National College of Ireland

A novel workflow for improved access to microservices using ZKP-based methods

MSc Research Project
Cloud Computing

Abdul Wasee
Student ID: 23388790

School of Computing
National College of Ireland

Supervisor: Dr Giovani Estrada

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Abdul Wasee
Student ID:	23388790
Programme:	Cloud Computing
Year:	2025
Module:	MSc Research Project
Supervisor:	Dr Giovani Estrada
Submission Due Date:	11/08/2025
Project Title:	A novel workflow for improved access to microservices using ZKP-based methods
Word Count:	3700
Page Count:	20

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Abdul Wasee
Date:	12th September 2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

A novel workflow for improved access to microservices using ZKP-based methods

Abdul Wasee
23388790

Abstract

Microservices commonly rely on bearer tokens such as JSON Web Tokens (JWTs). Although fast, these tokens reveal user claims to every service and can be reused if stolen. This thesis designs and evaluates a privacy-preserving alternative based on zero-knowledge proofs (ZKPs) that verifies each request under a zero-trust model without disclosing identity attributes. The authentication prototype service is built with two implementations (TypeScript/NestJS and Rust) and compared with a JWT baseline in privacy, access control correctness, and performance.

Across six concurrency levels (1, 10, 15, 20, 25, 30) over 15-second runs, CPU and memory usage were sampled every 250 ms, and means were reported across the concurrency points. The workflow achieves zero attribute disclosure while keeping verification practical: verification adds 1.8 ms per microservice hop with a native Rust verifier and ∼130 ms with a NestJS verifier. Proof generation is the main computational cost (∼0.7s in NestJS; ∼1.4s in Rust), with NestJS demonstrating superior performance due to efficient WASM witness generation and reuse of artefacts. These measurements were taken on the same host with an otherwise idle machine to control variance. Formal statistical significance would require multiple independent repetitions; the present analysis reports central tendencies and observed stability.

Contributions are: (i) a working ZKP-based authentication workflow suitable for microservices, (ii) a dual-stack prototype and comparative evaluation against JWT, and (iii) evidence that privacy can be improved without prohibitive verification overhead.

1 Introduction

Microservices commonly rely on bearer tokens such as JSON Web Tokens (JWTs). Although fast, these tokens reveal user claims to every service and can be reused if stolen. This thesis designs and evaluates a privacy-preserving alternative based on zero-knowledge proofs (ZKPs) that verifies each request under a zero-trust model without disclosing identity attributes. The authentication prototype service is built with two implementations (TypeScript/NestJS and Rust) and compared with a JWT baseline in privacy, access control correctness, and performance.

In today's industrial practice, authentication between microservices typically relies on token-based mechanisms that assume a considerable level of interservice trust. Common solutions (for example, OAuth 2.0 with OpenID Connect) issue JWTs to represent the user's identity and permissions, which are then included with each request to downstream

services[1]. Although this single-sign-on approach streamlines authentication across a distributed system, it also means that every microservice receiving the token gains access to all encoded identity claims (e.g. user attributes or roles) whether that service needs them or not (de Almeida and Canedo; 2022). This disclosure of all identity claims breaches the principle of least privilege, which in a microservices setting means that a service should receive only the information strictly necessary for its function. Applied on-demand, this principle requires that such information is provided only at the moment it is needed, rather than being exposed by default, thereby reducing privacy risks. If any service or its logs are compromised, sensitive user information can be exposed. Moreover, many designs implicitly trust traffic within the cluster as "internal" and benign, contradicting zero-trust best practices and can allow an attacker who compromises one service to escalate across the system.

To address these vulnerabilities, the academic literature advocates the Zero Trust Architecture (ZTA) *"never trust, always verify"* for cloud and microservice environments (Sarkar et al.; 2022). Under a zero-trust model, no request is automatically trusted based on its origin; instead, every user-to-service or service-to-service call must be authenticated and authorised as if it came from an open, untrusted network. In a microservices context, this translates into enforcing strong identity verification and minimal privileges on every single API call. Techniques like mutual TLS (mTLS) can ensure the authenticity of communicating services, but mTLS alone does not convey user-level permissions and still relies on the sharing of user identity (via tokens or certificates), which can introduce privacy. Continuously re-authenticating each request (for example, validating a JWT and checking its permissions on every call) improves security but adds performance overhead and still entails exposing some user data in each token exchange. Thus, implementing a zero-trust philosophy strictly with conventional tokens can be both inefficient and insufficient from a privacy standpoint.

Zero-Knowledge Proofs (ZKPs) have emerged as a promising technology to achieve strong security without sacrificing privacy in authentication systems. A ZKP is a cryptographic protocol in which a prover convinces a verifier that a statement is true while revealing nothing beyond the validity of the statement (that is, no witness / secret is leaked) (Feige et al.; 1988). Applied to microservice authentication, ZKPs enable a service to cryptographically verify that a client holds valid credentials or possesses a required attribute *without* the client ever exposing those credentials or attributes. For example, rather than sending a JWT that reveals the identity and roles of a user, a client could present a cryptographic proof stating "I am authorised with the role X for this request" and the service can verify the proof without learning the user's identity or any other roles (Chen et al.; 2023).

This design upholds privacy-by-design principles: the service learns only that the proof is valid (i.e., the request is authorised), but no personal data are disclosed. Moreover, the proof does not reveal attributes or reusable credentials. To prevent replay, each proof must be bound to a one-time challenge (e.g. nonce/timestamp and, optionally, request context); without such binding, an identical proof/commitment pair could be replayed. In essence, ZKPs enable zero-trust enforcement with minimal information exposure since each request can be verified cryptographically while sensitive details remain concealed. Equally important, modern ZKP schemes (e.g., efficient zk-SNARKs and zk-STARKs) have dramatically improved performance. Proofs can now be generated and verified in a matter of a few milliseconds (Motlagh et al.; 2025), making it feasible to require a new

[1] `https://tinyurl.com/ykksf53v`

proof on every service call without introducing significant latency.

Given these developments, this dissertation proposes to explore a novel microservices authentication framework based on zero-knowledge proofs, with the goal of mitigating the weaknesses of the current token-based models. The following sections outline the research gap that motivates this study, the specific research question being addressed, and the key objectives that will be pursued.

