POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

ZK-KYC-DSIG: An eIDAS2 Compliant Privacy Preserving Identity Verification Framework via Zero Knowledge Proof and Digital Signature

Tesi di Laurea Magistrale in
Computer Science Engineering - Ingegneria Informatica

Author: **Matteo Savino**

Student ID: 994779
Advisor: Prof. Francesco Bruschi
Co-advisors: Marco Esposito
Academic Year: 2023-24

Abstract

Digital identity has become a key aspect of life in a world where a significant portion of activities take place online. As a result, multiple regulations have emerged such as the EU's eIDAS which is currently at its second iteration. It provides a comprehensive framework aimed at promoting smooth online interactions between EU member states. On the other hand, individuals continue to face challenges in order to maintain control over their personal information in the digital environment. Namely, eIDAS 2 places greater emphasis on privacy and provides a framework that supports emerging technologies like ZKPs and DLTs. This thesis leverages these technologies to develop a tailored identity verification solution that complies with the mentioned regulations and adheres to the principles of SSI. The proposed approach combines eIDAS 2 compliant digital signatures with ZK-SNARKs and blockchain, ensuring decentralized, secure and privacy preserving identification. The results have been evaluated across multiple dimensions, including computational efficiency and gas fees. The outcome of this analysis highlights practical feasibility and strong future potential, despite some remaining limitations.

Keywords: Zero Knowledge Proof, Digital Identity, eIDAS 2, Digital Signature, Blockchain, DLTs

Abstract in lingua italiana

L'identità digitale è diventata un aspetto fondamentale della vita in un mondo in cui una parte significativa delle attività si svolge online. Di conseguenza, sono emerse diverse normative, come l'eIDAS dell'Unione Europea, attualmente alla sua seconda iterazione. Essa fornisce un quadro normativo completo volto a promuovere interazioni online fluide tra gli Stati membri. Dall'altro lato, gli individui continuano ad affrontare difficoltà nel mantenere il controllo sulle proprie informazioni personali nell'ambiente digitale. In particolare, eIDAS 2 pone una maggiore enfasi sulla privacy e fornisce un quadro che supporta tecnologie emergenti come le prove a zero conoscenza (ZKP) e le tecnologie a registro distribuito (DLT). Questa tesi sfrutta specificamente tali tecnologie per presentare una soluzione su misura per la verifica dell'identità, compatibile con le normative citate e con il concetto di SSI. L'approccio proposto combina firme digitali conformi a eIDAS 2 con ZK-SNARKs e blockchain, garantendo una verifica dell'identità sicura e decentralizzata, mantenendo allo stesso tempo la privacy dell'individuo. I risultati sono stati valutati e analizzati in diverse dimensioni, tra cui l'efficienza computazionale e i costi delle transazioni, dimostrando così la fattibilità pratica e un forte potenziale futuro nonostante alcuni limiti rimanenti.

Parole chiave: Prove a zero conoscenza, Identità digitale, eIDAS 2, Firma digitale, Blockchain, Tecnologie a registro distribuito

Contents

Introduction

0.1. Context and Motivations

In the last decades, digital services have expanded significantly in both the public and private sectors. The constant online presence of individuals has resulted in a vast amount of personal data shared across various platforms. On one hand, people can easily access a wide range of global services, such as healthcare, education, financial transactions, e-commerce and more, simply by using their digital devices. On the other hand, this interconnected environment requires people to disclose an ever increasing amount of personal information to governments, companies and other organizations. This dynamic has given rise to the concept of digital identity which can be broadly defined as the comprehensive set of data and online footprints that each individual leaves behind. As digital identities become essential to online interactions, there is also a pressing need for reliable identification and verification mechanisms. Governments, companies and organizations providing online services must be certain who they are dealing with and/or be able to confirm a user's claims in order to allow citizens and consumers to exercise their rights confidently and efficiently [1]. Standard approaches to identity proofing typically involve sharing personal data with institutions. Hence, personal information has become more valuable over time and is now referred to as PII. Unfortunately, this process of commoditization of PII has also been recognized by fraudsters [2]. As a matter of fact, identity theft is a growing trend which require an intervention in order to be solved [3]. Additionally, users of online services may not want to share some information if not strictly necessary. There are multiple reasons for ensuring privacy [4]. In short, privacy allow individuals to negotiate and set boundaries between themselves and others. "This allows individuals to set their personal privacy boundary in terms of whom they interact with, how they interact with others, and when and where these interactions occur, taking into account the costs and benefits of such interactions" [5].

The concept of SSI gained popularity in response to the aforementioned circumstances. It is a subcategory of the wider notion of decentralized identity [6]. The core principles of this paradigm, which revolve around a user-centric identity model (figure 1), have been

extensively discussed in recent years and they can be grouped into three main categories: security, controllability and portability. In essence, security entails safeguarding personal data and restricting its disclosure to what is strictly required, while still allowing for a persistent identity. Controllability ensures that individuals maintain full control over their identity (creation, management, removal) and that each operation is performed with their explicit consent. This principle also includes the "right to be forgotten", which must coexist without undermining the identity's persistence. Meanwhile, portability ensures that users can exercise control over their digital identities wherever they want without being tethered to a specific provider [7].

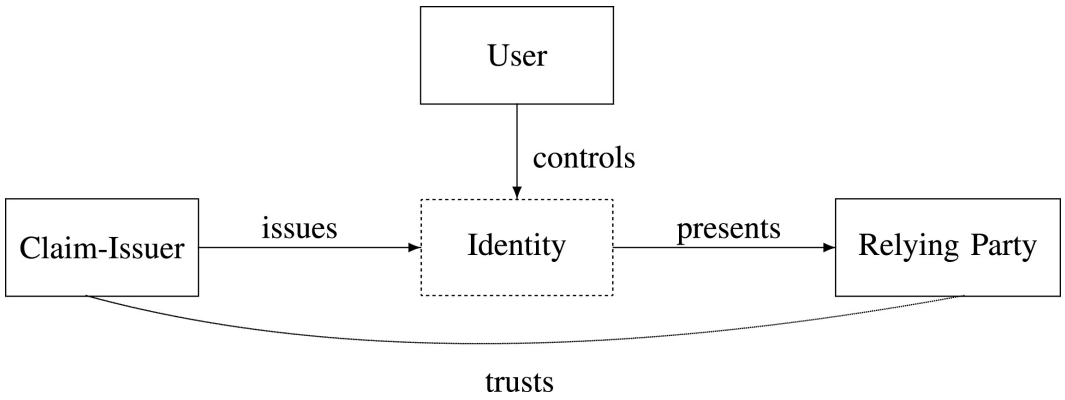

Figure 1: Self-Sovereign Identity actors.

Last but not least, regulators have acknowledged the urgent need to actively address digital identity challenges. Since the objective of this chapter is to provide a clear context, only a concise summary of the regulatory framework is included: further details can be found in the "State of the art" chapter. In the EU, the original eIDAS Regulation established the legal and technical foundations needed for secure and straightforward cross border digital transactions. This first release insufficiently addressed privacy and did not define a standard for the technologies that should be used. A new version of the regulation was launched in 2024 to address newly emerged challenges and to overcome the limits of the previous framework. Although eIDAS 2 is technologically neutral, it incorporates emerging technologies and new features such as DLTs, selective disclosure and ZKP in the regulatory framework. By doing so, eIDAS 2 has paved the way for innovative approaches that align better with the principles of SSI ("Without legal compliance SSI would remain academic") [8, 9].

It should be clear now that without careful considerations on how digital identity systems are designed, there is a real risk of sliding into a scenario where people lose trust on online services and could find themselves locked out of opportunities or worse, exposed to

security risks. This highlights the critical importance of good digital identity: one that not only meets the needs of verification and security but also respects user autonomy, respect privacy principles and fosters trust [1].

0.2. Problem statement

Despite widespread interest, SSI often struggles to fully achieve its core principles within the constraints imposed by existing infrastructures and regulations. As a result, the practical adoption of SSI in real world settings is still limited. On top of that, a significant technical challenge lies in executing advanced cryptographic operations directly on the blockchain. ZKPs and related cryptographic techniques, while beneficial for privacy preserving verification, are computationally intensive. Therefore, it is essential to prioritize the optimization of both execution time and gas fees. Performing ZKP operations on an EVM compatible blockchain also requires optimized circuits and considering the offloading of some operations. Balancing these performance requirements with regulatory needs further adds constraints to the design of a digital identity verification framework.

0.3. Goals and Contributions

The main goal of this thesis consists of designing and implementing a privacy preserving identity proofing system that addresses key challenges of digital identity management. The focus of the project is to exhibit compliance with the eIDAS2 regulation and SSI principles while having good enough performances for a practical usage. Specifically, the system enables a user to demonstrate ownership of a valid tax ID (Italian Codice Fiscale) that has been verified by a trusted organization. This organization is needed to establish the trust relationship but does not actively participate in either proof validation or verification. Additionally, if required (e.g. in a lawsuit), the taxID must be disclosed to a governmental authority, making the approach suitable for KYC scenarios where a user's identity has to be revealed under legal conditions.

Conversely, the core contribution lays in the design of the ZK-KYC-DSIG framework. This project consists of an identity verification solution which leverages the usage of eIDAS2 compliant digital signature, EVM compatible blockchain and ZK-SNARKs. A testable prototype is provided and a comprehensive analysis of performance and compliance aspects is presented, providing insights on both the framework's strengths and limitations.

0.4. Thesis structure

The thesis is organized into eight chapters and the structure is as follows. Chapter 1 (Preliminary Background) introduces the foundational concepts and theoretical underpinnings of the research. It covers key cryptographic principles, privacy requirements and relevant regulatory frameworks. Chapter 2 (State of the art) reviews the state of the art by examining existing solutions and related work in the field of identity verification, highlighting their strengths and limitations. Chapter 3 (Methods and Materials) describes the methodology and the metrics used to evaluate the proposed framework. It also discusses how tests are designed and conducted to assess both performance and compliance. Chapter 4 (Proposed Solution) provides a detailed overview of the system architecture, including the design of the ZKP circuit, the integration of digital signatures and the role of blockchain. Chapter 5 (Implementation) delves into the technical details of the implementation. It outlines the system components, data flow, configurations and development challenges, demonstrating the practical implementation of the designed architecture. Chapter 6 (Evaluation and Results) presents and discusses the experimental results, covering aspects such as computational overhead, gas fees and any observed trade-offs. Chapter 7 (Future developments) explores opportunities for further development or enhancements to the proposed system, including additional use cases and areas for optimization. Chapter 8 (Conclusions) concludes the thesis by summarizing the main findings, reflecting on their broader implications and suggesting how they can impact future research and real-world applications. Appendix (Appendix) provides supplementary materials including code and data omitted from the main chapters. At the end, a comprehensive list of acronyms used throughout the thesis is included.

1 | Preliminary Background

1.1. PKI and digital signature

The primary goal of certificates and PKI is to bind an identifier to a public key [10]. However, an identifier differs from an identity. An entity (subscriber) may claim ownership of a specific identifier but other entities (relying parties) require a method to verify the validity of that claim, confirming the identity. This necessitates a process to authenticate the accuracy of these bindings. For this purpose, CAs and certificates are employed. CAs are trusted authorities whose certificates must be stored in the trust stores of every device. They function as certificate issuers for subscriber entities, which are identified through their certificates.

1.1.1. PKI Standards and Certificate Fundamentals

The entire PKI is built around public key cryptography, which utilizes key pairs: private and public keys. These keys enable a dual functionality:

- **Encrypt**: Using the public key to encrypt data. The only way to decrypt it is through the private key.
- **Sign**: Using the private key to sign data. Anyone with the corresponding public key can verify the signature, proving which key produced it.

There is an issue with the presented system, as these operations are not feasible if the relying party does not possess the subscriber public key. The aforementioned certificates address precisely this task. They consist of a data structure containing several pieces of information, including the public key and an identifier. Subsequently, this data structure is also signed by the issuing CA. Conversely, the CA certificate is already known as it resides in the trust store of each device. The certificates of root CAs are self signed. In practice, there could also be a certificate chain involving a root CA and multiple intermediate CAs, which sequentially sign certificates, ultimately leading to the end entity leaf certificate.

As explained, there is no central entity, so the expiration date of certificates must be

embedded within the certificates themselves. The X.509 standard includes fields for an issued-at time, a not-before time and a not-after time. Additionally, CRLs exist and they are lists of revoked certificates that can invalidate a certificate before its expiration (for instance, in the case of a compromised key). However, the topic of CRLs is complex as it requires actively checking this lists and many systems may fail to do so, potentially accepting invalid certificates. For this reason, alternative techniques are being explored, though they fall outside the scope of this chapter. A straightforward best practice worth noting is to favor short lived certificates whenever possible, allowing them to expire naturally without renewal in the event of a compromise.

Delving into more technical details, the certificates used on the web are X.509 certificates. The PKI that employs this type of certificate is known as PKIX. Data within these certificates is structured according to a standard notation called ASN.1, which serves as a framework for defining data types. It supports common data types such as integers, strings, sets and sequences as well as more complex data types like OIDs which are sequences of integers that act as universally unique identifiers. Various encoding rules exist but the ones most commonly used in X.509 certificates are undoubtedly BER and DER. DER, by far the more prevalent, consists of straightforward binary values. Since binary data can be cumbersome to handle, it is typically packaged into PEM files, which encode the binary in base64 format and sandwich it between a header and a footer. Alternatively, there are more sophisticated packaging options known as envelope formats, which define larger data structures in ASN.1. The PKCS suite of standards outlines the envelope formats which are more likely to encounter.

