizing our optimization techniques including a combination of fast C and highly efficient ASM GF operations. The results are presented in cycles. Measurements were performed on the ARM Cortex-M4, with 168 MHz clock frequency, using the GCC ARM compiler with the -O3 flag set. . . . .	69
7.12	Memory requirements of the OPAQUE messages generated and exchanged between the client and server, measured in bytes. . . . .	70
8.1	Measured Clock Frequencies of ARM Cores . . . . .	72
8.2	Comparison of implementations on CPUs with different ARM cores	72

8.3	The processor memory stack requirements for the OPAQUE functions used during the generation of OPAQUE messages in both the registration and authentication phases on the client-side were measured in bytes. These measurements were performed on functions implemented in pure C. . . . .	73
1	Debugger Commands and Descriptions . . . . .	97



# List of Abbreviations

---

**AKE** Authenticated Key Exchange.

**aPAKE** Asymmetric Password-Authenticated Key Exchange.

**ARM** Advanced RISC Machines.

**ASM** Assembly Language.

**CPU** Central Processing Unit.

**DWT CYCCNT** Data Watchpoint and Trace Cycle Counter.

**ECC** Elliptic Curve Cryptography.

**GPU** Graphics Processing Unit.

**HKDF** HMAC-based Extract-and-Expand Key Derivation Function.

**HMAC** Keyed-Hash Message Authentication Code.

**HTTPS** Hypertext Transfer Protocol Secure.

**IoT** Internet of Things.

**ITM** Instrumentation Trace Macrocell.

**JWT** JSON Web Token.

**KDF** Key Derivation Function.

**KSF** Key Stretching Function.

**LSB** Least Significant Bit.

**MAC** Message Authentication Code.

**MCU** Microcontroller Units.

**n-POG** Non-Prime-Order Group.

**NIST** National Institute of Standards and Technology.

**OPRF** Oblivious Pseudorandom Function.

**OS** Operating System.

**PAKE** Password Authenticated Key Exchange.

**PC** Personal Computer.

**PKI** Public Key Infrastructure.

**PLL** Phase-Locked Loop.

**POG** Prime-Order Group.

**PRF** Pseudorandom Function.

**RAM** Random Access Memory.

**RG255** Ristretto255 Group.

**RNG** Random Number Generator.

**SHA** Secure Hash Algorithm.

**SLL** Secure Sockets Layer.

**SRP** Secure Remote Password.

**TLS** Transport Layer Security.

**TRNG** True Pseudorandom Number Generator.

**USB** Universal Serial Bus.

**VPN** Virtual Private Network.

**X3DH** Extended Triple Diffie-Hellman.

**ZKP** Zero Knowledge Proof.

# List of Symbols

---

$\parallel$	Byte concatenation
$\oplus$	Binary XOR operation
$\mathbb{GF}$	Galois Field
$Ea, b :$	Elliptic curve with parameters $a, b$
$\mathbb{F}$	Field of odd characteristic
$:=$	Assignment symbol in pseudocode
$\leftarrow$	Assignment symbol in algorithm
$*$	Curve point multiplication by scalar
$ a $	Absolute value of number $a$
$\sim$	Negation
$\gg$	Right shift operator
$\ll$	Left shift operator
$\&$	Bitwise and operator

# Introduction

---

In the digital world, we use passwords to prove who we are when accessing our online accounts. However, this authentication system has problems. People have been using secret phrases or symbols to gain access to protected things for a long time. In the digital age, we started using passwords to protect our computer and online accounts. Traditional passwords have many issues. They are often easy to guess, many people use the same password for multiple accounts, and hackers can guess passwords or trick people into revealing them. As digital threats have become more advanced, the problems with traditional passwords have become more obvious. We need better ways to keep our accounts safe. That's where solutions like OPAQUE protocol (hereafter referred to as OPAQUE) [1] come in. They use mathematical techniques to prove our identity without revealing our actual password, they make sure our information is safe as it travels online, and they protect our privacy by hiding our passwords from the services we use. In summary, traditional passwords have problems, and we need better ways to keep our online accounts safe. Advanced solutions like OPAQUE offer a safer and more private way to prove our identity online. There are couple of reasons why we should consider OPAQUE when establishing secure authentication process. One of the reasons you might prefer OPAQUE over technologies like JWT (JSON Web Tokens) [2] or similar authentication mechanisms is the need for a more secure channel, especially in certain contexts such as embedded systems. It is known that JWTs are typically used for representing claims in a compact and self-contained way, and they are often used in combination with HTTPS (TLS/SLL) to secure the communication between the client and server. While JWTs themselves do not establish secure channels, they rely on the underlying transport layer security provided by HTTPS. Using PKI (Public Key Infrastructure) would be overkill from a performance perspective for embedded systems.

Clearly, we need something more lightweight. There are multiple methods of authentication processes. Some are simple and fast, while others are more secure. All of them have pros and cons. They are susceptible to eavesdropping,

potentially exposing useful information to attackers, whether it is a salt or, in a worst-case scenario, plaintext credentials.

OPAQUE is designed to provide secure channel establishment during the authentication process. It ensures that the communication between the user and the server is encrypted and protected from eavesdropping. This is particularly important for maintaining the confidentiality of user credentials during the authentication process. OPAQUE is specifically designed to protect user password privacy. It ensures that the server never learns the actual password during the authentication process, making it a preferred choice when privacy is a concern. Also, OPAQUE ensures that the server never sends its private information through a potentially insecure communication channel, making it impossible for a threat actor to eavesdrop on any useful information. OPAQUE is well-suited for embedded systems and scenarios where secure authentication might be challenging due to resource constraints.

In this thesis, we contribute with our own implementation of OPAQUE based on elliptic curve cryptography (ECC). We chose the elliptic curve Curve25519 for its speed and security aspects. It is well-known that incorporating Curve25519, a non-prime order group curve, into protocols relying on prime order group elliptic curves requires some adjustments due to a cofactor greater than 1. Following the official OPAQUE configuration stack, we used Ristretto255, specifically designed to resolve the cofactor problem and provide high-speed abstraction for mapping points from non-prime order groups into prime order groups.

This work will be structured in the following way:

The first chapter of this thesis briefly introduces the OPAQUE protocol, its purposes, and its usage in real-world applications. This chapter also includes essential cryptographic primitives and techniques used in OPAQUE.

In the second chapter, we describe the concept of password-authenticated key exchange, a non-prime-order elliptic curve (Curve25519), Ristretto255 transformation, and its importance in cryptographic protocols based on prime-order curves.

The third chapter focuses on a detailed description of OPAQUE, particularly the offline registration phase and the client-to-server authentication phase.

The fourth chapter describes the development environment and tools used during the development and testing of our implementation of the Ristretto255 transformation and subsequently the OPAQUE protocol.

Chapter five is dedicated to describing implementation strategies used in our implementations of the OPAQUE protocol and Ristretto255 transformation, es-

pecially approaches like constant-time, portable endian agnostic, buffer wiping, and a method of big-endian testing in the QEMU emulator.

In the sixth chapter, we describe optimization techniques employed in our implementations targeting embedded systems. We followed an official standards and focused on integrating a variety of implementation and optimization techniques, which have been successfully included in our library.

Finally, the seventh chapter describes the experimental results obtained by testing various parts of our implementation, focusing on achieving bit-exact results and performance measurements.

# 1 OPAQUE Protocol Overview

---

Authentication via passwords is a commonly employed method across various applications. Typically, a user sends their identification and password to a server through a secure channel. Nevertheless, this approach poses risks such as potential servers mishandling or data breaches. Additionally, even when passwords are transmitted through secure channels like TLS [3], vulnerabilities to attacks or malfunctions persist.

OPAQUE [1] stands as a first secure Asymmetric Password Authenticated Key Exchange (aPAKE) protocol that operates independently of PKI and is resilient against pre-computation attacks. It offers forward secrecy, safeguards passwords from server exposure, and allows users to enhance protection against offline dictionary attacks. An aPAKE [4] protocols are designed to authenticate passwords securely and facilitate key exchange without relying on PKI or exposing passwords to servers. A secure aPAKE should provide optimal security, acknowledging that certain attacks are inevitable.

OPAQUE operates in two phases, registration and authenticated key exchange phase also known as login stage. During registration, a user registers their password with the server and preserves information necessary for credential recovery, accessible only to those who possess the password. Subsequently, users leverage their passwords to acquire these credentials and then uses them within an AKE [5] protocol in login stage.

Furthermore, OPAQUE is adaptable, allowing users to store and access application data on servers solely through their passwords, which is side product of OPAQUE registration phase. OPAQUE functions as a "compiler" that transforms any suitable AKE protocol into a secure aPAKE protocol. In this thesis we employed variation of OPAQUE that is based on X3DH [6], however alternative configurations are available [1].

OPAQUE contains two fundamental components, an oblivious pseudorandom function (OPRF) [7] and an AKE protocol. OPAQUE relies on various cryptographic primitives, elaborated upon:

- Oblivious Pseudorandom Function: A method enabling users to compute a pseudorandom function without disclosing the input to the server [7].
- Key Derivation Function: A mechanism for generating encryption keys from a shared secret, such as a password [8].
- Message Authentication Code: A technique ensuring message integrity and authenticity during exchange between parties, for example HMAC [9].
- Cryptographic Hash Function: A function converting data into a fixed-size output, ensuring data integrity and irreversible transformation [10].
- Memory-Hard Function: A function requiring significant memory for computation, discouraging attackers from brute-force attempts. A good example is password-hashing function designed specifically for resistance against side-channel attacks and GPU cracking called Argon2 [11].
- Authenticated Key Exchange protocol: A procedure enabling two parties to establish a shared encryption key securely while authenticating each other [5].

## 1.1 Offline Registration Phase

During registration phase (referred to as the "offline registration stage" in the specification) in OPAQUE, a secure communication channel between the user and the server is required. This can be done through a physical connection, an out-of-band method, or one based on Public Key Infrastructure (PKI). Note that this is the only phase in OPAQUE that requires a server-authenticated channel that provides confidentiality and integrity. Registration starts on the client-side by blinding a client's password and sending it to the server. The server then signs that blinded password using an *opr<sub>f</sub>\_seed* (also referred to in other literature as a seed) and sends it back to the client. It's important to note that the *opr<sub>f</sub>\_seed* and the server's public and private keys (referred to as "general" keys later in this thesis) are generated prior to the OPAQUE registration phase. Subsequently, the client unblinds the signed password and uses it to generate its general public key (note that client's general key pair is not stored and needs to be recovered later in authentication phase), masking key, and a special structure called an envelope, which is used later during client's credentials recovery in the online authentication process [1]. A masking key is deterministically generated from the



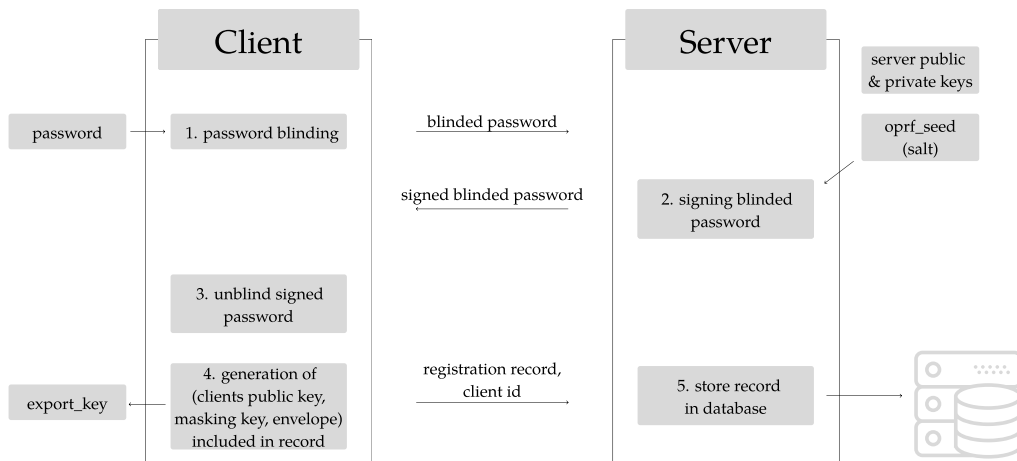


Figure 1.1: OPAQUE registration phase overview

unblinded signed password and is used by server later in the login phase to encrypt an envelope. These three parameters make up a registration record, which is then sent to the server along with the client’s ID, as depicted in Figure 1.1. The user also generates an *export\_key* as a side-product of registration phase, usable for application specific tasks like encrypting additional data for storage on the server, inaccessible to the server. The server obtains a record related to the user’s registration, stores it in a database alongside other user information as required. More information about OPAQUE registration is detailed in the specification [1] and explained in Chapter 3.

## 1.2 Online Authentication Phase

During the online authentication phase, clients retrieve their previously registered credentials from the server, recover their private key by utilizing their password, and subsequently use this key in the Authenticated Key Exchange (AKE) protocol (see Figure 1.2).

The authentication phase begins similarly to the registration phase. The client uses its password to compute a blinded password and sends it to the server. However, in this phase, clients also generate their private and public keys for the Triple Diffie-Hellman (X3DH) protocol [6], which is used to generate a shared secret session key. We have chosen X3DH as the AKE method in accordance with the OPAQUE specification [1], although other options are available as outlined in the specification.

Once receiving the blinded password from the client, the server performs a signing operation using the *opr\_f\_seed*, similar to the registration phase. Additionally, the server generates a *masked\_response*, which serves as an encrypted

envelope, and its own set of X3DH key pair (server public and private X3DH keys). These (except server public X3DH key) are then sent to the client in a single message. Before sending, the server also generates a session key based on all available keys (client's public general and X3DH keys, as well as its own server general key pair and server X3DH key pair). When client receives a message from the server containing the signed blinded password, server general public key, and server X3DH public key, the client unblinds the signed password. Subsequently, it uses this unblinded password to recover an envelope from the *masked\_response*. Additionally, an *export\_key* (similar to that used in the registration phase) is generated. Once the envelope is successfully recovered, the client uses it to retrieve its general public key and private key pair. Similar to the server, the client generates a session key based on all available keys. Think of *session\_key* as classic symmetric key that is used to encrypt communication between two parties. It is worth noting that OPAQUE provides forward secrecy, meaning that every successful authentication produces a different session key, however the *export\_key* remains the same [1]. Please bear in mind that these phases are simplified versions of the OPAQUE registration and authentication phases. More detailed information is provided in the specification [1] and explained in Chapter 3.

OPAQUE presents a valuable building block for applications where secure password-based authentication and key exchange is crucial. It effectively addresses numerous concerns, encompassing the safeguarding of user passwords against server mismanagement, mitigation of offline dictionary attacks and assurance of forward secrecy. Several domains where OPAQUE has potential to offer significant advantages include online services and websites, internet of things (IoT) devices, mobile applications, secure messaging platforms, remote access and virtual private networks (VPNs), external cloud storage and many more [12].

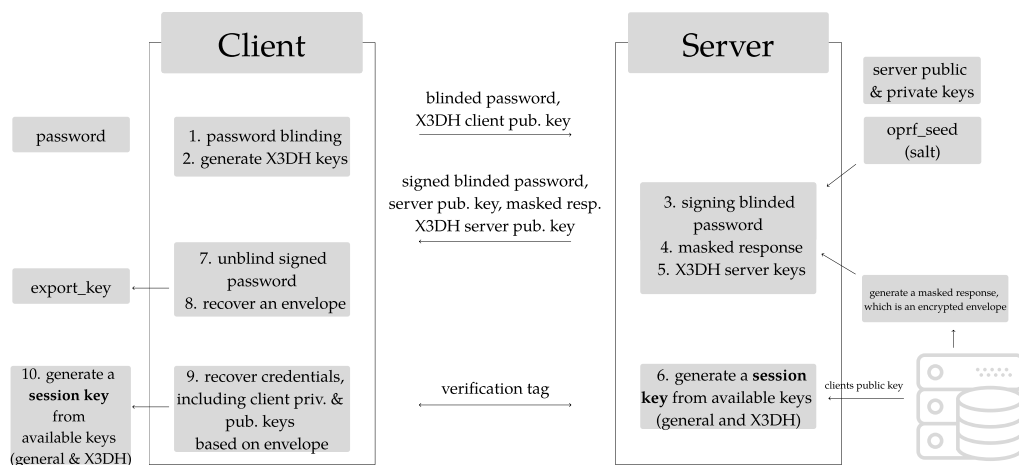


Figure 1.2: OPAQUE authentication phase overview

## 2 Basic Cryptographic Building Blocks and Concepts Used in OPAQUE

---

This section introduces the fundamental cryptographic building blocks and concepts essential to the OPAQUE protocol [1], as depicted in the pyramid diagram shown in Figure 2.1. The OPAQUE protocol forms the highest layer of this pyramid, built upon foundational blocks such as Galois Field ( $\mathbb{GF}$ ) arithmetic, elliptic curve cryptography (specifically Curve25519 in our implementation [13]), and the Ristretto255 transformation [14]. All the blocks depicted in this pyramid are described in this chapter and also implemented in our library, available in the Appendix A. OPAQUE is briefly introduced in Section 1 and further elaborated upon in Section 3, which serves as supplementary material to the official RFC specification [1]. This supplementary material aims to enhance the reader's understanding of the concepts of OPAQUE.

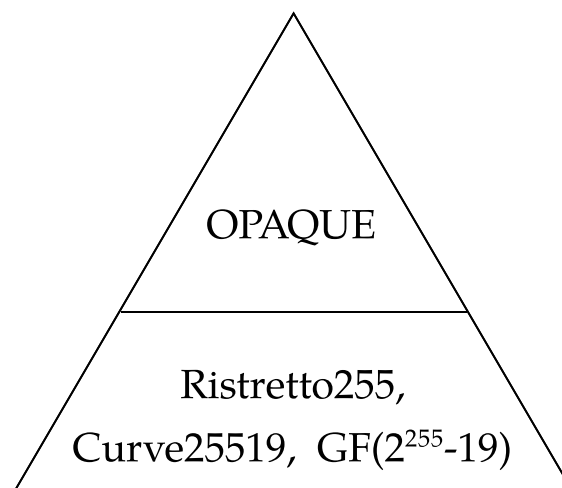


Figure 2.1: Fundamental cryptographic building blocks used in OPAQUE

The primary focus of this section is to provide a concise overview of three key components: the elliptic curve Curve25519 [13] the Ristretto255 transformation [14] and password authenticated key exchange [4].

## 2.1 Group Theory Overview

In this section, we will frequently refer to group. Therefore, we will provide a brief explanation of what a group actually is. A group is a mathematical structure that consists of a set of elements along with a binary operation. Combining any two elements with binary operation produce a third element in the set. Structure must satisfy certain properties to be considered a group [15]. These properties are:

- **Closure:** For any two elements  $a$  and  $b$  in the group, their combination using the group operation (denoted as  $\cdot$ , i.e.,  $a \cdot b$ ) must also belong to the same set.
- **Associativity:** The group operation is associative, meaning for all elements  $a, b$ , and  $c$  in the group,  $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ .
- **Identity Element:** There exists an identity element  $e$  in the group such that for any element  $a$  in the group,  $a \cdot e = e \cdot a = a$ .
- **Inverse Element:** For each element  $a$  in the group, there exists an inverse element  $a^{-1}$  in the group such that  $a \cdot a^{-1} = a^{-1} \cdot a = e$ , where  $e$  is the identity element.

An Abelian group, also known as a commutative group, is a special type of group where the group operation is commutative. Meaning, for all elements  $a, b$  we have  $a \cdot b = b \cdot a$  [16].

One important concept in mathematics is group addition. Addition is a group operation, characterized by an identity element and prime order (in our case prime number  $L$ ). Adding the same element  $L$ -times results in the identity element. Adding the identity element to any other element results in the unchanged element. Negating an element results in a value that, when added to its original, gives the identity element. Subtraction can be viewed as adding the negation of an element, and scalar multiplication involves repeatedly adding the same element.

## 2.2 Modern Elliptic Curve

In the field of cryptography, where keeping information secure is crucial, the choice of mathematical structures and cryptographic protocols plays a significant role in ensuring the safety of digital communications.

Elliptic Curve Cryptography (ECC) is a modern and widely-used encryption technique employed in securing communication, relying on operations with points on elliptic curves. Many different types of elliptic curves are utilized in ECC, such as NIST [17], Bernstein's [13], Edward's [18], and more. These curves vary in properties like the order of the curve, the cofactor, the shape of the curve, and their computational complexity (speed) and security. While the NIST curves work with groups that have prime order [19], not all of them are considered trustworthy [20].

A common building block in modern cryptography is prime-order groups (POGs), which form the foundation for important cryptographic algorithms and protocols. POGs are widely used in various cryptographic protocols, like the Zero-knowledge proof [21], PAKE and more. Prime-order groups (POGs) facilitate secure key exchange, ensuring that communication between entities remains safeguarded from unauthorized access. The mathematical foundation of POGs enhances resistance against various cryptographic attacks, including discrete logarithm and factorization, thereby strengthening the overall security of cryptographic systems.

However, in practice, many implementations either choose to use non-prime order groups (n-POGs) for efficiency reasons, or they can be manipulated to operate in n-POGs. If we want these implementations to be applied in a protocol that uses POG, a problem arises. One example is the popular Bernstein elliptic curve (EC) Curve25519 [13], which has a non-prime order group (n-POG). To enable the utilization of this curve in protocols designed for prime order groups (POGs), a transformation capable of converting a group with non-prime order to one with prime order is necessary.

For this reason, it makes sense to consider the highly efficient Curve25519, even though it does not have a cofactor equal to 1 and is not a prime-order group.

The elliptic curve Curve25519, designed by mathematician and cryptographer Daniel J. Bernstein, has garnered significant attention in the realm of cryptography [13]. The emphasis in the design of Curve25519 is strongly placed on security, providing resilient defence against various cryptographic attacks. Curve25519 provides a cryptographic security level of 128 bits. This positioning establishes Curve25519 as a crucial element in ensuring the security of communication protocols, such as TLS 1.3 [22]. It is also applied for encryption in WhatsApp [23], Signal [24], and numerous other systems and protocols [16]. The Curve25519 curve exhibits excellent properties in terms of speed, security and implementation simplicity. Compared to other elliptic curves, Curve25519 has several advantages. It

is designed to be resistant to side-channel attacks on elliptic curves. Additionally, it is very fast, and operations on this curve can be optimized (e.g., scalar multiplication using the Montgomery ladder[16]).

Curve25519 is an elliptic curve defined over the finite field  $\mathbb{GF}(p)$ , where  $p$  is a prime number given by  $2^{255} - 19$ , hence the name of the elliptic curve. It's worth noting that the number  $2^{255} - 19$  is the largest prime number that can fit into 256 bits. The Curve25519 can be described using the Montgomery equation [18] as follows:

$$y^2 = x^3 + 48662x^2 + x \quad (2.1)$$

Montgomery representation allows for the efficient implementation of various operations on the curve, including point addition, subtraction, or scalar multiplication of an elliptic curve point. Importantly, it offers a high level of security for asymmetric public key encryption. Additionally, it facilitates the rapid computation of public and private keys through an algorithm based on multiplying a point on the curve by a scalar.