1.1 Research Gap

Current microservice architectures lack effective solutions to balance distributed authentication with strict data minimisation for privacy. In practice, most organisations use OAuth 2.0, API gateways, or JWT tokens to handle identity in microservices, emphasising functionality and interoperability over privacy controls. This often means that user attributes (claims) are passed freely between services, sharing far more personal data with each microservice than is necessary for authorisation. Such overexposure conflicts with data protection principles like data minimisation mandated by regulations (e.g. GDPR) and creates unnecessary privacy risk. However, it remains the norm: A recent systematic review of 290 microservices security studies found that only 9 ($\sim$3%) of them even considered user privacy or compliance requirements (Berardi et al.; 2022). In short, privacy-preserving authentication in microservice environments is largely an open problem, not adequately addressed by industry and academia.

Some emerging approaches have begun to address some parts of this issue, but a satisfactory solution remains open. For example, the Selective Disclosure JWT (SD-JWT) standard allows clients to reveal only specific attributes from an identity token to a service, rather than the entire identity profile. This limits unnecessary data sharing, but the service still learns a subset of user information. In contrast, an ideal solution would use ZKPs so that a service can verify an authorisation criterion (e.g., "user is over 18") without seeing any personal data at all. So far, proof-based authentication that preserves privacy is not standard in enterprise microservices. Interestingly, other domains demonstrate the feasibility of this approach.

In the field of the Internet of Things (IoT), researchers have shown that zero-knowledge authentication can validate device or user credentials without revealing identities, drastically reducing privacy leakage (Chen et al.; 2023). In the same paper, various ZKP-based IoT schemes were critically reviewed and concluded that eliminating direct credential disclosure significantly mitigates privacy risks in distributed systems, even if performance tuning is needed for constrained devices. Likewise, in the domain of decentralised identity, a standard Attribute-Based Access Control model was extended with blockchain smart contracts and ZKPs to allow policy checks "without disclosing the value of sensitive attributes" (Maesa et al.; 2023). These studies show that ZKP technology can be integrated into complex, distributed environments to enhance privacy. They also suggested ways to overcome performance hurdles, for example, batching multiple proof verifications into one ZK-rollup has shown a $\sim$86% reduction in authorisation overhead under high loads (Lin et al.; 2023).

However, none of the aforementioned advances has been translated into mainstream microservice platforms for user or service-to-service authentication. Today, no popular framework or reference architecture incorporates zero-knowledge verification in microservices yet. This leaves a clear gap that our research addresses.

1.2 Research Question

In light of the research gap mentioned above, a detailed study is proposed here. The key research question can be described as follows.

- How can zero-knowledge proof techniques be used to design a fast microservice authentication and access control mechanism that preserves user privacy and achieves zero-trust security?

1.3 Research Objectives

To answer the research question, this study pursues three objectives.

1. **RO1: Prototype a ZK Proof–Based Authentication Microservice:** Design and implement an end-to-end workflow in which (i) a registration endpoint derives a per-user secret/nonce and public commitment from minimal attributes, (ii) a proof-generation endpoint (implemented server side in our prototype) produces ZK proofs , and (iii) a verification endpoint validates the proof and commitment on each request to enforce zero-trust authentication.

2. **RO2: Assess Security and Privacy Gains over JWT:** Compare the prototype with a standard JWT-based flow, analysing the resistance to replay or token theft attacks, and measuring the reduction in exposed user data (i.e. zero attribute disclosure).

3. **RO3: Measure Performance Overhead and Scalability:** Benchmark authentication latency, throughput, and CPU/memory usage under representative load, quantifying the trade-off between stronger privacy/security and system performance.

Together, these objectives show that a lightweight ZKP mechanism can replace bearer tokens in microservices, delivering verifiable zero-trust authentication without revealing sensitive credentials and with performance costs that are explicitly quantified rather than assumed.

The document is organised as follows. Work related to token-based authentication in microservices, zero trust, ZKP, and performance considerations will be introduced in Section 2. The methodology related to dual prototypes (NestJS/Rust), integration/deployment context, cryptographic design, and threat model will be introduced in Section 3. The design specification related to the three innovations (zk-SNARK embedded verification, zero-trust enforcement per request, and hybrid authentication flow) will be presented in Section 4. Section 5 will discuss the details of the implementation of the circuit setup, the back-end architecture, API endpoints, and libraries/tools employed. Section 6 will then discuss the results of three proposed use cases (privacy, authorisation correctness, and performance / scalability). Finally, Section 7 will present the main research findings and directions for future work.

2 Related Work

2.1 Token-Based Authentication in Microservices: Benefits and Limits

Token-based schemes (OAuth2 / OIDC with JWT) are widely used in microservices due to their statelessness and interoperability (Venčkauskas et al.; 2023). However, JWTs are vulnerable to theft and replay and often carry overly broad claims. A leaked token can be reused across services if strict scoping is not enforced, affecting security (de Almeida and Canedo; 2022). To mitigate this, some authors (Venčkauskas et al.; 2023) separate external user tokens from short-lived audience-scoped internal tokens issued by a central authoriser. This reduces lateral movement but adds per-service validation overhead and coordination with the issuer; the evaluation compares mainly against a centralised baseline and does not cover non-token alternatives such as mTLS. In general, token-based auth remains practical but fragile without careful scoping, rotation, and validation across heterogeneous services (Venčkauskas et al.; 2023; de Almeida and Canedo; 2022).

2.2 Zero Trust for Cloud-Native Systems

Zero trust ("never trust, always verify") aligns with native and distributed cloud microservices by requiring continuous identity, authorisation, and encryption for every call (Sarkar et al.; 2022). Surveys emphasise micro-segmentation and least privilege, but also note adoption barriers: operational complexity, human factors, privacy considerations, and the difficulty of maintaining consistent policy as services scale and move (Sarkar et al.; 2022).

At the implementation level, eZTrust tags and filters traffic with eBPF based on workload identity rather than IP, restricting lateral movement in containerised environments (Zaheer et al.; 2019). Although effective against dependency chain attacks, network layer controls such as eZTrust primarily enforce passively and require complementary monitoring and runtime risk assessment; large, dynamic clusters also challenge policy freshness and scalability (Zaheer et al.; 2019; Sarkar et al.; 2022). The literature converges on the verification of the identity per request and continuous authorisation as foundations to reduce the radius of the explosion in microservices, provided that integration and performance costs are managed (Sarkar et al.; 2022).

2.3 Zero-Knowledge Proofs for Authentication

Zero-knowledge proofs (ZKPs) authenticate the possession of a secret without revealing it. The Feige–Fiat–Shamir team introduced zero-knowledge identification via interactive challenge response, showing that password-less authentication is possible, although with round-trip overhead (Feige et al.; 1988). Modern systems use interactive and non-interactive ZKPs to avoid sending passwords or raw attributes.