1.1.2. CMS digital signature

The [11] document outlines the CMS, which establishes a syntax for digitally signing, authenticating, digesting, or encrypting arbitrary messages. It builds upon the PKCS#7 version 1.5 standard and supports PKI adopting X.509 certificates. An implementation adhering to CMS must include the ContentInfo protection content along with the data, signed-data, and enveloped-data content types. Conversely, other content types remain optional. In practice, ZK-KYC-DSIG utilizes only a subset of these, which are detailed in the 1.1 figure. Generally, all data within a digital signature is BER-encoded. However, signed attributes and authenticated attributes require DER encoding to enable verification by recipients.

```
ContentInfo ::= SEQUENCE {
  contentType ContentType,
  content [0] EXPLICIT ANY DEFINED BY contentType }

ContentType ::= OBJECT IDENTIFIER

SignedData ::= SEQUENCE {
  version CMSVersion,
  digestAlgorithms DigestAlgorithmIdentifiers,
  encapContentInfo EncapsulatedContentInfo,
  certificates [0] IMPLICIT CertificateSet OPTIONAL,
  crls [1] IMPLICIT RevocationInfoChoices OPTIONAL,
  signerInfos SignerInfos }

DigestAlgorithmIdentifiers ::= SET OF DigestAlgorithmIdentifier

SignerInfos ::= SET OF SignerInfo

EncapsulatedContentInfo ::= SEQUENCE {
  eContentType ContentType,
  eContent [0] EXPLICIT OCTET STRING OPTIONAL }

SignerInfo ::= SEQUENCE {
  version CMSVersion,
  sid SignerIdentifier,
  digestAlgorithm DigestAlgorithmIdentifier,
  signedAttrs [0] IMPLICIT SignedAttributes OPTIONAL,
  signatureAlgorithm SignatureAlgorithmIdentifier,
  signature SignatureValue,
  unsignedAttrs [1] IMPLICIT UnsignedAttributes OPTIONAL }

SignerIdentifier ::= CHOICE {
  issuerAndSerialNumber IssuerAndSerialNumber,
  subjectKeyIdentifier [0] SubjectKeyIdentifier }

SignedAttributes ::= SET SIZE (1..MAX) OF Attribute

Attribute ::= SEQUENCE {
  attrType OBJECT IDENTIFIER,
  attrValues SET OF AttributeValue }

AttributeValue ::= ANY

SignatureValue ::= OCTET STRING
```

Code 1.1: CMS data structure.

Several extensions of the CMS exist. Notably, [12] defines the format of electronic signatures, focusing on long term validity and including the non repudiation property of the signature. Among the presented formats is CAdES-BES, which satisfies the legal requirements for electronic signatures and exhibiting compliance with the European Directive on Electronic Signatures.

1.2. DLTs and blockchain

Blockchain was first introduced by Satoshi Nakamoto with his renowned research on a peer to peer network for electronic cash transactions named Bitcoin [13].

Blockchain exhibits unique and fundamental features:

- **Decentralization**: No third party is required to validate transactions. Each node can directly transact with another using only its address, eliminating the need for intermediaries.
- **Transparency**: All algorithms and transactions are fully visible and accessible to every user, fostering openness within the network.
- **Autonomy**: Blockchains operate as trustless networks, where cryptographic principles replace the need for a governing authority to establish trust among participants.
- **Immutability**: Once a transaction is recorded on the blockchain, it cannot be altered or removed. It can only be validated and verified, ensuring permanent integrity.
- **Irreversibility**: Once a transaction is committed on the blockchain, it cannot be reversed.
- **Auditability**: A DL records all transactions with timestamps, allowing the complete history of actions to be traced.

A blockchain is a set of sequentially ordered blocks containing transactional records which form a DL. This structure gives rise to the broader term DLT. Each block includes a reference to the previous block via a hash value, ensuring continuity and integrity. Participants in a blockchain, known as nodes, are entities that maintain a copy of the DL. Any node can initiate a transaction by applying a digital signature by leveraging public key cryptography (similar yet distinct from the mechanisms discussed in PKI). Subsequently, a specific type of node, called a miner, must "mine" the transaction which is the process of appending it to the latest block. To do so, miners tackle a complex computational task, earning the right to add the transaction upon success. If they succeed, they receive a reward from both the protocol itself and a small fee paid by the node initiating the transaction. This consensus mechanism is known as PoW. The increasing demand for computational power driven by the competition to obtain the rewards have made PoW unsustainable and environmentally unfriendly. As a result, alternative consensus mechanisms have been explored and implemented, such as PoS. PoS replaces miners with validators. They are are selected to confirm transactions based on the amount of cryptocurrency they hold and

commit as a stake [14]. A prominent example of a PoS blockchain is Ethereum, which transitioned from PoW to PoS over the course of its life cycle.

1.2.1. Smart contracts and DAPPs

The concept of smart contracts was introduced in the 1990s by Nick Szabo [15], but it was primarily with the advent of blockchain technology, notably Ethereum, that they gained widespread practical application. Ethereum was introduced to the public through its white paper in 2014 by Vitalik Buterin [16], laying out the vision for a blockchain platform supporting smart contracts and decentralized applications, before officially launching in 2015.

Smart contracts enhance traditional contracts by automating agreement execution between multiple parties once predefined conditions are met. In practice, they encode agreements into executable code, minimizing human error and avoiding disputes. The Ethereum blockchain implements a Turing complete virtual machine, known as the EVM, enabling developers to write sophisticated smart contracts using the Solidity programming language. This flexibility supports the creation of a wide range of DAPPs. Within these contracts, functions are classified as either read only, which incur no fees (called "gas"), or write functions which require a gas fee due to the need to record state transitions in a new block on the blockchain. Additionally, the cost needed for each operation prevents indefinite contract execution and potential abuse. Despite their advantages, smart contracts face challenges that hinder widespread adoption. Legally speaking, each country has different regulations. Additionally, contract immutability conflicts with rights like the "right to be forgotten" enshrined in laws such as the GDPR. Moreover, smart contracts often depend on off chain data, necessitating a mechanism to retrieve and deliver external information to the blockchain. This mechanism relies on oracles, which serve as trusted intermediaries to fetch and supply off chain data to the blockchain. While existing oracles are well tested and employ mechanisms to minimize trust requirements, they remain potential points of failure that smart contracts must account for. Immutability also complicates correcting development errors once a contract is deployed. Finally, scalability remains a significant hurdle: while centralized systems handle thousands of transactions per second, DLTs struggle to achieve comparable performance. Nevertheless, the technology holds immense promise and active research is already addressing these limitations. One notable advancement is the adoption of Layer 2 solutions which process transactions on a secondary blockchain offloading much of the workload from the main chain and recording only aggregated results on the primary blockchain thereby enhancing efficiency and scalability [17].

1.3. eIDAS 2 and EUDI frameworks

Facebook, Google and Apple developed their federated identity systems (figure 1.1).

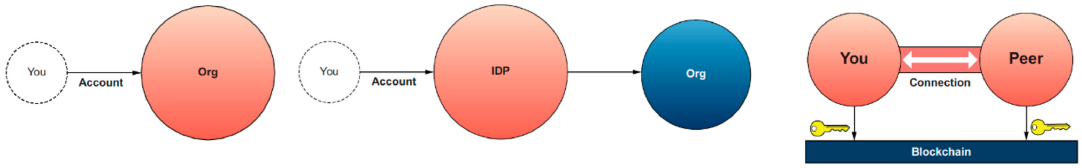

Figure 1.1: Centralized, federated and Self Sovereign Identity models.

Many services adopted these authentication systems and users willingly adopted them to ease credential management through a single identity. On the other hand, these systems offer limited control over which information is shared and expose additional user behavior metadata to identity providers, which may be shared with brokers. On top of that, these systems are closed between each other and do not allow for any interoperability revealing the need for a standard guideline for digital identity. To address these issues, the EU introduced the first iteration of the eIDAS regulation.It's objective was to enable secure digital transactions among public organizations, private companies and EU citizens, reducing reliance on federated solutions. The regulation established the concept of attested attributes related to data with a higher level of truth [18]. These attributes are digitally certified characteristics or credentials that individuals can use to prove specific aspects of their identity or qualifications with greater trustworthiness. In a subsequent document, named ARF, the EUDI architecture was formalized and the EUDIW was introduced as a key component. The EUDIW is a digital tool designed to allow EU citizens to store, manage, and share their electronic identity credentials and attested attributes across borders in a standardized and privacy preserving way. However, both eIDAS and the EUDIW have numerous limitations. First, the attested attributes defined by eIDAS were restricted to governmental identification schemes. Hence, many other attributes tied to a broader concept of identity were excluded, such as diplomas or certifications which are often still managed through paper based documents. Moreover, eIDAS almost entirely disregarded privacy principles. The more recent ARF, introducing the EUDIW, included some references to privacy principles but still lacked the necessary standardization. Additionally, there was no legal certainty for the application of technologies like DLTs and ZKP, leaving their implementation an unpredictable and uncertain variable legally speaking. For these reasons, eIDAS 2 was discussed and subsequently released in 2024. Its primary goal is to extend and refine the original regulation rather than replace it [8].

eIDAS 2 introduces, on one hand, the EUDIW as "as an electronic identification means

which allows the user to securely store, manage and validate person identification data and electronic attestations of attributes for the purpose of providing them to relying parties and other users of European Digital Identity Wallets, and to sign by means of qualified electronic signatures or to seal by means of qualified electronic seals". On the other hand, it introduces the EAA as "as an attestation in electronic form that allows attributes to be authenticated". In practice, EAAs are credentials and certifying acts issued by recognized authorities, such as a driver's license, university degree or certificate of extended family status. Additionally, selective disclosure of these attributes is introduced to minimize the sharing of personal information as much as possible. Another goal of eIDAS 2 is to empower users with control over their own data through decentralization (see Figure 1.1). This eliminates the need for a central third party repository, thereby removing a potential central point of failure or vulnerability. Specifically, this embodies the SSI model principles. Furthermore, the regulation also provides legal recognition to technologies like electronic ledgers, defined as "a sequence of electronic data records ensuring the integrity of those records and the accuracy of their chronological ordering," as well as ZKPs. Electronic ledgers clearly encompass DLTs, which support the aforementioned decentralization principle. It appears evident that many limitations have been addressed. However, some details remain unclear. The regulation stipulates that these principles should be applied but does not specify how to integrate them, leaving the responsibility to member states. This same issue previously posed challenges in earlier versions of the regulation [9, 19].

1.4. ZKP and ZK-SNARKs

ZKPs are a cryptographic primitive that allow one party, the prover $\mathcal{P}$, to convince another party, the verifier $\mathcal{V}$, that a given statement is true without disclosing any information beyond the validity of the statement itself. This powerful capability makes ZKPs particularly valuable in privacy preserving applications, where it is necessary to verify specific properties or claims without exposing sensitive underlying data. In this context, a statement refers to a mathematically verifiable assertion, such as knowledge of a secret value or the correct execution of a computational process. Formally, a ZKP allows the prover $\mathcal{P}$ to demonstrate knowledge of a secret value w, known as the witness, such that a function $\mathcal{F}(x; w) = y$ holds, where x and y are public inputs and outputs, respectively. The prover achieves this by generating a proof that attests to the truth of the statement $\mathcal{F}(x; w) = y$, without revealing any details about w. This process ensures that the verifier learns only that the statement is true, preserving the confidentiality of the witness [20, 21].

A ZKP must satisfy three fundamental properties:

- **Completeness:** If the statement is true, an honest prover can convince a verifier.
- **Soundness:** If the statement is false, no dishonest prover can convince an honest verifier of its truth, except with negligible probability. This property ensures the reliability of the verification process.
- **Zero-Knowledge:** If the statement is true, the verifier gains no knowledge about the witness or any private inputs beyond the fact that the statement holds.

1.4.1. Types of ZKP Constructions

Building on the foundational concept of ZKPs, several specialized constructions have emerged, each designed to balance privacy, efficiency and security requirements. This subsection briefly describes four prominent types, with their key attributes summarized in Table 1.1 and asymptotic characteristics detailed in Table 1.2:

- **ZK-SNARKs**: They are non interactive protocols known for small proof sizes and constant time verification complexity. Groth16 ZK-SNARKs operate by arithmetizing a computation into a R1CS, which is then converted into a QAP of polynomials. A trusted setup generates prover and verifier keys for each circuit, allowing the prover to create a succinct proof that any verifier can efficiently check without recomputing. However, this setup is computationally intensive, relies on elliptic curve cryptography (vulnerable to quantum attacks) and requires significant prover computation.
- **ZK-STARKs**: They eliminate the trusted setup by using publicly verifiable randomness, achieving post quantum security through collision resistant hash functions instead of elliptic curve cryptography. The computation execution trace is arithmetized into polynomials, verified through low degree testing with FRI commitments and compacted into a Merkle tree, whose root acts as a commitment for efficient verification. They provide scalability via Merkle trees but produce larger proofs and demand more computational resources for generation and verification.
- **MPCitH**: They leverage secure multiparty computation, where the prover simulates multiple parties locally and executes an MPC protocol by assigning secret shares of inputs among them to compute a circuit. The prover generates views (messages and data for each simulated party), commits these views using a standard commitment scheme and interactively verifies a subset for consistency. The Fiat-Shamir transform makes this process non-interactive. Requiring no trusted setup and offering post-quantum security, they yield bigger proof sizes with lin-

ear verification complexity, balanced by flexibility in optimizing proof generation through MPC techniques.

- **VOLE-ZK**: They are interactive protocols that use information theoretic message authentication codes (IT-MACs) via VOLE correlations, where the prover (VOLE sender) and verifier (VOLE receiver) operate over vectors satisfying $u_i = v_i + x_i \cdot \Delta$. The prover commits to authenticated wire values in a circuit and proves consistency without revealing the witness, achieving high scalability. They produce smaller proofs without a trusted setup and ensure post quantum security, but their designated verifier nature (requiring the verifier to hold a secret Δ) restricts public verifiability.