The mathematical operations on the Curve25519 can be implemented with less code and on less powerful devices. A proof of this is the cryptographic library TweetNaCl[25]. TweetNaCl is a library based on the publicly available NaCl library (Network Communication, Cryptography, and Security library). The implementation of the TweetNaCl library is very compact, and after compression with a Python script, the size of the entire library is smaller than 100 tweets. Curve25519 is used in various cryptographic protocols, including Diffie-Hellman key exchange, digital signatures, and encryption. For more information about TweetNaCl, see chapter 6.2.1.

Curve25519 parameters are deliberately selected so that the order of the group  $n$  (representing the number of all points on the curve) is the product of the cofactor  $c$  and the order of the prime-order subgroup  $L$ :

$$n = c \cdot L \quad (2.2)$$

where  $L = 2^{252} + 27742317777372353535851937790883648493$  and cofactor  $c$  is equal to 8.

The cofactor  $c$  in elliptic curve cryptography (ECC) denotes the ratio between the total number of points on the EC group (the order) and the order of a specific subgroup within that curve. It essentially represents the ratio between the overall number of points on the EC and the points in a smaller subgroup [16]. The choice of the cofactor  $c$  is significant in ECC as it has implications for the security of the

cryptographic system. Opting for an elliptic curve with a cofactor  $c$  equal to 1 is desirable as it minimizes the risk of certain types of attacks. Elliptic curves with  $c > 1$  may introduce vulnerabilities, making it generally recommended to choose elliptic curves with  $c = 1$  in cryptographic applications.

This is the most significant drawback of Curve25519 because it is designed to use a cofactor  $c = 8$ . In some applications, computational complexity could be increased due to the need to work with elements that are not part of the main subgroup. This may negatively impact the speed and efficiency of certain cryptographic operations. In other applications, additional modifications are made to maintain security. However, there is not an universal solution that can be applied to all applications. In many existing protocols, the complexity of managing this abstraction is increased through ad-hoc protocol modifications[16]. However, these modifications often become a recurring source of vulnerabilities and subtle complications in design, usually preventing the application of security proofs for the abstract protocol. The cofactor 8 issue can be efficiently addressed by Ristretto255 transformation.

## **2.3 Ristretto255 Group and Transformation**

Ristretto255[14] is a technique used to construct an abstract POG group of elliptic curves of prime order. Ristretto255 enables existing libraries containing protocols based on n-POG like the elliptic curve Curve25519 to implement a prime-order group with fast, thin high-level abstraction.

The Ristretto255 transformation (RG255) is a modern cryptographic transformation introduced in 2019 [14]. It extends the Decaf approach proposed by the author M. Hamburg[26]. RG255 is utilized to map EC points from the Curve25519 group, which has a n-POG, to a POG. The goal of this transformation is to enable the use of efficient Curve25519 in cryptographic protocols that require a POG, preserving the security and efficiency of Curve25519.

Ristretto255 is a fast thin layer transformation that unambiguously maps each point on Curve25519 to a Ristretto representation, allowing the creation of an abstract interface for a POG using the points of the elliptic Curve25519.

An Ristretto255 element is an abstract element of a POG, and its encoding is a unique, reversible representation of a group element. In the context of group element encoding, this implies that if an element is encoded in a specific form, it can be decoded back to its original form using a reversible encoding process. The internal representation is understood to be the point on Curve25519. Each

element of a group may have multiple equivalent internal representations. The order of the abstract group formed is the same as the order of the subgroup of Curve25519, with a value of  $L$ .

By creating an abstraction of a prime group, Ristretto255 ensures that all elements of that group, except the neutral element, are generators. However, for interoperability, a canonical generator was chosen, which can serve as an internal representation of the base point of the Curve25519. Note, that in the context of RG255, terms canonical and non-canonical simply mean less than  $2^{255} - 19$  or greater than or equal to  $2^{255} - 19$  respectively. The encoded form of the canonical generator  $g$  2.3 is as follows:

$$g = \text{e2f2ae0a 6abc4e71 a884a961 c500515f} \\ \text{58e30b6a a582dd8d b6a65945 e08d2d76} \quad (2.3)$$

The Ristretto255 group is an additive abelian group designed primarily to securely perform operations on a curve, such as addition and scalar multiplication, with minimal overhead. Consequently, before any curve operation, the initial step involves mapping byte strings into Ristretto255 points, executing operations like point addition, and subsequently mapping back to byte strings. Internally, a Ristretto point is represented by an Edwards point, typically in extended twisted Edwards coordinates [27]. It's worth noting that two EC points in extended twisted Edwards coordinates,  $P$  and  $Q$ , may represent the same Ristretto255 point, similar to how different projective  $x, y, z, t$  coordinates may represent the same Edwards point [28].

Edwards curves are a special type of elliptic curves is that they are birationally equivalent to elliptic curves (Weierstrass elliptic curves) [29]. Edwards curves have several advantages for cryptographic purposes, including complete [18] addition formulas, which simplifies operations like point addition and doubling compared to other elliptic curve forms. The general form of an Edwards curve [30] can also be written as:

$$E_{a,d} : ax^2 + y^2 = 1 + dx^2y^2 \quad (2.4)$$

where  $a, d \in \mathbb{K}$  with  $ad(a - d) \neq 0$ . The  $\mathbb{K}$  is a field of odd characteristic [29]. Edwards curves are a particular type of twisted Edwards curve, in which the parameter  $a$  can be adjusted or rescaled to equal 1.



Affine addition formulae for twisted Edwards curves is defined by an equation 2.5.

$$(x_1, y_1) + (x_2, y_2) = \left( \frac{x_1y_2 + y_1x_2}{1 + dx_1y_1x_2y_2}, \frac{y_1y_2 - ax_1x_2}{1 - dx_1y_1x_2y_2} \right) = (x_3, y_3). \quad (2.5)$$

More information about Twisted Edwards curve can be found in paper [18] and [29].

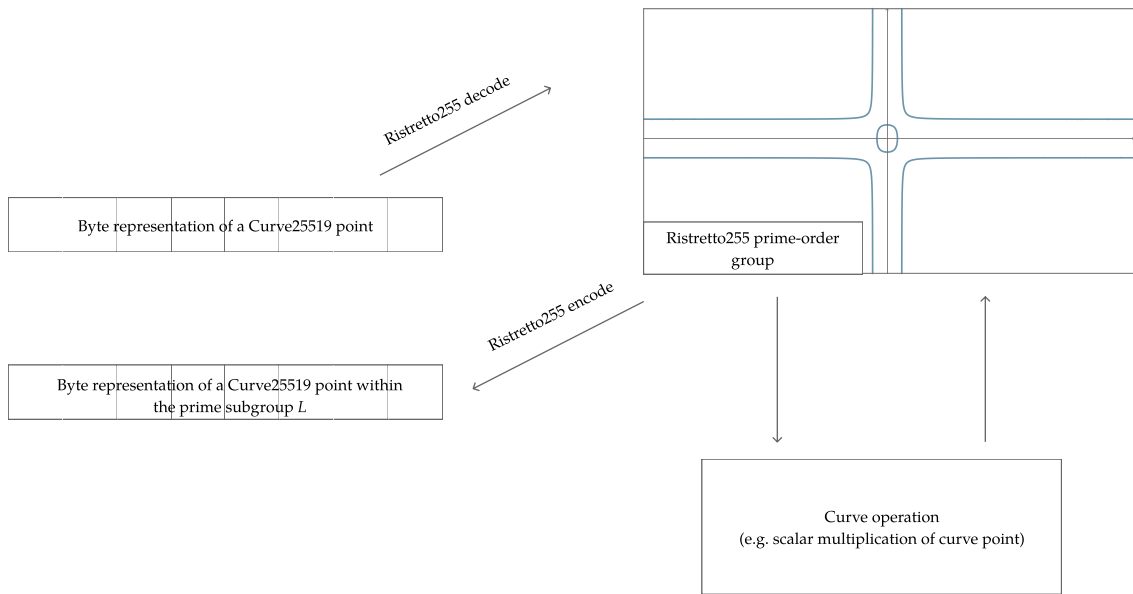


Figure 2.2: Typical usage of RG255, where Curve25519 point in byte form is transformed into the Ristretto255 group. Subsequently, a curve point operation such as scalar multiplication of the curve point is performed, and then transformed back into a Curve25519 point represented in byte form within the prime subgroup  $L$

Algorithm 2 illustrates the decoding process, which is the mapping from byte strings to Ristretto255 points. Similarly, encoding[14] can be utilized to map the Ristretto255 point back to the byte string. All underlying steps in Algorithms 1, 2, 4 and 3 are executed over  $\mathbb{GF}(2^{255} - 19)$ , including internal functions like the inverse square root [28], denoted as  $\text{INV\_SQRT}()$  [26]. Note that  $\mathbb{GF}(2^{255} - 19)$  arithmetic is interesting because it is specifically designed to perform arithmetic operations over  $\mathbb{GF}$  efficiently. Such efficient approaches utilized in many fast cryptographic libraries like TweetNaCl (see Section 6.2.1), or in implementation [31]. There are various approaches to implementing the inverse square root [32][31]. During the development of our Ristretto255 group implementation, we found inspiration in the CycloneCRYPTO [31] approach to the inverse square root

computation. Our implementation incorporates additional adjustments with the goal of reducing the number of utilized variables, thus enhancing the efficiency of stack manipulation [33]. The operations within the abstract Ristretto255 group include encoding, decoding, equality verification, one-way mapping, along with two operations on Curve25519, an addition and scalar multiplication. This section provides a brief overview of chosen operations.

### 2.3.1 Encoding from Ristretto255 Group

Encoding is a function from abstract elements to byte strings. Internally, an abstract element  $s$  might have more than one possible representation, for example the implementation might use projective coordinates [29]. When encoding, all equivalent representations of the same element are encoded as identical byte strings. Decoding the output of the encoding function always succeeds and returns an element equivalent to the encoding input [14]. Pseudocode 1, defined in the official Ristretto255 specification [14], internally uses constants that are marked as bold (**SQRT\_M1**, **INVSQRT\_A\_MINUS\_D**, **ONE\_MINUS\_D\_SQ**, etc.) in Algorithms 1, 2, 4 and 3, which are also defined in the same specification [14].

---

#### Algorithm 1 Ristretto255 Encode

---

**Input:**  $(x_0, y_0, z_0, t_0)$  representing a group element on Curve25519  
**Output:**  $s$  as the encoded as 256-bits field element in abstract group

- 1:  $u_1 \leftarrow (z_0 + y_0) \cdot (z_0 - y_0)$
- 2:  $u_2 \leftarrow x_0 \cdot y_0$
- 3:  $(\_, invsqrt) \leftarrow INV\_SQRT(1, u_1 \cdot u_2^2)$
- 4:  $den_1 \leftarrow invsqrt \cdot u_1$
- 5:  $den_2 \leftarrow invsqrt \cdot u_2$
- 6:  $z\_inv \leftarrow den_1 \cdot den_2 \cdot t_0$
- 7:  $ix_0 \leftarrow x_0 \cdot \mathbf{SQRT\_M1}$
- 8:  $iy_0 \leftarrow y_0 \cdot \mathbf{SQRT\_M1}$
- 9:  $e\_den \leftarrow den_1 \cdot \mathbf{INVSQRT\_A\_MINUS\_D}$
- 10:  $rotate \leftarrow IS\_NEGATIVE(t_0 \cdot z\_inv)$
- 11:  $x \leftarrow CT\_SELECT(iy_0 \text{ if rotate else } x_0)$
- 12:  $y \leftarrow CT\_SELECT(ix_0 \text{ if rotate else } y_0)$
- 13:  $z \leftarrow z_0$
- 14:  $den\_inv \leftarrow CT\_SELECT(e\_den \text{ if rotate else } den_2)$
- 15:  $rotate \leftarrow IS\_NEGATIVE(x \cdot z\_inv)$
- 16:  $y \leftarrow CT\_SELECT(-y \text{ if rotate else } y)$
- 17:  $s \leftarrow CT\_ABS(den\_inv \cdot (z - y))$
- 18: **return**  $s$

---

### 2.3.2 Decoding to Ristretto255 Group

In cryptography, it is common to use a byte string to represent a cryptographic object, whether it is a private or public key, which forms the core of asymmetric encryption. Therefore, we need some kind of conversion tool that helps us convert between the byte string representation of an element and a point on the curve. In Ristretto255, this is achieved through decoding and encoding functions. Decoding is a function that converts byte strings  $s$  to abstract elements (point on the curve) with built-in validation, ensuring that only the canonical encodings of valid elements are accepted. This built-in validation eliminates the need for explicit invalid curve checks [14].

---

#### Algorithm 2 Ristretto255 Decoding

---

**Input:**  $s, a$  the encoded as 256-bits field element in abstract group

**Output:**  $(x, y, 1, t)$  representing the group element on Curve25519 after decoding

```

1: if  $IS\_NEGATIVE(s)$  then
2:   return 1 {Decoding fails}
3: end if
4:  $ss \leftarrow s^2$ 
5:  $u_1 \leftarrow 1 - ss$ 
6:  $u_2 \leftarrow 1 + ss$ 
7:  $u_{2\_sqr} \leftarrow u_2^2$ 
8:  $v \leftarrow -(\mathbf{D} \cdot u_1^2) - u_{2\_sqr}$ 
9:  $(was\_square, invsqrt) \leftarrow INV\_SQRT(1, v \cdot u_{2\_sqr})$ 
10:  $den\_x \leftarrow invsqrt \cdot u_2$ 
11:  $den\_y \leftarrow invsqrt \cdot den\_x \cdot v$ 
12:  $x \leftarrow CT\_ABS(2 \cdot s \cdot den\_x)$ 
13:  $y \leftarrow u_1 \cdot den\_y$ 
14:  $t \leftarrow x \cdot y$ 
15: if  $was\_square = 0$  then
16:   return 1 {Decoding fails}
17: else if  $IS\_NEGATIVE(t)$  then
18:   return 1 {Decoding fails}
19: else if  $y = 0$  then
20:   return 1 {Decoding fails}
21: end if
22: return  $(x, y, 1, t)$ 

```

---

### 2.3.3 Hash to Ristretto255 Group

The element derivation function maps deterministically from byte strings of a fixed length to abstract elements. In practice this can be used to map hash digest

of user's password to point on the curve. The element derivation function has two important properties. First, if the input is a uniformly random byte string, then the output is (within a negligible statistical probability) a uniformly random abstract group element. This means the function is suitable for selecting random group elements [14].

To obtain such an input from an arbitrary-length byte string, a hash function such as SHA512 can be used. Subsequently, the uniformly distributed 64-byte string is divided into two halves. The first half (the first 32 bytes of the 64-byte input string) is processed by the MAP function presented in Algorithm 3, which maps the 32-byte string into point P1. The same process is performed on the second half (the last 32 bytes of the 64-byte input string), resulting in point P2.

Finally, point addition  $P1 + P2$  is performed, which is the output value of the element derivation function (see Algorithm 4). It's worth noting that both halves need to be masked. Masking is performed on the most significant bit, which is equivalent to interpreting the whole string (one of the 32-byte halves) as an unsigned integer in little-endian representation and then reducing it modulo  $2^{255}$ , which is denoted in Algorithm 4 on first and second line by the function called *masked()*. Pseudocode of the MAP function can be seen in Algorithm 3.

---

**Algorithm 3** Ristretto255 MAP

---

**Input:**  $t, \mathbf{D}$  (input values)

**Output:** Group element represented by internal representation  $(w_0 \cdot w_3, w_2 \cdot w_1, w_1 \cdot w_3, w_0 \cdot w_2)$

```

1:  $r \leftarrow \mathbf{SQRT\_M1} \cdot t^2$ 
2:  $u \leftarrow (r + 1) \cdot \mathbf{ONE\_MINUS\_D\_SQ}$ 
3:  $v \leftarrow (-1 - r \cdot \mathbf{D}) \cdot (r + \mathbf{D})$ 
4:  $(\text{was\_square}, s) \leftarrow \mathbf{INV\_SQRT}(u, v)$ 
5:  $s\_prime \leftarrow -\mathbf{CT\_ABS}(ss \cdot t)$ 
6:  $ss \leftarrow \mathbf{CT\_SELECT}(s \text{ if was\_square else } s\_prime)$ 
7:  $c \leftarrow \mathbf{CT\_SELECT}(-1 \text{ if was\_square else } r)$ 
8:  $N \leftarrow c \cdot (r - 1) \cdot \mathbf{D\_MINUS\_ONE\_SQ} - v$ 
9:  $w_0 \leftarrow 2 \cdot ss \cdot v$ 
10:  $w_1 \leftarrow N \cdot \mathbf{SQRT\_AD\_MINUS\_ONE}$ 
11:  $w_2 \leftarrow 1 - ss^2$ 
12:  $w_3 \leftarrow 1 + ss^2$ 
13: return  $(w_0 \cdot w_3, w_2 \cdot w_1, w_1 \cdot w_3, w_0 \cdot w_2)$ 

```

---

---

**Algorithm 4** Ristretto255 Hash to group

---

**Input:**  $b$  - a uniformly distributed 64-bytes byte-string**Output:**  $P$  - point on Curve25519 in extended Edwards twisted form1:  $t_0 \leftarrow \text{masked}(b[0..31])$ 2:  $t_1 \leftarrow \text{masked}(b[32..64])$ 3:  $P_1 \leftarrow \text{MAP}(t_0)$ 4:  $P_2 \leftarrow \text{MAP}(t_1)$ 5:  $P \leftarrow P_1 + P_2$ 6: **return**  $P$ 

---

## 2.4 Password-authenticated Key Exchange

The protocols for key exchange are protocols that allow two communicating parties to share a security key for their connection (session key). Typically, a session key is used by both parties as the symmetric cipher key to encrypt communication between the client and the server. Generally, both parties have pre-generated keys stored on cryptographic devices such as USB tokens or smart cards. However, implementing such devices significantly increases system complexity and decreases user-friendliness. An alternative solution to this problem is the use of easily memorable passwords. However, this solution compromises the security of the system, as easily memorable passwords are selected from a much smaller set than secure passwords or cryptographic keys stored on the mentioned devices. Therefore, password-based systems are highly vulnerable[34]. To address the security issues of such protocols, authentication protocols based on passwords PAKE are employed.

The PAKE protocol, introduced by Bellare and Merritt [35], is a special kind of protocol for swapping cryptographic keys. These protocols help two parties, usually a client and a server, agree on a shared key using public-key cryptography. The first well-known protocol for this purposes was the Diffie-Hellman protocol[36]. However, it has a weakness, it's vulnerable to man-in-the-middle attacks[34]. PAKE protocols are designed to enable two or more parties to achieve mutual authentication and exchange a secure connection key over an insecure channel using a short and easily memorable password, without relying on a public key infrastructure (except for user registration in case of OPAQUE). A notable feature of these protocols is that they should also protect the client's password. The password is not disclosed to the server or any other entity. On the other hand, the client never learns how the password hash was computed on server or what salt was used. A salt, typically stored on the server, is a randomly generated string

added during password processing to ensure that identical passwords produce different outcomes for different users. When it comes to not revealing a user's password to the server, there is a large group of cryptographic protocols known as Zero-knowledge proofs (ZKP) protocols. These protocols enable one party (the prover) to prove to another party (the verifier) that a statement is true without revealing any additional information, apart from the validity of the statement itself (for example, knowledge of a password).

In simple terms, after an authentication attempt (whether valid or invalid), both parties (client and server) should only learn whether the client's password matches the expected server value and no additional information.

A secure PAKE should provide the best possible security for user authentication using passwords. In practice, PAKE protocols can be used wherever secure communication over an insecure channel is needed, such as in Bluetooth communication, securing wireless networks, IoT devices, or as a replacement for current internet login protocols.

Despite the apparent security advantage provided by PAKE protocols compared to existing approaches used for server authentication, they are still not widely adopted. This may be due to a lack of good PAKE implementations in programming languages used in web app and mobile app industry, making it challenging to use these protocols. Another reason may be the limited awareness of PAKE protocols. Many people may not be aware of their existence and potential applications. Nevertheless, some PAKE protocols have found practical use[37]. One of our main goals in this thesis was to find out whether a secure aPAKE protocol like OPAQUE can be usable in MCU and, if so, how much time such a protocol consumes on widely used MCUs.

One of the most widespread and relatively old PAKE protocols is the SRP protocol [38],[39], standardized as an authentication method for the TLS protocol. Created in 1998 by T. Wu [40], SRP is implemented in cryptographic libraries such as OpenSSL or libsodium[41]. Despite its age, SRP is still used in various projects. For example, Apple uses the SRP protocol in its iCloud Key Vault method[42], which backs up user passwords for websites and email accounts on Apple iCloud. One disadvantage of the SRP protocol is its vulnerability to rainbow table attacks, as it does not ensure that only the server knows the salt. When an attacker initiates an SRP protocol session, they gain access to the salt, allowing them to create potentially hashed password values and launch an attack on the server. Another drawback is that SRP cannot be used with elliptic curves efficiently[39]. A suitable replacement for the SRP protocol is the relatively new OPAQUE protocol,

designed in 2018 by S. Jarecki, H. Krawczyk, and J. Xu [1]. OPAQUE works fine with elliptic curves and is resistant to rainbow table attacks[4]. In fact, OPAQUE provides several recommended configurations that include ECC. In our implementation, we used elliptic curve Curve25519 utilizing the Ristretto255 transformation.

## 3 Detailed Description of OPAQUE Phases

---

OPAQUE, a cryptographic protocol, is a secure asymmetric password authenticated key-exchange (aPAKE) that supports mutual authentication in a client-server setting without reliance on PKI[1]. The OPAQUE is unique because it is the first aPAKE that employs secret salt. This brings a huge differentiation amongs other aPAKE protocols and makes OPAQUE first to be secure against pre-computation attacks upon server compromise. Prior aPAKE protocols either use salt in clear form that is transmitted through communication channel from server to user, which can essentially lead to building pre-computed dictionaries, or do not use salt at all. In addition, OPAQUE provides ability to hide users passwords from server even during registration phase, which means that server does not store user passwords.

OPAQUE consists of two phases: *registration* and *authenticated key exchange*. In the first phase, a client registers its password with the server and stores its credentials on the server. Recovering these credentials can only be done with knowledge of the client password. In the second phase, a client uses its password to recover those credentials and subsequently uses them as input to an AKE protocol. This phase has additional mechanisms to prevent an active attacker from interacting with the server to guess or confirm clients registered via the first phase[1].

Before these phases, the client and server decide on a setup that completely describes the cryptographic algorithm requirements necessary to run the protocol. The server generates a set of keys *server\_private\_key* and *server\_public\_key* for the authenticated key exchange (AKE)[5] and selects a starting point called *opr\_f\_seeds* (in some literature also known as salt) for the OPRF. The server can utilize the same *server\_private\_key* and *server\_public\_key* with several clients. Moreover, the server has the option to utilize distinct *opr\_f\_seeds* for each client, as long as they remain consistent throughout both the registration and online AKE phases and are consistently maintained for each client.