In hardware attestation, zk-SNARKs have been used to prove that a device's PUF output matches an enroled value without revealing the output (Zhong et al.; 2023). This provides resistance to clones and privacy, but assumes a trusted setup (CRS) and imposes testing costs on constrained devices (Zhong et al.; 2023). More generally, ZKPs support multifactor authentication (MFA) and attribute checks (e.g., proving age or role) so that servers learn only compliance, not underlying secrets. The trade-off is computational

overhead and the need to integrate specialised libraries and ceremonies (for SNARKs), which motivates selective use, offloading, or hybrid designs.

2.4 Performance and Scalability of ZKPs

Performance remains the central constraint for ZKP adoption. Comparative studies in constrained settings report distinct trade-offs among systems: Bulletproofs offer short, setup-free proofs with logarithmic sizes and relatively fast verification for small statements, but verification scales linearly with statement bits (Bünz et al.; 2018; Motlagh et al.; 2025). zkSTARKs remove the trusted setup and offer strong scalability and post-quantum security at the cost of larger proofs and heavy FFT / FRI based proving (Motlagh et al.; 2025). In head-to-head IoT workloads, Bulletproofs produced smaller proofs and faster verification than STARKs, whereas STARKs can become favourable for very large statements due to sublinear verification and parallelizable proofing (Motlagh et al.; 2025). Across systems, real-world performance is strongly dependent on implementation quality, parallelism (e.g., GPU-accelerated FFTs), and memory footprint; many deployments offload proving or limit circuit complexity.

For microservices, where tail latency and throughput are paramount, these costs necessitate careful protocol choices (e.g., SNARKs with millisecond-scale verification), batching/aggregation where appropriate, and hybrid patterns that minimise proof frequency while preserving zero trust semantics.

Table 1: Related works, highlighting main contributions and limitations.

Cited Work	**Main Contributions**	**Limitations**
Venčkauskas et al. (2023)	Separates external JWTs from short-lived internal tokens, reducing lateral movement in microservices.	Adds per-service validation overhead and still risks abuse if a token is stolen; tested only against a basic baseline.
de Almeida and Canedo (2022)	Systematic review of microservice auth (OAuth2/OIDC, JWT, gateways) and associated security gaps.	Pure survey; no new technique or quantitative evaluation and limited open-source evidence.
Sarkar et al. (2022)	Comparative survey of zero-trust models for cloud; stresses identity checks and micro-segmentation.	Focuses on the network layer; lacks empirical data and app-level implementation guidance.
Zaheer et al. (2019)	eZTrust uses eBPF to tag traffic and block lateral movement via per-packet identity.	Network-only, passive enforcement; no global monitoring and untested under heavy loads.
Feige et al. (1988)	First zero-knowledge identity proof enabling password-less authentication.	Interactive, latency-heavy and limited to simple identity statements.
Zhong et al. (2023)	zk-SNARK attestation of IoT PUFs for clone-resistant device authentication.	Needs trusted setup and heavy proving on constrained devices.
Motlagh et al. (2025)	Benchmarks Bulletproofs vs. STARKs on IoT workloads, quantifying size/time trade-offs.	Covers only two protocols and single-proof cases; integration effort not addressed.
Bünz et al. (2018)	Bulletproofs give short, setup-free range proofs for confidential transactions.	Cost grows with circuit size; less efficient for complex statements than SNARKs.

Closest Work and Distinction. The work of (Lin et al.; 2023) is conceptually the closest one, but differs in deployment and proof workflow: that work targets IoT access control with blockchain logging, batching many requests into a verified rollup on-chain. In contrast, the approach presented here verifies a fresh Groth16 proof entirely off-chain within a cloud-native microservice pipeline, validating each request independently under zero trust with millisecond-scale verification. Both approaches reduce credential leakage. This work prioritises per-request checks and standard microservice integration, while the rollup design sacrifices granularity to improve blockchain scalability. The threat model and evaluation further indicate that JWT-centric designs expose more identifying information in practice, while the ZKP approach considered here avoids disclosing secrets or attributes.

3 Methodology

The methodology integrates implementation workflows, cryptographic design decisions, threat modelling, and system limitations.

3.1 Development Approach

Two functionally equivalent prototypes of the ZK authentication service were built to compare stacks: a **NestJS/TypeScript** service and a **Rust/Actix** service. Both consume the same *Circom* circuit (compiled to R1CS with Groth16 keys), so the cryptographic statement proved and verified is identical across stacks. The NestJS service uses SnarkJS (WASM) for witness/proof generation; the Rust service uses native ark-crypto libraries. Each implements the REST API defined once in §5; to avoid duplication, the endpoint and schema details are omitted here.

The dual build enabled an apples-to-apples comparison of ergonomics and run-time behaviour under identical statements. For the remainder of the thesis, **NestJS/TypeScript** is adopted as the reference implementation for its fast iteration, broad library support, and lower proof generation latency in the evaluated setup, while **Rust/Actix** is retained for portability and performance-oriented verification tests.

3.2 Integration & Deployment Context

The ZK authenticator is a standalone microservice used in a *zero-trust* setting. In successful proof verification, downstream services either (a) accept that verification result directly or (b) use a short-lived JWT issued by the auth service to amortise the ZK cost between calls (pattern used for the baseline comparison in §6). The functionality was validated locally via Docker on Apple Silicon (macOS, Node 18.x, Rust 1.70), exercising the services over localhost networking. TLS was disabled only for local testing; production use assumes TLS on all links.

3.3 Cryptographic Design

Hashing. *Poseidon* is a cryptographic hash function designed to be efficient inside zk-SNARK circuits. It is used because of low constraint footprint in SNARK circuits, keeping prover memory/time modest while maintaining standard security properties.

Proving system. *Groth16* is a pairing-based zk-SNARK proof system that produces constant-size proofs with fast verification. It requires a trusted set-up per circuit (common reference string, CRS).

Circuit authoring. *Circom* is a domain-specific language (DSL) and compiler for arithmetic circuits. Provides concise, auditable components (e.g. Poseidon in `circomlib`) and automatic witness generation. Alternatives (e.g., PLONK / Noir, arkworks) were considered; for the target (small statement, high verify rate) **Groth16 + Circom + SnarkJS** provided the most direct and performant path. Transparent or universal-setup systems (e.g., PLONK variants, STARKs) remain viable replacements when setup hygiene is prioritised over minimal proof size/latency.