Construction	Key Advantages	Key Disadvantages
zk-SNARKs	Succinct, Publicly Verifiable	Trusted Setup Required, Computationally Expensive to Prove, Not Post-Quantum
zk-STARKs	No Trusted Setup, Post-Quantum Secure, Scalable Prover, Publicly Verifiable	Larger Proof Sizes, Slow Verification
MPCitH	No Trusted Setup, Post-Quantum Secure, Publicly Verifiable	Slow Verification, Computationally Expensive Proving
VOLE-ZK	Highest Scalability, No Trusted Setup, Post-Quantum Secure	Slow Verification, Designated Verifier

Table 1.1: Core Attributes of Popular ZKP Constructions.

	zk-SNARKs	zk-STARKs	MPCitH	VOLE-ZK
Prover complexity	$O(n \log(n))$	$O(n\text{poly-log}(n))$	$O(n)$	$O(n)$
Verifier complexity	$O(1)$	$O(\text{poly-log}(n))$	$O(n)$	$O(n)$
Proof size	$O(1)$	$O(\text{poly-log}(n))$	$O(n)$	$O(n)$
Trusted setup	✓	✗	✗	✗
Non-interactive	✓	✓	✓	✗
Post-quantum secure	✗	✓	✓	✓
Practical proof size	120-500 bytes	10 KB - 1 MB	10-1000 KB	5-200 KB

Table 1.2: Asymptotic attributes of presented ZKP constructions.

These constructions represent a spectrum of trade offs in succinctness, security and computational overhead, as outlined in Table 1.1, making each suited to distinct applications within the ZKP domain. It is clear that the best suited approach for on-chain verification are ZK-SNARKs due to their reduced proof size and fast verification time.

1.4.2. ZK-SNARKs: Technical Overview and Properties

ZK-SNARKs are distinguished by their ability to produce compact proofs that can be verified quickly without requiring interaction between the prover and verifier, making them particularly valuable in resource constrained environments like blockchain networks.

Core Properties of ZK-SNARKs

ZK-SNARKs exhibit three defining characteristics: succinctness, non interactivity and arguments of knowledge.

- **Succinctness**: ZK-SNARKs produce proofs that are compact in size, regardless of the complexity of the underlying computation or the size of the input data. This property enables scalable and cost effective verification.
- **Non Interactivity**: Unlike interactive ZKPs, which require multiple rounds of communication between the prover and verifier, ZK-SNARKs are non interactive. This means that the prover can generate a single proof that the verifier can independently check without further communication. Non interactivity is achieved through the use of a CRS, which is shared between the prover and verifier during the setup phase.
- **Arguments of Knowledge**: ZK-SNARKs not only prove the validity of a statement but also ensure that the prover possesses explicit knowledge of the witness that substantiates the statement. This property is known as knowledge soundness.

Therefore, in addition to the general ZKP properties of completeness, soundness and zero knowledge, ZK-SNARKs exhibit a related property called **knowledge soundness**. This ensures that if the prover successfully convinces the verifier of a statement truth, the prover must actually know the corresponding witness. Unlike basic soundness, which prevents a cheating prover from validating a false statement, knowledge soundness ties the validity of the proof to the prover possession of the witness, reinforcing trust in the system integrity. Knowledge soundness is a stronger form of soundness.

Applications of ZK-SNARKs

ZK-SNARKs have found widespread applications in blockchain technology, particularly in Layer 2 scaling solutions such as ZK rollups. In ZK rollups, ZK-SNARKs are used to prove the validity of state changes on a Layer 2 blockchain without requiring the Layer 1 blockchain to execute the corresponding transactions. This enables significant cost savings and scalability improvements, as the Layer 1 blockchain only needs to verify the succinct proofs generated by the Layer 2 rollup. Additionally, ZK-SNARKs can also provide privacy preserving properties.

Lifecycle of a ZK-SNARK

The life cycle of a ZK-SNARK transforms a high-level computation into a verifiable proof through distinct phases: the front end, an intermediate arithmetization step and the back end. In the front end, a computation is converted into an arithmetic circuit. This circuit operates over a finite field defined by a large prime number, breaking down operations into a series of interconnected gates. These gates represent basic arithmetic steps like multiplication and addition. Next, the circuit is arithmetized into a R1CS, which is a system that encodes each gate operation as a set of constraints. Each constraint ensures that the operation at a gate holds true, verifying the computation correctness across all steps. In the back end, the R1CS is transformed into a QAP, which represents the constraints as polynomial equations. These polynomials capture the relationships between inputs and outputs, with one polynomial acting as a solution and another designed to equal zero at specific points tied to each constraint. A trusted setup generating keys for the prover and verifier through a CRS. Using these keys, the prover produces a succinct proof consisting of polynomial evaluations. The verifier checks this proof against public polynomial values, confirming validity efficiently without redoing the entire computation. The trusted setup requires a secure ceremony that uses private random values provided by participants to create the CRS essential for ZK-SNARK integrity. These values, known as toxic waste, must be discarded after the ceremony. If the process is flawed or the toxic waste retained, an attacker could exploit it to forge proofs, compromising security.

ZK-DSLs

A ZK-DSL is a specialized programming language designed to simplify interaction with the complexities of ZKPs by enabling developers to write high level code that translates into low level circuits. A prominent example is Circom, which allows developers to define arithmetic circuits targeting the R1CS representation for the Groth16 ZK-SNARK system. This abstraction significantly reduces the burden of manually crafting circuits, offering a clear advantage in productivity and maintainability. Despite these benefits, ZK-DSLs introduce specific challenges. Memory management becomes complex due to cryptographic operations. Consequently, efficient strategies are needed to handle large scale circuits and data structures. Furthermore, traditional programming paradigms are difficult to implement, requiring developers to adapt their coding practices. Some examples are recursion, due to the lack of a direct stack and the determinism of paths, conditional operators (excluding ternaries), mutable variables and user defined structures. Additionally, effective use of ZK-DSLs for development demands a deep understanding of cryptographic primitives, computational strategies, elliptic curves and ZKP mechanisms, which can steepen

the learning curve. Code reuse is further constrained by circuit size limits and gas usage considerations in blockchain environments. To address this issue, gadget libraries have emerged, providing modular components for circuit construction to enhance reusability and mitigate redundancy.

2 | Methods and Materials

2.1. Methodology

2.1.1. Analysis of Existing Solutions

This research does not aim to provide a comprehensive state of the art review. Although numerous papers addressing the identity verification problem have been examined, only a selection is included in the State of the art chapter. The objective of the SOTA is to trace the evolution of research in this field, highlight the technologies adopted and evaluate the strengths and limitations of various approaches.

2.1.2. System Design and Development Overview

After a thorough evaluation of the SOTA, requirements were gathered and an initial strategy was determined for the development of the identity verification solution. Subsequently, the technologies to adopt were selected (more in the Materials and Technologies selection section). A hybrid development approach has been opted for. At the beginning, the overall design and critical milestones were clearly defined and the development was segmented in periods. At the end of each period, a review was conducted to ensure alignment with project objectives, enabling iterative adjustments without disrupting the overall workflow.

2.1.3. Compliance by Design

Compliance with EU regulations and adherence to SSI principles have been enforced by design. A large collection of documents and scientific papers from the fields of law and IT was gathered and analyzed. The literature was limited to works published between 2020 and 2025. Particular attention was paid to eIDAS 2 which was released in 2024, for which more recent studies were prioritized. In the end, the initial set of 20 documents was refined to a final selection of 5 documents [6, 8, 9, 18, 19]. The design phase of ZK-KYC-DSIG was conducted with clear regulatory constraints and compliance with eIDAS2 and

SSI principles was systematically verified at every stage.

2.1.4. Experimental Design

Compliance has been assessed through inspection rather than execution based testing, as stated in the Compliance by Design subsection. On the other hand, functional tests of ZK-KYC-DSIG framework aimed at verifying the successful completion of identity verification procedures and collecting time performances of each phase. In addition, also key sizes have been analyzed. The devised tests had to cover the entire cycle, from proof generation to proof verification. The latter had also to be tested on chain to simulate a real world application. Gas costs were analyzed using a smart contract with a minimal use case. Moreover, since ZKP proof generation can be computationally intensive and its performance may vary depending on processing power, tests were conducted on at least two different machines to assess execution feasibility in different conditions.

2.1.5. Evaluation Metrics

For regulatory compliance, the evaluation is considered successful if no significant obstacles are identified within the framework. In terms of performance, the approach is to set reasonable objectives to determine whether the proposed method is suitable for further refinement, rather than imposing stringent targets more appropriate for a production ready product. Namely, ZKP operations are deemed acceptable if proof generation is completed within 3 minutes and proof verification within 10 seconds. Another key metric is the size of the proving and verification keys, for which no precise limit has been defined, only the requirement that they remain feasible for a real production environment. Additionally, gas consumption for blockchain transactions is considered acceptable if the cost stays well below the equivalent of 1 euro.

2.2. Materials and Technologies selection

The adopted technologies were meticulously selected to ensure that the proposed solution meets all specified requirements while remaining easily integrable with both current and future standards. Consequently, priority was given to widely adopted languages, multi platform technologies and rigorously audited libraries.

The ZK-KYC-DSIG framework is composed of a pre-processing layer, an off chain ZKP proof generation circuit and an on chain ZKP verification smart contract.

2.2.1. Hardware Environment for Development and Testing

Development and testing were both carried out on two Windows based systems. The primary system, which has been used for both phases is equipped with an *Intel i7-4790K CPU and 16GB of DDR3 RAM*, while the secondary system, mainly used for testing, features an *Intel i9-13900K CPU and 64GB of DDR5 RAM*. The latter system has greater processing power in order to evaluate the developed solution behavior under various conditions.

2.2.2. Test Data

Tests of ZK-KYC-DSIG involve the usage of digitally signed files and of the corresponding certificates. In order to ensure the suitability of the solution for a practical application, it had to be tested with at least one real case scenario. Hence, a sample of real data were used. Additionally, more test case have been generated through a Power Shell script which exploit the `OpenSSL` library to generate the necessary certificates and sign the file.

2.2.3. Blockchain and Cryptographic Tools

In order to respect all the requirements discussed in the Introduction chapter, the ZKP approach adopted in the ZK-KYC-DSIG framework was required to exhibit the following properties:

- Wide adoption and an active community.
- Thorough documentation and examples.
- Proven effectiveness in practical applications, not just academic experiments.
- Ease of adoption for developers without cryptographic expertise.
- Generation of succinct proofs that could be efficiently verified on chain.
- Publicly verifiable proofs.
- Support execution in different environments.

Therefore, while some ZKP frameworks achieve better performance in specific benchmarks, ZK-KYC-DSIG requires a solution that is both well established and easy for new developers to adopt. These considerations outweighed the pursuit of absolute performance, leading to favor a mature, popular approach over a purely optimal one.

It can be easily inferred from the content of Preliminary Background chapter that ZK-

SNARKs are the most suitable class of proofs since they inherently produce small proofs, have rapid verification times and allow for verification without any interaction between parties. They also align perfectly with cases where a proof may be verified multiple times over a relatively static piece of data (e.g. personal information that rarely changes). In addition, being SNARKs an approach that has been broadly experimented in the last decade, they have seen widespread use over the last years, resulting in robust ecosystem support, numerous production projects and established libraries that integrate seamlessly with blockchain [20]. Next, an investigation on which specific ZK-SNARK implementation to adopt has been conducted, comparing commonly used tools and languages. The following table shows relevant statistics about the most popular ZK-SNARKS frameworks. It is easy to observe in table 2.1 that Circom is by far the most popular approach.

Language	**Projects**	**Circuits**
Circom	518	6.4K
Leo	96	396
Zinc	31	944
Halo2	83	187
Plonky2	17	1.5k
Noir	66	544
Gnark	43	3.6k
ZoKrates	301	5.9k

Table 2.1: Statistics of different ZK domain-specific languages for constructing arithmetic circuits, collected from Github publicly available codebases.

Another recent survey [20] highlights Gnark as one of the best alternatives since it has excellent documentation and active enough community while it outperforms the Circom + SnarkJS approach in raw performance metrics. Unfortunately, their benchmarks did not include Rapidsnark which allows both faster proof generation time and mobile device support while accepting as input SnarkJS generated files ensuring compatibility. At the same time, [22] states "Both gnark and rapidsnark stand out with the most efficient provers. However, gnark's superior performance is offset by its suboptimal memory efficiency". Ultimately, the final verdict was in favor of Circom. The deciding factors revolved around its established track record in real world projects such as Tornado Cash [23] with its technical merits despite the legal issues, the support for proof generation on browsers and mobile devices and, last but not last, its greater popularity. This extensive adoption and platform flexibility minimizes barriers to entry and facilitates both maintenance and future enhancements. Consequently, ZK-KYC-DSIG has been built with Circom and

SnarkJS, employing the Groth16 protocol to generate and verify proofs. This choice of the protocol also retains compatibility with Rapidsnark which is the selected variant for a production deployment.

For the purpose of constructing the ZKP circuits, various "gadgets" have been exploited from the audited `@zk-mail/circuits` library [24]. The natural choice for a blockchain platform was Ethereum or an EVM compatible layer 2 network, as SnarkJS can automatically generate Solidity verifier contracts and Ethereum remains the most popular platform for smart contract deployment (and also for on chain identity solutions [25]). A dedicated Power Shell script has been created to generate certificates and sign files using OpenSSL, which enabled consistent testing with crafted data. Finally, gas costs of on chain transactions were estimated via the REMIX IDE built in function which provide a gas cost approximation. While these values are only indicative, they are enough to evaluate the feasibility of on chain proof verification.