Note that in this chapter, we provide supplementary material to the official RFC specification [1] to help readers better understand certain concepts of the OPAQUE protocol. In subsequent sections, we will describe the types of messages used in the OPAQUE registration and authentication phases, which we have implemented in our library (see Appendix A). We have developed a well-documented library with numerous implementation and optimization techniques, some of which are described in Chapters 5 and 6. Our library is designed for small embedded platforms, and the complete code is available in Appendix A.

## 3.1 Client to Server Registration phase

Registration is performed with three messages. A registration request, a registration response, and ends with the client sending a registration record to the server. This record contains an envelope as well as additional information such as the client's public key and masking key, which will be explained later in this chapter. An envelope is then recovered on client-side during authentication phase. During the registration phase, OPAQUE utilizes a technique called Oblivious Pseudorandom Function (OPRF)[7]. In this process, the server signs the user's blinded hashed password, which is then used to create an envelope. Its very important to note that registration phase is the only phase in OPAQUE that requires a server-authenticated channel that provides confidentiality and integrity. In other words it is necessary to perform registration of a client through secure channel employing either physical, out-of-band, or using PKI-based approaches.

### 3.1.1 Oblivious Pseudorandom Function

An Oblivious Pseudorandom Function (OPRF) [7] is a collaborative two-party protocol involving a client and a server. Its purpose is to compute a Pseudorandom Function (PRF) where the server possesses the PRF key, and the client supplies the input to the function. During this process the client gains no knowledge about the PRF except for the resulting output, while the server remains oblivious to both the client's input and the output of the function. The OPRF ensures a secure and mutually confidential interaction between the two parties, preserving the privacy of each party's sensitive information[1].

The mathematical foundation of OPAQUE for Oblivious Pseudo-Random Functions (OPRF), commonly known as blind signing, is elegantly straightforward. The simplicity of the mathematical processes involved contributes to the protocol's accessibility, facilitating a clear understanding of its operations. Despite this

simplicity, OPAQUE maintains a high level of cryptographic integrity, demonstrating the effectiveness of its design. In summary, OPAQUE combines mathematical clarity with cryptographic robustness in the realm of blind signing. It looks like this:

1. Client holds input  $t$  (this could be password), Server holds secret key  $x$ .
2. Client generates random *blind* value  $r$ , which is just a random number from  $\mathbb{GF}$ .
3. Client computes  $T = H_1(t)$  and then blinds it by computing  $r * T$ .
4. Client sends  $M = r * T$  to Server, note that  $M$  is known as `blinded_element` in our case.
5. Server computes  $Z = x * M$  and returns  $Z$  to Client.
6. Client computes  $(1/r) * Z = x * T = N$  and stores the pair  $(t, N)$  for some point in the future.

Why do we actually need OPRF, and what is its purpose in OPAQUE? Well, the whole registration phase is pretty much the OPRF protocol with some additional tweaks. Let's look at OPAQUE's registration diagram (Figure 3.1):

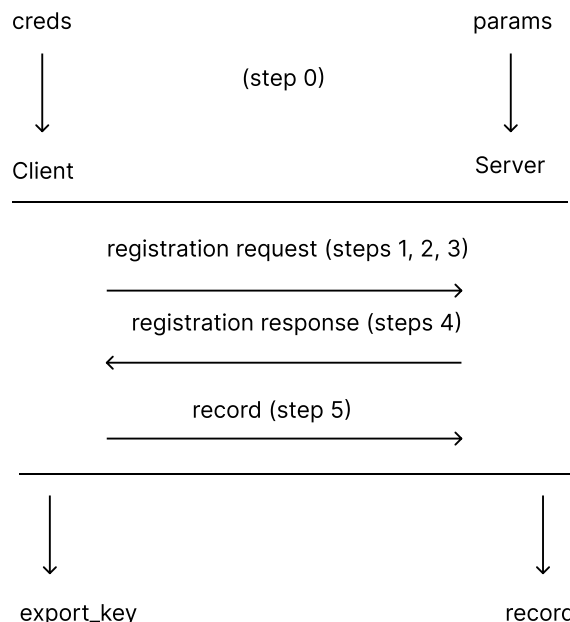


Figure 3.1: OPAQUE offline registration: sequence of messages sent between client and server

As you can see in the diagram above, steps 1-3 are used during the construction of the registration request. In the process of creating the registration response, step 4 takes place. Lastly, the construction of the record includes step 5. Note that OPRF steps are the core of the registration phase. The registration phase contains non-OPRF steps and additional tweaks, such as the creation of the envelope, etc. The envelope is a special structure created by the client and subsequently sent to the server. The server then stores just the envelope, no client password, no additional password salt, etc. Now back to the question, "What is the purpose of OPRF in OPAQUE?" As shown in the scheme above, the client uses its password to later compute  $N$  (*opr\_f\_output*) from the server's  $Z$ . This *opr\_f\_output* is then used to create the so-called *randomized\_password* using the `hkdfExtract` function [43]. Based on the *randomized\_password*, the client generates an *envelope*, *client\_public\_key*, *masking\_key*, and *export\_key* [1]. An *envelope*, *client\_public\_key* and *masking\_key* creates final registration record that is send to server by client as depicted on diagram 3.1 and their purpose will be explained later in this chapter. For now, just keep in mind that they are stored on server after client registers to server. Also note that client output of registration phase is an *export\_key*. This is very unique property of OPAQUE protocol and it can be used for application specific reason. An *export\_key* is a value that is available only for user and it stays the same for specific user. This means that every time a user logs in, the same *export\_key* will be generated on client side. Usage of such key can be used as a symmetric key to encrypt some files and store those encrypted files on remote storage. Purpose of *exported\_key* can differ from application to application or it might not be used at all. Think of it as a additional feature provided by OPAQUE protocol[1].

Client's envelope with other values discussed above is generated using `Store()` function during offline registration and recover (using function `Recover()`) later during online login phase [1]. HMAC-based Extract-and-Expand Key Derivation Function (HKDF) is a key derivation function (KDF), that takes some source of initial keying material and uses it to derive one or more cryptographically strong keys[43]. In OPAQUE, there are 2 main HKDF function used (based on SHA512[10] in case of our implementation suit) `Extract()` and `Expand()`[43].

The `Extract()` function extracts a pseudorandom key of fixed length from input keying material *ikm* and an optional byte string salt.

The `Expand()` function expands a pseudorandom key *prk* using the optional string info into selected amount of bytes of output keying material.

When discussing the creation and retrieval of the *envelope*, clients generate

their *envelope* using the *Store()* function specified in the RFC during registration, and later retrieve it via the *Recovery* function, also specified in the RFC [1], when logging in.

## 3.2 Client to Server Authentication Phase

The online login phase is a bit more complicated. Login consists of two parts, OPRF and AKE [1], [5]. In this phase, a client uses its password to recover his credentials (OPRF part) and subsequently uses them as input to an AKE protocol. The AKE protocol is used to generate the *session\_key*. Note that the *session\_key* differs from login to login. In other words, every time a user logs in to a server, a new *session\_key* will be generated to encrypt communication.

Not only the *session\_key* is generated, but also the *export\_key*. Note that the *export\_key* is not unique for every communication, it remains the same (the same *export\_key* as generated during the registration phase). One thing that has not been mentioned so far is the presence of multiple private and public keys. To make things clear, we will explain the magic behind those keys. First of all, we'll talk about the "general" public and private keys.

The server generates its private general key and public general key using the *DeriveKeyPair(server\_private\_key, server\_public\_key)* function before communication is established. In fact these are the keys discussed in Chapter 3 when we talk about initial setup for OPAQUE protocol, that is prior both registration and authentication phase. On the other hand, the client generates its general public key (*client\_public\_key*) during the registration phase, using the *Store()* function when creating the registration request. Note that client's general private key (*client\_private\_key*) is not going to be used in this moment. Therefore, in our implementation we decided to clear it from processor memory stack using approach described in Section 5.3. The client's private key will be safely restored later.

Both the client and the server also generate so-called AKE public and private keys. The client generates AKE keys (*client\_public\_keyshare, client\_secret* - private AKE key) during the first AKE message (see Figure 3.2). The server also generates AKE keys (*server\_public\_keyshare, server\_secret* - private AKE key). This is done in the AKE part of the login when the server constructs the second AKE message (Figure 3.3). Note that when talking about AKE and KE messages in this thesis, we refer to the same thing, and we will be using these two terms interchangeably. We can see the flow of the login phase in the diagram 3.2.

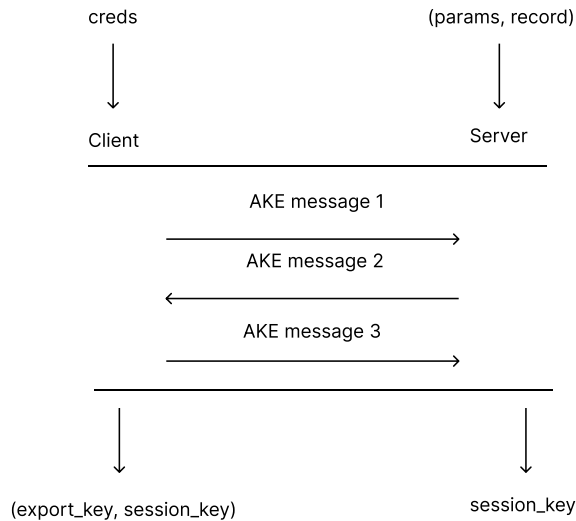


Figure 3.2: OPAQUE authentication: sequence of messages sent between client and server

### 3.2.1 First AKE Message

The login phase is initiated by the client. The client needs to generate the KE1 message and send it to the server. The KE1 message consists of two structures called *CredentialRequest* and *AuthRequest*. In other words, the client needs to generate a blinded message (OPRF phase, the same as during registration). This is stored in the *CredentialRequest* structure. In the *AuthRequest* structure, client's public information, such as the client's public AKE key and public nonce (public nonce will be discussed later), is stored on the server. The client also creates its own structure to hold private information like the password, blind (random scalar - OPRF phase), AKE private key (client\_secret), and also a copy of the KE1 message. See Figure 3.3.

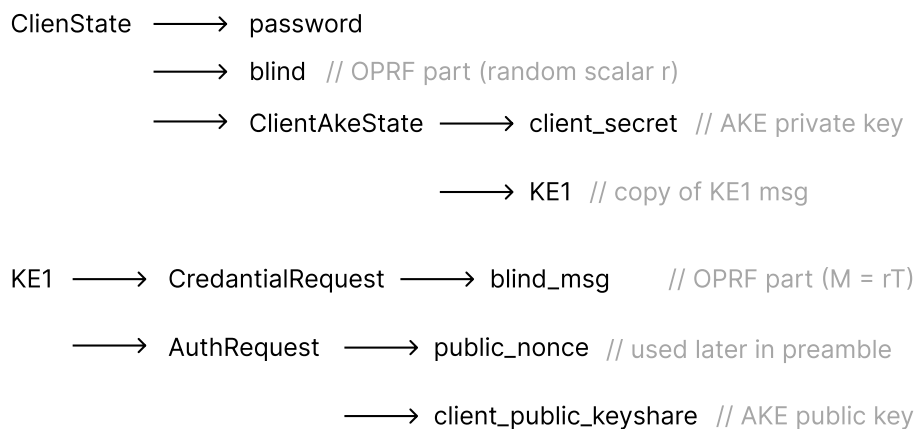


Figure 3.3: First AKE message generation: client-side flowchart.

### 3.2.2 Second AKE Message

When the server receives the KE1 message from the client, it calculates the *evaluated\_message*, which is the OPRF step on the server-side. This is the same as registration step 4 discussed previously in section 3.1.1. Besides that, the server also generates a *masking\_nonce*, which is a value retrieved from random number generator (RNG) and calculates the *masked\_response*. Be aware, that for simplicity sake, we decided to implement linear congruent RNG (as depicted in code 7.1). The *masking\_nonce* is used in the process of creation *credential\_response\_pad* as depicted in equation 3.8. The masked response is calculated in a very specific manner:

$$masking\_nonce = rng() \quad (3.1)$$

$$masked\_response = credential\_response\_pad \oplus (server\_public\_key||envelope) \quad (3.2)$$

Where *server\_public\_key* is the server's "general" public key. Note that the envelope is a structure that was created during registration and is stored in the server's database. One missing piece from the formula above is *credential\_response\_pad*, which is a value expanded (using `hkdfExpand`) from the *masking\_key*. The credential response label is simply a byte-string defined in the OPAQUE protocol specification. Recall that *masking\_key* is a value that is generated by client during registration and stored on server. These values form the CredentialResponse structure. Note that the *masked\_response* is later used by the client to recover an envelope (AKE3). In the KE2 message, the AuthResponse structure is also present, which the server needs to form. It consists of *server\_nonce* (from RNG), *server\_public\_keyshare* (server's AKE key), and *server\_mac*. We will go deeper in a bit, but first, a quick summary.

We can divide the forming of the second AKE message into two steps. The first step is to create the *evaluated\_message* (OPRF step 4, see section 3.1.1), use stored the *masking\_key* created in registration phase, and calculate the *masked\_response*. The next step is the logic behind the actual AKE. Now let's talk about the AKE logic. In this particular implementation, we use the Triple Diffie-Hellman (X3DH) protocol[6], a 3-message AKE that satisfies forward secrecy. This means that on every login, client and server will agree upon different *session\_key*, which is a key that serves as symmetric encryption key. Basically the purpose of X3DH is to create *ikm* (input key material) by combining the general and AKE private server

keys with the client's general and AKE public keys on the server-side. Later, when forming the AKE3 message, the client will generate  $ikm$  from the general and AKE private client keys with the server's general and AKE public keys. When forming AKE2 on the server-side, formula for server-side  $ikm$  looks like this:

$$\begin{aligned}
 ikm = & server\_secret(AKE) \cdot client\_public\_key\_shared(AKE) \\
 & || server\_private\_key \cdot client\_public\_key\_shared(AKE) \\
 & || server\_Secret(AKE) \cdot client\_public\_key
 \end{aligned} \tag{3.3}$$

where "||" denotes byte-concatenation.

Why do we actually need something like  $ikm$ ? From  $ikm$ , the server can generate three things:  $km2$ ,  $km3$ , and  $session\_key$ . As mentioned above, X3DH provides forward secrecy because, on every login, new AKE keys are generated, thus  $newsession\_key$ . Key material  $km2$  is then used to form the  $server\_mac$ , and  $km3$  is used to form the  $expected\_client\_mac$ .

These MAC (Message Authentication Code) messages are used to verify the client/server, respectively. The server sends its MAC to the client, and the client (in AKE3 step) generates the  $expected\_server\_mac$ . These must match (verification step) [1]. On the other hand, the client also sends its  $client\_mac$  (in AKE3 message) to the server, which already calculates (in AKE2 step) the  $expected\_client\_mac$ . If they match, they can use the  $session\_key$  for encrypted communication, this is known as the finalization phase of AKE. Note that to form  $km2$ ,  $km3$ , and  $session\_key$ , we need a bit more than just  $ikm$  and that missing piece is called preamble. A preamble is just a byte-concatenation of  $client\_identity$ , KE1,  $server\_identity$ , and KE2 (without  $server\_mac$ ).

Recall the  $public\_nonce$  generated by client during creation of KE1, which takes part in constructing a preamble. The public nonce indeed introduces randomness into the first message (KE1) structure in OPAQUE. This randomness enhances security by ensuring that each authentication attempt is unique, thereby preventing replay attacks. Additionally, including randomness in the preamble, further strengthens security by making it more difficult for an attacker to predict or manipulate the authentication process. See the diagram 3.4 to better understand what the server will hold and generate when forming AKE2.



Figure 3.4: Second AKE Message Generation: Server-Side Flowchart.

### 3.2.3 Third AKE Message

In third AKE message, the client receives the KE2 message from the server but also holds a copy of the KE1 message in its state ( $ClientState \rightarrow ClientAkeState \rightarrow KE1$ ), along with other private values. These values were generated by the client and saved during the KE1 creation process. Now, the client is ready to form the KE3 message. This can be divided into two steps: building *RecoverCredentials* structure and AKE finalization. First, let's discuss the *RecoverCredentials* phase. This is the last step of the OPRF protocol, where the client needs to perform the OPRF finalization step, which is to calculate  $(1/r) * Z = x * T = N$ . Recall the OPRF step from the registration phase of the OPAQUE protocol. The next step is to retrieve the client's *envelope*, which is stored on the server. To do so, the client needs to use the HKDF function for extraction of the *randomized\_password*. A *randomized\_password* is an output of *hkdExtract* function that takes byte string. Such byte-string is represented as concatenation of  $hash(N)$ , which is specifically hashed OPRF output  $N$  and user password, and hardened  $hash(N)$ . It may look like this:

$$opr\_output = hash(N || pwd), \quad (3.4)$$

$$randomized\_password = hkdExtract(opr\_output || Harden(opr\_output)), \quad (3.5)$$

where the *pwd* represents client's password and the function *Harden()* could be any Key Stretching Function (KSF). In this case, it is the identity function,



which is the same as in the OPAQUE specification. The identity function returns the same value on the output as the input. In other words,  $msg = Identity(msg)$ , which could be simply implemented as memcopy. So, in our case, the *randomized\_password* would look like this:

$$opr\_output = hash(N \parallel pwd), \quad (3.6)$$

$$randomized\_password = hkdfExtract(opr\_output \parallel opr\_output), \quad (3.7)$$

Now, when the client has its randomized password, the OPRF is done and user can use randomized password as input to AKE. Subsequently randomized password is used to create the *masking\_key* using *hkdfExpand*. The client also recreates *credential\_response\_pad* (recall *credential\_response\_pad* from AKE2), which is the same process as it was on the server-side and also results in the same value. It looks like this:

$$credential\_response\_pad = hkdfExpand( \\ masking\_key, (masking\_nonce \parallel "CredentialResponsePad") ) \quad (3.8)$$

Note that while the client can generate the masking key, the server needs to get it during the registration step (in the client's record response message). So far, the client has retrieved the *randomized\_password* and expand *credential\_response\_pad* from it. But why does the client actually need *credential\_response\_pad*? It's simple, recall that the server sends the *masked\_response* in KE2 to the client, which means that at this point, the client already has *masked\_response* received from  $KE2 \rightarrow CredentialResponse \rightarrow masked\_response$ . Masked response will be used to retrieve *server\_public\_key* and the *envelope*.

Server calculates *masked\_response* as:

$$masked\_response = credential\_response\_pad \oplus (server\_public\_key \parallel envelope) \quad (3.9)$$

Since the client recreated *credential\_response\_pad*, it is easy for the client to get *server\_public\_key* and *envelope*.

All he needs to do is to XOR *credential\_response\_pad* and *masked\_response*.

$$(server\_public\_key \parallel envelope) = masked\_response \oplus credential\_response\_pad \quad (3.10)$$

The Client is now able to generate the *client\_private\_key* and *export\_key* using the *Recover()* function [1], which input values are the *randomized\_password*, *server\_public\_key*, *envelope*, *server\_identity*, and *client\_identity*. Server's identity is also something publicly known. This is typically a domain name, e.g., example.com. If not specified, it defaults to the server's public key. Server also needs to generate its general public and private key and choose a *opr\_f\_seeds* (salt) for the OPRF process. Server's identity, general public and private keys, and *opr\_f\_seeds* are generated prior to both registration and authentication phase as discussed previously.

Besides the generation of the *client\_private\_key* and *export\_key*, *Recover()* also internally recreates the *auth\_tag* and verifies it against the *auth\_tag* from the envelope [1].

$$expected\_tag == envelope.auth\_tag \quad (3.11)$$

If those are not the same, an error will occur. This is a prevention against attackers because only the server that the client is registered on has a valid envelope. Recall that registration is done out-of-band or using PKI, so we assume that registration is done safely. This is end of *RecoverCredentials* part of AKE3.

A brief summarization for clarity before we go further. During third step of authentication phase, user retrieve *opr\_f\_output* which is a product of OPRF. The *opr\_f\_output* coupled with it's hardened version is then used to create so called *randomize\_password*, which is an essential value for expansion of masking key and other values. Note that at this point, client also generates *client\_private\_key* and *export\_key*, which is the same as in registration phase.

Subsequently *masking\_key* is used to form *credential\_response\_pad* that is necessary to obtain *server\_public\_key* and the *envelope*. Once the *envelope* is recovered, client will generate an *expected\_tag* and compare it to *auth\_tag* in the *envelope*.

The second part is the AKE finalization step. This is actually pretty straightforward, as we already discussed most of the concepts previously. The client generates *ikm*, which is very similar to what the server did in AKE2. However, now *ikm* formation is on the client-side and looks like this:

$$\begin{aligned} ikm = & client\_secret(AKE) \cdot server\_public\_keyshare(AKE) \\ & || client\_secret \cdot server\_public\_key \\ & || client\_private\_key \cdot server\_public\_keyshare \end{aligned} \quad (3.12)$$

where  $\|$  denotes byte-concatination. After the server generates  $ikm$ , it also needs to construct the preamble (similar to what the server did in AKE2). Now, the client is able to generate its  $KM2$ ,  $KM3$ , and  $session\_key$ . Client computes  $expected\_server\_mac$  from  $KM2$ , which is then compared to the  $server\_mac$  that comes from the server in the KE2 message. The client also creates the  $client\_mac$  from  $KM3$  and sends it to the server. Note that the server already holds the  $expected\_client\_mac$ , which it calculates during KE2 formation. The last step of the entire OPAQUE protocol is on the server-side. The server needs to verify if the  $client\_mac$  (in the KE3 message) is identical to the  $expected\_client\_mac$ . If so, the client and server can use the  $session\_key$  to encrypt their conversation.

## 4 Development Environment

---

The development of our RG255 implementation and subsequently OPAQUE implementation aimed at embedded systems, requires the incorporation of fast and memory-efficient approaches throughout the development process. To create such implementations, we utilized various development tools and environments that helped us incorporate optimization approaches, evaluate the correctness of our implementation, perform speed measurements, and more.

We had multiple ideas that we wanted to integrate into our library. One such example is endian-agnostic code, a method that allows code to be written once and run on both little and big endian architectures without conditional compilation. However, this approach may not be efficient for embedded systems, which are our targeted platforms. Therefore, we decided to create two archives: one for PC (personal computer) platforms utilizing the endian-agnostic approach, and another targeted specifically for embedded systems, particularly those running on the ARM Cortex-M4 core.

This chapter introduces the specific development environment and tools utilized throughout the entire development process.

### 4.1 Development Platform

In this section discuss development environment used during creation of both archives discussed above. Development of whole codebase was performed on PC with following specifications:

- **CPU:** Intel i5-8250U (1.60GHz)
- **RAM:** 8.00 GB
- **Operating System:** Windows 10 Pro 64-bit

### 4.1.1 Programming environment

We prepared C implementations of Ristretto255 and OPAQUE on a PC platform and performed various tests as described in Chapter 7. During the development of our implementation we used a latest version GCC compiler (version 13.2.0) was used, targeting both 64-bit and 32-bit little-endian architectures. This ensures compatible types that are platform independent. We stumbled upon several problems when using type `size_t`, that results in compilation errors on 64-bit platform and no error on 32-bit platform and vice-versa, therefore we decided not to use some critical types and rather use platform independent types like `uint32_t`, `int32_t` etc. Implementation details and optimization techniques utilized in our library are provided in Sections 5 and 6. Additionally, we have put a lot of effort into writing a library that is well-documented and heavily commented, the library is available in Appendix A.