3.4 Threat Model and Security Assumptions

Adversary: The threat model assumes a network attacker capable of observing, blocking, modifying, and replaying messages, and of compromising a single microservice (including access to logs and environment secrets). The adversary cannot break standard cryptographic primitives (hashes, signatures, zk-SNARK soundness).

Security goals: (1) *Credential secrecy*: do not disclose reusable secrets to relying services; (2) *Freshness*: each authorisation must be bound to the current request to prevent replay; (3) *Containment*: no implicit trust between services; privileges are least privilege and short-lived.

Threats & mitigations

- **Credential exposure.** Services verify a zero-knowledge proof against a Poseidon commitment, learning no reusable credential. In production, proving should occur on the client side so that the server never receives the raw secret; In this prototype, proving is performed in the authentication service, so secrecy relies on TLS and strict handling of memory and log hygiene (see §6.5).
- **Replay.** Without a freshness binding, an intercepted proof/commitment can be replayed. The intended design binds each proof to a one-time challenge (nonce / timestamp and optionally a request hash); this binding is not yet implemented in the prototype and is called a limitation (§6.5). Short-lived JWTs (when used to amortise cost) and TLS further reduce replay viability, but do not replace in-circuit freshness.
- **Lateral movement.** Every service call requires its own authorisation (proof or scoped, short-lived JWT); services do not inherit one another's privileges. Least-privilege scoping limits the blast radius if a token is stolen or a service is compromised.

Assumptions

- End-to-end TLS in production and secure custody of signing/verification keys.
- Honest Groth16 setup for this circuit (multi-party ceremony recommended).
- Correctness of the selected libraries and configurations; high-entropy randomness.
- No global cryptographic break (hash preimage resistance, zk-SNARK soundness).

Scope of Evaluation: Both stacks were benchmarked under identical workloads; verification throughput and latency, as well as resource profiles, were gathered.

4 Design Specification

The proposed system's design departs from previous microservice authentication models through three major innovations.

4.1 First Innovation: Embedded zk-SNARK Verification

Prior methods include internal JWT schemes per request (Venčkauskas et al.; 2023) and network-level zero trust perimeterization such as eZTrust (which tags and verifies workloads through eBPF) (Zaheer et al.; 2019). The workflow embeds a zk-SNARK verification step directly into the microservice call flow. Each client request carries a succinct Groth16 proof that is verified on the fly by the target microservice, eliminating the need for continual token issuance or a trusted network gateway. This off-chain integration of ZK proofs means that, in contrast to blockchain-dependent frameworks (Lin et al.; 2023; Zhong et al.; 2023), no distributed ledger or external consensus is required to establish trust. The microservice itself becomes the point of verification, which simplifies the architecture and reduces multi-hop latency by handling authentication locally.

4.2 Second Innovation: Per-Request Zero-Trust Principles

The system follows a strict zero-trust mantra: Never trust a request by default; always verify credentials anew. Unlike internal token schemes that might trust a signed token for multiple calls (until it expires or is revoked) or require sharing a signing key across services (Venčkauskas et al.; 2023), our design treats every single API call as untrusted until a proof is verified for that call. This means that no service ever accepts a request based solely on an earlier authentication. Consequently, long-lived secrets or tokens are not distributed between services, thereby sidestepping the key-management complexity inherent in token-based approaches. Privacy is inherently enhanced as well: the zero-knowledge proof reveals zero user attributes or identity info to the verifier, whereas even a minimally scoped JWT still exposes some user claims. To keep this per-request verification practical, the proof system was optimised. The Circom circuit is intentionally small (checking just a hash equality), and SNARK-friendly hashing (Poseidon) and the efficient Groth16 prover/verifier. As a result, generating a proof and verifying it can be done with only a slight increase in latency (e.g. verification in a few milliseconds). This is a crucial point: The design achieves "never trust, always verify" on each call without imposing an unacceptable performance penalty.

4.3 Third Innovation: Hybrid Authentication Approach

The workflow combines zero-knowledge authentication with traditional token-based methods to maximise compatibility. ZK proofs are the default for clients and services that support them, but if a client cannot produce proofs (e.g., a legacy component), the system can gracefully fall back to a standard JWT-based authentication flow. This dual-mode operation ensures that adopting our framework does not break integration with existing identity infrastructure; it can operate alongside OAuth2/JWT. Importantly, the ZKP

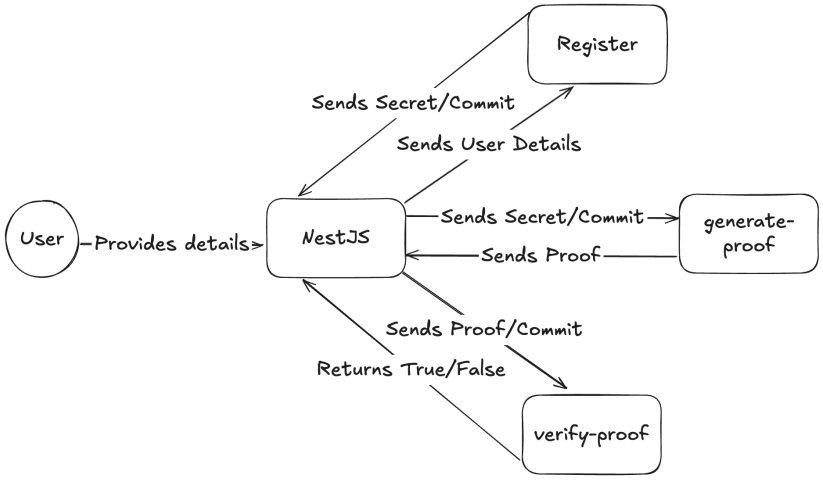

Figure 1: System overview and request flow. The user submits attributes to the NestJS microservice. The `/register` handler derives a secret/nonce and a public commitment; the `/generate-proof` handler takes (secret, commitment) and returns a Groth16 proof; the `/verify-proof` handler validates the proof against the commitment and responds 201 with the literal boolean `true` on success, or 401 *Unauthorized* on failure.

verification is implemented as an independent module that can run next to each microservice or in an API gateway. This module takes a proof and commitment and returns a yes/no authorisation decision. It does not rely on any centralised identity provider at request time, unlike typical OAuth2 or mTLS setups that depend on a trusted authority or shared certificates (Venčkauskas et al.; 2023).

In summary, the design improves privacy (no credentials or personal data are shared with services), provides strong per-request security, and yet remains flexible. If ZKP verification is available, zero trust is enforced with minimal data disclosure; if not, the system can use a short-lived JWT as a stopgap. This hybrid approach had not been explored in prior microservice auth research and is a key enabler for real-world adoption, letting organisations incrementally deploy ZKP authentication without abandoning existing mechanisms.