2.2.4. Programming Languages, Libraries and Frameworks

The pre-processing layer of ZK-KYC-DSIG was implemented in TypeScript. This language was chosen because it meets the identified technical requirements. First, is widely adopted and benefits from strong community support. As a superset of JavaScript, TypeScript enables seamless integration with the SnarkJS library. Furthermore, rapidsnark can be executed in any environment that supports JavaScript including mobile platforms. This capability ensures that the solution is easy to integrate across a wide range of applications, leaving more freedom to developers of future standards.

Similarly, since the requirements of SSI do not allow server side code execution (to avoid third parties), the libraries chosen for processing digital signatures were deliberately selected to be independent from Node or or similar environments. Accordingly, we employed `asn1js` [26] and `pkijs` [27] for parsing digitally signed files and certificates. Additionally, the `@zk-mail/helpers` library [28] was utilized to provide utility functions for preparing input data for the ZKP circuit which, as already stated in the Blockchain and Cryptographic Tools subsection, was written in Circom language. The testing phase was executed using Power Shell scripts and the Jest framework inside a Node environment. Jest was used not only for testing but also for its automatic reporting capabilities, thereby automating the collection of test execution times. It is worth noting that, while all core components of ZK-KYC-DSIG are platform independent, the testing phase specifically relies on Power Shell scripts, which are limited to Windows systems.

Last but not least, the smart contracts were implemented in Solidity, which is currently

the most popular language for blockchain development. Furthermore, it takes significant inspiration from JavaScript, making it easier for developers to become proficient with the language [29, 30]. The OpenZeppelin security library was included to mitigate well known reentrancy vulnerabilities and enhance overall security of the smart contracts.

3 | State of the art

Numerous solutions exist for authentication. Notably, as briefly noted in the Preliminary Background chapter, there are several established federated identity solutions. Notably, many leverage the widely known OpenID based authentication combined with the OAuth authorization model. Additionally, many governments have implemented their systems for identity management. Therefore, as highlighted in the Introduction chapter, these centralized systems represent prime targets for data breaches. Their appeal and vulnerability make them susceptible to attacks, which can lead to the compromise of sensitive personal data and hence identity theft. Consequently, the research community has rapidly embraced the SSI concept as a solution to these concepts, leading to the development of numerous solutions. The next logical step could hardly be anything other than the adoption of blockchains. In the Existing solutions and analysis section, a group of solutions proposed in the literature is presented and analyzed.

3.1. Existing solutions and analysis

The authors of [25] gathered and investigated a comprehensive set of proposed solutions for on chain identity management, analyzing their key features. Their investigation addressed six central questions, shedding light on the landscape of blockchain based identity systems:

- **Adopted DLTs**: The analysis revealed that Ethereum is, by far, the most commonly adopted blockchain technology for these solutions, underscoring its dominance in the field.
- **Prevalent Concepts**: The most frequently utilized concepts alongside blockchain technology are ZKPs, PKI and trust propagation, which form the backbone of many identity systems.
- **Key Entities**: The solutions typically involve several entities: a user, an issuer of credentials, a verifier, and in some cases, a trust anchor. Additionally, certain systems include a legal authority with privileged access to the user real world identity.
- **On and Off chain**: The analyzed solutions leverage blockchain in varied ways.

Some perform verification directly on chain, while others integrate additional operations, highlighting different architectural approaches.

- **Implementation Status and Maturity**: Most solutions have progressed beyond theoretical proposals, with at least a proof of concept implementation, demonstrating practical applicability.
- **Bridging Real World Data**: The study examined how real world data is attested on chain, identifying a spectrum of approaches. Some solutions overlook this issue entirely, others introduce a trust anchor without detailing its implementation and a few rely on governmental digital IDs as a root of trust. The authors note that this final aspect remains an unresolved challenge in the field.

In [31], three operational SSI solutions are examined: Sovrin, uPort, and Jolocom. These approaches adopt DIDs and VCs. The firsts are unique and user controlled identifiers, usually registered on a blockchain. Instead, the latter are cryptographically signed assertions of identity data. Each DID is linked to a DID document for resolution, containing the associated public key, additional public credentials and network addresses essential for trusted interactions with the identity owner who controls the document.

- Sovrin: It operates on the Hyperledger Indy public permissioned blockchain and stores public DIDs, credential schema, definitions and revocation registries directly on chain. Conversely, VCs are exchanged off chain through encrypted private channels between edge agents, such as mobile devices.
- uPort: Built on the Ethereum public permissionless blockchain, it registers DIDs in an on chain uPort Registry. VCs are managed within a mobile app and shared off chain as signed JWTs, with DID documents retrieved from IPFS for verification.
- Jolocom: Also based on Ethereum, it records DIDs on chain and stores DID documents on IPFS by default. VCs are managed locally in a wallet, with support for child DIDs to conceal that multiple credentials belong to the same individual, enhancing privacy.

The primary focus of the paper was to analyze GDPR compliance of those blockchain based SSI systems. Privacy considerations reveal significant GDPR tensions, notably the immutability of on chain DIDs and hashes, which conflicts with the right to erasure and the ambiguity in identifying data controllers for on chain data, requiring a case specific analysis.

A distinct approach, the TCID solution, detailed in [32], delivers a robust SSI framework via a peer to peer architecture, integrating network level anonymization through multi

hop channels and pseudonym based credential management. This design ensures self sovereignty, credibility and anonymity without depending on centralized servers. Despite its innovative approach, the deep commitment on SSI principles limits interoperability with established legal frameworks, existing infrastructures and standardized trust anchors, hindering widespread adoption.

The protocol proposed in [33] integrates DIDs and VCs with ZKP (SNARKs) through ZoKrates toolbox. Users obtain a VC from a registry office, encrypt their fiscal code using a symmetric key shared with a judicial authority via an elliptic curve protocol and submit a ZKP to an Ethereum smart contract to activate their address without revealing identity details. A key feature is its design for KYC and AML compliance, allowing a judicial authority to unveil identities under specific conditions. However, a notable disadvantage is its reliance on a custom VC issuer (registry office), requiring development rather than leveraging existing infrastructures such as PKI.

A more recent study [34] presented a comparable approach utilizing existing PKI and CMS/PKCS#7 digital signatures within a zkVM framework for ZKP. The presented use case focuses on government tax refunds. A notable feature is the adoption of eIDAS compliant digital signature standards. The solution leverages RISC Zero zkVM to process an off chain signed document containing the user tax ID and blockchain address, producing a proof of valid signature verification in approximately one minute. This proof is subsequently verified on chain by a smart contract, facilitating secure fund transfers without revealing sensitive data. While promising and with good performance, the approach ensures eIDAS compliance only for digital signatures, not addressing broader eIDAS 2 requirements. Additionally, it delegates proof generation to the Bonsai service, partially undermining independence from third parties.

The solution that will be proposed in this research leverages DLTs, ZKPs, PKI, CMS digital signature and tries to overcome the issues of existing solutions with particular care in remaining compliant with the EU regulations.

4 | Proposed Solution

In this chapter, the ZK-KYC-DSIG framework architecture is introduced and its core components and functionalities are presented.

4.1. Assumptions

In order to build a working prototype, some key assumptions have been made to simplify the development. Most of these simplifications can be easily generalized with additional work. Only a few ones could require careful optimization but further discussion on these optimizations can be found in the "Evaluation and Results" and "Future developments" chapters.

4.1.1. Algorithms

Firstly, it is assumed that the RSA key size is 2048 bits with a fixed exponent of 65537 and that the hashing algorithm SHA256 is adopted. Although some providers still adopt these standards, many are currently implementing 4096 bit keys already. The same assumption has been applied to the RSA keys (more on the reason of their inclusion in the "System Architecture" section of this chapter) of the relying party in order to be able to reuse existing code.

4.1.2. Restricted Geographic Scope

Secondly, the prototype is designed for Italian individuals. Extending the scope to include all EU citizens would be a trivial task.

4.1.3. CA Chain

Thirdly, the certificate chain is currently limited to a root CA which attests end entity leaf certificates. Extending it to support a variable length certificate chain and to incorporate intermediate CAs would require only a few additional coding steps.

4.1.4. Signature type and settings

The ZK-KYC-DSIG expects attached digital signatures, hence the eContent field must be included within the signature. Analogously, the CAdES-BES standard, which is eIDAS and eIDAS 2 compliant, has been adopted for digital signatures. This standard expects that signed attributes field is always present even if according to the broader CMS standard it is optional [11, 12]. Given the widespread adoption of the CAdES-BES standard in both the public and private sectors, one could argue that this is not a simplifying assumption but an inevitable choice. Anyway, generalization to other standards could be feasible with more development.

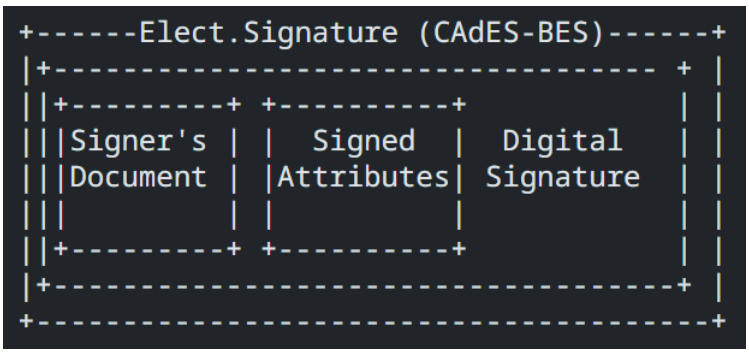

Figure 4.1: CMS CAdES-BES digital signature.

4.1.5. Certificate Revocation Lists and Expiration Date

The current version of the prototype doesn't handle either the CRL or the notBefore and notAfter fields (validity period) of the signature. The considerations behind this decision are discussed in a later chapter.

4.1.6. File type

The framework requires the usage of a plain text file (e.g. `.txt`) for the digital signature. This requirement stems from the fact that the signature's signed attributes include a message digest (hash) of the file content to ensure data integrity. If the file is altered in any way, the hash changes, thereby invalidating the signature. Non plain formats such as PDF incorporate complex structures (e.g. embedded metadata, forms, incremental updates) and may apply compression algorithms (e.g. deflate). These characteristics make data manipulation and hash integrity checks exceedingly difficult within a ZKP circuit. At present, no readily available Circom gadgets can efficiently handle such intricate formats, and even a custom solution would incur significant computational overhead inside the

ZKP circuit. As a result, the system only supports straightforward plain text files.

4.1.7. Legal Authority Public Key

The system relies on a Legal Authority's public key. The current prototype simply expects that key to be provided in isolation. In a production environment, a complete certificate chain would be necessary to establish trust in the key. However, the project already incorporates certificate parsing capabilities, making any extension to handle full certificates relatively straightforward. Furthermore, the precise role and responsibilities of the Legal Authority lie outside the scope of this thesis, thus, working solely with the public key is sufficient for demonstration purposes.

4.2. System Architecture

Up to this point, a complete description of the framework structure has not been provided. In this section, a detailed overview of the architecture is presented. Before delving into the technical details, the involved actors are clarified. The ZK-KYC-DSIG framework introduces an additional actor compared to the classical SSI model depicted in figure 1. Nevertheless, the core SSI principles remain intact. The updated model is shown in figure 4.2.

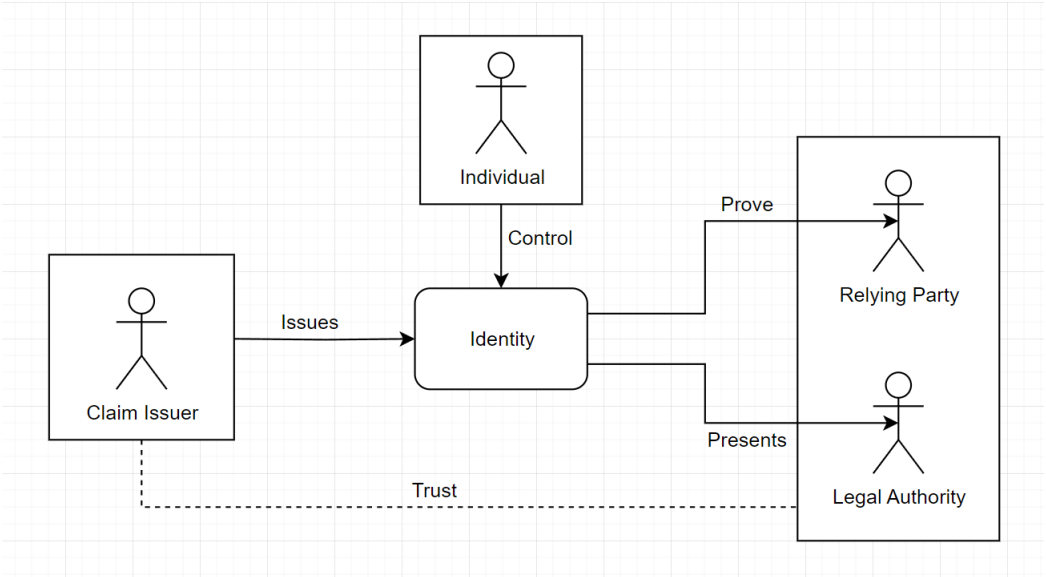

Figure 4.2: ZK-KYC-DSIG actors.

The system architecture is composed of three primary components.