### 4.1.2 Python Prototype

As part of the development of our library, we also used the Python programming language of version 3.9.11 [44]. Python serves as a high-level environment for debugging and reference calculations and was used for rapid prototyping of Ristretto255, which helped me deeply analyze RG255. In the optimization phase, we employed computational methods over finite fields using modulo  $2p$ , where Python was used to identify suitable points in RG255 for additional reduction, thus bringing numbers back within the modulo  $p$  interval. More details about this optimization technique and how we used Python to overcome some difficulties are provided in Section 6.6.

### 4.1.3 Embedded Platform

Our main goal was to develop a fast, compact and secure implementation resistant to side-channel attacks aimed for embedded systems. Therefore, we prepared an archive specifically targeting ARM Cortex-M4 core, where some of the optimization techniques described in Chapter 6. The whole archive for MCU (available in Appendix A) was tested on STM32F4DISCOVERY development board, where we performed measurements (see Chapter 7). The STM32F4DISCOVERY board is equipped with an STM32F407VG microcontroller, which is built on the 32-bit ARM Cortex-M4 core and operates at a clock frequency of up to 168 MHz, with 1 MB of Flash memory and 192 kB of RAM [45]. NIST has acknowledged the widespread use of the Cortex-M4 in academic post-quantum literature and

has recommended it to submission teams as an optimization target for the second round [46].

The board uses an 8 MHz crystal for processor clocking, and the core processor clock frequency was adjusted up to a maximum of 168 MHz using a Phase-Locked Loop (PLL) configuration. Measurements were performed using the GCC ARM compiler from ARM GNU Toolchain, Version 13.2.Rel1, released on October 30, 2023, available at <https://developer.arm.com/downloads/-/arm-gnu-toolchaindownloads>. The board incorporates an integrated ST-LINK/V2 debugger and programmer, enabling us to program and debug the microcontroller through the integrated ST-LINK interface, eliminating the need for an external debugger. In our experiment, we leveraged this feature by connecting the board to a computer via USB, providing both power and a communication interface for debugging.

Furthermore, in our experiments we utilized the Instrumentation Trace Macrocell (ITM) and DWT CYCCNT (Data Watchpoint and Trace Cycle Counter), both integral components of the ARM Cortex-M debug and trace architecture. We employed ITM in conjunction with ST-LINK/V2, enabling the redirection of the `printf` function to the ITM ports to facilitate code instrumentation. DWT CYCCNT is a component located in the core of the ARM Cortex-M processor, and it is a cycle counter that increments with every central processing unit (CPU) cycle. It was employed for measuring execution time and conducting performance profiling.

## 4.2 QEMU for Big Endian Code on Little Endian Devices

One of our goals was to write code that is endian-agnostic, meaning there is no need for conditional compilation to ensure the program runs correctly on both big-endian or little-endian architectures. To test for bit-exact results (correct results compared to official test vectors) using this approach, we utilized the QEMU emulator [47], which emulates the Debian 11 operating system running on a big-endian platform called PowerPC (Power-PC64 version 5.2.0). During testing of the endian-agnostic approach on Debian OS, we used `gcc powerpc64-linux-gnu-gcc`, version 10.2.1.

This section is dedicated to describe an approach of testing code written for big endian systems. Section 5.4, we explains how we built an endian-agnostic

implementation of Ristretto255. When employing such an approach, errors can easily be introduced into the code, especially when doing so for the first time. Therefore, we need a way to test our code on a big endian device. However, during the development of our Ristretto255 implementation, we did not have access to such a device. Thus, we needed to find a way to test big endian code on little endian devices. During our research, we stumbled upon an emulator, QEMU, in combination with the GNU debugger, which offers exactly what we needed.

QEMU [47] is an open-source machine emulator and virtualizer that enables users to execute operating systems and programs intended for one machine architecture (such as an ARM-based system) on a different machine architecture (such as an x86-based PC), without requiring the original hardware. It supports emulation for a variety of architectures, including x86, ARM, PowerPC, and others.

QEMU supports several different operation modes. One commonly used mode is System Emulation mode, which provides a virtual model of an entire machine, including the central processing unit (CPU), memory, and emulated devices, to run a guest OS. This mode may also work with a hypervisor such as KVM, Xen, or Hypervisor.

Another supported mode of QEMU is User Mode Emulation, in which QEMU is capable of executing processes compiled for a different instruction set. In this mode, system calls are adapted to accommodate endianness differences and 32/64-bit mismatches. The primary function of user-mode emulation is to facilitate fast cross-compilation and cross-debugging. This mode, User Mode Emulation, is the one we utilized for testing and debugging our endian-agnostic code for big endian systems.

When we searched for the best way to set up QEMU and GNU debugger to test code on big endian devices, we came across a very helpful article[48], where the author describes a step-by-step setup and debugging process. However, the article is tailored for Linux-based operating systems, which may not be a perfect solution for everyone, especially Windows users. Therefore, we provide a solution for Windows users in the following sections Appendix C.

# 5 Implementation Strategy of Ristretto255 Transformation

---

In this section, we discuss multiple implementation strategies utilized in our implementation of the Ristretto255 transformation. These strategies encompass several key techniques, including constant-time operations, management of negative elements in  $\mathbb{GF}(p)$ , secure cleaning of local variables stored on the processor memory stack, and maintaining endian agnosticity. Each of these approaches was crucial in our implementation for efficiency, security, and platform compatibility. We have incorporated all these strategies into our implementation, which are detailed in Appendix A.

## 5.1 Constant Time Approach

High-quality cryptographic protocols require constant-time operations to prevent side-channel attacks. This is why all operations should be implemented in constant time, as suggested in RFC 8032 [49]. By constant-time implementation, we mean an implementation that is resistant to timing attacks, which is a subclass of side-channel attacks. A timing attack is a sophisticated method of bypassing the security mechanisms of an application and intentionally gaining leaked information that could be used by an attacker for malicious purposes. The idea behind timing attacks is quite simple and is based on time differences in data processing. An attacker supplies various inputs to the algorithm or application, times the process. If the algorithm is susceptible to timing attacks, different inputs should be processed in different time lengths. By analyzing time differences based on certain inputs, the attacker can guess the valid input.

Algorithm 5 shows a typical non-constant time algorithm for byte-string comparison that is susceptible to timing attacks. It is obvious that the execution time of this algorithm is based on the input parameter because its execution ends when



two compared elements differ from each other. This means that if the input is the same as the value that is being compared, we need to loop through the whole array, thus it takes the maximum time and effort. On the other hand, if the input differs right in the first element, the comparison ends immediately[50].

---

**Algorithm 5** non\_ct\_compare( $a, b$ ), where  $a, b$  are 32-bit unsigned integers

---

```

1:  $n \leftarrow$  length of  $a$ 
2: for  $i \leftarrow 0$  to  $n - 1$  do
3:   if  $a[i] \neq b[i]$  then
4:     return false
5:   end if
6: end for
7: return true

```

---

Table 5.1 shows the time taken for Algorithm 5 to run with different inputs. We can clearly see that Algorithm 5 is susceptible to side-channel attacks because the processing duration varies with different inputs. The more similar the strings are, the longer the execution time of the comparison takes. Table 5.1 shows various strings that were compared to a reference string along with their evaluation times. Each string contains 32 characters, and measurements were performed using a Python script that follows Algorithm 5. A Python script is provided in Appendix B. However, in practice, password evaluation may be more complex than simple string comparison, and time differences may be more apparent when a non-constant-time algorithm is used.

Table 5.1: Comparison of String with Reference and Time Measurement

String to Compare	Reference String	Time Evaluation [ns]
"eeeeeeeeeeee..."	"RefString1AB..."	601
"Reeeeeeeeeeeee..."	"RefString1AB..."	813
"RefStrieeeee..."	"RefString1AB..."	1585
"RefString1AB..."	"RefString1AB..."	3731

Algorithm 6 is one of the most common ways of implementing byte-string comparison in constant time. It ensures that, regardless of the input, we loop through the whole array. This may not be the most efficient way of byte-string comparison in terms of performance, but it is certainly safer than Algorithm 5.

---

**Algorithm 6** `ct_compare(a, b)`, where  $a, b$  are 32-bit unsigned integers

---

```

1:  $mask \leftarrow 0$ 
2:  $n \leftarrow \text{length of } a$ 
3: for  $i \leftarrow 0$  to  $n - 1$  do
4:    $mask \text{ or} = a[i] \oplus b[i]$ 
5: end for
6: return  $((u32)(mask \text{ or } (\sim mask + 1))) \gg 31$ 

```

---

## 5.2 Concept of Negative Elements Used $\mathbb{GF}(p)$

One more important concept deserves attention. The evaluation of a negative byte string element. It may sound a bit confusing because all elements in  $\mathbb{GF}$  are essentially positive. However, the concept of a negative element has been introduced, which is uncommon in traditional mathematics but is employed in cryptography. Although the utilization of negative elements in a finite field is specified in Ristretto255 [14] and RFC8032 [49], we have not found its mathematical justification. Therefore, in this section, we will provide a straightforward description of this concept.

In As per RFC8032, a field element  $e$  is considered negative if the least non-negative integer representing  $e$  is odd, and it's considered FALSE if it is even [49]. For a given element  $e$  in  $\mathbb{GF}$ , if its Least Significant Bit (LSB) is 1, then `IS_NEGATIVE(e)` is true; otherwise, it's false. Implementing this evaluation in constant time is recommended [1]. In RG255, an evaluation of negative elements can be seen in multiple places. For example, in the encoding algorithm on the tenth and fifteenth rows of Algorithm 1, or in the decoding Algorithm 2 at rows one and seventeen. For further details, please refer to Appendix A.

This approach is straightforward. Let's examine the evaluation of negative elements in a  $\mathbb{GF}$  from a different perspective. Suppose  $e$  is a  $\mathbb{GF}(p)$  element, then its negation, denoted as  $-e$ , is calculated as  $-e \equiv p - e \pmod{p}$ , where  $p$  is a prime number. All calculations are performed modulo  $p$  since operations are conducted within the finite field.

Now, consider a  $\mathbb{GF}$  element  $e$  and its negation  $-e$ . How do we determine which element is negative and which is positive? One convention is to define the odd element as negative and the even element as positive.

For example, let's take  $e$  equal to 4 and  $p$  equal to 7. The negation  $-e$  is calculated as  $-e = 7 - 4$ , resulting in  $-e$  equal to 3, which in binary representation is 0011, while 'e' remains 4, represented as 0100 in binary. Following the conven-

tion where the term ‘negative element’ denotes the one with the least significant bit (LSB) set to 1 (i.e., an odd number), the  $\mathbb{GF}$  element  $-e$  is identified as negative, and  $e$  as positive. It’s notable that subtracting an odd  $\mathbb{GF}(p)$  element from the prime  $p$  always yields an even element, and vice versa.

### 5.3 Secure Wiping of Local Variable

Wiping or clearing the processor memory stack after performing cryptographic calculations in a function is a security practice aimed at minimizing the risk of sensitive data exposure. In cryptography, the processor stack is a region of memory used to store local variables and function call information. When cryptographic operations involve sensitive information like encryption keys or intermediate results, it’s crucial to take measures to protect this data from potential attacks. Values assigned to local variables are not volatile and persist in the processor stack even if the called function finishes. They can only be rewritten by other values, which can introduce potential information leakage. A very simple yet efficient method of preventing data leakage is zeroing buffers. This is done at the end of each function that could lead to a potential leakage of sensitive information. In our library, we drew inspiration from multiple cryptographic libraries such as MonoCypher [51], TweetNaCl [52], and CycloneCRYPTO [32] (see Section 6.2).

From our research on these libraries, we found that, unlike MonoCypher [51], TweetNaCl [52] and CycloneCRYPTO [32] do not wipe internal buffers. Drawing inspiration from MonoCypher, we decided to incorporate buffer wiping into our implementation, ensuring that we clear memory responsibly. Wiping function can be seen in code 5.1.

```

1 void crypto_wipe(void *secret, int32_t size)
2 {
3     volatile u8 *v_secret = (u8*)secret;
4     int32_t idx;
5     ZERO(idx, v_secret, size);
6 }

```

Source Code 5.1: Buffer wiping function inspired by MonoCypher library

In the context of a crypto library, using volatile to prevent the compiler from optimizing out memory zeroing is a safety precaution. The compiler’s optimization may decide to skip the zeroing operation if it believes the memory is never read afterward. By using volatile, we’re essentially telling the compiler not to make that assumption and to perform the zeroing operation regardless of whether

the memory is explicitly read afterward. This helps ensure that sensitive data is properly wiped from memory, which is crucial for security purposes. In other words, we need to use `volatile` because the compiler might skip zeroing the buffer due to optimization and make the assumption that "Why would I write into the buffer when it is never read?" One more comment on `volatile` keyword. The "volatile" keyword is used to indicate to the compiler that a variable or object can change its value at any time without any action being taken by the code the compiler finds nearby. This is typically used to prevent the compiler from optimizing away certain accesses to memory, especially in cases where the memory might be modified by hardware, another thread, or some other external entity that the compiler cannot predict. Last but not least, the difference between `volatile` on a pointer and `volatile` on a variable is that `volatile` on a pointer indicates that the data being pointed to may change, while `volatile` on a variable indicates that the variable itself may change at any time [53].

The concept of buffer wiping is intensively utilized in our library, for example, in low-level  $\mathbb{GF}$  arithmetic and in the RG255 functions such as encoding, decoding, and many supportive functions (see Appendix A). However, this is an optional feature that enhances security but can be disabled by simply commenting out the buffer wiping functions

## 5.4 Portable Endian Agnostic Code

The development and testing of our library were performed on a little-endian device. From the outset, our goal was to write code that would run seamlessly on both little and big-endian platforms. However, many other libraries are designed to use conditional compilation. In such cases, the code typically includes something like an `#ifdef BIG_ENDIAN` block followed by specific code for handling endianness. However, during our research on endian independence, we stumbled upon an article [54] that states it is achievable without conditional compilation. This highly inspired us to avoid the conditional compilation approach in our implementation. Instead, we write code that simply runs on both little-endian and big-endian systems. Here's how we did it. Firstly, we need to realize that a number represented as  $8 \times u32$  (an array of eight `uint32_t` elements) has the same byte order on both little and big endian systems. Therefore, we do not need to worry about endianness when processing  $8 \times u32$  number representations. The problem arises when we need to manipulate bytes, in other words, when dealing with  $32 \times u8$  number representations, because this is endian-dependent. In our

Ristretto255 implementation, we need to perform several operations with  $32 \times u8$  numbers, which means that we need to convert  $8 \times u32$  into  $32 \times u8$  number representation and perform some operations afterward. Combining the knowledge gathered from the article [54], we created two functions that perform conversions from  $8 \times u32$  into  $32 \times u8$  little-endian order number representation (even if we are on a big-endian device).

The functions `bytes_to_int` and `int_to_bytes` perform conversions from  $32 \times u8$  to  $8 \times u32$  and vice versa, guaranteeing that the final product of those functions will always be a number represented in little-endian format, regardless of the device's endianness.

```

1 void bytes_to_int(u32* uint32Array, const u8* uint8Array) {
2     for (uint8_t i = 0; i < 8; i++) {
3         uint32Array[i] = (uint8Array[i * 4 + 0] << 0) |
4                         (uint8Array[i * 4 + 1] << 8) |
5                         (uint8Array[i * 4 + 2] << 16) |
6                         (uint8Array[i * 4 + 3] << 24);
7     }
8 }

```

Source Code 5.2: Definition of `bytes_to_int` C function

```

1 void int_to_bytes(u8* uint8Array, const u32* uint32Array) {
2     for (uint8_t i = 0; i < 8; i++) {
3         uint8Array[i * 4 + 0] = ((uint32Array[i] >> 0) & 0xFF);
4         uint8Array[i * 4 + 1] = ((uint32Array[i] >> 8) & 0xFF);
5         uint8Array[i * 4 + 2] = ((uint32Array[i] >> 16) & 0xFF);
6         uint8Array[i * 4 + 3] = ((uint32Array[i] >> 24) & 0xFF);
7     }
8 }

```

Source Code 5.3: Definition of `int_to_bytes` C function

Following this method, we developed the Ristretto255 protocol, ensuring endian independence with just two conversion functions. Note that if we were working solely on a little-endian platform, these conversions would be unnecessary and could introduce a small overhead because we could simply cast a `uint32_t` array into a `uint8_t` array. Additionally, it is important to note that all inputs passed

```
Example for big - endian device

u8 in [32] -> big endian order input bytes
u8 out [32] -> temporary array

in_little_endian := convert 'in ' from big to little endian
ristretto255_decode ( ristretto_point , in_little_endian )
ristretto255_encode ( out , ristretto_point )

at this point out [32] is filled with bytes
in little endian order so it needs to be
concerted to big - endian

out_big_endian := convert 'out ' from little to big endian
```

Figure 5.1: An example pseudocode showing usage of our endian-agnostic RG255 implementation

into our endian-agnostic Ristretto255 function need to be in little-endian format (see code 5.1).

## 6 Optimization Techniques for embedded platform

---

In this thesis, we are targeting embedded systems, which are typically resource-constrained, have a small memory and low computational power. Therefore, our goal is to create an implementation capable of running on embedded platforms that is reasonably fast, compact and resistant to side-channel attacks. Additionally, the size of the processor memory stack for embedded systems is significantly smaller compared to typical CPUs used in PC platforms, such as the Intel i5-8250U processor that we used in developing our library (see Section 4). Therefore, function calls cannot utilize a large stack depth. Accordingly, we made special efforts to create an implementation that minimizes requirements on stack size. When optimizing the speed of our library, we weren't restricted to using only the C language, instead we aimed to find and combine efficient C codes with highly optimized assembly (ASM) codes [31], [55] to achieve a very fast and optimized implementation. This section discusses various optimization techniques that have been integrated into our library, detailed in Appendix A.

### 6.1 Available High-level Ristretto255 Libraries

Non-prime-order curves can introduce security issues due to cofactor, potentially leading to various attacks on more complex protocols. Typically, these concerns are managed through additional tweaks in protocols [16], but some modifications become recurring sources of vulnerabilities and subtle design complications. These alterations often prevent applying the security proofs of the abstract protocol [26].

Ristretto255 provides the reliable abstraction necessary for implementing complex protocols, offering the simplicity, efficiency, and speed characteristic of non-prime-order curves like Bernstein's Curve25519. Several cryptographic libraries and frameworks are available that have implemented Ristretto255 for Curve25519

across various programming languages. Some of these implementations include:

- Rust: *curve25519-dalek*, by Isis Lovecruft and Henry de Valence [56].
- Go: *ristretto255*, by George Tankersley, Filippo Valsorda, and Henry de Valence [57].
- C: *libsodium*, by Frank Denis [58].
- C: *libristretto*, by Tony Arcieri [25].
- AssemblyScript: *wasm-crypto*, by Frank Denis [59].
- Javascript with Typescript: *noble-ed25519*, by Paul Miller [60].
- Javascript: *ristretto255-js*, by Valeria Nikolaenko and Kevin Lewi [61].
- Zig: the Zig programming language includes *ristretto255* in its standard library, implemented by Frank Denis [62].

However, none of these implementations were suitable for our goal of incorporating Ristretto255 into embedded systems. We faced challenges in finding a compact, high-speed, lightweight implementation of Ristretto255 written in C. This led us to develop our implementation, taking into account these specific requirements.

The goal of our implementation was to achieve speed with minimal memory requirements and maintain constant-time operation execution. Our goal was to create a compact, high-speed, constant-time, and memory-efficient implementation of Ristretto255, which forms the core of this thesis, followed by the complete OPAQUE protocol. We also wanted to create code that would run on little endian as well as on big endian devices. Combination of these properties makes a perfectly compliant implementation for embedded systems.

## 6.2 State of the Art Embedded C libraries for GF25519

The purpose of Ristretto255 is to provide a way to perform operations on Curve25519 in more secure manner. From the perspective of high-level protocols, Ristretto255 can be view as a intermediate layer between Curve25519 and higher-level protocol like OPRF [7] or PAKE [4]. Such hierarchy can be seen in Figure 6.1.



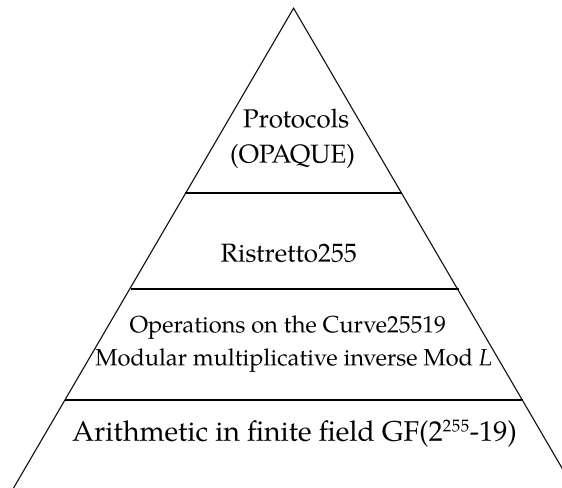


Figure 6.1: Hierarchy of operations in the Ristretto255 Group supporting higher-level protocols, such as the OPAQUE protocol.

Figure 6.1 at the highest level shows the OPAQUE protocol that is described in detail in the specification [1], with its fundamental concepts explained in Chapter 3. Moving from the top to the bottom of the pyramid (Figure 6.1), Ristretto255, positioned in the second layer, represents a significant aspect of our optimization efforts, constructed from simpler mathematical building blocks. The very bottom layer of the pyramid (Figure 6.1) illustrates the use of arithmetic in the finite field  $\mathbb{GF}(2^{255} - 19)$ .

Taking a good look at Figure 6.1, we can see that finite field arithmetic is involved in all upper layers of the pyramid making it obvious pivot point for optimization. Therefore, during the optimization technique thinking process, our primary focus was on optimizing the lowest layer of the pyramid, as optimizations at this level have a significant impact on all upper layers.

We searched for the appropriate approach to implement algorithms for operations over  $\mathbb{GF}(2^{255} - 19)$ , and we identified three compact yet efficient cryptographic libraries that inspired us: TweetNaCl[52], MonoCypher[51], and CycloneCRYPTO[32]. Throughout the research we deeply analyzed all of these libraries and identified optimized approach that suits our needs. We specifically aimed for combination of fast implementation with memory efficient internal representation of a field element.

### 6.2.1 TweetNaCl

TweetNaCl[52], written by D.J. Bernstein, is a small but practical cryptographic library with strong security properties in particular, it uses constant-time algorithms to prevent side-channel attacks [16]. TweetNaCl is based on the NaCl

(Networking and Cryptography Library) with the sole purpose of being as compact and practical as possible. It fits into only 100 tweets and seamlessly incorporates all 25 NaCl functions. It contains operations over  $\mathbb{GF}(2^{255} - 19)$ . However, its main disadvantage is the use of an internal representation of a field element of size 64x16 bits in radix  $2^{16}$ , making it too large for embedded systems.