5 Implementation

This section details the implementation of our ZK proof authentication microservice as described in the Research Objectives (1-3). The goal is to enforce a zero-trust policy within a microservices architecture, allowing each service to authenticate requests without learning any sensitive identity attributes.

Our prototype uses a Circom circuit (**SecretProof**) that binds a user's `secret` to a public `commitment`, and a NestJS backend that exposes three HTTP endpoints (registration, proof generation, and proof verification) to manage the ZK proof workflow. The details of cryptographic choices and authoring tools were discussed in Section 3.

Scope. This section implements **RO1** by building a minimal ZK-proof authentication microservice. Specifies three endpoints (registration, proof generation, and verification) and details their core logic in Table 2.

5.1 Circuit Design and Setup

The system uses a minimal Circom circuit, **SecretProof** (v2.1.7). Calculate $h = \text{Poseidon}(\text{secret})$ and enforce $h = \text{commitment}$, so a proof is valid only when the supplied commitment matches the hash; the commitment is emitted as the public signal. This compact design keeps proofs small and proving/verification fast. The circuit is compiled to R1CS and WASM, and a one-time Groth16 set-up generates the verification and verification keys used by the service.

5.2 Backend Architecture and API Overview

The back-end is implemented in NestJS, a Node.js framework well suited to building modular microservices. Our `AuthService` encapsulates the logic for generating secrets, commitments, proofs, and verification, while an `AuthController` exposes REST endpoints. Deploying this service as an independent microservice allows other components in a cloud application to delegate authentication to it, following zero-trust principles.

API summary

Table 2 summarises the three endpoints. Each endpoint uses JSON for requests and responses and must be called over HTTPS to protect the secret payload.

Table 2: Summary of authentication API endpoints

Endpoint	Method	Purpose and key fields
`/register`	POST	*Registration.* Accepts user attributes (email, name, age, country code, date of birth) and returns a freshly derived secret/nonce pair and the resulting commitment. No sensitive input is persisted on the server.
`/generate-proof`	POST	*Proof generation.* Accepts a previously issued secret (hex) and commitment, computes a witness for the circuit and produces a Groth16 proof and public signals. Returns the proof, public signals and pre-formatted Solidity call parameters.
`/verify-proof`	GET	*Proof verification.* Accepts a commitment, proof and public signals, verifies the proof against the stored verification key, and confirms that the commitment matches the public signal. Returns success or an error.

5.3 Data Handling and Security Considerations

The design intentionally minimises the exposure and storage of sensitive data.

- Personal attributes (email, name, age, country, date of birth) are used only transiently during registration to derive the secret and commitment. They are never persisted on the server.

- The secret and nonce are returned to the client and not stored on the server side. Only the commitment is kept or used as a public identifier.
- Proofs are generated per session and need not be stored after verification. If stored (for example, in logs), they reveal no user information and cannot be reused by an attacker without the secret.
- All communication is assumed to occur over TLS. In this prototype, the client's secret is transmitted to the server for proof generation (never persisted and sent only over TLS); a future version will generate client-side proofs to avoid sending the secret at all.

5.4 Libraries and Tools

Several open-source components support this implementation:

- **Circomlib and circomlibjs** provide the Poseidon hash implementation used both in the circuit and in the JavaScript code.
- **snarkJS** is the primary SNARK toolkit used for proof generation, export, and verification. In particular, `groth16.prove`, and `groth16.verify` are used.
- **Keccak** implements SHA3/Keccak-256 for hashing user attributes.
- **Node.js** `crypto` generates cryptographically secure random *nonces*.
- **NestJS with class-validator** structures the backend and enforces input validation.

Development and testing were performed on macOS using *Node.js* and `pnpm`. The *Circom* and *snarkJS* libraries were installed globally via `pnpm`. The module `generate-proof.ts` wraps the logic to load the compiled circuit artefacts and compute the proofs, allowing reuse in both the server and the potential client-side contexts.

In summary, this implementation delivers a fast zero-knowledge authentication mechanism that aligns with zero-trust principles. It allows a microservice to verify a client's authorisation without learning any sensitive attributes, demonstrating a practical alternative to JWT-based flows with stronger privacy guarantees.

6 Evaluation

This section evaluates the prototype through three use cases mapped to the research objectives: (UC-A) a privacy-focused comparison against JWT (RO2), (UC-B) an access / authorisation correctness study (RO2), and (UC-C) a performance and scalability benchmark (RO3). Together, these use cases test whether ZK proofs achieve zero-attribute disclosure, uphold access decisions under adversarial inputs, and at what runtime cost compared to JWT. The experiments were designed to answer three questions:

1. (UC-A, RO2) Does the ZK approach reduce the amount of user data exposed to services and logs relative to JWT?

2. (UC-B, RO2) Does the verifier accept only valid proof/commitment pairs and robustly reject tampering or replay attempts?

3. (UC-C, RO3) How do latency, throughput, and resource consumption compare to a JWT baseline under load?

To address these questions, each implementation (Rust / ZK, NestJS / ZK, and NestJS / JWT) was instrumented with identical benchmark scripts, and latency, throughput, CPU, and memory metrics were collected across a range of concurrency levels. The results reveal clear performance trade-offs between privacy-preserving ZK proofs and conventional token authentication, and also show that the current prototype would accept a replayed proof/commitment pair because a per-request nonce is not yet bound. This is a known limitation; the planned mitigation is to bind a one-time challenge to each proof (see §6.5).

6.1 Use Case A (RO2): Privacy — Zero Attribute Disclosure vs JWT

Goal Quantify how much user information each scheme exposes to services and logs, aligning with Research Objective 2 (assess security and privacy gains over JWT).

Method The artefacts presented to the microservice—and those persisted or logged by default in each flow—are compared. An *Attribute Disclosure Count (ADC)* is defined as the number of user attributes or identifiers available to the application code or logs during authentication. For ZK, the credential is a proof and a public *commitment*; for JWT, the credential is a signed token with registered claims (e.g., `sub`, `iat`, `exp`) and often application claims (e.g., email, name).