- **Pre-Processing Layer**: This layer provides all the functionality for reading and

extracting information from the digitally signed file (P7M/CMS package) and the cited certificates (.cer/.pem format). It parses the signature, verifies its basic structure, checks the attached certificates and converts the parsed data (such as the signer's public key, hashed content, and ASN.1-encoded fields) into a format suitable for the ZKP circuit. The output of this phase is a structured dataset in JSON format, which serves as the witness input for proof generation.

- **Proof Generation Circuit**: This component is responsible for applying the ZKP logic on the data prepared by the pre-processing layer. Using a Circom based circuit, it enforces the cryptographic constraints that validate the signature and identity attributes without exposing any sensitive information. Once all constraints are satisfied, the circuit outputs a proof attesting that the holder of the signed file meets the identity verification requirements, while preserving his privacy. The only input signals that must remain publicly accessible are the public key of the legal authority and the public key of the CA. This ensures the establishment of a verifiable trust relationship, as illustrated in Figure 4.2, allowing third parties to authenticate the legitimacy of issued claims without compromising the privacy of the user.
- **Proof Verification Contract**: The final step involves on-chain verification of the proof by a specialized smart contract, automatically generated by `snarkjs` toolkit. This Solidity contract, deployed on an EVM-compatible blockchain, checks the proof to confirm its validity. If the proof is correct, the contract signals a successful identity verification on-chain, otherwise, the verification fails. The verification contract is incorporated into a demonstration contract called ZKIdentityVault, which serves to validate the proof verification mechanism through a representative test case.

4.2.1. System inputs

The system requires the following inputs:

- **Digitally Signed File**: A plain text file, containing encrypted data, signed with the CAdES-BES format from the user.
- **CA Certificate**: The certificate of the CA that issued the digital signature of the user.
- **Legal Authority Public Key**: The public key of the authority responsible for overseeing KYC data in legal proceedings.
- **Content of the Signed File**: A cipher text of a data structure containing the taxID, encrypted with the public key of the legal authority. A random salt must

be incorporated in the data structure to ensure that repeated encryptions of the same data produce different cipher texts, thereby avoiding known vulnerabilities. Typically, this is addressed by using non deterministic padding, but the circuit mandates only deterministic operations.

In practice, only the digitally signed file would need to be supplied by the user, due to the fact that in a production environment the certificates could be automatically retrieved or included by the platform adopting the ZK-KYC-DSIG framework. The system aims to provide proof that the user possesses a valid tax ID. To achieve this, ZK-KYC-DSIG leverages the non repudiation property of digital signatures, which are certified by a CA. A digitally signed file inherently includes the signer certificate which still needs to confirm its validity against the CA certificate (as discussed in the Preliminary Background chapter). Consequently, both a digitally signed file and the CA certificate are required. Furthermore, a legal authority must be able to access the data for legal proceedings if needed. To ensure that only authorized entities can decrypt the relevant information, it is encrypted using the public key of that authority.

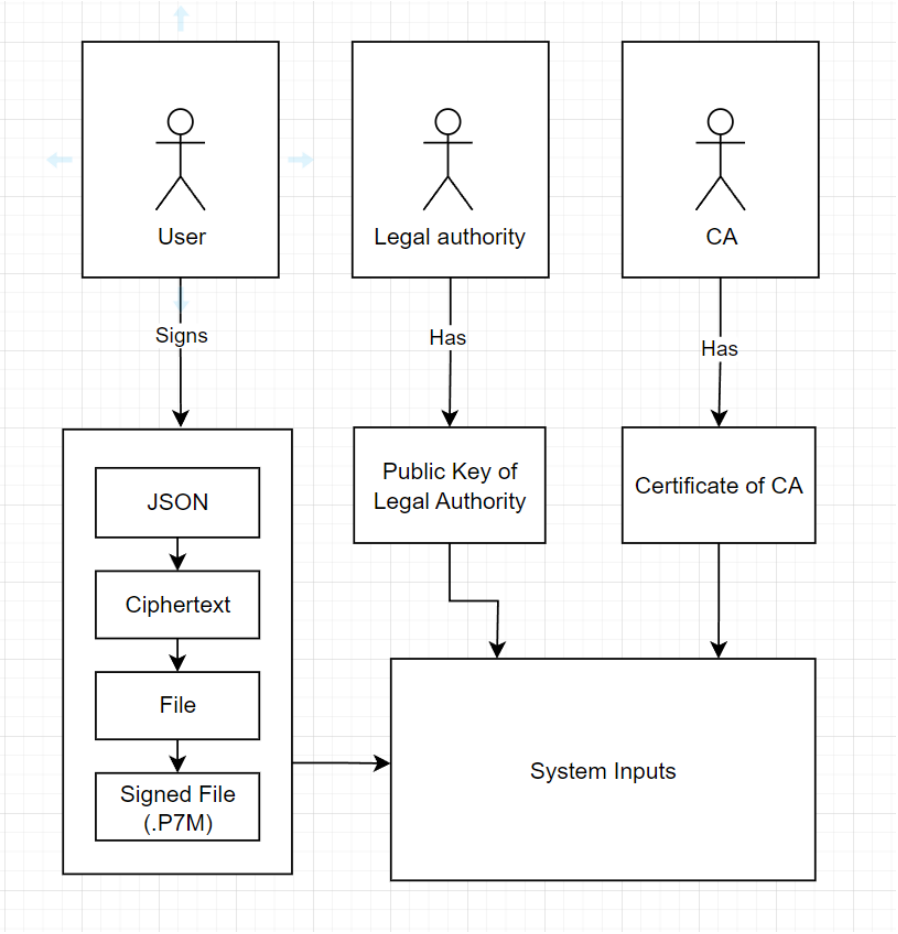

Figure 4.3: ZK-KYC-DSIG inputs.

4.2.2. ZKP circuit

The core of ZK-KYC-DSIG is certainly the ZKP circuit. In order to establish that the user possesses a specific taxID the flow of the proof follow this process:

- Verify the signature of the digitally signed file (i.e. Padded Hash of Signed Attributes = Digital Signature decrypted with the public key of the user).
- Verify the signature of the user's certificate (i.e. Padded Hash of Tbs = Certificate Signature decrypted with the public key of the CA).
- Verify the message digest (i.e. Message Digest = Hash of the file content).
- Verify the content. (i.e. Content = Data Structure with personal data encrypted with the Legal Authority Public Key).
- Verify the user taxID contained in the content with the one contained in the Tbs of the certificate.

4.2.3. Smart contracts

The verifier smart contract was generated automatically using SnarkJS built in functionality, which produces the necessary Solidity code for on-chain proof verification. To simulate a practical use case scenario, a rudimentary Vault smart contract was implemented: ZKIdentityVault. This Vault contract allows users to deposit funds and subsequently withdraw them only after a successful ZKP verification via the verifier contract. Although this implementation is solely for testing purposes rather than deployment, it provides a robust environment for evaluating the framework proof verification and integration with financial operations.

4.2.4. Trusting the CA

The current proof only validates that the digitally signed file contains a specific content and that the user file signature chains back to a CA certificate. However, any entity can generate a custom CA certificate (as demonstrated in the Implementation chapter), so the smart contract must enforce trust by accepting only approved CA certificates. A straightforward method involves maintaining an on chain list of trusted CA public keys and verifying that the CA key provided by the prover matches one of the entries on this list. Although this mechanism is not yet implemented in the prototype, adding it would be straightforward from a technical perspective. However, the primary challenge lies in designing a suitable mechanism and designating clear responsibilities for maintaining the

trusted CA list in a cost-effective and secure manner.

4.2.5. Storage of cipher text

An important technical and legal challenge concerns how the cipher text containing the user personal data is stored so that the designated legal authority can still access it when required. In the current ZK-KYC-DSIG prototype, this cipher text is placed on chain as a public input to the ZKP circuit. From a cryptographic standpoint, the data is protected because it is encrypted under the public key of the legal authority. Hence, access is infeasible for unauthorized parties, assuming the authority's key remains securely managed.

However, despite the encryption, GDPR and related data protection regulations may still view this cipher text as personal data. In particular, the "Right to Erasure" / "Right to Be Forgotten" poses a significant issue for immutable ledgers: once data is recorded on chain, it cannot be simply deleted or altered. Theoretically, one might render the cipher text permanently unusable by destroying the corresponding decryption key (i.e. the legal authority's private key). Yet this is generally impractical in normal operational scenarios and would not address user specific erasure requests on a case by case basis. Consequently, a conflict may arise if the cipher text is retained indefinitely on the blockchain. To satisfy regulatory requirements and avoid potential liabilities, a permanent legal justification is typically needed. Anyway a case by case analysis may be necessary from a legal stand point [31].

In the Future developments chapter, we discuss alternative architectures that could mitigate these concerns. For instance, a straightforward option is to avoid placing the user encrypted personal data on an immutable blockchain and instead, store only a pointer to the cipher text, which resides off chain in a mutable data store. Another approach is to limit how long a particular public key (of the authority) is valid, thereby balancing legal requirements with data minimization. Ultimately, consultation and collaboration with the relevant legal authority is crucial, since that authority will be responsible for key custody and probably also for management of the architecture holding the personal data of users.

4.3. Software as a product regime

As previously mentioned in the Preliminary Background chapter, the eIDAS2 regulation leaves the design of technical and legal frameworks to EU member states. Due to this

fact, multiple regimes may emerge, fragmenting the digital market unnecessarily. The author of [19] analyzed two solutions from a legal standpoint: trust service and software product regimes. The conclusion reached is that the software as a product model aligns better with ZKP adoption within existing regulations. "Hence, in a pure product scenario of ZKP generation and validation, all aspects, including liability, cybersecurity, software quality, privacy, and market surveillance, are effectively addressed."

Moreover, the software as a product regime remains completely in line with the SSI principles. The user would need to download a digital product (e.g. a mobile app) and manage autonomously its personal data. By contrast, a service based approach would require interaction with a third party server for every operation, contradicting the user centric data management philosophy of SSI. This rationale reinforces the choice of SnarkJS with Groth16 in this project, as it allows immediate switching to Rapidsnark, which can be easily adopted on mobile devices.

5 | Implementation

This chapter provides a detailed technical description of the implementation of the three core components of ZK-KYC-DSIG. The full code of the prototype can be found publicly available in this GitHub repository [35].

5.1. Folder structure

As already explained, the project has been developed as a Node project, even though the framework is designed to operate independently of it. This design choice facilitates the use of automated testing tools such as the Jest testing library. Figure 5.1 shows the folder structure of the project.

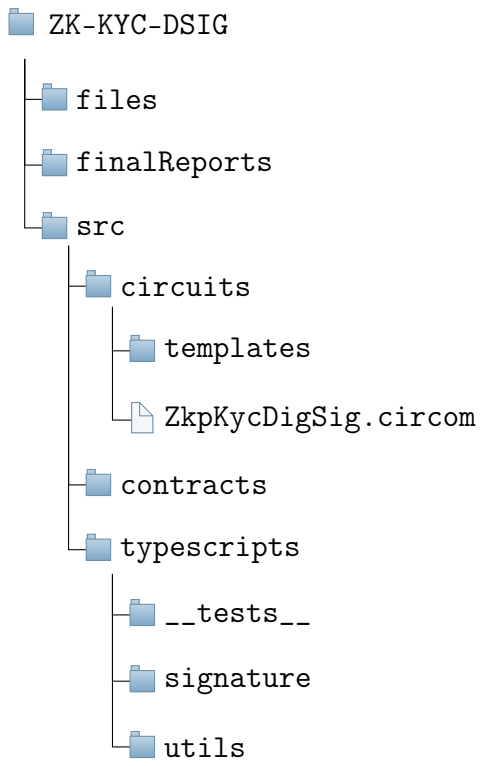

Figure 5.1: Folder structure of ZK-KYC-DSIG.

The project root directory contains two Power Shell scripts and the configuration files for Node and the Jest testing library. The files folder holds input files used by the Pre Processing layer, while the finalReports folder stores all reports generated by tests. Results of these tests are presented in the Evaluation and Results chapter.

The src folder comprises the entire source code of the prototype and is organized into three subfolders: typescript, circuits, and contracts. The typescript folder includes the Pre Processing layer code, together with test scripts and various utility modules used during development and testing. The circuits folder contains the main ZKP circuit along with all subsidiary circuits it invokes. Finally, the contracts folder includes the automatically generated verifier smart contract and a demo smart contract implementing a simple Vault service for testing the verifier functionality.

5.2. Pre Processing Layer

The pre processing layer extracts information from input files and prepares data for the ZKP Circom circuit. This avoids unnecessary operations inside the circuit, thereby improving proof generation performance. As the first layer of ZK-KYC-DSIG, it processes the raw inputs. Circom accepts only integers and arrays of integers as input signals. Hence, the required format conversions are performed at this phase. The framework requires the legal authority PKI certificate, the digitally signed file (which contains the ciphertext of a data structure of user personal data encrypted with the legal authority public key) and the CA public key certificate.