### 6.2.2 MonoCypher

MonoCypher[51] is a small cryptographic library written in C, comprising fewer than 2000 lines of code. The entire code base consists of just two files, making it user-friendly and easy to deploy. MonoCypher excels in maintaining high-performance cryptographic primitives without unnecessary sacrifices. Remarkably, it stands up well against libsodium [58], despite its compact size being closer to TweetNaCl. MonoCypher uses a 32x10-bit internal representation of a field element, ensuring highly efficient computations over  $\mathbb{GF}(2^{255} - 19)$ , particularly on larger systems like desktops. MonoCypher's field arithmetic is strongly inspired by SUPERCOP's ref10 [63].

### 6.2.3 CycloneCRYPTO

CycloneCRYPTO [32] is a cryptographic toolkit designed for use in embedded systems. It provides a comprehensive set of cryptographic primitives and building blocks, including various hash functions, symmetric encryption algorithms, algorithms for asymmetric cryptography, elliptic curve cryptography algorithms, and operations over  $\mathbb{GF}(2^{255} - 19)$ , using an internal representation of  $32 \times 8$ -bits.

A  $32 \times 8$  internal representation is chosen for its perfect fit with the 256-bit prime number  $p = 2^{255} - 19$ , effectively utilizing each bit. In contrast, a  $32 \times 10$ -bit representation might lead to inefficient memory usage, with unused bits. Conversely, a larger word size can offer techniques that enable highly efficient computations.

Our analysis indicates that, of all the C implementations we examined, MonoCypher has the fastest low-level operations over  $\mathbb{GF}$ . However, we chose the format from CycloneCRYPTO library for several reasons, including its compactness, decent speed, and the elimination to/from internal element conversions. Unlike other libraries such as TweetNaCl or MonoCypher, CycloneCRYPTO doesn't require pack/unpack or from bytes/to bytes conversions, which would introduce significant overhead to our implementation. Another reason for choosing Cy-

cloneCRYPTO format is its small  $32 \times 8$ -bit internal representation of a  $\mathbb{GF}$  element. From our perspective, an 8-word representation of the  $\mathbb{GF}$  element is considered optimal for MCUs, and several other libraries also utilize an 8-words representation.

### 6.3 Function for Computation of Inverse Square Root

The Ristretto255 transformation requires computing the inverse square root ( $INV\_SQRT$ ) as depicted in the decoding algorithm (2) at line 3, the encoding algorithm (1) at line 9, or the MAP function (3) at line 4. In fact, computing the inverse square root ( $INV\_SQRT$ ) is one of the most computationally intensive operations in the Ristretto255 transformation. Therefore, we carefully selected the best available implementation for our library. Let  $p = 2^{255} - 19$ , then computation of  $INV\_SQRT$  (specified in [14]) is computed by Algorithm 7.

---

#### Algorithm 7 Algorithm for computing $INV\_SQRT$

---

**Input:**  $(u, v)$  - elements from  $\mathbb{GF}$  such that  $u < p$  and  $v < p$

**Output:**  $(was\_square, r)$  -  $r$  is an element from  $\mathbb{GF}$  and  $was\_square$  represents a boolean value indicating whether  $u/v$  was a square

- 1:  $r \leftarrow (uv^3)(uv^7)^{(p-5)/8}$
  - 2:  $c \leftarrow vr^2$
  - 3:  $correct\_sign\_sqrt \leftarrow (c = u)$
  - 4:  $flipped\_sign\_sqrt \leftarrow (c = -u)$
  - 5:  $flipped\_sign\_sqrt\_i \leftarrow (c = -iu)$
  - 6:  $r \leftarrow ir$  if  $flipped\_sign\_sqrt$  or  $flipped\_sign\_sqrt\_i$
  - 7:  $r \leftarrow -r$  if  $r$  is negative
  - 8:  $was\_square \leftarrow correct\_sign\_sqrt$  or  $flipped\_sign\_sqrt$
  - 9: **return**  $was\_square, r$
- 

From Algorithm 7, it can be seen that the first step is to compute  $r$ , which involves the computation of  $x^{(p-5)/8}$ . This computation might appear challenging at first sight. However, the computation of  $x^{(p-5)/8}$  is already required for Ed25519 decoding. Therefore, we can draw inspiration from existing libraries. Note that all calculations in Algorithm 7 are performed within the field  $\mathbb{GF}(p)$ . During our research, we stumbled upon multiple libraries that include such computations. It is important to note that  $x^{(p-5)/8}$  is equivalent to  $x^{(2^{252}-3)}$ , as shown in equations below. Therefore, such expressions can be found in multiple cryptographic libraries.

$$x^{(2^{255}-19-5)/8} \tag{6.1}$$

$$\frac{2^{255} - 24}{8} = \frac{2^{255}}{8} - \frac{24}{8} = 2^{252} - 3 \tag{6.2}$$

$$x^{2^{252}-3} \tag{6.3}$$

We analyzed libraries like TweetNaCl, MonoCypher, ristretto255-donna [64] and CycloneCRYPTO and compare implementation of  $x^{(p-5)/8}$  for each of them. We focused on the efficiency of computation of each implementation, in which we observed the amount of mathematical operations in  $\mathbb{GF}(p)$ , such as multiplication and squaring, which indirectly indicates the performance of the given implementations. Based on the measurements from the table 6.1, we can observe that the implementation from CycloneCRYPTO utilizes the fewest mathematical operations among all the analyzed implementations. This was one of the reason why we choose CycloneCRYPTO's implementation of  $x^{(p-5)/8}$  when implementing INV\_SQRT in our Ristretto255 transformation. Another reason is that CycloneCRYPTO's implementation uses simple approach of squaring and powering. It simply perform multiplication on squaring or multiple multiplication performed in for loop when powering [65]. Similar approach is implemented in TweetNaCl. On the other hand, MonoCypher and ristretto255-donna are using optimized approach of powering and squaring, that is a bit more complicated. MonoCypher, as well as ristretto255-donna, is using such internal representation of  $\mathbb{GF}$  element that allows for optimized approach of powering and squaring, suitable for desktops[64].

Table 6.1: Comparison of the implementation of  $x^{(p-5)/8}$  across various cryptographic libraries

Library	Multiplication	Squaring
TweetNaCl	250	246 (eq. to mult.)
ristretto-donna	11	254
MonoCypher	12	259
CycloneCrypto	21	243 (eq. to mult.)

As we choose CycloneCRYPTO format, we were able to truly take advantage of highly efficient ASM implementation of operations in  $\mathbb{GF}$ , which is ideal to combine with ASM implementations from libraries [31], [55] (see Section 6.6).

## 6.4 Functions for ModL Arithmetic

Due to Ristretto255's inclusion in various recommended OPAQUE configurations, we found it necessary to integrate modular inversion modulo  $L$  into our implementation. This integration is essential for the creation of more complex protocols, such as OPRF.

Oblivious Pseudorandom Function [7] is essentially a blind signature scheme. Let's consider a simple but practical example where a blind signature of a password is performed. Blind signing starts on the client side. Firstly, the client chooses its password and passes it to a hash function, which produces a product known as a hash digest. Afterwards, the client blinds its password digest with a random number  $r$  and sends the blinded digest to the server. The server signs the blinded password and sends the blinded signature back to the client. At the end of the blinding scheme, the client needs to perform unblinding (as shown in the equation below), which requires the computation of modular inversion.

$$R' = H(P)^{r \cdot s} \quad \Rightarrow \quad (R')^{r^{-1}} = H(P)^{r \cdot s \cdot r^{-1}} = H(P)^s, \quad (6.4)$$

where  $H(P)$  is the digest of the user's password,  $r$  is the randomly generated number used by the user to blind the password,  $s$  is a random number provided by the server to sign the user's password hash, and  $r^{-1}$  is the modular inverse of the user's randomly generated number  $r$  used to unblind the signed password.

Now that we have clarified why we need modular inversion, it's evident that we need some way of multiplication and reduction modulo  $L$  to construct such modular inversion. Just a quick reminder, we want to work with POGs, such as group with prime order  $L$ , because every element within a POG has a multiplicative inverse. This opens up opportunities for logical optimizations. In our implementation, we provide support for two approaches, enabling users to choose between two methods for computing the modular inverse modulo  $L$  using either Barrett's or Montgomery's reduction algorithms. Our implementation of modular inversion draws inspiration from the MonoCypher library.

Both the Barrett algorithm and the Montgomery reduction algorithm have the capability to speed up modular reduction. They shared some commonalities like requirement of precomputing various constants for a given modulus  $L$  etc. [66].

Montgomery reduction involves converting numbers into and out of "Montgomery form", which can be costly operations that require a true modulo operation in each direction. In contrast, Barrett reduction works directly with regular numbers. As a result, Montgomery reduction is well-suited for modular exponentiation but less suitable for handling unrelated numerical operations.

Montgomery relies on modular congruences and exact division, while Barrett operates by approximating the true reciprocal with bounded precision.

## 6.5 Minimizing Processor Stack Requirements via Shared Local Variables

Another optimization technique that was incorporated into our implementation is efficient stack management. During the development process, we focused on manipulating the processor stack efficiently, utilizing as few variables as possible while maintaining code readability. This was achieved through the use of macros. The code snippet below illustrates a simple function, *calculate\_1*, which takes one input and one output parameter. Internally, it allocates three local variables and three buffers of length 256 bits to perform basic  $\mathbb{GF}$  calculations.

While the code snippet 6.1 (*calculate\_1*) is easy to read, it can be optimized from a memory perspective. It's unnecessary to use three buffers for such calculations. Instead, we can reuse buffer *a* and save space on the processor stack, which is advantageous, particularly on an MCU. The subsequent code snippet 6.3 (*calculate\_2*) demonstrates a possible optimization where buffer *a* is reused, making buffer *c* unnecessary, thus allowing us to save 256 bits on the stack. However, such code can become confusing and less readable, especially when multiple buffers are reused. To efficiently manipulate the stack, using as few variables as possible while preserving code readability, we can employ macros, as demonstrated in code snippet 6.2 (*calculate\_3*).

These code snippets are relatively small, so we can't really see big optimization here. However, using macros becomes advantageous in situations involving more buffers to reuse. If we want to truly take advantage of buffer reuse, while preserving code readability we can even redefine macros if needed. For a detailed demonstration of how this optimization technique was implemented in our library, please refer to Appendix A, specifically the files *ristretto255.c* and *opaque.c*, where this technique is extensively used.

```

1 void calculate_1(uint32_t out[8], const uint32_t in[8]){
2     uint32_t a[8], b[8], c[8];
3     fmul(a, in, in);    // a = in * in
4     fneg(b, a);        // b = -a
5

```

```

6 // NOTE: we no longer use variable 'a'
7 fsum(c, b, in); // c = b - in
8 fmul(c, b, c); // c = b * c
9 fsub(out, c, in); // out = c - in
10 }

```

Source Code 6.1: Original implementation without sharing local variables

```

1 void calculate_2(uint32_t out[8], const uint32_t in[8]){
2     uint32_t a[8], b[8];
3     fmul(a, in, in); // a = in * in
4     fneg(b, a); // b = -a
5
6     // NOTE: we reused variable 'a'
7     fsum(a, b, in); // a = b - in
8     fmul(a, b, a); // a = b * a
9     fsub(out, a, in); // out = a - in
10 }

```

Source Code 6.2: Utilizing macros to maintain readability by sharing local variables

```

1 void calculate_3(uint32_t out[8], const uint32_t in[8]){
2     uint32_t temp1[8], temp2[8];
3     #define a temp1
4     #define b temp2
5     fmul(a, in, in); // a = in * in
6     fneg(b, a); // b = -a
7
8     // NOTE: we reused variable 'a' and preserve code
9     readability
10    #define c temp1
11    fsum(c, b, in); // c = b - in
12    fmul(c, b, c); // c = b * c
13    fsub(out, c, in); // out = c - in

```

Source Code 6.3: Original implementation utilizing shared local variables

## 6.6 Approach to Using Existing Highly Optimized ASM Routines for $\mathbb{GF}(p)$ Operations

Continuing with optimization, we aimed to speed up the most critical parts of our implementation, which involved low-level operations over  $\mathbb{GF}(2^{255} - 19)$ . As we chose to use CycloneCRYPTO's 8-word representation of the field element, we were highly motivated to integrate fast operations such as multiplication, addition, and other operations using the 8-word representation written in ASM [31], [55]. This implementation is noticeably faster than the CycloneCRYPTO implementation written in C, but there was more to optimize.

The assembly operations modulo  $(2^{256} - 36)$ , making internal computations faster. It is required to reduce the result at the end of every function to bring it back from modulo  $2p$  to modulo  $p$ . In our implementation, we use an 8-word representation operating on modulo  $2p$ , which employs efficient ASM functions for intermediate calculations. However, reducing at the end of every function is not necessary, and we can optimize it. Instead of reducing in every function, we identified and performed reduction only at some required places in the Ristretto255 implementation to bit-exact results. However integration of such approach can be challenging and time consuming, therefore we used Python script to prototype high-level implementation of RG255 to find as few as possible places in code, where reduction was needed. Note that Python allows us to operate with much larger numbers than the C language does, so our prototype does not need to use any special library to work with big numbers. A Python script is provided in Appendix A, along with the entire optimized library written in C.

When referring to specific places in the code where reduction needs to be applied, we are mostly addressing situations involving operations on the  $32 \times u8$  representation. For example, the *IS\_NEGATIVE* function checks the LSB of a  $\mathbb{GF}$  element, or the calculation of the absolute value of a  $\mathbb{GF}$  element (see the discussion of negative elements in Section 5.2). The utilization of the *IS\_NEGATIVE* function and the evaluation of the absolute value (*CT\_ABS* function) of a  $\mathbb{GF}$  element can also be observed in the encoding algorithm (Algorithm 1) and the decoding algorithm (Algorithm 2).

You can observe a demonstration of such reduction in Appendix A, specifically provided in the source file *ristretto.c*, where the *fe25519\_reduce\_emil* function is used to reduce  $\mathbb{GF}$  elements modulo  $p$ .

Additionally, we test our implementation on a set of test vectors to confirm the correctness of our implementation.



We continued our optimization efforts at higher levels of the pyramid by employing appropriate algorithms, specifically optimizing the calculation of Mod  $L$  inversion in the Montgomery domain. For this purpose, we drew inspiration from the Monocypher and Cyclone libraries. We identified a critical part of the algorithm, focusing on multiplication and reduction. These parts were rewritten in ASM, utilizing the loop unrolling [46] method in combination with UMAAL and UMULL instructions.

Following the pyramid depicted in Figure 6.1, our optimization process involved the implementation of the following tasks:

- Minimizing the utilization of local variables on the stack.
- Maintaining the constancy of execution time for  $\mathbb{GF}$  operations and other supporting functions (*CT\_SELECT*, *CT\_ABS* etc).
- Performing algorithmic optimization for modulo  $L$  arithmetic at the intermediate level (Barrett and the Montgomery reduction, see Section 6.4).

The performance measurements are described in Section 7.

## 6.7 Additional Optimization Approaches

In our implementation of OPAQUE, we utilize a simple library [67] that specifically focuses on SHA (Secure Hash Algorithm) function and the HKDF (HMAC-based Key Derivation Function). This library provides everything we need, from SHA functions to HKDF-extract and HKDF-expand functions. However, this library provides broader support, offering variations from SHA224 to SHA512. For our purposes, we only require the SHA512 variation, so we removed unnecessary components. Additionally, we identified inefficiencies in memory management within the code of this library. One of the core functions of the SHA512 algorithm was utilizing a buffer of length 80x64 bits, which was unnecessarily large. Drawing inspiration from Jeffrey Walton's implementation [68], which is more efficient, we adjusted to utilize a buffer of length 16x64 bits. This optimization allowed us to save approximately 512 bytes of stack memory. For details, please refer to Appendix A of the MCU archive, specifically in the file `/dependencies/sha384-512.c`.

# 7 Experimental Results

---

This section describes the experimental results obtained by testing various parts of our implementation. Firstly, we performed tests focused on achieving bit-exact results, meaning accurate outcomes compared to official test vectors. These tests were performed on platforms utilizing both little-endian (see Section 7.1) and big-endian architectures in Section 7.2.

Subsequently, we performed extensive numerical tests (see Section 7.3) to ensure the correctness of our RG255 implementation, as test vectors may not cover all edge cases.

Secondly, performance measurements specifically targeted at ARM Cortex M4 platforms were performed using the STM32F4DISCOVERY development board, which operates on a little-endian platform.

Since we developed a library that can be used on multiple platforms, we also wanted to check intermediate results during extensive numerical tests. For this purpose, we employed the modern and highly efficient hashing function xxHash [69] to quickly generate and display hash digest of intermediate results.

## 7.1 Testing for Little Endian Platforms

Our code development began on a little-endian platform (see details in Appendix A) using the development environments and tools outlined in Chapter 4. While implementing the Ristretto255 transformations and the OPAQUE protocol, we closely adhered to their official specifications.

Our primary goal was to deliver clean and well-documented library that allows readers to easily follow and verify each operation against the official specifications. Additionally, we incorporated strategies described in Chapter 5 into our library and applied optimization techniques detailed in Chapter 6. To ensure bit-exact results, our implementations were tested against official test vectors [14], [1]. Our endian-agnostic implementation of RG255 was developed and initially tested on a PC platform running Windows 10 Pro 64-bit OS, which operates on

a little-endian architecture with an Intel i5-8250U CPU, as depicted in Figure 7.1. The properties of PC platform are discussed in detail in Chapter 4. Test include a coplex testing utilizing xxHash and approach described in Section 7.3.

```

P-NEGATIVE VECTOR TEST no.7: SUCCESS!
non-canonical VECTOR TEST no.0: SUCCESS!
non-canonical VECTOR TEST no.1: SUCCESS!
non-canonical VECTOR TEST no.2: SUCCESS!
non-canonical VECTOR TEST no.3: SUCCESS!
Non-square x^2 VECTOR TEST no.0: SUCCESS!
Non-square x^2 VECTOR TEST no.1: SUCCESS!
Non-square x^2 VECTOR TEST no.2: SUCCESS!
Non-square x^2 VECTOR TEST no.3: SUCCESS!
Non-square x^2 VECTOR TEST no.4: SUCCESS!
Non-square x^2 VECTOR TEST no.5: SUCCESS!
Non-square x^2 VECTOR TEST no.6: SUCCESS!
Non-square x^2 VECTOR TEST no.7: SUCCESS!
Negative xy VECTOR TEST no.0: SUCCESS!
Negative xy VECTOR TEST no.1: SUCCESS!
Negative xy VECTOR TEST no.2: SUCCESS!
Negative xy VECTOR TEST no.3: SUCCESS!
Negative xy VECTOR TEST no.4: SUCCESS!
Negative xy VECTOR TEST no.5: SUCCESS!
Negative xy VECTOR TEST no.6: SUCCESS!
Negative xy VECTOR TEST no.7: SUCCESS!
ModL VECTOR TEST: SUCCESS!
s = -1 VECTOR TEST: SUCCESS!

*****
-----CONCLUSION-----

ALL TESTS RAN SUCCESSFULLY!
*****
Test no.0, xxHash: 6fd7b14d
Test no.2, xxHash: 31b558a5
Test no.4, xxHash: 73d80146
Test no.6, xxHash: 9514ac5d
Test no.8, xxHash: 0f9c1e29

*****
-----COMPLEX TEST-----

ALL TESTS RAN SUCCESSFULLY!
*****

Final xxHash Digest: c8ac3345
C:\SHARED\ristretto255>S

```

Figure 7.1: Terminal output of successful Ristretto255 tests performed on a PC platform running Windows 10 Pro 64-bit OS, operating on a little-endian architecture

## 7.2 Testing Big Endian in QEMU

This section is dedicated to testing code written for big endian systems. In section 5.4, we explained how we built an endian-agnostic implementation of Ristretto255, meaning it runs on both little endian and big endian systems without conditional compilation. When employing such an approach, errors can easily be introduced into the code, especially when doing so for the first time. Therefore, we need a way to test our code on a big endian device. However, during the development of our Ristretto255 implementation, we did not have access to a device with such hardware. Thus, we needed to find a way to test big endian code on little endian devices. During our research, we stumbled upon an emulator, QEMU [47], in combination with the GNU debugger, which offers exactly what

we need. For more information refer to Section 4.2.

```

Non-square x^2 VECTOR TEST no.0: SUCCESS!
Non-square x^2 VECTOR TEST no.1: SUCCESS!
Non-square x^2 VECTOR TEST no.2: SUCCESS!
Non-square x^2 VECTOR TEST no.3: SUCCESS!
Non-square x^2 VECTOR TEST no.4: SUCCESS!
Non-square x^2 VECTOR TEST no.5: SUCCESS!
Non-square x^2 VECTOR TEST no.6: SUCCESS!
Non-square x^2 VECTOR TEST no.7: SUCCESS!
Negative xy VECTOR TEST no.0: SUCCESS!
Negative xy VECTOR TEST no.1: SUCCESS!
Negative xy VECTOR TEST no.2: SUCCESS!
Negative xy VECTOR TEST no.3: SUCCESS!
Negative xy VECTOR TEST no.4: SUCCESS!
Negative xy VECTOR TEST no.5: SUCCESS!
Negative xy VECTOR TEST no.6: SUCCESS!
Negative xy VECTOR TEST no.7: SUCCESS!
ModL VECTOR TEST: SUCCESS!
s = -1 VECTOR TEST: SUCCESS!

*****
-----CONCLUSION-----

ALL TESTS RAN SUCCESSFULLY!
*****
Test no.0, xxHash: 6fd7b14d
Test no.2, xxHash: 31b558a5
Test no.4, xxHash: 73d80146
Test no.6, xxHash: 9514ac5d
Test no.8, xxHash: 0f9c1e29
*****
-----COMPLEX TEST-----

ALL TESTS RAN SUCCESSFULLY!
*****
Final xxHash Digest: c8ac3345debian@debian11:~/ristretto255$

```

Test vectors

Extensive numerical tests

Final digest - identical to digest on little-endian platform

Figure 7.2: Terminal output of successful Ristretto255 tests performed on big-Endian platform running on Power-PC64 in QEMU emulator

QEMU is an open-source machine emulator and virtualizer that enables users to execute operating systems and programs intended for one machine architecture (such as an ARM-based system) on a different machine architecture (such as an x86-based PC), without requiring the original hardware. It supports emulation for a variety of architectures, including x86, ARM, PowerPC, and others. QEMU supports several different operation modes. One commonly used mode is System Emulation mode, which provides a virtual model of an entire machine, including the central processing unit (CPU), memory, and emulated devices, to run a guest OS. This mode may also work with a hypervisor such as KVM, Xen, or Hypervisor. Another supported mode of QEMU is User Mode Emulation, in which QEMU is capable of executing processes compiled for a different instruction set. In this mode, system calls are adapted to accommodate endianness differences and 32/64-bit mismatches. The primary function of user-mode emulation is to facilitate fast cross-compilation and cross-debugging. This mode, User Mode Emulation, is the one we utilized for testing and debugging our endian-agnostic code for big endian systems.