Table 3: Attribute exposure comparison between ZK and JWT during authentication

Channel / Artefact	**ZK Proof**	**JWT**
Credential presented to service	Proof + commitment (no PII)	Signed token containing claims
Data accessible to application code	Commitment only	Claims (registered + app)
Default logging surface (headers/payload)	Proof length/status; no attributes	Token/claims may appear in traces/logs
Persisted server state	Commitment (pseudonymous)	Token/claims may be cached or stored
Cross-service linkability	Same commitment implies linkability	Token identifier linkable across services

Outcome In the prototype, *ADC* is minimised for ZK (no attributes are disclosed to the service; only a commitment is visible), while JWT exposes at least the registered claims and, in common deployments, additional application claims. This demonstrates

RO2's privacy advantage: ZK achieves authentication with zero attribute disclosure to the microservice and a reduced logging surface.

6.2 Use Case B (RO2): Access — Authorisation Correctness & Abuse Cases

Goal Evaluate whether the verifier enforces access correctly (accepts only valid inputs) and how it behaves under adversarial conditions, aligning with **RO2** (§2).

Setup A dummy resource is protected with the ZK verifier (endpoint `/verify-proof`, GET) and compared with a variant protected by JWT. *Acceptance criteria:* a valid credential yields **HTTP 201 Created** with body `true`; invalid, tampered, or mismatched inputs yield **HTTP 401 Unauthorized**.

Adversarial tests

1. **Random proof:** submit a well-formed JSON with random proof bytes. *Expected:* reject (401).
2. **Mismatched pair:** submit a valid proof bound to commitment C_1 with a different commitment C_2. *Expected:* reject due to the commitment check.
3. **Bit-flip tamper:** flip one bit in the proof object. *Expected:* reject.
4. **Replay (credential theft):** reuse a previously accepted credential from another client or time window. *Expected codes:* JWT: **200 OK** until expiration unless server-side revocation is used. ZK prototype: **200 OK** if the identical proof / commitment pair is re-created.

Outcome The verifier strictly accepts only a valid proof/commitment pair and rejects tampering, demonstrating correct access enforcement. However, in the current prototype an identical proof/commitment pair will be accepted if replayed, because the circuit lacks a per-request nonce or service challenge. This is an acknowledged limitation of the present design. The intended fix is to link a one-time challenge (e.g. nonce and / or timestamp, optionally a request hash) to each proof, which prevents replay. This mitigation is described in §6.5 (Limitations).

6.3 Use Case C (RO3): Performance and Scalability Benchmarks

6.3.1 Benchmark setup

The benchmarks used Autocannon, a Node.js HTTP load generator[2], and exercised each API endpoint for a fixed duration of 15 seconds at six concurrency levels (1, 10, 15, 20, 25 and 30 parallel connections). Prior to each run, the script performed a one-time registration to obtain a secret and commitment, and then pre-computed a single ZK proof by calling the `/generate-proof` endpoint. This pre-computed proof was reused for the `/verify-proof` benchmarks to ensure that all frameworks verify the same proof. During

[2] `https://www.npmjs.com/package/autocannon`

each run, the script also sampled the server process every 250 ms using `pidusage` to estimate the average CPU and memory usage.

To ensure fairness, the same benchmark script was used for all three implementations, running on the same host (Apple Silicon M3) with no other significant load. The Rust server was built in release mode and launched with four concurrent proving workers; the NestJS/ZK server used SnarkJS via WebAssembly; the JWT version used the same NestJS stack but replaced the proof generation and verification logic with conventional token signing and verification.

6.3.2 Performance metrics summary

Table 4 summarises the average latency, throughput, CPU and memory consumption across all concurrency levels for each endpoint and framework. The means are computed over six concurrency points; per-concurrency results are reported in the endpoint-specific figures and tables in this section.

Table 4: Average performance metrics for each endpoint. "Avg Latency" is the mean over the six tested concurrency levels (1, 10, 15, 20, 25, 30). Lower latency and CPU/memory usage are better; higher requests per second (RPS) are better. The Rust implementation excels on registration and verification, while the Node.js implementation produces faster proofs. Traditional JWT signing/verifying is significantly faster than ZK-based alternatives.

Endpoint & Framework	**Avg Latency (ms)**	**RPS (req/s)**	**CPU (%)**	**Memory (MB)**
/register				
Rust (ZK)	0.87	11,984	10.58	28.17
NestJS (ZK)	6.63	3,114	6.63	141.06
NestJS (JWT)	16.75	1,408	6.49	135.12
/verify-proof				
Rust (ZK)	1.82	15,292	41.22	188.04
NestJS (ZK)	130.47	1,586	22.08	166.70
NestJS (JWT)	4.71	24,101	6.49	199.15
/generate-proof				
NestJS (ZK)	687.65	75	38.72	306.19
Rust (ZK)	1,404.91	42	29.95	524.92
NestJS (JWT login)	20.62	2,818	6.58	157.01

Latency trends This section presents the average latency versus concurrency for each endpoint.

- `/register` (Figure 2): *Rust/ZK* stays sub-2 ms across 1–30 (0.2–1.8 ms); *NestJS/ZK* rises from 0.3 ms (1 thread) to 11 ms (30 threads); *JWT* increases nearly linearly, reaching ∼30 ms at maximum load (mean 16.75 ms).
- `/verify-proof` (Figure 3): *Rust/ZK* remains fastest, with a maximum of 3 ms at 30 threads; *JWT* follows, scaling from 1 ms to 8 ms; *NestJS/ZK* degrades from 11 ms to 240 ms as concurrency increases.

- `/generate-proof` (Figure 4): proof generation dominates cost—*Rust/ZK* grows nearly linearly to 2.6 s at maximum concurrency; *NestJS/ZK* follows a similar trend but peaks at 1.3 s; *JWT login* remains consistently sub-40 ms.

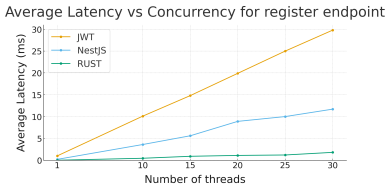

Figure 2: Latency-concurrency relationship for `/register`

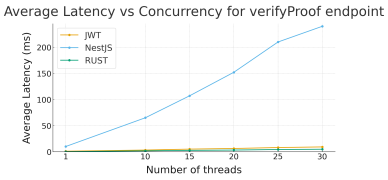

Figure 3: Latency-concurrency relationship for `/verify-proof`

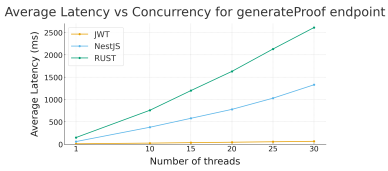

Figure 4: Latency-concurrency relationship for `/generate-proof`

Resource consumption trends This section analyses CPU and memory usage patterns across different concurrency levels. The trends reveal how each implementation handles computational load and memory management under increasing demand.