5.2.1. Parsing PKI files

The code to extract and prepare significant data from the signature is shown in this subsection. Not all of the code is included, as similar operations are performed for each case. As anticipated in the Methods and Materials chapter, the parsing class extensively utilizes the two libraries [26, 27], while the formatter class makes use of the [28] library.

```
private extractContentInfo() {
  this.ContentInfo = new pkiLib.ContentInfo({ schema: this.FileAsn1Format });
  if (!this.ContentInfo) {
    throw new Error("Failed to decode ContentInfo structure");
  }
}
```

Code 5.1: Signature Parsing: extracting ContentInfo.

```
private extractSignedData() {
  if (this.ContentInfo.contentType !== "1.2.840.113549.1.7.2") {
    throw new Error("Not a valid SignedData structure");
  }
  this.SignedData = new pkiLib.SignedData({ schema: this.ContentInfo.content });
  if (!this.SignedData) {
    throw new Error("Failed to decode SignedData structure");
  }
}
```

Code 5.2: Signature Parsing: extracting SignedData from ContentInfo.

```
private extractSignerInfos() {
  this.SignerInfos = [];
  if (!this.SignedData.signerInfos || this.SignedData.signerInfos.length === 0) {
    throw new Error("Failed to extract SignerInfos from SignedData");
  }
  for (const signerInfo of this.SignedData.signerInfos) {
    this.SignerInfos.push(new pkiLib.SignerInfo(signerInfo));
  }
}
```

Code 5.3: Signature Parsing: extracting SignerInfos from SignedData.

```
private extractSignatureAndSignedAttributes() {
  this.Signatures = [];
  this.SignedAttributes = [];
  for (const signerInfo of this.SignerInfos) {
    if (!signerInfo.signature) {
      throw new Error(
        "Failed to extract Signature from SignerInfo with index: " + this.SignerInfos.
            indexOf(signerInfo) + "."
      );
    }
    if (!signerInfo.signedAttrs) {
      throw new Error(
        "Failed to extract SignedAttributes from SignerInfo with index: " + this.
            SignerInfos.indexOf(signerInfo) + "."
      );
    }
    this.Signatures.push(signerInfo.signature);
    this.SignedAttributes.push(new pkiLib.SignedAndUnsignedAttributes(signerInfo.
        signedAttrs));
  }
}
```

Code 5.4: Signature Parsing: extracting Signature and SignedAttributes from SignerInfos.

```
const signature: string[] = toCircomBigIntBytes(this.Data.Signature);
const [signedAttributesPadded, signedAttributesPaddedLength] = sha256Pad(this.Data.
    SignedAttributes, this.MaxSignAttributesLength);
const signedAttributesPaddedString: string[] = Uint8ArrayToCharArray(
    signedAttributesPadded);
```

Code 5.5: Input formatting: converting Signature and SignedAttributes for the circuit.

5.3. Proof Generation Circuit

The ZKP Circom circuit invokes several sub circuits for different operations. In this section, its key characteristics are uncovered.

5.3.1. Circuit Parameters

Circom does not support dynamically sized arrays. Instead, a fixed maximum length must be defined for each input signal and the array size remains static. Consequently, parameters are used to specify the maximum length of arrays to accommodate various signal sizes. Additionally, longer signals must be split into multiple fixed size chunks because Circom only supports integer signals. The circuit has four parameters:

- **maxSignedAttributesLength**: Maximum length of the SignedAttributes field.
- **maxCertificateTbsLength**: Maximum length of the certificate TBS field.
- **chunksBitLength**: Bit length of each chunk for long signals.
- **totalChunksNumber**: Total number of chunks available for representing a large signal.

```
component main {public [CaPublicKeyModulus, JudgePublicKeyModulus, Content]} = ZkpKycDigSig
    (512,2048,121,17);
```

Code 5.6: Circuit instantiation.

The values 121 and 17 are chosen because their product slightly exceeds the 2048 bit RSA key length. Although 2048 bits would theoretically be sufficient, the extra bits provide a buffer to account for any operational overhead. This ensures that the entire 2048 bit signature is fully captured when it is split into chunks, thereby preventing any loss of data during proof generation.

5.3.2. Circuit code

In this subsection, the main circuit code is presented. It outlines the execution flow and the invocation of sub circuits to verify the identity embedded in the signature. Detailed implementations of the sub circuits are provided in the Appendix. The circuit inputs match those listed in the Proposed Solution chapter, supplemented by dynamic length parameters for certain arrays and indices for specific patterns within the digital signature (e.g. particular OIDs). These supplementary parameters prevent the need to parse the digital signature from scratch within Circom. As a result, the verification process is streamlined and computational overhead is reduced.

```
template ZkpKycDigSig(maxSignedAttributesLength, maxCertificateTbsLength, chunksBitLength
    , totalChunksNumber) {

    signal input SignedAttributes[maxSignedAttributesLength];
    signal input SignedAttributesLength;
    signal input Signature[totalChunksNumber];
    signal input PublicKeyModulus[totalChunksNumber];

    signal input CertificateTbs[maxCertificateTbsLength];
    signal input CertificateTbsLength;
    signal input CertificateSignature[totalChunksNumber];
    signal input CaPublicKeyModulus[totalChunksNumber];

    signal input JudgePublicKeyModulus[totalChunksNumber];

    //Index necessary for extracting the message digest from the signed attributes
    signal input MessageDigestPatternStartingIndex;
    var maxMessageDigestLength = 32;

    var maxFiscalCodeLength = 16;

    //The content must be as long as the key so 2048 bits = 256 bytes
    var maxDecryptedContentLength = 256;
    //Value 344 is mandatory since it's the base64 length of an rsa 2048 signature
    var maxContentLength = 344;
    signal input Content[maxContentLength];
    signal input DecryptedContent[maxDecryptedContentLength];
    signal input DecryptedContentLength;
    signal input FiscalCodeIndexInDecryptedContent;

    //Indexes necessary for extracting the fiscal code and the public key from the
        certificate tbs
    signal input FiscalCodePatternStartingIndexInTbs;
    signal input PublicKeyModulusPatternStartingIndexInTbs;
```

Code 5.7: Main circuit header and inputs.

```
//1-Verify the SIGNATURE of the signed attributes
//2-Verify the SIGNATURE of the certificate
component signatureVerify = FormatterAndSignatureVerifier(maxSignedAttributesLength
    ,2048, chunksBitLength, totalChunksNumber);
component certificateSignatureVerify= FormatterAndSignatureVerifier(
    maxCertificateTbsLength,2048, chunksBitLength, totalChunksNumber);

signatureVerify.data <== SignedAttributes;
signatureVerify.dataLength <== SignedAttributesLength;
signatureVerify.signature <== Signature;
signatureVerify.publicKey <== PublicKeyModulus;

certificateSignatureVerify.data <== CertificateTbs;
certificateSignatureVerify.dataLength <== CertificateTbsLength;
certificateSignatureVerify.signature <== CertificateSignature;
certificateSignatureVerify.publicKey <== CaPublicKeyModulus;
```

Code 5.8: Main circuit implementation: Verify signatures.

```
//3-Extract the MESSAGE DIGEST from the signed attributes
component messageDigestExtractor = ExtractMessageDigestFromSignedAttributes(
    maxSignedAttributesLength, maxMessageDigestLength);
messageDigestExtractor.SignedAttributes <== SignedAttributes;
messageDigestExtractor.SignedAttributesLength <== SignedAttributesLength;
messageDigestExtractor.MessageDigestPatternStartingIndex <==
    MessageDigestPatternStartingIndex;
signal MessageDigest[maxMessageDigestLength] <== messageDigestExtractor.MessageDigest
    ;

//Verify it corresponds to the hash of the content
component verifyHash = VerifyHash(maxContentLength);
verifyHash.bytes <== Content;
verifyHash.expectedSha <== MessageDigest;
```

Code 5.9: Main circuit implementation: Verify message integrity through message digest.

```
//4-Verify that the CONTENT is a cipher encrypted with the judgle public key and
    which plain text contains the fiscal code
component verifyRsa = VerifySimpleRsaEncryptionBase64AndExtractSubstring(
    maxDecryptedContentLength, maxContentLength, maxFiscalCodeLength, chunksBitLength
    , totalChunksNumber);
verifyRsa.SignatureBase64 <== Content;
verifyRsa.PublicKeyModulus <== JudgePublicKeyModulus;
verifyRsa.Message <== DecryptedContent;
verifyRsa.IndexOfPartialMessage <== FiscalCodeIndexInDecryptedContent;
verifyRsa.MessageLength <== DecryptedContentLength;
signal FiscalCode[maxFiscalCodeLength] <== verifyRsa.Substring;
```

Code 5.10: Main circuit implementation: Verify content structure and encryption with the Legal Authority public key.

```
    //5- Verify that the FISCAL CODE and the PUBLIC KEY corresponds with the ones
        contained in the CERTIFICATE TBS
    component verifyFiscalCodeAndPubKey = VerifyFiscalCodeAndPubkeyFromCertTbs(
        maxCertificateTbsLength, maxFiscalCodeLength, chunksBitLength, totalChunksNumber);
    verifyFiscalCodeAndPubKey.CertificateTbs <== CertificateTbs;
    verifyFiscalCodeAndPubKey.CertificateTbsLength <== CertificateTbsLength;
    verifyFiscalCodeAndPubKey.FiscalCode <== FiscalCode;
    verifyFiscalCodeAndPubKey.PublicKeyModulus <== PublicKeyModulus;
    verifyFiscalCodeAndPubKey.FiscalCodePatternStartingIndex <==
        FiscalCodePatternStartingIndexInTbs;
    verifyFiscalCodeAndPubKey.PublicKeyModulusPatternStartingIndex <==
        PublicKeyModulusPatternStartingIndexInTbs;
}
```

Code 5.11: Main circuit implementation: Verify identity through taxID and data contained within the certificate.

5.4. Proof Verification Contract

The Verifier smart contract was generated automatically by the SnarkJS library using the following command:

```
snarkjs zkey export solidityverifier <CircuitKeyPath> <VerifierContractPath>
```

Command 5.1: Generation of the Solidity Verifier smart contract.

SnarkJS also provides an utility command to generate the input for the Verifier smart contract:

```
snarkjs generatecall
```

Command 5.2: Generation of the input for the Verifier smart contract.

The generated verifier contract is integrated into a demo vault DAPP called ZKIdentity-Vault. Security is ensured using the OpenZeppelin library. The smart contract implements three main functions:

- **deposit**: Users can deposit ETH into the contract at anytime.
- **proveIdentity**: Users submit a ZKP to prove their identity.
- **withdraw**: Verified users can withdraw a specified amount of deposited ETH.

The implementation of the three function is given below:

```
function deposit() external payable whenNotPaused nonReentrant {
    require(msg.value > 0, "No ETH sent");
    balances[msg.sender] += msg.value;
    emit Deposited(msg.sender, msg.value);
}
```

Code 5.12: Deposit function of ZKIdentityVault smart contract.

```
function proveIdentity(
    uint256[2] calldata _pA,
    uint256[2][2] calldata _pB,
    uint256[2] calldata _pC,
    uint256[378] calldata _pubSignals
) external whenNotPaused {
    bool isValid = verifier.verifyProof(_pA, _pB, _pC, _pubSignals);
    require(isValid, "Invalid proof");
    verified[msg.sender] = true;
    emit IdentityProven(msg.sender);
}
```

Code 5.13: ProveIdentity function of ZKIdentityVault smart contract..

```
function withdraw(uint256 amount) external whenNotPaused nonReentrant {
    require(verified[msg.sender], "Not verified");
    require(amount > 0, "Must withdraw > 0");
    require(balances[msg.sender] >= amount, "Insufficient balance");
    balances[msg.sender] -= amount;
    (bool success, ) = msg.sender.call{value: amount}("");
    require(success, "ETH transfer failed");
    emit Withdrawn(msg.sender, amount);
}
```

Code 5.14: Withdraw function of ZKIdentityVault smart contract.

As already anticipated in the Proposed Solution chapter, the CA whitelist is still not included in the prototype at this step of development. Additionally, an isVerified function is provided to determine whether a user is correctly verified. The current version is not included here since allows unrestricted access to address verification information which is not desirable. Only accredited services should be permitted to perform these checks maybe by creating a list of trusted services. One straightforward solution is to tie identity verification to either a specific trusted service or a predefined list of approved services, giving users the power to choose their preferred option.

5.5. Testing

The Jest based testing process extensively leverages two Power Shell scripts. They have a straightforward and simple design. In fact, the tests only verify that the invoked script returns an "OK" status, thus their code is not included in this document.

5.5.1. Scripts for Test Automation

The GenerateCertsAndSignFile.ps1 script is responsible for crafting certificates and signing files with the generated certificates to enable extensive testing. It leverages the OpenSSL library. Conversely, the GenerateProofAndVerifyIt.ps1 script performs the Circom circuit compilation, orchestrates the complete ceremony, creates the verifier smart contract, generates proofs and verifies them off chain. The complete source code for the scripts is available in Appendix.

5.5.2. Testing commands

There are four commands to run the prototype. The first, which could require hours, handles the entire workflow from circuit compilation through proof generation and verification. It must be run the first time, since it generates the proving and verification keys. Another command generates and signs certificates for the prototype. Then, the command running the first practical test uses real data to carry out a prove-and-verify process. Analogously, another command attempts the same procedure using the newly crafted certificates.

```
test:full
```

Command 5.3: Full test: compile, ceremony, generated solidity smart contract, prove and verify.

```
test:create
```

Command 5.4: Craft certificates and sign file.

```
test:one
```

Command 5.5: Practical test one (real data): prove and verify.

```
test:two
```

Command 5.6: Practical test two (crafted data): prove and verify.

5.6. Use Cases

The ZK-KYC-DSIG framework can be tailored to a wide range of applications requiring secure, privacy preserving identity verification. Below are a few illustrative scenarios:

- **Financial Services On Chain**: The Vault demo is a basic example of a smart contract service needing KYC. The same approach can be extended to DEFI protocols, exchanges or earn platforms that must verify users for legal or regulatory compliance.
- **Off Chain KYC Checks**: Through the isVerified function, any off chain service can verify the user status without revealing or acquiring sensitive data. This could include traditional banking, insurance providers or any other party wanting to confirm an address or identity attribute securely. The system design also enables authorities to trace actors in financial operations, ensuring compliance where required.
- **E-Commerce for Digital Services**: An online marketplace could integrate the framework to check a buyer identity without storing personal details. Users would simply prove they are verified, then purchase goods or services.
- **Government Services**: Public administration offices could adopt the framework. This would reduce the need to handle personal information directly on government platforms. Obviously, this would only apply to services where data information disclosure is not mandatory for the type of service.