When we searched for the best way to set up QEMU and GNU debugger to test code on big endian devices, we came across a very helpful article [48], where

the author describes a step-by-step setup and debugging process. However, the article is tailored for Linux-based operating systems, which may not be a perfect solution for everyone, especially Windows users. Therefore, we provide a solution for Windows available in Appendix A.

### 7.3 Deep Testing of Ristretto255

Ensuring the accuracy and reliability of our implementation, we've designed a comprehensive testing strategy. This strategy includes two distinct sets of tests, each serving a specific purpose in our verification process. The first set of tests utilizes traditional testing methodologies with official test vectors serving as a foundational reference point. While this approach provides valuable insights into the correctness of our implementation, we recognize its limitations. Official test vectors, though rigorous, may not encompass every possible scenario or edge case that our RG255 implementation might encounter in real-world usage. Thus, while testing with test vectors forms an essential component of our validation efforts, we acknowledge the need for additional measures to ensure wider coverage. Consequently, we've developed a supplementary suite of tests, which we refer to as extensive numerical tests. Unlike traditional vector tests which compare result values to test vectors, these numerical tests are designed to simulate real-world usage conditions more closely. They are designed to run potentially millions of iterations, providing a rigorous examination of our implementation's behavior across a wide spectrum of inputs and scenarios. These tests allow users to choose the number of iterations according to their specific requirements, ensuring adaptability to diverse use cases.

The construction of these extensive numerical tests involves the utilization of multiple Ristretto255 encoding and decoding function calls (recall Algorithms 1 and 2). These functions are employed for comparing two points on the curve or hashing a curve point using the xxHash hash function. The transformation of a curve point into its byte-string form is necessary for actual hashing. This transformation is achieved using encoding Algorithm 1.

It's important to note that instead of directly comparing two points using their internal representation, we first perform an encoding transformation to retrieve a byte-string representation of each point. Subsequently, we compare the two points based on their byte-string form and then transform them back to a curve point using decoding functions, solely for testing RG255 functions.

We also employed modular arithmetic modulo  $L$ , including modular multi-

plicative inversion. Our numerical test design is presented in detail in Algorithm 8.

We also integrated a fast non-cryptographic hash function called xxHash [69] into our extensive numerical tests. Its purpose is to hash intermediate calculations to ensure that the result of our numerical test is not calculated correctly by accident, such as unawarely introducing errors during the calculation process that would still yield the correct result. For incorporating this approach, we used the xxHash hash function.

The xxHash stands out as an exceptionally rapid hash algorithm, capable of processing at RAM speed limits. The codebase exhibits high portability and ensures hash consistency across all platforms, regardless of endianness. An official library written by Yann Collet [69] contains 3 implementations of xxHash:

- **XXH32**: Generates 32-bit hashes via 32-bit arithmetic.
- **XXH64**: Generates 64-bit hashes via 64-bit arithmetic.
- **XXH3** (since v0.8.0): Generates either 64- or 128-bit hashes using vectorized arithmetic, with the latter variant termed XXH128.

Note that even though we've created and tested the Ristretto255 group transformation and subsequently developed the entire OPAQUE protocol optimized for ARM Cortex-M4 microprocessors, our implementation is not yet ready for production. Currently, our library does not utilize a True Pseudorandom Number Generator (TRNG). Instead, for simplicity's sake, we employ a linear congruential generator [70], as shown in code-block 7.1. We also used our RNG in extensive numerical tests (refer to Algorithm 8) where we first initialized a seed (see line 2) and then generated pseudo-random scalars (see line 8). Utilization of the same seed is crucial to replicate numerical tests across multiple platforms and get identical results.

Additionally, as mentioned in Section 3, we haven't integrated any key stretching function (KSF) like Argon. Instead, we've simply used the Identity function, which essentially functions as a memcpy. Since the main goal of this thesis is to create a compact implementation of the OPAQUE protocol for embedded systems, presenting and integrating various optimization techniques and other useful features, such as writing endian-agnostic code that can run on both little and big endian systems, we have chosen not to focus on KSF or TRNG to keep the code simple. For more information please refer to Appendix A.

---

**Algorithm 8** Deep Testing of Ristretto255 Transformation
 

---

**Input:** COMPLEX\_TEST\_ITERATIONS - an integer indicating the number of iterations

**Output:**  $s$  error/success - report message

```

1: xxHash_init()
2:  $prng\_seed \leftarrow s\_rand()$ 
3: Initialize  $k$  to 0 {Variable used at the end of test}
4: Generate valid initial Ristretto point  $P_{init}$ 
5:  $P1 \leftarrow P_{init}$ 
6: xxHash_update( $P1$ )
7: for  $i = 1$  to COMPLEX_TEST_ITERATIONS do
8:   Generate a pseudo-random scalar  $SCALAR$ 
9:   Calculate  $SCALAR^{-1} \equiv$  inverse of  $SCALAR \pmod{L}$ 
10:  Calculate  $P2 := P1 \times SCALAR$ 
11:  xxHash_update( $SCALAR$ )
12:  xxHash_update( $SCALAR^{-1}$ )
13:  xxHash_update( $P2$ )
14:  if  $(P2 \times SCALAR^{-1}) \pmod{L} = P1$  then
15:     $k \leftarrow k \times SCALAR \pmod{L}$ 
16:  else
17:    Test has failed
18:  end if
19:   $P1 \leftarrow P2$ 
20: end for
21:  $P_{final} \leftarrow P1$ 
22: if  $P_{init} = P_{final} \times k^{-1} \pmod{L}$  then
23:   Test ran successfully
24: else
25:   Test has failed
26: end if

```

---

```

1 #define RAND_MAX ((1U << 31) - 1)
2
3 static inline int32_t rand() {
4     return rseed = (rseed * 1103515245 + 12345) & RAND_MAX;
5 }

```

Source Code 7.1: Linear congruential generator

## 7.4 Target ARM Cortex M4 platform/board

This section presents an experimental results obtained from the development board STM32F4DISCOVERY discussed in detail in Section 4. Experimental measurements were performed using the GCC ARM compiler from ARM GNU Toolchain, Version 13.2 Rel1, available on website<sup>1</sup>. The compiler was primarily configured with the optimization level set to `-Os`, a flag that prioritizes code size optimization during the compilation process.

The board features an integrated ST-LINK/V2 debugger and programmer, allowing for seamless programming and debugging of the microcontroller through the built-in ST-LINK interface, eliminating the need for an external debugger. In our experiments, the board was connected to a computer via USB, providing power and a communication interface for debugging. Additionally, board utilizes the Instrumentation Trace Macrocell (ITM) and DWT CYCCNT (Data Watchpoint and Trace Cycle Counter) in conjunction with ST-LINK/V2. The ITM was employed to redirect the `printf` function to ITM ports for code instrumentation. At the same time, DWT CYCCNT, situated in the ARM Cortex-M4 processor core, served as a cycle counter for measuring execution time and conducting performance profiling. This setup was already pre-configured and ready to use conveniently.

Firstly, we focus on the measurement of arithmetic operations in  $\mathbb{GF}(p)$ , which was our main priority during the optimization of our implementation. This layer is fundamental and forms the basis for all other layers in the pyramid, as depicted in Figure 6.1.

The measurements (see Table 7.1 and Table 7.2) indicate that MonoCypher yields the fastest performance among all examined implementations (written in C) of  $\mathbb{GF}(2^{255} - 19)$  operations. Nevertheless, the optimized version of the  $8 \times 32$ -bits internal representation of the field element written in ASM stands out as

<sup>1</sup><https://developer.arm.com/downloads/-/arm-gnu-toolchain-downloads>



the absolute fastest as we expected, eliminating the need for additional conversion routines. Utilizing optimized ASM implementation for the  $8 \times 32$ -bits internal representation of the field element enables computations within the modulo  $(2^{256} - 36)$  domain, with additional reduction to modulo  $(2^{255} - 19)$  required in only a few logical places.

Table 7.1: Speed measurements for the most critical functions operating over  $\mathbb{GF}(2^{255} - 19)$  in various libraries. These libraries employ distinct  $\mathbb{GF}$  element representations, and the results are presented in cycles. Measurements were performed on the ARM Cortex-M4, with 168 MHz clock frequency, using the GCC ARM compiler with the -Os flag set.

Implementations	GF MUL (CPU cycles)	GF SQR (CPU cycles)	GF ADD (CPU cycles)
TweetNaCl	9887	NA	232
Monocypher	833	505	130
Cyclone (C)	1671	1671	400
Our Optimized (ASM)	249	195	80

Table 7.2: Continuation of speed measurements from Table 7.1. for the most critical functions operating over  $\mathbb{GF}(2^{255} - 19)$  in various libraries. Measurements were performed on the ARM Cortex-M4, with 168 MHz clock frequency, using the GCC ARM compiler with the -Os flag set.

Implementations	GF SUB (CPU Cycles)	PACK/UNPACK (CPU Cycles)	GF RED (CPU Cycles)
TweetNaCl	241	4359/266	NA
Monocypher	130	460/369	NA
Cyclone (C)	426	NA	NA
Our Optimized (ASM)	93	NA	88

By integrating a relatively small piece of code written in ASM (approximately 2.5 kB, see Table 7.3 and Table 7.4) and employing the little Fermat’s theorem along with calculations in Montgomery’s domain, we can significantly enhance performance. This strategic inclusion of highly efficient ASM code optimizes the overall implementation, resulting in a significant decrease in the execution time of the modular inversion, which can be seen in TABLE 7.5.

The speed measurements for modular multiplicative inversion modulo  $L$  (InvModL), including supporting functions and various optimization techniques, are presented in Table 7.5. The results reveal that the unoptimized C variant consumes twice as many CPU cycles as the optimized implementation, wherein the

most critical part was replaced with ASM code. Table 7.3 illustrates the memory usage in bytes for each function written in ASM. This emphasizes that employing a small ASM code snippet can enhance the speed of RG255 as well as modular multiplicative inversion modulo  $L$ .

Table 7.3: Memory usage for functions operating within the modulo  $(2^{256} - 36)$  domain in highly-efficient ASM implementation. The memory requirements of each function are presented in bytes.

Implementations	GF MUL (Bytes)	GF SQR (Bytes)	GF ADD (Bytes)	GF SUB (Bytes)
Our Optimized (ASM)	528	412	104	126

Table 7.4: Continuation of Table 7.3 for memory usage of functions operating within the modulo  $(2^{256} - 36)$  domain in highly-efficient ASM implementation. The memory requirements of each function are presented in bytes.

Implementations	GF RED (Bytes)	REDC (Bytes)	MUL256 (Bytes)
Our Optimized (ASM)	152	524	420

Table 7.5: Speed for InvModL, including supporting functions and various optimization techniques. Measurements were performed on the ARM Cortex-M4, with 168 MHz clock frequency, using the GCC ARM compiler with the -Os and -O3 flag set.

Function	-Os	-O3
MUL256 (C)	886	822
MUL256 (ASM)	234	237
Inv_Mont (C)	1,405,478	833,262
Inv_Mont (ASM)	672,957	453,760

Next, we performed measurements for individual parts of our library such as Ristretto255 transformation (encode, decode, hash\_to\_group), InvModL, and the entire OPAQUE, focusing on measurements from the client side, as clients typically have fewer resources than servers in practice.

Table 7.6 illustrates measurements performed on the RG255 functions, presented in cycles. The evaluations were performed using the GCC ARM compiler, with the -Os and -O3 flags configured for functions implemented in C. These flags were also utilized when measuring functions implemented using the optimized C+ASM approach. From the table, it can be seen that optimization with ASM

significantly improves performance compared to C implementation. Additionally, compilation with the `-O3` flag demonstrates slightly better performance than with the `-Os` flag for ASM implementation. It's worth noting that the optimized (C + ASM) versions of RG255 functions, such as encode and decode make up only around 2.5% of the execution time of scalar multiplication of curve points, compared to approximately 5% for the unoptimized (pure C) versions. Therefore, we can conclude that RG255 adds just a thin layer abstraction and presents very small overhead, as shown in table 7.6, where encoding and decoding require considerably fewer cycles compared to scalar multiplication or modular inversion.

Table 7.6: Speed for RG255 core functions and InvModL. The results are presented in cycles. The measurements were performed on the ARM Cortex-M4, with 168 MHz clock frequency, using the GCC ARM compiler, with the `-Os` and `-O3` flag set.

Function	C (CPU Cycles)		C+ASM (CPU Cycles)	
	-Os	-O3	-Os	-O3
DECODE	515 021	381 982	57 335	54 926
SCALARMULT	11 319 857	7 926 035	2 560 876	2 148 528
ENCODE	520 415	386 192	58 825	56 103
HASH_TO_GROUP	1 593 455	1 180 044	184 358	174 865
InvModL	1 665 584	978 602	932 763	602 244
InvModL (no wipe buffers)	1 405 478	833 362	672 957	453 760

Table 7.7 presents the memory usage for RG255 functions that have been implemented in pure C. The memory requirements for each function are provided in bytes, offering a clear breakdown of the resources required by these functions.

Table 7.7: Memory usage for RG255 functions implemented in pure C. The memory requirements of each function are presented in bytes.

Function	Size (Bytes)
DECODE	415
SCALARMULT	293
ENCODE	435
HASH_TO_GROUP	947

The measurements of the OPAQUE protocol were conducted on the client side, specifically during the registration and authentication phases performed by the user. The measurements are presented in Table 7.8 and Table 7.9. Tables 7.8 and

7.9 detail measurements in cycles for each message generated on the client side. This approach was chosen under the assumption that the server is potentially not heavily resource-constrained. From Tables 7.8 with the optimization flag `-Os` and 7.9 with the optimization flag `-O3`, we can see that the Registration phase is around 2.5 times faster than the Authentication (login) phase in each setup. Additionally, our optimized approach, where we combine a fast C implementation and highly efficient ASM implementations of  $\mathbb{GF}$  operations [31], [55], coupled with various optimization techniques discussed in Chapter 6, is about 3.8 times faster compared to an implementation that is written purely in C. Our optimized library incorporates secure buffer wiping (see Section 5.3), which is an optional feature that can be disabled. When buffer wiping is disabled, performance slightly increases, as shown in Tables 7.8 - 7.11

Table 7.8: Speed of the OPAQUE registration and authentication phases were performed on the client-side, using our optimization techniques, including a combination of fast C and efficient ASM  $\mathbb{GF}$  operations. The results are presented in cycles. Measurements were performed on the ARM Cortex-M4, with 168 MHz clock frequency, using the GCC ARM compiler with the `-Os` flag.

Setup	Registration Phase (CPU Cycles)	Login Phase (CPU Cycles)
ASM no wipe	7 883 913	19 548 243
ASM with wipe	10 215 861	25 314 003
Pure C no wipe	38 424 486	98 641 278

Table 7.9: Speed of the OPAQUE registration and authentication phases were performed on the client-side, using our optimization techniques, including a combination of fast C and efficient ASM  $\mathbb{GF}$  operations. The results are presented in cycles. Measurements were performed on the ARM Cortex-M4, with 168 MHz clock frequency, using the GCC ARM compiler with the `-O3` flag.

Setup	Registration Phase (CPU Cycles)	Login Phase (CPU Cycles)
ASM no wipe	7 230 664	18 179 008
ASM with wipe	8 442 296	21 158 093
Pure C no wipe	27 927 997	71 063 248

Table 7.10: Extension of Table 7.8 shows speed measurements for each OPAQUE message generated on the client-side during both registration and authentication phases, utilizing our optimization techniques including a combination of fast C and highly efficient ASM  $\mathbb{G}_F$  operations. The results are presented in cycles. Measurements were performed on the ARM Cortex-M4, with 168 MHz clock frequency, using the GCC ARM compiler with the -Os flag set.

Setup	Registration Phase		Login Phase	
	Reg. Request	Reg. Record	KE1	KE3
ASM no wipe	2 289 999	5 593 924	4 438 381	15 109 863
ASM with wipe	2 988 137	7 227 744	5 823 925	19 490 089
Pure C no wipe	13 409 562	25 014 934	25 268 451	73 372 828

Table 7.11: Extension of Table 7.8 shows speed measurements for each OPAQUE message generated on the client-side during both registration and authentication phases, utilizing our optimization techniques including a combination of fast C and highly efficient ASM  $\mathbb{G}_F$  operations. The results are presented in cycles. Measurements were performed on the ARM Cortex-M4, with 168 MHz clock frequency, using the GCC ARM compiler with the -O3 flag set.

Setup	Registration Phase		Login Phase	
	Reg. Request	Reg. Record	KE1	KE3
ASM no wipe	2 189 663	5 041 000	4 234 479	13 944 523
ASM with wipe	2 548 919	5 893 381	5 893 381	16 210 575
Pure C no wipe	9 635 946 2	18 292 050	18 118 144	52 945 098

Table 7.12 provides memory requirements for OPAQUE messages exchanged between the client and server during both the registration and authentication phases. In the registration phase, the client generates and sends a RegistrationRequest and RegistrationRecord, while the server generates and sends a RegistrationResponse message. During authentication, the client generates KE1 and KE3 messages and the server generates KE2. Table 7.12 illustrates that the messages exchanged between the client and server are relatively small in size. This can be particularly advantageous for embedded systems, where resources such as memory and processing power are often constrained.

Table 7.12: Memory requirements of the OPAQUE messages generated and exchanged between the client and server, measured in bytes.

Message Type	Size (Bytes)
RegistrationRequest	32
RegistrationResponse	64
RegistrationRecord	192
KE1	96
KE2	320
KE3	64

To summarize, our approach integrates the efficient  $8 \times 32$ -bits representation of  $\mathbb{GF}$  elements with a compact yet highly efficient ASM code segment [31], [55], alongside our sophisticated reduction technique. Such a combination appears to yield an optimal solution for ARM Cortex-M4. Using this technique and several other optimization techniques discussed in Chapter 6, we prepared a fast, compact and memory-efficient Ristretto255 transformation that was subsequently utilized in our OPAQUE protocol implementation. Our library serves as a compliant aPAKE operating at the 128-bit security level.

## 8 Discussion

---

In this discussion, we compare an existing implementation provided in research [71] of the OPAQUE protocol designed for embedded systems to our own implementation. Specifically, we examine an implementation intended for embedded environments that relies on resource-intensive cryptographic libraries, which may not be optimal for small resource-constrained embedded devices.

Examined implementation [71] focuses on evaluating the feasibility and efficiency of this approach, particularly on more powerful MCUs. By analyzing these measurements, we aim to identify potential trade-offs in our optimized implementation for OPAQUE's deployment in resource-constrained embedded systems.

### OPAQUE Variants for Robust ARM Architectures

During our research, we could not find any optimized implementations of the OPAQUE protocol for relatively less powerful platforms, so we created our own fast and compact implementation targeting ARM Cortex-M4. However, there are a couple of implementations discussed in research [71] which provide measurements performed on single-board computers like the ODROID-N2<sup>1</sup> with Linux OpenWrt 2.6.36.4brcmarm (featuring ARM Cortex-A73) and the Wi-Fi router Asus RT-AC66U B1<sup>2</sup> running Linux OpenWrt with ARM Cortex-A9.

This subsection aims to showcase measurements performed on various implementations of the OPAQUE protocol using different cores, comparing them with our implementation designed for the ARM Cortex-M4.

---

<sup>1</sup>ODROID-N2: <https://www.hardkernel.com/shop/odroid-n2-with-4gbyte-ram-2/>

<sup>2</sup>Asus RT-AC66U B1: <https://www.asus.com/networking-iot-servers/wifi-routers/asus-wifi-routers/rt-ac66u-b1/>

Table 8.1 shows the clock frequencies of all the cores that we are comparing.

ARM Core	Clock Frequency
Cortex A73	1.8 GHz
Cortex-A9	1 GHz
Cortex M4	168 MHz

Table 8.1: Measured Clock Frequencies of ARM Cores

Table 8.2: Comparison of implementations on CPUs with different ARM cores

CPU Type	Implementation	Reg. Phase [ms]	Auth. Phase [ms]
ARM Cortex-A73	<i>opaque_sha</i>	0.789	5.034
ARM Cortex-A9	<i>opaque_sha</i>	2.222	36.666
ARM Cortex-M4	Our Implementation	50.251	125.941

Table 8.2 displays measurements for the ARM Cortex A73 and Cortex-A9 implementations as presented in research [71] called *opaque\_sha*, in comparison to our optimized implementation targeting the ARM Cortex M4. The clock frequency measurements indicate notable differences among the ARM Cortex cores. The Cortex A73 exhibits the highest clock frequency among the compared cores, followed by the Cortex-A9 and Cortex M4, in decreasing order of performance.

The higher clock frequencies of the ARM Cortex A73 and ARM Cortex-A9 cores, with an OPAQUE implementation based on utilizing the libsodium [58] and TweetNaCl [52] cryptographic libraries, suggest greater computational capabilities but may come at the cost of increased power consumption compared to the ARM Cortex M4, which is optimized for efficiency in embedded applications.

It's also important to note that while we used the Identity function (see section 3.2.3), the implementation provided in [71] utilizes a slightly better option a SHA256 hash function. Additionally, the implementation [71] utilizes the libsodium cryptographic library, which employs an 8-word internal representation of  $\mathbb{GF}$  elements designed to run faster on more powerful CPUs.

Although the research in [71] does not provide information about the processor memory stack requirements of the OPAQUE protocol, we have decided to include this information for readers who may be interested in such details for potential use. Additionally, all implementation details are provided directly in well-documented code, available in Appendix A.



Table 8.3 presents the processor memory stack requirements of the functions used during the generation of OPAQUE messages in both the registration and authentication phases on the client-side. These measurements were specifically conducted for OPAQUE functions implemented in pure C. The stack memory usage, represented in bytes, offers valuable insights into the resource demands of critical cryptographic operations within the OPAQUE protocol

Table 8.3: The processor memory stack requirements for the OPAQUE functions used during the generation of OPAQUE messages in both the registration and authentication phases on the client-side were measured in bytes. These measurements were performed on functions implemented in pure C.

Message Type	Size (Bytes)
RegistrationRequest	1191
RegistrationRecord	1963
KE1	1330
KE3	3046

## 9 Conclusion

---

This thesis describes the OPAQUE protocol, a modern asymmetric password-authenticated key exchange (aPAKE), its potential usage in real-world applications, and provides a detailed description of how it actually works. The thesis also focuses on fundamental principles when integrating a non-prime order group into the OPAQUE protocol, such as Curve25519. We addressed security issues by applying a thin layer of transformation to transition into a more secure abstract group called the Ristretto255 group.

Subsequently, the thesis focuses on optimizing the Ristretto255 transformation (RG255), which is suitable for safely implementing higher-level and complex cryptographic protocols. We analyzed RG255 and identified the most time-critical parts, particularly operations in  $\mathbb{GF}(2^{255} - 19)$ .