CPU Utilisation

- `/register` (Figure 5): Rust/ZK holds 9–12 % while NestJS/ZK and JWT both stay below 7 %.
- `/verify-proof` (Figure 6): Rust/ZK peaks around 41 % (pairing operations), NestJS/ZK is approximately 22 %, and JWT remains below 8 %.
- `/generate-proof` (Figure 7): NestJS/ZK is highest at about 38 %, Rust/ZK sits in the 26–30 % band, and JWT stays under 7 %.

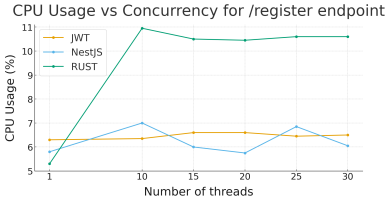

Figure 5: CPU for `/register`

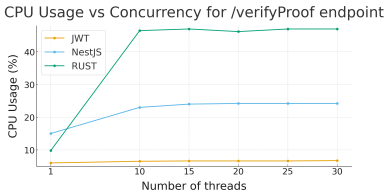

Figure 6: CPU for `/verify-proof`

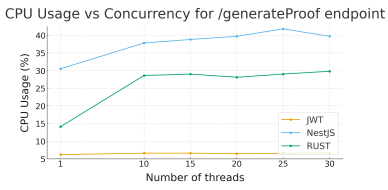

Figure 7: CPU for `/generate-proof`

Memory Footprint

- `/register` (Figure 8): Rust/ZK is minimal at 25–30 MB, while NestJS/ZK uses 120–145 MB and JWT 130–140 MB.
- `/verify-proof` (Figure 9): all stacks are stable—JWT around 200 MB, Rust/ZK 185 MB, and NestJS/ZK 165 MB.
- `/generate-proof` (Figure 10): Rust/ZK peaks near 525 MB, NestJS/ZK requires about 306 MB, and the JWT baseline remains close to 160 MB.

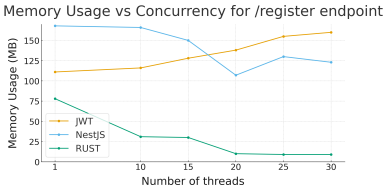

Figure 8: Memory for `/register`

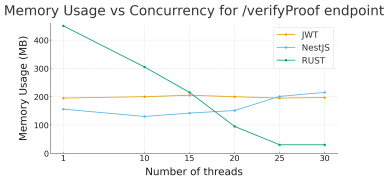

Figure 9: Memory for `/verify-proof`

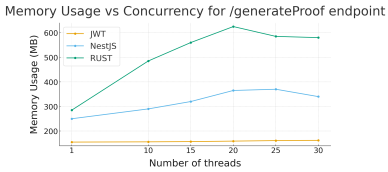

Figure 10: Memory for `/generate-proof`

6.4 Discussion

6.4.1 Performance asymmetry and causes.

Table 4 and Figures 2–4 show the core asymmetry of zk-SNARK workflows: *verification is fast and predictable*, while *proof generation is the bottleneck*. The verification of Groth16 in Rust is complete in ≈2–3 ms even at higher concurrency, while proof generation takes ≈0.69 s (NestJS) to ≈1.4 s (Rust) on our circuit. The gap stems from (i) constant-time pairings for verify vs large arithmetic circuits for prove, and (ii) implementation details: the NestJS path reuses the WASM witness calculator, while the Rust prover reloads/rehydrates artefacts and data structures per request. This also explains resource profiles: verification is CPU-light and memory-stable; proof generation drives CPU and memory (Rust peaking ∼525 MB vs. NestJS ∼306 MB).

6.4.2 JWT contrast and privacy trade-off.

JWT baselines are unsurprisingly faster: sub-40 ms for the log-in and < 10 ms for the verification with high RPS. The cost is privacy and blast radius: The JWTs are tokens that bear claims that are routinely visible to services and logs, whereas the ZK flow authenticates with *zero attribute disclosure* (UC-A) and still enforces correct access under tampering (UC-B). In short: JWT optimises latency; ZK minimises disclosure.

However, JWTs encode all user claims and are bearer tokens by design; any service that possesses the token can access all encoded attributes. This violates the principle of least privilege and exposes personal data if a service or log is compromised. In contrast, zero-knowledge proofs allow one to prove the possession of a credential without revealing the credential itself. The European Digital Identity framework therefore recommends ZK proofs to enable selective disclosure and unlinkability between services. Thus, while JWTs are faster, they carry inherent privacy risks that ZK-based schemes mitigate.

6.4.3 Rust vs. NestJS for ZK implementations

The Rust prototype delivers superior latency in lightweight operations (`/register`, `/verify-proof`), while NestJS leads in the computationally heavy `/generate-proof` endpoint. This performance asymmetry stems from implementation differences in how each stack handles circuit artefacts and witness generation.

- **Circuit artefact handling** – The Rust implementation reloads Circom WASM / R1CS artefacts per proof due to library constraints, adding I/O overhead at high concurrency. NestJS loads the WASM witness calculator once and reuses it across requests.

- **Concurrency/runtime model** – Rust uses a bounded blocking pool (4 workers) to protect resources, while Node.js offloads compute to WASM and keeps the event loop responsive, yielding smoother scaling for proof generation.
- **Memory profile** – During proving, Rust peaks around ∼525 MB vs. NestJS ∼306 MB. For lightweight operations (verify/register), Rust remains very lean.
- **WASM efficiency** – The NestJS path benefits from optimised WASM witness generation, while Rust incurs overhead from repeated artefact loading and field element conversions.

Overall: Rust is excellent when verification dominates (ultralow tail latency, small footprint). However, many practical flows are *prove-heavy* and benefit from artefact reuse, easier parallelisation, and ecosystem integration. *NestJS is favoured* as the default runtime for end-to-end ZK authentication (owing to faster proof generation, simpler operations, and strong library support), with Rust used as a targeted optimisation for verifier-heavy or performance-critical roles.

6.5 Limitations

Although research and proof-of-concept implementation demonstrates the feasibility of ZK proof-based microservice authentication, further enhancements are still possible.

- **Trusted setup alternatives.** Participating in a public ceremony or adopting universal setups (PLONK, Marlin) would eliminate the need for trusted circuit setups.
- **Replay protection and session binding.** Incorporating secure random nonces or service-specific challenges into the proof would prevent reuse of proofs across sessions.
- **Rate limiting and abuse prevention.** To mitigate denial-of-service risks, the endpoints could implement rate limits or require CAPTCHAs for repeated requests for proof generation.
- **Session tokens or delegated credentials.** After a proof is verified, the service could issue a short-lived token (for example, a signed JWT or macaroons) to avoid requiring a proof on every downstream call while still respecting the least privilege.
- **Smart contract integration.** The returned Solidity parameters can be passed to an on-chain verifier, enabling the commitment to serve as an on-chain identity in decentralised applications.