In essence, any system needing reliable KYC or identity checks, either on chain or off chain, can integrate the ZK-KYC-DSIG framework.

6 | Evaluation and Results

In this chapter, we present the experimental results obtained from tests performed on ZK-KYC-DSIG. The experiments were conducted on two different hardware setups, as described in the Hardware Environment for Development and Testing subsection. For each category of experiments an evaluation of the results is presented.

6.1. Time Performance

6.1.1. Results

Table 6.1 reports the time required for proof generation. The experiments include independent runs using both real data obtained from a digitally signed document and synthetically crafted data. For the real data, the signature was issued by an individual certified by the Italian provider Aruba Spa. These initial experiments provide valuable insights into the computational requirements of our implementation.

Proof Generation	**Older Machine (s)**	**Newer Machine (s)**
Real Data 0	78.107	39.564
Real Data 1	124.057	45.173
Real Data 2	109.893	41.044
Crafted Data 1	102.806	40.369
Crafted Data 2	115.517	40.690

Table 6.1: Proof generation performance.

Table 6.2 shows the time required for proof verification. The verification times are comparable across both machines, with only a slight improvement on the faster one.

Proof Verification	Older Machine (s)	Newer Machine (s)
Real Data 0	1.986	1.509
Real Data 1	2.071	1.463
Real Data 2	1.744	2.084
Crafted Data 1	1.715	2.050
Crafted Data 2	1.861	1.717

Table 6.2: Proof verification performance.

6.1.2. Discussion

The experimental results indicate that proof generation time is significantly affected by the computational capabilities of the hardware. On the older machine, proof generation times range from approximately 78 to 124 seconds, whereas on the newer machine they are consistently around 40 seconds. This difference reflects the higher processing power, particularly in terms of CPU performance and memory speed, of the newer system. In contrast, proof verification times remain nearly constant across both machines, with values between 1.5 and 2.1 seconds. This outcome is in line with theoretical expectations for SNARK schemes such as Groth16, where verification is designed to run in constant time (O(1)) regardless of circuit complexity [20]. Such efficiency is crucial for on chain applications where inexpensive and rapid verification is required. The succinctness of SNARK proofs enables extremely fast verification. This trade off is acceptable in many practical scenarios, especially when proofs are generated infrequently but verified multiple times—for example, in digital identity verification frameworks.

Overall, the results are in line with the proposed goals and the designed tests. The verification time is completely acceptable and the proof generation time is quite good even on older computers. Furthermore, the adoption of Rapidsnark, which has been previously presented, could significantly improve the proof generation capabilities of ZK-KYC-DSIG.

6.2. Gas Fees Costs

6.2.1. Results

As discussed in the Storage of cipher text subsection, a version of ZK-KYC-DSIG which do not share the cipher text containing the personal data on chain is advisable. Therefore, two versions of the protocol are defined:

- **Large Input** version: the input includes the cipher text containing the user's en-

crypted personal data.

- **Small Input** version: The cipher text is omitted from the input.

The gas costs were evaluated using the Remix IDE configured with the Cancun EVM simulation, which emulates Ethereum blockchain conditions. In order to understand the results, recall that the Gas Limit represents the maximum gas allocated for a transaction, the Transaction Cost denotes the total gas consumed by the transaction and the Execution Cost is the gas consumed solely by the smart contract execution. For this analysis, the Transaction Cost is the most relevant metric.

For the Large Input version, the following results were obtained:

- **Gas Limit**: 3,236,468 gas
- **Transaction Cost**: 2,814,320 gas
- **Execution Cost**: 2,730,420 gas

For the Small Input version, instead, the results were as follows:

- **Gas Limit**: 548,822 gas
- **Transaction Cost**: 477,236 gas
- **Execution Cost**: 441,472 gas

It is evident that the Small Input version incurs significantly lower gas costs, confirming that omitting data from the transaction input substantially reduces the overall transaction fee.

6.2.2. Discussion

Based on experiments conducted as of February 20, year 2025, table 6.3 presents the gas fees converted into EUR. Although deploying on Ethereum mainnet would result in prohibitively high costs, the use of Layer 2 EVM compatible chains significantly reduces gas fees.

EVM Blockchain	Cost (Large Input) (EUR)	Cost (Small Input) (EUR)
Ethereum	14.6278	2.4815
Optimism	0.7314	0.1241
Avalanche	0.0659	0.0112
Polygon	0.0211	0.0036

Table 6.3: Gas fee conversion in EUR for the Large Input and Small Input versions of ZK-KYC-DSIG.

The results clearly demonstrate that the *Small Input* version incurs substantially lower gas costs, making it more sustainable, particularly on a Layer 2 chain. Moreover, from both a privacy and compliance standpoint, it is advantageous not to store the cipher text containing the user's personal data on chain. Once more, it is possible to conclude that the objectives have been reached.

6.3. Trusted setup

6.3.1. Results

A typical aspect of ZK-SNARKs is the necessity of performing a trusted setup ceremony. Table 6.4 shows the results from this process. The ceremony encompasses several operations, including active contributions from multiple parties. For simplicity, the reported time includes the full process required to prepare for proof generation, excluding only the Circom compilation step which has been recorded separately.

Phase	Older Machine (s)	Newer Machine (s)
Trusted Setup	90,165.613	17,631.757
Circuit Compile	119.399	62.296

Table 6.4: Time taken for the trusted setup and circuit compilation phases.

6.3.2. Discussion

The power of tau ceremony required over 25 hours on the older machine, while the newer machine completed the process in under 5 hours. It is a significant improvement, though still relatively long. Since the trusted setup is performed only once per ZKP circuit, its impact on routine operations is marginal. Nonetheless, its high computational cost and duration are noteworthy, especially in contexts where circuit updates may be necessary.

Last but not least, security is also at stake, as participants in the ceremony must securely delete their secret random contributions, often referred to as "toxic waste", once the trusted setup is complete. Failure to do so could allow an attacker to recover these parameters, potentially compromising the integrity of the entire ZK-SNARK system. Hence, strict protocols and verifiable procedures must be implemented to ensure that all sensitive data is irrevocably erased after the ceremony, preserving the overall security of the system.

6.4. Proving and Verification Key Sizes

6.4.1. Results

An important aspect of the system is the size of the SNARK circuit keys, which include the proving key for generating proofs and the verification key for on chain validation. Keeping these key sizes within acceptable limits is critical: proof generation must be feasible in a portable environment (in accordance with SSI principles) while on chain verification benefits from smaller verification keys. Table 6.5 reports the key sizes for the two circuits versions already introduced.

Key Type	Large Input Circuit (KB)	Small Input Circuit (KB)
Proving Key	1,271,629.000	1,271,615.000
Verification Key	70.200	12.000

Table 6.5: Proving and verification key sizes for circuits with large and small inputs.

6.4.2. Discussion

The results indicate that the Small Input circuit produce considerably smaller verification keys (12 KB) compared to the Large Input one (70.2 KB), which is advantageous for on chain verification to reduce gas costs and improve efficiency. In contrast, the proving key remains approximately 1.27 GB in both cases. This size is clearly too large for browser based environments. However, it is acceptable for mobile deployments, given that the proving key would only need to be downloaded once. As discussed earlier, the legal model best suited for ZK-KYC-DSIG is "software as a product," which is compatible with mobile applications. For this reason, this result can be considered successful. These results reveal additional incentives for choosing the mobile application model and the Small Input approach.

6.5. Limitations

After testing, a comprehensive inspection for legal compliance issues and technical limitations was performed. This section contains relevant observations originated from this activity.

6.5.1. Exposure of Public Information

As discussed, the public keys of both the Legal Authority and the CA are publicly available by design. This openness enables verifiers to establish a trusted base for proof validation. However, it also allows external observers to potentially infer some information from each proof. For example, if a user possesses a digital signature issued by Aruba Spa and the corresponding proof employs a specific Legal Authority public key, an observer might deduce that the signature likely originates from Italy. Although this information does not directly reveal sensitive personal data, its permanent availability on the blockchain may raise privacy concerns, especially in the event of a data breach at the Legal Authority servers.

6.5.2. eIDAS2 Compliance

The current implementation of the framework adheres to the fundamental principles established by EU regulations, ensuring that identity verification is conducted in a secure, privacy preserving and interoperable manner. Although the prototype satisfies key regulatory requirements, further adaptations are necessary to evolve the system from a prototype to a fully deployable solution. In particular, the evolution toward a comprehensive infrastructure, including the complete mobile application, must be executed in a manner that maintains regulatory compliance at every development stage. This careful approach is essential to achieve full compliance and ensure that the final system meets all legal and technical standards.

7 | Future developments

In the future, the primary focus should be on addressing the current limitations discussed in the Evaluation and Results chapter and then developing new features for the solution.

7.1. Proof Generation Times

Although our proof generation times meet the original objectives and compare favorably with other approaches in the literature, the absolute duration remains a non negligible issue. Since the system is designed for mobile devices, the performance measured on the slowest hardware, approximately 2 minutes, is particularly relevant. Nielsen, in his renowned book [36], advises that operations should ideally complete within 10 seconds and that users should receive timely feedback of what is being executed. While current KYC and identity verification processes may take hours or days due to the need for transmitting and processing document images (or sometimes even manual inspection), reducing operation times closer to the 10 second target would significantly improve user experience. Moreover, Nielsen recommends providing users the option to cancel operations that exceed this threshold. However, these guidelines do not fully account for the capabilities of mobile devices. Therefore, a more practical approach would be to allow proof generation to continue in the background while users may switch to another application. As already explained, although the prototype utilizes SnarkJS, Rapidsnark is the chosen framework for production deployment and it has demonstrated significantly better performance compared to SnarkJS. This improvement should help approaching the 10 second threshold. The only potential directions for further reducing proof generation times are circuit design optimization, hardware enhancement or solver improvement. First, the proposed circuit may be scrutinized for additional refinements. However, since it was developed with performance as a design objective, substantial gains in this area are likely to be limited. Direct hardware improvements are not feasible as users cannot be compelled to upgrade their devices, although general hardware performance is expected to improve over time. Therefore, the primary focus should be on enhancing the efficiency of the proof generation process itself. In particular, improvements in the Circom language

and the solver represent the most viable options. With such optimizations, it should be possible to significantly reduce proof generation times further.

7.2. Gas costs and size of transaction inputs

Gas costs have been evaluated favorably with respect to the initial goals. However, some solutions in the literature, such as [34], achieve slightly lower gas fees. Gas fees are closely related to the size of the transaction input, as showed in the Evaluation and Results chapter. A straightforward optimization is to adopt the Small Input version of the prototype, which excludes the cipher text of the user's personal data from the public input. Instead, only an hash can be compared on chain. This can be implemented by having the Legal Authority provide an API that accepts the cipher text and returns its hash. Then, the hash can be provided on chain via an oracle or a direct call to the smart contract. The hash can be verified with an output of the proof with an additional control mechanism. It is important to note that this approach does not compromise the core principles of SSI. In this model, the Legal Authority does not cover the role a third party responsible for identity validation, rather it functions as an additional relying party mandated by legal requirements to have access to the cipher text. In addition, costs could be substantially reduced by omitting the full CA public key and the Legal Authority's public key from the transaction inputs and instead passing only their respective hashes. As discussed in the Trusting the CA subsection, an on chain whitelist of trusted CAs is already planned, so incorporating only the hash of the public key would require minimal effort. A similar approach can be applied to the Legal Authority's public key.

7.3. Power of tau ceremony and Post Quantum Computers security

ZK-SNARK systems require a trusted setup called the Power of Tau ceremony. During this phase, each participant must contribute random values and subsequently destroy them. For this reason, these values are also called "toxic waste." Failure to dispose of the toxic waste properly may allow an adversary to recover the secret values and maliciously forge proofs. Moreover, the underlying cryptographic assumptions of traditional Groth16 SNARKs are not post quantum secure. The potential advent of large scale quantum computers may pose another security vulnerability for systems adopting this ZKP technology. Additionally, experimental results in the Evaluation and Results chapter have shown that the trusted setup may require many hours. Although the setup is performed only once

per circuit, its high computational overhead is concerning when circuit updates are necessary. Due to the context of deployment, circuits may be required to change over time due to evolving legal or regulatory demands. However, such updates should not occur frequently. Therefore, the primary concern remains security, with the setup's time overhead being a secondary issue. Many researchers have already analyzed these issues and proposed improvements to ZK-SNARKs as well as alternative solutions [37]. A complete paradigm shift may not be necessary since compatible solutions exist and better ones may arise. Optimizing the ceremony process and integrating quantum resistant solutions into existing SNARKs remain promising avenues for future research.

7.4. Key Sizes

The experimental evaluation revealed that the proving keys for the SNARK circuits are quite large, yet still acceptable within the described context for ZK-KYC-DSIG. Nevertheless, reducing the size of the proving key remains a worthwhile objective. Optimizing the circuit design, leveraging advanced compression techniques or exploring alternative cryptographic primitives could lower the key sizes without compromising security. Such improvements would enhance storage efficiency, reduce network transfer times and extend the framework applicability to environments with tighter resource constraints.

7.5. Signature Algorithms

For the sake of generality, the current implementation, which employs RSA2048, should be extended to support RSA 4096 as many CAs are increasingly adopting this higher security standard. Some Circom projects have already implemented RSA 4096 compatibility [38]. Moreover, support for alternative encryption algorithms should be considered to ensure versatility. It is important to note, however, that employing RSA 4096 or other more complex algorithms may adversely impact system performance. Once again, further optimization of cryptographic operations is required.