During optimization, we analyzed multiple implementations written in the C language and performed speed measurements. Drawing inspiration from analyzed libraries, we created our implementation that uses highly efficient ASM code to accelerate the most time-critical operations.

Experimental measurements were conducted to determine the effectiveness of individual implementations. We proposed and followed a strategy for optimization, which ultimately led us to the conclusion that we successfully optimized the entire RG255 as well as much more complex cryptographic protocol OPAQUE.

We also considered minimal memory requirements, efficient stack manipulation, and maintained constant-time operations. Additionally, we integrated buffer wiping into our implementation to ensure responsible memory clearance.

In this thesis, we focused on optimizing RG255, which constituted a significant part of our effort. More specifically, our aim was to optimize computations in  $\mathbb{GF}(2^{255} - 19)$ . The RG255 is reasonably optimized and suitable for integration into other protocols that require working with POGs. The optimization of computations in  $\mathbb{GF}(2^{255} - 19)$  accounts for approximately 20 % of the total execution time of the OPAQUE protocol, providing opportunities for further optimization in future research.

Finally, it is worth noting that some of the results from this study have already been published in the article titled "Optimization of Ristretto255 Group Implementation for Cortex-M4 based Cryptographic Applications" authored by E. Kupcova, P. Zelenak, M. Pleva, and M. Drutarovsky, presented at the International Conference Radioelektronika in April 2024 [72].

# Bibliography

---

1. BOURDREZ, Daniel; KRAWCZYK, Dr. Hugo; LEWI, Kevin; WOOD, Christopher A. *The OPAQUE Augmented PAKE Protocol*. Internet Engineering Task Force, 2024-03. Internet-Draft, draft-irtf-cfrg-opaque-14. Internet Engineering Task Force. Available also from: <https://datatracker.ietf.org/doc/draft-irtf-cfrg-opaque/14/>. Work in Progress.
2. FETT, Daniel; YASUDA, Kristina; CAMPBELL, Brian. *Selective Disclosure for JWTs (SD-JWT)*. Internet Engineering Task Force, 2024-03. Internet-Draft, draft-ietf-oauth-selective-disclosure-jwt-08. Internet Engineering Task Force. Available also from: <https://datatracker.ietf.org/doc/draft-ietf-oauth-selective-disclosure-jwt/08/>. Work in Progress.
3. RESCORLA, Eric; OKU, Kazuho; SULLIVAN, Nick; WOOD, Christopher A. *TLS Encrypted Client Hello*. Internet Engineering Task Force, 2024-03. Internet-Draft, draft-ietf-tls-esni-18. Internet Engineering Task Force. Available also from: <https://datatracker.ietf.org/doc/draft-ietf-tls-esni/18/>. Work in Progress.
4. GREEN, Matthew. *Let's talk about PAKE*. 2018. Available also from: <https://blog.cryptographyengineering.com/2018/10/19/lets-talk-about-pake/>.
5. SAINT GUILHEM, Cyprien Delpech de; FISCHLIN, Marc; WARINSCHI, Bogdan. Authentication in Key-Exchange: Definitions, Relations and Composition. In: *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*. 2020, pp. 288–303. Available from doi: 10.1109/CSF49147.2020.00028.
6. MARLINSPIKE, M.; PERRIN, T. *The X3DH Key Agreement Protocol*. 2016. Available also from: <https://signal.org/docs/specifications/x3dh/x3dh.pdf>.
7. DAVIDSON, Alex; FAZ-HERNANDEZ, Armando; SULLIVAN, Nick; WOOD, Christopher A. *Oblivious Pseudorandom Functions (OPRFs) Using Prime-Order*

- Groups* [RFC 9497]. RFC Editor, 2023. Request for Comments, no. 9497. Available from DOI: 10.17487/RFC9497.
8. LAKE J. *What is a key derivation function (KDF)? A comprehensive guide*. 2023. Available also from: <https://www.comparitech.com/blog/information-security/key-derivation-function-kdf/>.
  9. KRAWCZYK, Dr. Hugo; BELLARE, Mihir; CANETTI, Ran. *HMAC: Keyed-Hashing for Message Authentication* [RFC 2104]. RFC Editor, 1997. Request for Comments, no. 2104. Available from DOI: 10.17487/RFC2104.
  10. HANSEN, Tony; 3RD, Donald E. Eastlake. *US Secure Hash Algorithms (SHA and HMAC-SHA)* [RFC 4634]. RFC Editor, 2006. Request for Comments, no. 4634. Available from DOI: 10.17487/RFC4634.
  11. BIRYUKOV, Alex; DINU, Daniel; KHOVRATOVICH, Dmitry; JOSEFSSON, Simon. *Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications*. Internet Engineering Task Force, 2021-03. Internet-Draft, draft-irtf-cfrg-argon2-13. Internet Engineering Task Force. Available also from: <https://datatracker.ietf.org/doc/draft-irtf-cfrg-argon2/13/>. Work in Progress.
  12. VASYLENKO O. *The OPAQUE Asymmetric PAKE Protocol. Make authentication secure again*. 2023. Available also from: <https://medium.com/@oleksiiv.vasylenko/the-opaque-asymmetric-pake-protocol-make-authentication-secure-again-366f821a319d>.
  13. BERNSTEIN, Daniel J. Curve25519: new Diffie-Hellman speed records. In: *Public Key Cryptography-PKC 2006*. Springer, 2006, pp. 207–228. Available also from: <https://cr.ypt.to/ecdh/curve25519-20060209.pdf..>
  14. VALENCE, Henry de; GRIGG, Jack; HAMBURG, Mike; LOVECRUFT, Isis; TANKERSLEY, George; VALSORDA, Filippo. *The ristretto255 and decaf448 Groups* [RFC 9496]. RFC Editor, 2023. Request for Comments, no. 9496. Available from DOI: 10.17487/RFC9496.
  15. DUBINSKY E. Dautermann J., Leron U. et al. On learning fundamental concepts of group theory. *Educational Studies in Mathematics*. 1994, vol. 27, pp. 267–305. Available from DOI: 10.1007/BF01273732.
  16. KLEPPMANN, MARTIN. *Implementing Curve25519/X25519: A tutorial on elliptic curve cryptography*. 2020. Available also from: <https://martin.kleppmann.com/papers/curve25519.pdf..>

17. BROWN, Michael; HANKERSON, Darrel; LÓPEZ, Julio; MENEZES, Alfred. Software implementation of the NIST elliptic curves over prime fields. In: *Topics in Cryptology—CT-RSA 2001*. Springer, 2001, pp. 250–265.
18. EL HOUSNI, Youssef. *Edwards curves*. 2018. Available also from: <https://hal.science/hal-01942759/document>..
19. NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY (NIST). *Elliptic Curve Cryptography (ECC) Project*. National Institute of Standards and Technology (NIST), Accessed 2024. Available also from: <https://csrc.nist.gov/projects/elliptic-curve-cryptography>.
20. KANNWISCHER, Matthias J; RIJNEVELD, Joost; SCHWABE, Peter; STOFFELEN, Ko. *pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4*. CSRC, 2019. Available also from: <https://csrc.nist.gov/CSRC/media/Events/Second-PQC-Standardization-Conference/documents/accepted-papers/kannwischer-pqm4.pdf>.
21. THALER, Justin. *Proofs, Arguments, and Zero-Knowledge*. 2023. Available also from: <https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.pdf>.
22. RESCORLA, Eric. *The Transport Layer Security (TLS) Protocol Version 1.3*. 2018. RFC, 8446. Internet Engineering Task Force (IETF). Available from DOI: 10.17487/RFC8446.
23. WHATSAPP. *WhatsApp Encryption Overview*. 2017. Available also from: <https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf>. Archived at <https://perma.cc/QD7M-GPG5>.
24. MARLINSPIKE, Moxie; PERRIN, Trevor. *The X3DH Key Agreement Protocol*. 2016. Available also from: <https://www.signal.org/docs/specifications/x3dh/>. Archived at <https://perma.cc/MSA4-DP4G>.
25. ARCIERI, T. *libristretto*. 2019. Available also from: <https://github.com/Ristretto/libristretto255>.
26. VALENCE, H de; GRIGG, J; HAMBURG, M; LOVECRUFT, I; TANKERSLEY, G; VALSORDA, F. *RFC 9496: The ristretto255 and decaf448 Groups*. RFC Editor, 2023. Available also from: <https://datatracker.ietf.org/doc/draft-hdevalence-cfrg-ristretto/>..
27. HAMBURG, Mike. *Decaf: Eliminating cofactors through point compression*. 2015. Available also from: [www.shiftright.org/papers/decaf/decaf.pdf](http://www.shiftright.org/papers/decaf/decaf.pdf)..

28. HISIL, Huseyin; WONG, Kenneth Koon-Ho; CARTER, Gary; DAWSON, Ed. Twisted Edwards curves revisited. In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2008, pp. 326–343. Available also from: <https://eprint.iacr.org/2008/522..>
29. NGUYEN, Dang. *Correspondence Between Elliptic Curves in Edwards-Bernstein and Weierstrass Forms* [Department of Mathematics and Statistics, University of Ottawa]. 2017. Available also from: <https://mysite.science.uottawa.ca/mnevins/papers/NguyenMScProj2017.pdf>. Supervisor: Professor Monica Nevins.
30. HISIL, Huseyin; WONG, Kenneth Koon-Ho; CARTER, Gary; DAWSON, Ed. *Twisted Edwards Curves Revisited* [Cryptology ePrint Archive, Paper 2008/522]. 2008. Available also from: <https://eprint.iacr.org/2008/522>. <https://eprint.iacr.org/2008/522>.
31. HAASE, B. *fe25519*. Available also from: <https://github.com/BjoernMHaase/fe25519>.
32. ORYX EMBEDDED. *CycloneCRYPTO*. Available also from: <https://oryx-embedded.com/products/CycloneCRYPTO.html>.
33. ZELENAK, P. *OPAQUEC*. Available also from: <https://github.com/Alg0ritmus/OPAQUE%5C-C..>
34. MENEZES, Alfred J.; OORSCHOT, Paul C. van; VANSTONE, Scott A. *Handbook of Applied Cryptography*. CRC Press, 2001. Available also from: <http://www.cacr.math.uwaterloo.ca/hac/>.
35. BELLOVIN, Steven M.; MERRITT, Michael. Augmented encrypted key exchange: a password-based protocol secure against dictionary attacks and password file compromise. In: *Proceedings of the 1st ACM Conference on Computer and Communications Security*. Fairfax, Virginia, USA: Association for Computing Machinery, 1993, pp. 244–250. CCS '93. ISBN 0897916298. Available from DOI: 10.1145/168588.168618.
36. RESCORLA, Eric. *Diffie-Hellman Key Agreement Method* [RFC 2631]. RFC Editor, 1999. Request for Comments, no. 2631. Available from DOI: 10.17487/RFC2631.
37. HU, Xinyi; ZHANG, Junzuo; ZHANG, Zhenfeng; AL., et. Universally composable anonymous password authenticated key exchange. *Sci. China Inf. Sci.* 2017, vol. 60, p. 52107. Available from DOI: 10.1007/s11432-016-5522-z.

38. LEMON, Ted; CHESHIRE, Stuart. *Service Registration Protocol for DNS-Based Service Discovery*. Internet Engineering Task Force, 2024-03. Internet-Draft, draft-ietf-dnssd-srp-25. Internet Engineering Task Force. Available also from: <https://datatracker.ietf.org/doc/draft-ietf-dnssd-srp/25/>. Work in Progress.
39. GREEN, Matthew. *Should you use SRP?* Available also from: <https://blog.cryptographyengineering.com/should-you-use-srp/>.
40. TAYLOR, David; PERRIN, Trevor; WU, Thomas; MAVROGIANNOPOULOS, Nikos. *Using the Secure Remote Password (SRP) Protocol for TLS Authentication [RFC 5054]*. RFC Editor, 2007. Request for Comments, no. 5054. Available from DOI: 10.17487/RFC5054.
41. SOCARDE P. *SRP: A Zero-Knowledge Protocol for Password Authentication*. 2023. Available also from: <https://medium.com/@psocarde/srp-a-zero-knowledge-protocol-for-password-authentication-1e19582aab29>.
42. KRSTIC J. *Behind the Scenes with iOS Security*. 2016. Available also from: <https://www.blackhat.com/docs/us-16/materials/us-16-Krstic.pdf>.
43. KRAWCZYK, Dr. Hugo; ERONEN, Pasi. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF) [RFC 5869]*. RFC Editor, 2010. Request for Comments, no. 5869. Available from DOI: 10.17487/RFC5869.
44. PYTHON TEAM. *Python*. Available also from: <https://www.python.org/>.
45. STMICROELECTRONICS. *Discovery kit with STM32F407VG MCU \* New order code STM32F407G-DISC1*. Available also from: <https://www.st.com/en/evaluation-tools/stm32f4discovery.html>.
46. HAASE, Bjorn; LABRIQUE, Benoit. AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT. *IACR Transactions on Cryptographic Hardware and Embedded Systems*. 2019, vol. 2019, no. 2, pp. 1–48. Available from DOI: 10.13154/tches.v2019.i2.1-48.
47. QEMU. *QEMU - A generic and open source machine emulator and virtualizer*. Available also from: <https://www.qemu.org/>.
48. VEGA. *QEMU + GNU Debugger Basic Tutorial*. 2023. Available also from: <https://mariokartwii.com/showthread.php?tid=1998>.



49. JOSEFSSON, Simon; LIUSVAARA, Ilari. *Edwards-Curve Digital Signature Algorithm (EdDSA)* [RFC 8032]. RFC Editor, 2017. Request for Comments, no. 8032. Available from doi: 10.17487/RFC8032.
50. ROPE SECURITY. *Timing Attack*. Available also from: <https://ropesec.com/articles/timing-attacks/>.
51. VAILLANT, L. *Monocypher*. Available also from: <https://monocypher.org/>.
52. BERNSTEIN, D.J.; GASTEL, B.; ET.AL. *TweetNaCl*. Available also from: <https://tweetnacl.cr.yp.to/>.
53. *Why is a point-to-volatile pointer, like "volatile int \* p", useful?* [online]. 2022-12. [visited on 2024-04-07]. Available from: <https://stackoverflow.com/questions/9935190/why-is-a-point-to-volatile-pointer-like-volatile-int-p-useful>. Stack Overflow.
54. CENTER, Command. *The byte order fallacy*. 2016. Available also from: <https://commandcenter.blogspot.com/2012/04/byte-order-fallacy.html>.
55. EMIL, L. *X25519-Cortex-M4*. Available also from: <https://github.com/Emill/X25519-Cortex-M4>.
56. LOVECRUFT, I.; H., VALENCE. *Crate curve25519\_dalek*. Available also from: [https://doc.dalek.rs/curve25519\\_dalek/..](https://doc.dalek.rs/curve25519_dalek/)
57. TANKERSLEY, G.; VALSORDA, F.; VALENCE, H. *ristretto255*. Available also from: <https://github.com/gtank/ristretto255>.
58. FRANK, D. *libsodium*. Available also from: <https://doc.libsodium.org/advanced/point-arithmetic/ristretto..>
59. MILLER, P. *noble25519*. Available also from: <https://github.com/paulmillr/noble-ed25519..>
60. FRANK, D. *wasm-crypto*. Available also from: <https://github.com/jedisct1/wasm-crypto..>
61. NIKOLAENKO, V.; K., Lewi. *ristretto255-js*. Available also from: <https://github.com/facebook/ristretto255-js>.
62. FRANK, D. *Zig, a general-purpose programming language*. Available also from: <https://ziglang.org/..>

63. VAMPIRE LAB. *System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives*. Available also from: <https://github.com/floodyberry/supercop>.
64. ISIS AGORA LOVECRUFT. *ristretto-donna* [<https://github.com/isislovecruft/ristretto-donna>].
65. ORYX-EMBEDDED. *CycloneCRYPTO: Cryptographic library for embedded systems* [<https://github.com/Oryx-Embedded/CycloneCRYPTO>].
66. NAYUKI. *Barrett Reduction Algorithm* [<https://www.nayuki.io/page/barrett-reduction-algorithm>]. 2019.
67. MASSAR, Jeroen. *rfc6234*. Available also from: <https://github.com/massar/rfc6234/tree/master>.
68. WALTON, Jeffrey. *SHA-Intrinsics*. Available also from: <https://github.com/noloader/SHA-Intrinsics/tree/master>.
69. COLLET, Yann. *xxHash* [<https://xxhash.com/>].
70. CODE, Rosetta. *Linear congruential generator*. Available also from: [https://rosettacode.org/wiki/Linear\\_congruential\\_generator#C](https://rosettacode.org/wiki/Linear_congruential_generator#C).
71. KUPCOVA, Eva. *Implementácia OPAQUE protokolu pre IoT zariadenia*. Košice, 2023. Technická univerzita v Košiciach, Fakulta elektrotechniky a informatiky.
72. KUPCOVA, E.; ZELENAK, P.; PLEVA, M.; DRUTAROVSKY, M. Optimization of Ristretto255 Group Implementation for Cortex-M4 based Cryptographic Applications. In: *2024 34th International Conference Radioelektronika (RADIOELEKTRONIKA)*. Žilina, Slovakia, 2024.

# List of Appendixes

---

**Appendix A** Structure of Our Optimized Library.

**Appendix B** Non-constant Time Algorithm for String Comparison.

**Appendix C** Testing Big-Endian Code on QEMU Emulator in VirtualBox.

# Appendix A

---

During the development and testing codebase for this thesis, we had multiple ideas that we wanted to integrate into our library. One notable example is implementing endian-agnostic code, enabling programs to operate seamlessly across both little and big endian architectures without the need for conditional compilation. However, we recognized that this approach might not be optimal for the embedded systems we are focusing on, especially those using ARM Cortex-M4 cores (which is platform we were focused on). Consequently, we opted to create two distinct archives: one tailored for PC platforms utilizing the endian-agnostic technique (see Section 9 in Appendix A), and another specifically optimized for embedded systems, particularly those running on ARM Cortex-M4 cores (Section 0.4 in Appendix A).

## Main OPAQUE library

This section presents the code structure of our library designed for PC platforms, leveraging the endian-agnostic technique discussed in Section 5.4. This technique can be seen implemented in the `./dependencies/ristretto255.c` file. As illustrated by the code structure below, the code is organized into multiple folders. The `./dependencies/` directory contains the Ristretto255 transformation along with all supporting files, such as `gf25519.c`, which contains the C functions for  $\mathbb{GF}(p)$  operations discussed throughout this thesis.

The archive integrates all implementation approaches discussed in Chapter 5 and optimization techniques discussed in Chapter 6. Additionally, it includes the `./opaque_in_details/` folder, which serves as a detailed description of how OPAQUE works step-by-step. Within this folder, a file `opaque_simulation.c` simulates the client-server registration and authentication process for OPAQUE. The complete archive is available at <https://github.com/Algoritmus/OPAQUE-C>.

## Main archive

```
|
|_ dependencies/
|_ opaque_in_details/
|_ ristretto255/
|_ client_side.c
|_ client_side.h
|_ main_config.h
|_ Makefile
|_ opaque.c
|_ opaque.h
|_ oprf.c
|_ oprf.h
|_ OPRF.txt
|_ README.md
|_ server_side.c
|_ server_side.h
|_ test.c
|_ test.py
```

### dependencies/

```
|_ hkdf.c
|_ hmac.c
|_ README.md
|_ rfc6234.txt
|_ sha-private.h
|_ sha.h
|_ sha384-512.c
|_ usha.c
```

### opaque\_in\_details/

```
|_ importer.h
|_ initial_configuration_step_0.c
|_ offline_reg_step_1.c
|_ offline_reg_step_2.c
|_ offline_reg_step_3.c
|_ online_login_step_4.c
|_ online_login_step_5.c
|_ online_login_step_6.c
|_ online_login_step_7.c
|_ opaque_simulation.c
```

```
|_ summary_of_opaque.txt
```

## **ristretto255/**

```
|_ config.h
|_ gf25519.c
|_ gf25519.h
|_ helpers.h
|_ Makefile
|_ modl.c
|_ modl.h
|_ prng.c
|_ prng.h
|_ py_modl_l_inverse.py
|_ README.md
|_ ristretto255.c
|_ ristretto255.h
|_ ristretto255_constants.h
|_ ristretto_main.c
|_ test_config.h
|_ utils.c
|_ utils.h
|_ xxhash.c
|_ xxhash.h
```

## **OPAQUE-MCU library**

This section presents the code structure of our library specifically tailored for embedded platforms running on ARM Cortex-M4 cores, leveraging a combination of fast C libraries and highly efficient ASM  $\mathbb{GF}(p)$  operations discussed in detail in Section 6.6. This archive was used for the measurements described in Chapter 7 of this thesis, which were performed on the STM32F4DISCOVERY development board.

It's important to note that this archive is designed for little-endian architectures and contains fewer files compared to the main archive, focusing primarily on client-side functionality. Some files, such as *opaque.c*, *oprf.c*, *test.c*, *ristretto255/-ristretto255.c*, and others, differ from those in the main archive (see *README.md* for more information).

The complete archive can be accessed at [https://github.com/Algorithmus/OPAQUE-C/tree/MCU\\_version](https://github.com/Algorithmus/OPAQUE-C/tree/MCU_version).

## Archive for MCUs

```
|
├─ dependencies/
├─ ristretto255/
├─ Makefile
├─ opaque.c
├─ opaque.h
├─ oprf.c
├─ oprf.h
├─ README.md
├─ test.c
```

### dependencies/

```
|
├─ hkdf.c
├─ hmac.c
├─ README.md
├─ rfc6234.txt
├─ sha-private.h
├─ sha.h
├─ sha384-512.c
├─ usha.c
```

### ristretto255/

```
|
├─ config.h
├─ gf25519.c
├─ gf25519.h
├─ helpers.h
├─ Makefile
├─ modl.c
├─ modl.h
├─ prng.c
├─ prng.h
├─ ristretto255.c
├─ ristretto255.h
├─ ristretto255_constants.h
├─ utils.c
├─ utils.h
```

# Appendix B

---

In this thesis, we emphasize the importance of constant-time execution of algorithms in cryptographic protocols and libraries to mitigate side-channel attacks, specifically timing attacks (refer to Chapter 5.1). Constant-time algorithms ensure that the execution time of an algorithm is independent of input values. As an example, we discuss a simple string comparison algorithm that does not operate in constant time, resulting in varying execution times highly dependent on inputs.