Regarding RO2, UC-A and UC-B show that ZK achieves zero attribute disclosure and correct rejection of tampered inputs, with replay resistance requiring nonce binding. Meanwhile, the RO3 results (UC-C) quantified the overheads (proof generation versus verification) and the relative gap to JWT under load.

7 Conclusion

This research was about whether zero-knowledge proofs (ZKPs) can provide fast, privacy-preserving zero-trust authentication for microservices. Integrating a Groth16-based workflow that substitutes JWTs with per-request proofs, the evaluation shows that services can validate clients without revealing client secrets, addressing the privacy gap noted by (Berardi et al.; 2022). Experiments confirmed that the prototype eliminates credential leakage and maintains verification below $\approx 3\,\text{ms}$, and proof generation remains roughly two orders of magnitude slower than JWT signing. These results show that privacy-by-design authentication is already practicable and provide a concrete alternative to token-centric models that rely on implicit internal trust (Zaheer et al.; 2019).

Lessons learnt A key insight is that per request zk-SNARK verification can be integrated with millisecond-scale overhead, making strict zero-trust checks feasible in microservices. Equally important, it was observed that *replay prevention* requires binding each proof to a one-time challenge (nonce/timestamp and, optionally, request context); omitting this binding (as in our initial prototype) allows a captured proof/commitment pair to be replayed. Finally, the evaluation confirmed the practical performance trade-off: Proof generation is $\gg$ JWT token issuance, shaping system design choices.

Future work. Reduce proof-generation latency and enhancing operational robustness. First, embed secure random nonces/timestamps into the circuit to make every proof single-use, then cache circuit artefacts and keep witness calculators warm so the service avoids redundant I/O. Second, adopt proving systems with transparent or updatable setups (e.g., PLONK or zk-STARK) to remove trusted setup risks while experimenting with GPU/FPGA accelerators, parallel witness computation, and proof aggregation strategies that amortise costs across requests (Lin et al.; 2023; Motlagh et al.; 2025). Third, off-load heavy proving to a dedicated micro-cluster or edge nodes and pipeline proof generation asynchronously, issuing a short-lived bearer token while the definitive ZKP is being prepared. In summary, the thesis designed and implemented a dual stack ZKP authentication prototype (NestJS and Rust), introduced an Attribute Disclosure Count metric, and conducted an empirical evaluation in terms of privacy, access control correctness, and performance. These findings answer the research question. It is indeed true that ZKPs can enforce zero-trust authentication with practical performance, subject to the quantified caveats (notably the need for nonce binding to prevent replay and the cost of proof generation). It remains open to validate the optimised workflow under production-scale traffic and integrate the verifier as a plug-in for service meshes or API gateways. It could then turn the prototype into a deployable, commercial-grade solution that couples rigorous data minimisation with high-throughput zero-trust security across cloud and edge environments.

References

Berardi, D., Giallorenzo, S., Mauro, J., Melis, A., Montesi, F. and Prandini, M. (2022). Microservice security: a systematic literature review, *PeerJ Computer Science* **8**: e779. **URL:** *https://doi.org/10.7717/peerj-cs.779*

Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P. and Maxwell, G. (2018). Bulletproofs: Short proofs for confidential transactions and more, *2018 IEEE Symposium*

on Security and Privacy (SP), IEEE, pp. 315–334.
URL: *https://doi.org/10.1109/SP.2018.00020*

Chen, Z., Jiang, Y., Song, X. and Chen, L. (2023). A survey on zero-knowledge authentication for internet of things, *Electronics* **12**(5): 1145.
URL: *https://doi.org/10.3390/electronics12051145*

de Almeida, M. G. and Canedo, E. D. (2022). Authentication and authorization in microservices architecture: A systematic literature review, *Applied Sciences* **12**(6): 3023.
URL: *https://doi.org/10.3390/app12063023*

Feige, U., Fiat, A. and Shamir, A. (1988). Zero-knowledge proofs of identity, *Journal of Cryptology* **1**(2): 77–94.
URL: *https://doi.org/10.1007/BF02351717*

Lin, X., Zhang, Y., Huang, C., Xing, B., Chen, L., Hu, D. and Chen, Y. (2023). An access control system based on blockchain with zero-knowledge rollups in high-traffic iot environments, *Sensors* **23**(7): 3443.
URL: *https://doi.org/10.3390/s23073443*

Maesa, D. D. F., Lisi, A., Mori, P., Ricci, L. and Boschi, G. (2023). Self sovereign and blockchain based access control: Supporting attributes privacy with zero knowledge, *Journal of Network and Computer Applications* **212**: 103577.
URL: *https://doi.org/10.1016/j.jnca.2022.103577*

Motlagh, S. M. R., Pahl, C., Barzegar, H. R. and El Ioini, N. (2025). A comparative evaluation of zero knowledge proof techniques, *Proceedings of the 10th International Conference on Internet of Things, Big Data and Security (IoTBDS 2025)*, pp. 237–244.
URL: *https://doi.org/10.5220/0013269100003944*

Sarkar, S., Choudhary, G., Shandilya, S. K., Hussain, A. and Kim, H. (2022). Security of zero trust networks in cloud computing: A comparative review, *Sustainability* **14**(18): 11213.
URL: *https://doi.org/10.3390/su141811213*

Venčkauskas, A., Kukta, D., Grigaliūnas, Š. and Brūzgienė, R. (2023). Enhancing microservices security with token-based access control method, *Sensors* **23**(6): 3363.
URL: *https://doi.org/10.3390/s23063363*

Zaheer, Z., Chang, H., Mukherjee, S. and Van der Merwe, J. (2019). eztrust: Network-independent zero-trust perimeterization for microservices, *Proceedings of the 2019 ACM Symposium on SDN Research (SOSR)*, ACM, pp. 49–61.
URL: *https://doi.org/10.1145/3314148.3314349*

Zhong, Y., Hovanes, J. and Guin, U. (2023). On-demand device authentication using zero-knowledge proofs for smart systems, *Proceedings of the 2023 Great Lakes Symposium on VLSI (GLSVLSI '23)*, ACM, pp. 91–96.
URL: *https://doi.org/10.1145/3583781.3590275*