The code snippet below illustrates the algorithm used for the results presented in Table 5.1 in Chapter 5.1. The code first measures 10,000 dummy iterations of a for loop, which is then subtracted from 10,000 iterations of string comparisons to get more precise measurements. The resulting value is divided by 10,000 to obtain the execution time for one string comparison (see line 30).

```
1 import time
2 def measure_string_cmp():
3     # Example string comparisons to test
4     strings_to_compare = [
5         'eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee',
6         'Reeeeeeeeeeeeeeeeeeeeeeeeeeeee',
7         'RefStringeeeeeeeeeeeeeeeeeeee',
8         'RefString1ABCDEFGHIJKLMNPRSTXYZ'
9     ]
10    target_string = 'RefString1ABCDEFGHIJKLMNPRSTXYZ'
11
12    # Get baseline time for a dummy operation (empty loop)
13    start = time.perf_counter_ns()
14    for _ in range(10_000):
15        pass
16    end = time.perf_counter_ns()
17    baseline_time = end - start
```



```
18
19     # Time measurement for each string comparison
20     for s in strings_to_compare:
21         # Perform the string comparison multiple times to
22             estimate
23         # and takes mean of it
24         start = time.perf_counter_ns()
25         for _ in range(10_000):
26             compare(s, target_string)
27         end = time.perf_counter_ns()
28         comparison_time = end - start
29
30         # Estimate time per comparison based on relative times
31         cpu_cycles_per_comparison = (comparison_time -
32             baseline_time) / 10_000
33         print(f"Estimated time: {cpu_cycles_per_comparison:.0f}
34             nano-seconds for \n'{s}' == '{target_string}' \n")
35
36 measure_string_cmp()
```

Source Code 1: Python code to measure time execution of string comparisons with non-constant approach

# Appendix C

---

This section serves as a brief tutorial on testing big-endian code on the QEMU emulator running within VirtualBox.

## Testing Big-Endian Code on QEMU Emulator in VirtualBox

Our setup flow for non-Linux users requires a virtual machine running a Linux-based operating system, with QEMU emulator and GNU debugger installed. We opted for VirtualBox due to its status as a well-maintained, well-known open-source virtualization software developed by Oracle. VirtualBox (VB) supports a wide range of operating systems, including Windows (in various editions), macOS, and Linux, among others. Another requirement for testing and debugging is a VirtualBox image of a Linux-based operating system that will be used within VirtualBox. In this section, we will use a Debian 11 image, as recommended by the article's author. Finally, QEMU and the GNU debugger will be installed on the Linux-based OS within VirtualBox.

Firstly, we need to install the latest version of VirtualBox, available at <https://www.virtualbox.org/>. I used VirtualBox version Version 7.0.14 r161095. Installation is simple and straightforward; therefore, the installation of VirtualBox is beyond the scope of this thesis. What we provide here is a step-by-step guide for creating a virtual machine using a minimal installation Debian 11 image (<https://www.linuxvmimages.com/images/debian-11/>) and setting up shared folders for convenient testing and debugging of code.

Once VirtualBox is installed on your computer, create a folder that will serve as a shared folder. In my case, I created a folder called `SHARED` in `C:\SHARED`. The purpose of the shared folder is to provide a way of sharing files between the virtual machine (in this case Debian 11) and your host computer, which is Windows 10 in my case.

Suppose you have already downloaded the Debian 11 image. We aimed for

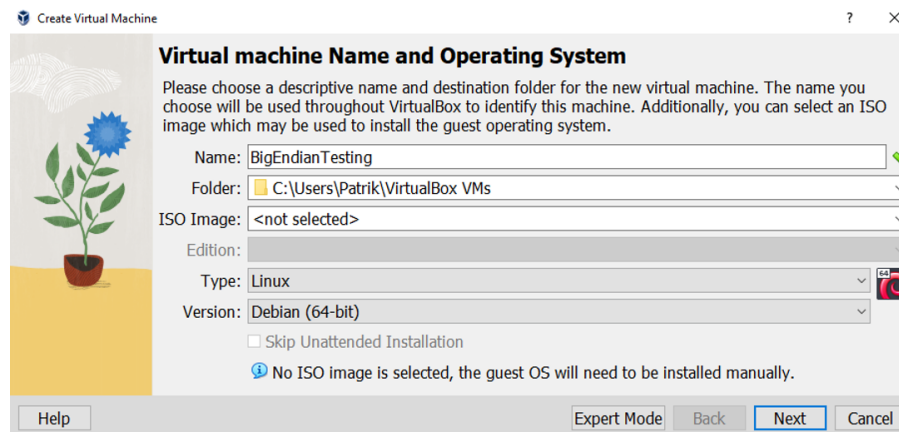


Figure 1: Creating a new virtual machine in VirtualBox.

a minimal installation VirtualBox image which is only 415 MB, but feel free to use a regular installation that provides a graphical user interface (GUI). Once you have downloaded the Debian image, we can create a virtual machine inside VirtualBox that will run the downloaded Debian image.

- Open VirtualBox and create a new virtual machine. Set the name of your virtual machine as you wish (in my case, I chose the name 'BigEndianTesting'). Select Linux as the type and Debian (64-bit) as the version, as shown in Figure 1.
- To run the downloaded Debian 11, it is crucial to choose the correct virtual disk. By correct, I mean selecting the downloaded Debian disk with .vdi extension. If you cannot find the downloaded Debian disk among the options, click the folder icon. A new window will pop up, where you can add the Debian disk. Once you have selected the virtual disk, finish the installation process(see Figure 2 and 3.
- Next, set up the shared folder. Open VM settings > Shared Folders > Add new and set your shared folder based on the name you chose. In my case, it's 'SHARED', as shown in Figure 4.

Next, we need to mount the shared folder in Debian and continue with the installation of QEMU with GNU debugger. This process will be exactly the same as described in the article [48]. Since I chose minimal installation Debian 11, only a terminal window is available, but feel free to download any GUI Debian VirtualBox image.

The simple way of mounting a shared folder in Debian is to use a series of commands:

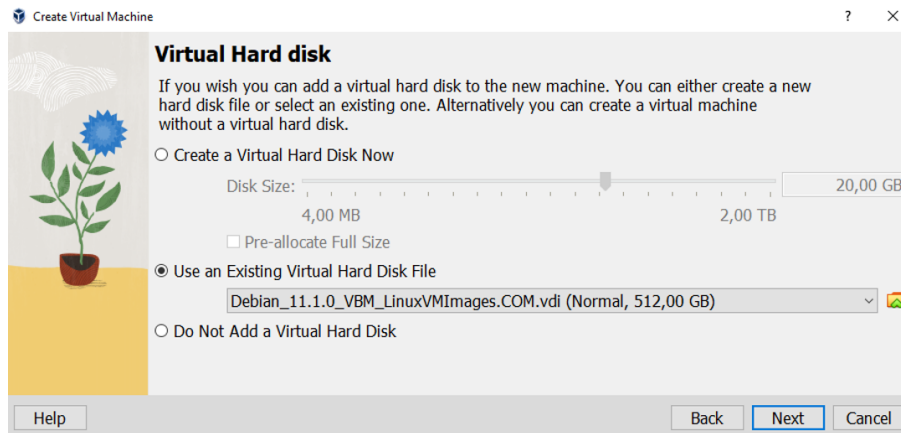


Figure 2: Selection of Debian 11 disk.

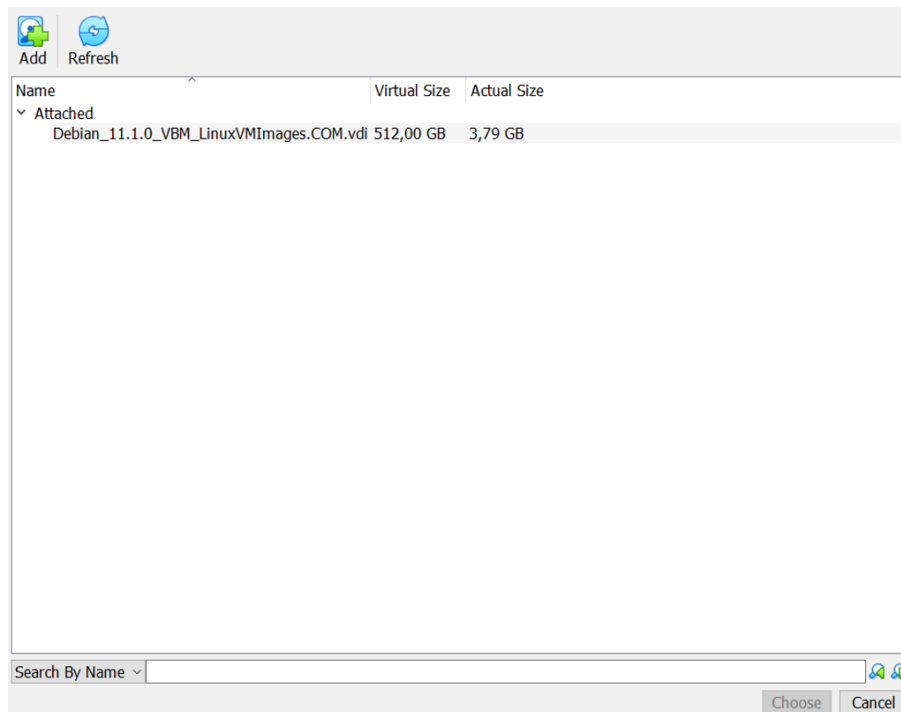


Figure 3: Adding new disk in VirtualBox.

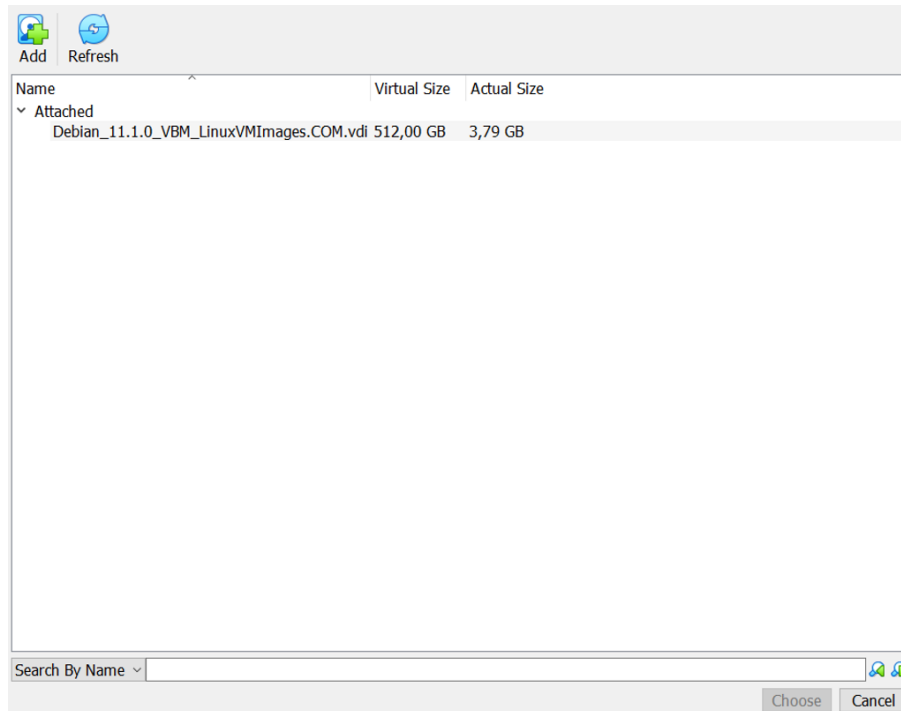


Figure 4: Setting up a shared folder in VirtualBox.

```

1 sudo mkdir /mnt/shared
2 sudo mount -t vboxsf SHARED /mnt/shared

```

Once the shared folder is mounted on the Debian VM, you can access files from the shared folder by navigating to the mount point. In this case, since we mounted the shared folder to `"/mnt/shared"`, you can access its contents using standard Linux commands.

```

1 cd /mnt/shared
2 ls

```

This will change the directory to the shared folder mount point and list the files and directories within it. You can then manipulate these files just like any other files on your Debian VM.

Let's continue with the preparation of an example file that we will be using to determine big endianness. In Debian, create a simple file that can look like an example file inspired by Stephan Brumme (<https://create.stephan-brumme.com/big-endian/>). See code below:

```

1 // //////////////////////////////////////
2 // endian.c
3 // Copyright (c) 2013-2015 Stephan Brumme. All rights reserved.
4 // see http://create.stephan-brumme.com/disclaimer.html

```

```

5 //
6 #include <stdio.h>
7 int main(int argc, char* argv[])
8 {
9     // on little endian systems, twoBytes is stored as 0x01, 0x00
10     and
11     // on big     endian systems, twoBytes is stored as 0x00, 0x01
12     short twoBytes = 0x0001;
13     // get first byte of twoBytes
14     char oneByte = *(char*) &twoBytes;
15     if (oneByte == 1)
16         puts("little endian");
17     else
18         puts("big endian");
19     return 0;
20 }

```

Source Code 2: Example C code to evaluate endianness

The final step is to install QEMU with the GNU debugger and support for a big-endian architecture provided by QEMU. When discussing big endian, QEMU supports multiple architectures such as ARM, MIPS, or PowerPC. We opt for a big endian PowerPC 64-bit architecture, which requires us to install the necessary tools. To do this, simply follow the command flow below:

```

1 sudo apt-get install gcc-powerpc64-linux-gnu \
2 binutils-powerpc64-linux-gnu binutils-powerpc64-linux-gnu-dbg

```

Proceed to install the QEMU Emulator and GNU debugger. Note that there are different QEMU packages available, but only 'qemu-user' and 'qemu-user-static' are necessary for our purposes.

Very important step to be made is compile an example file 2 for architecture we choose, a PowerPC 64-bit. To do so, follow subsequent code:

```

1 powerpc64-linux-gnu-gcc -ggdb3 -o endian endian.c -static

```

Where the flag -o serves to create an object file (executable), and the -ggdb3 flag generates debug information suitable for use with the GNU Debugger. The output is an executable file named 'endian,' which will be used as an example for the evaluation of endianness.

Since I chose a minimal installation of Debian 11 in VirtualBox, I only have access to one terminal. However, running the QEMU emulator and debugging

with the GNU debugger simultaneously requires multiple terminal windows. To overcome this limitation, there are several workarounds available.

Firstly, you can use the option to create additional terminal windows by pressing `Alt+Fn+(F1-F12)`, allowing you to open and switch between multiple terminal windows.

Another effective workaround is to use Tmux, a terminal multiplexer that enables the creation of multiple "pseudo terminals" within a single terminal session. Tmux facilitates the simultaneous operation of various programs via one terminal connection. Additionally, Tmux allows for detachment from the current terminal, ensuring uninterrupted background execution of all programs. Subsequently, reattachment to the same or alternative terminal is possible.

For our purposes, Tmux provides an ideal solution. You can install Tmux using the following command:

```
1 sudo apt-get install tmux
```

After installing Tmux on Debian, let's create two windows: one for running QEMU PowerPC emulation and one for the GNU debugger. To create new windows, enter the Tmux environment:

```
1 Tmux
```

When we are inside Tmux environment, create a new window by shortcut `Ctrl+B` and then press `C`. You can close active window by `Ctrl + B` and then `X`.

In Figure 5, the green panel at the bottom of the terminal indicates that the Tmux environment is running. Within the green panel, we can observe that two virtual windows have been created: `0 : bash-` and `1 : bash*`. The number before the separator (`:`) represents the index of each window. The hyphen indicates previously used window, while the asterisk denotes the window that is currently active. You can switch between windows using `Ctrl + B` followed by the window index. For example, if the active window is window 1, to switch to window 0, simply use the shortcut `Ctrl + B + 0`. For more information about basic Tmux shortcuts, see an article <https://www.redhat.com/sysadmin/introduction-tmux-linux>.

Now when we have two separate windows, we can run QEMU PowerPC64 emulator in one window and GNU debugger in the other window. Firstly run an example code in QEMU PowerPC64 emulator by command:

```
1 qemu-ppc64 -L /usr/powerpc64-linux-gnu -g 1234 ./endian
```

where `-L/user/xxx` let you to choose which elf interpreter to use and `-gxxx`

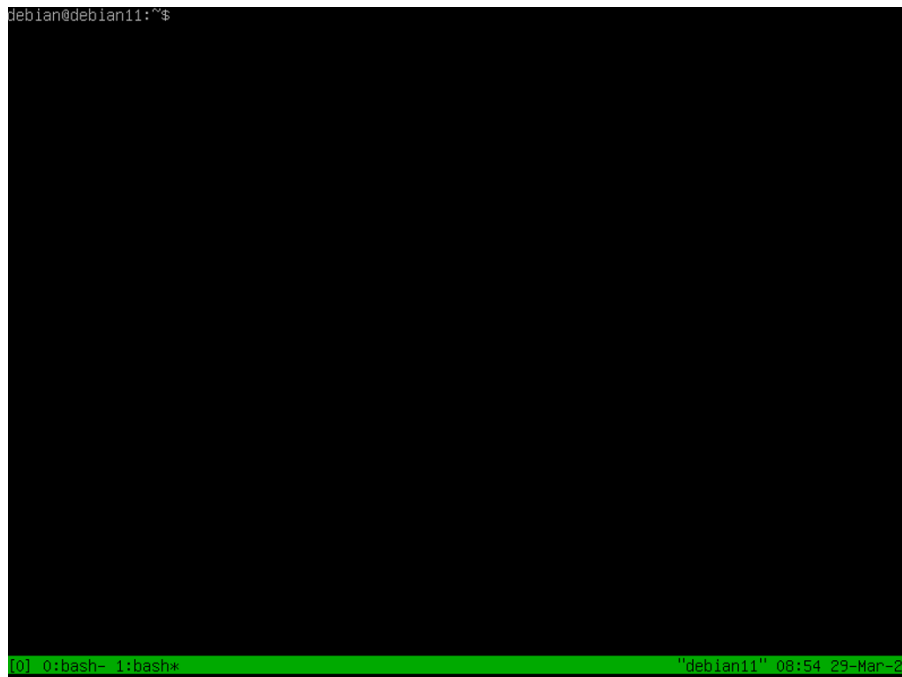


Figure 5: Showcase of Tmux terminal.

let you set port number for GDB connection.

Once you launch PowePC emulator, go to second windows using shortcut *Ctrl + B + index* and run debugger:

```

1 gdb-multiarch -q --nh \
2   -ex 'set architecture ppc64' \
3   -ex 'set sysroot /usr/powerpc64-linux-gnu' \
4   -ex 'file endian' \
5   -ex 'target remote localhost:1234' \
6   -ex 'break main' \
7   -ex continue \
8   -ex 'layout split' \
9   -ex 'layout next' \
10  -ex 'layout regs'

```

As you can see we are running debugger with multiple `-ex` tags. Basic description of used `-ex` tags is presented in table below:

- **Set architecture:** Sets the architecture we want to use.
- **Set sysroot:** Sets directory in which targeted libraries are located; this must match with ELF interpreter used when launching QEMU emulator.
- **file:** Sets the file you want to debug.



Table 1: Debugger Commands and Descriptions

Command	Description
Set architecture	Sets the architecture we want to use
Set sysroot	Sets directory in which targeted libraries are located; this must match with ELF interpreter used when launching QEMU emulator
file	Sets the file you want to debug
Target remote machine:port_number	Tells debugger what machine and port QEMU is running on
Break main	Places a breakpoint on the main function
continue	Tells debugger not to stop on the first assembly instruction
Layout split	Splits terminal into two halves
Layout regs	Tells debugger to place General Purpose Registers (GPRs) and Special Purpose Registers (SPRs) on the upper half of the layout

- **Target remote *machine:port\_number***: Tells debugger what machine and port QEMU is running on.
- **Break main**: Places a breakpoint on the main function.
- **continue**: Tells debugger not to stop on the first assembly instruction.
- **Layout split**: Splits terminal into two halves.
- **Layout regs**: Tells debugger to place GPRs (General Purpose Registers) and SPRs (Special Purpose Registers) on the upper half of the layout.

When you run QEMU PowerPC emulator in one window, it should be listening on port 1234 as we set in example, and in the other window a debugger should be running.

You can debug a program as you wish, but if you simply want it to be executed, you can use the *continue* command and let the program finish. After the program finishes in the debugger, you can close the debugger with the *quit* command (see Figure 6. In the previous window, the one where you are running an example file on the PowerPC architecture, you can see that the program actually finishes and

```
[ Register Values Unavailable ]

endianTest.c
B+ 6      short twoBytes = 0x0001;
7         // get first byte of twoBytes
8         char oneByte = *(char*) &twoBytes;
9         if (oneByte == 1)
10          puts("little endian");
11        else
12          puts("big endian");
13        return 0;
14        }

$exec No process in: L?? PC: ??
(gdb) continue
continuing.
Inferior 1 (process 1) exited normally]
(gdb)

0] 0:bash- 1:bash* "debian11" 09:09 30-Mar-2
```

Figure 6: Showcase of debugger for PowerPc64.

prints the result. As you can see from Figure 7, the example code indicates that we are on a big-endian architecture. Which is exactly what we wanted to achieve.

```
debian@debian11:~$ qemu-ppc64 -L /usr/powerpc64-linux-gnu -g 1234 ./endianTest
big endian
debian@debian11:~$ _
```

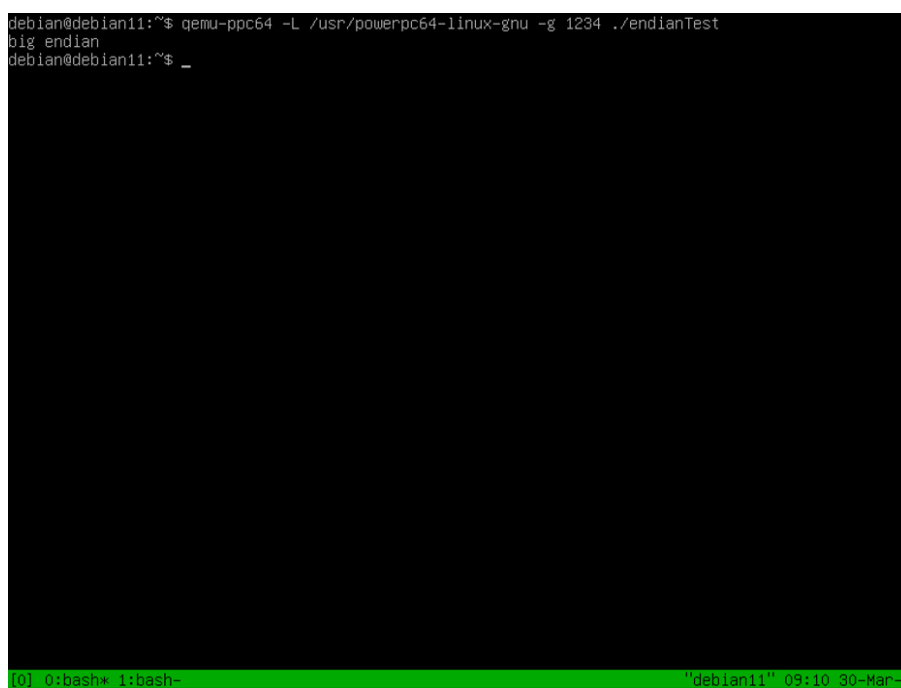
A terminal window with a black background and white text. The text shows a command being executed: 'qemu-ppc64 -L /usr/powerpc64-linux-gnu -g 1234 ./endianTest'. The output of the command is 'big endian'. The prompt 'debian@debian11:~\$' is visible at the beginning and end of the command line. At the bottom of the terminal, there is a green status bar with the text '[0] 0:bash\* 1:bash-' and a timestamp '"debian11" 09:10 30-Mar-2'.

Figure 7: First terminal window after debugging is finished on second window.