7.6. Inclusion of Certificate Expiry and CRLs

The current approach does not incorporate certificate validity periods (i.e. the notBefore and notAfter fields) or CRLs, even though these elements are critical for maintaining a robust trust framework. For end user certificates, which are intended to remain private, integrating a full CRL into the ZKP is challenging due to the potentially large size of these lists. Fortunately, since private certificates typically have short lifespans, their

expiration is managed by allowing them to lapse naturally, as is commonly done in many non ZKP applications. In contrast, CA certificates are designed to be long lived, making the ongoing verification of their validity essential. A practical solution is to maintain an on chain list of revoked CA public keys. However, a whitelist of trusted CAs is already stored on chain to enable verifiers to validate the source of trust. Instead of adopting a blacklisting approach like a CRL, it is possible to exploit the already present whitelist. Therefore, if a CA certificate is revoked, its corresponding entry can simply be removed from this whitelist rather than maintaining an extensive on chain CRL.

7.7. Multiple purpose system

ZK-KYC-DSIG has been presented as a system designed primarily for identity verification or KYC processes with the ultimate goal of providing proof of an individual identity. However, its capabilities can extend far beyond this core function, potentially incorporating a vast variety of different proofs. For instance, the age of a person could be easily derived from the tax ID, enabling the creation of a straightforward proof for that attribute. Furthermore, not all types of identification would require the legal authority to retain access to the information. As a result, the framework could be adapted to support a diverse array of use cases.

7.8. Legal Authority Security

The legal authority is entrusted with the critical responsibility of safeguarding and preserving a vast amount of sensitive personal data. To mitigate the risk of key compromise or unauthorized access, a multi-key system should be adopted. In such a system, decryption of the protected content would require the collaboration of multiple keys held by different authorized entities. This arrangement ensures that no single individual, whether a public administrator, judge, or technician, can access the information independently, thereby enhancing overall security. Other security measures depend on the created infrastructure.

7.9. Inclusion in the EUDI wallet

The final objective of this research is to integrate the ZK-KYC-DSIG framework into the EUDIW. This integration will allow citizens to benefit from a digital identity solution that maintains privacy. Achieving this goal requires resolving the current limitations of the prototype. Once these limitations are addressed, integration into the EU toolbox will be feasible. The initial step toward full integration is to ensure that future development

maintain strict compliance with the EU guidelines described in the ARF document [39]. This document contains the official recommendation for developing an EU Toolbox and forms the basis for project acceptance in the EUDIW. Note that the ARF document, published in 2021, does not cover ZKP, which was enabled only by eIDAS2 as previously discussed.

8 | Conclusions

This work has highlighted the numerous critical challenges of digital identity verification and introduced a framework that could serve as an initial step toward developing an effective solution to the problem. Among the main issues identified in the literature, several have been analyzed such as privacy and security concerns, regulatory compliance, interoperability and standardization needs as well as technology integration requirements.

This thesis presented the ZK-KYC-DSIG framework and analyzed it from multiple perspectives to assess its suitability in addressing the cited challenges. The solution adopts ZKP in order to offer privacy preserving user identification and DLT as a means to ensure trust in digital transactions since blockchain (EVM in this case) ensure immutability, uniqueness, integrity and authenticity of electronic transactions. Moreover, regulatory compliance has been included by design and reevaluated during each step of the development. On the other hand, since eIDAS2 delegates the creation of a clear standard to EU member states, no definitive standard exists yet. Therefore, the main priority has been the creation of a solution which guarantees ease of adoption in order to increase the likelihood of future inclusion in an official EU standard. This objective has been further enhanced by integrating the solution with the existing infrastructure. In fact, the classic PKCS#7/CMS digital signature is employed for identification purposes due to its properties like the non repudiation property.

During the analysis of ZK-KYC-DSIG, the following results have emerged. The ZKP circuit successfully verifies the identity of an individual through a precomputed set of information extracted from a digitally signed file. The inspected prototype incorporates several simplifying assumptions, however, methods and suggestions for handling them have been thoroughly discussed. The proving time is quite fast as most of the times it requires less than two minutes even on older machines and many optimizations are easily implementable such as using the rapid-snark library instead of the classic snark-js library which should considerably improve performance. Additionally, as expected with SNARKs, verification time requires just a handful of seconds. While time complexity is totally acceptable, the proving key size could require optimization, in fact the dimension can

easily exceed 1GB in size. Actually, it has been discussed already that this should not pose a significant problem since it is mitigated by the selected product regime. Analogously, the gas costs for on chain identity verification require improvements. In fact, transaction costs are relatively high, however using a Layer two EVM blockchain significantly mitigates the issue by reducing these costs to a reasonable level. Nonetheless, these costs remain slightly higher than those of some alternative solutions. Regarding regulations, the project aligns with eIDAS 2 regulation, demonstrating that SSI principles and regulatory compliance can coexist. Finally, potential threats and opportunities have been thoroughly investigated. In conclusion, although the proposed framework has some limitations that must be taken care of before adopting it in a practical environment, it has demonstrated great potential and it could serve as a solid foundation for a future standard implementation.

We will "snarkify" the world!

Bibliography

[1] World Economic Forum. Identity in a digital world, a new chapter in the social contract. Technical report, World Economic Forum, 9 2018.

[2] Uri Rivner. Identity crisis, detecting account opening fraud in the age of identity commoditisation. *Henry Stewart Publications*, 1(4):316–325, 7 2018.

[3] Identity Theft Resource Center. Itrc annual data breach report 2024, 2025.

[4] James Rachels. Why privacy is important. *Philosophy & Public Affairs*, 4(4):323–333, 1975.

[5] T. Tony Ke and K. Sudhir. Privacy rights and data security: Gdpr and personal data markets. *Management Science*, 69(8):4389–4412, 2023.

[6] Ben Biedermann, Matthew Scerri, Victoria Kozlova, and Joshua Ellul. A systematisation of knowledge: Connecting european digital identities with web3. In *2024 IEEE International Conference on Blockchain (Blockchain)*, page 605–610. IEEE, August 2024.

[7] Alexander Mühle, Andreas Grüner, Tatiana Gayvoronskaya, and Christoph Meinel. A survey on essential components of a self-sovereign identity. *Computer Science Review*, 30:80–86, 2018.

[8] Ignacio Alamillo-Domingo Steffen Schwalm. Self-sovereign-identity & eidas: a contradiction? challenges and chances of eidas 2.0. *European Review of Digital Administration & Law - Erdal*, 2:89–108, 2021.

[9] The European Parliament and the Council of the European Union. Regulation (eu) 2024/1183 of the european parliament and of the council of 11 april 2024 amending regulation (eu) no 910/2014 as regards establishing the european digital identity framework, 2024.

[10] Smallstep Labs. Everything you should know about certificates and pki but are too afraid to ask. Online article, 2024. Updated on: May 20, 2024.

[11] Russ Housley. Cryptographic Message Syntax (CMS). RFC 5652, September 2009.

[12] Nick Pope, Denis Pinkas, and John Ross. CMS Advanced Electronic Signatures (CAdES). RFC 5126, March 2008.

[13] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. White paper, 2008.

[14] Nazanin Moosavi, Hamed Taherdoost, Nachaat Mohamed, Mitra Madanchian, Yousef Farhaoui, and Inam Ullah Khan. Blockchain technology, structure, and applications: A survey. *Procedia Computer Science*, 237:645–658, 2024. International Conference on Industry Sciences and Computer Science Innovation.

[15] Nick Szabo. Smart contracts. Online article, 1994.

[16] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. White paper, 2014.

[17] Shafaq Naheed Khan, Faiza Loukil, Chirine Ghedira-Guegan, Elhadj Benkhelifa, and Anoud Bani-Hani. Blockchain smart contracts: Applications, challenges, and future trends. *Peer-to-Peer Networking and Applications*, 14(5):2901–2925, 2021.

[18] Jan Lindquist. Introducing privacy receipts into dlt and eidas. *Journal of ICT Standardization*, 11(2):117–134, 2023.

[19] Raül Ramos Fernández. Evaluation of trust service and software product regimes for zero-knowledge proof development under eidas 2.0. *Computer Law & Security Review*, 53:105968, 2024.

[20] Nojan Sheybani, Anees Ahmed, Michel Kinsy, and Farinaz Koushanfar. Zero-knowledge proof frameworks: A survey, 2025.

[21] Ryan Lavin, Xuekai Liu, Hardhik Mohanty, Logan Norman, Giovanni Zaarour, and Bhaskar Krishnamachari. A survey on the applications of zero-knowledge proofs, 2024.

[22] Jens Ernstberger, Stefanos Chaliasos, George Kadianakis, Sebastian Steinhorst, Philipp Jovanovic, Arthur Gervais, Benjamin Livshits, and Michele Orrù. zk-bench: A toolset for comparative evaluation and performance benchmarking of snarks. In Clemente Galdi and Duong Hieu Phan, editors, *Security and Cryptography for Networks*, pages 46–72, Cham, 2024. Springer Nature Switzerland.

[23] Roman Storm Alexey Pertsev, Roman Semenov. Tornado cash privacy solution version 1.4, 12 2019.

[24] @zk-email/circuits. npm package.

[25] Awid Vaziry, Kaustabh Barman, and Patrick Herbke. Sok: Bridging trust into the blockchain. a systematic review on on-chain identity, 2024.

[26] asn1js. npm package.

[27] pkijs. npm package.

[28] @zk-email/helpers. npm package.

[29] Solidity. Solidity language. Online documentation.

[30] Chainlink. Solidity language. Online article.

[31] Galia Kondova and Jörn Erbguth. Self-sovereign identity on public blockchains and the gdpr. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, page 342–345, New York, NY, USA, 2020. Association for Computing Machinery.

[32] Quinten Stokkink, Georgy Ishmaev, Dick Epema, and Johan Pouwelse. A truly self-sovereign identity system. In *2021 IEEE 46th Conference on Local Computer Networks (LCN)*, page 1–8. IEEE, October 2021.

[33] Francesco Bruschi, Tommaso Paulon, Vincenzo Rana, and Donatella Sciuto. A privacy preserving identification protocol for smart contracts. In *2021 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6, 2021.

[34] Paolo Moser, Marco Esposito, Francesco Bruschi, and Donatella Sciuto. Privacy-Preserving eIDAS Compliance in Blockchain Wallets via zkVM, 2025.

[35] Matteo Savino. ZK-KYC-DSIG. GitHub repository, 2025.

[36] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.

[37] Junkai Liang, Daqi Hu, Pengfei Wu, Yunbo Yang, Qingni Shen, and Zhonghai Wu. SoK: Understanding zk-SNARKs: The gap between research and practice. Cryptology ePrint Archive, Paper 2025/172, 2025.

[38] Rarimo. Passport zk circuits. GitHub repository.

[39] The European Parliament and the Council of the European Union. Commission recommendation (eu) 2021/946 of 3 june 2021 on a common union toolbox for a coordinated approach towards a european digital identity framework, 2021.

[40] Hongbo Wen, Jon Stephens, Yanju Chen, Kostas Ferles, Shankara Pailoor, Kyle Charbonnet, Isil Dillig, and Yu Feng. Practical security analysis of zero-knowledge proof circuits. Cryptology ePrint Archive, Paper 2023/190, 2023.

A | Appendix

A.1. Implementation: Other Code

A.1.1. PowerShell Scripts

The project includes two Power Shell scripts whose code has not yet been presented. One script generates synthetic data for experimental tests, while the other performs ZKP setup and execution process. The code of the scripts is included below, starting with the ZKP script.

```
param(
    [string] $CircuitName,
    [string] $CircuitDir,
    [string] $OutputDir = "build",
    [string] $InputFile = "input.json",
    [string] $NodeModulesDir = ".\node_modules",
    [switch] $Compile,
    [switch] $Setup,
    [switch] $Solidity,
    [switch] $Proof,
    [switch] $Verify,
    [switch] $Test1,
    [switch] $Test2
)

$env:NODE_OPTIONS = "--max-old-space-size=12288"

if($Test1 -and $Test2)
{
    Write-Error "You cannot use Test1 and Test2 flags at the same time."
    exit 1
}

$fullOuputDir = [System.IO.Path]::Combine($CircuitDir, $OutputDir)
$fullOuputDirTest1 = [System.IO.Path]::Combine($CircuitDir, "test_1")
$fullOuputDirTest2 = [System.IO.Path]::Combine($CircuitDir, "test_2")
$circomFile = -join($CircuitName,".circom")
$wasmFile = -join($CircuitName,".wasm")
$r1csFile = -join($CircuitName,".r1cs")
```

Code A.1: Power Shell Execution Script: Input and Setup 1.

```
$witnessFileFullPath = [System.IO.Path]::Combine($fullOuputDir, "witness.wtns")
$witnessFileFullPathTest1 = [System.IO.Path]::Combine($fullOuputDirTest1, "witness.wtns")
$witnessFileFullPathTest2 = [System.IO.Path]::Combine($fullOuputDirTest2, "witness.wtns")

$jsDir = [System.IO.Path]::Combine($fullOuputDir, -join($CircuitName, "_js"))

if($Compile){
    # Remove the build directory
    if(Test-Path $fullOuputDir)
    {
        Remove-Item -Recurse -Force $fullOuputDir | Out-Null
    }
    New-Item -ItemType Directory -Path $fullOuputDir -Force | Out-Null
}

$circomFilePath = [System.IO.Path]::Combine($CircuitDir, $circomFile)
$inputFilePath = [System.IO.Path]::Combine($CircuitDir, $InputFile)
$inputFilePathTest1 = [System.IO.Path]::Combine($CircuitDir,"test_1", $InputFile)
$inputFilePathTest2 = [System.IO.Path]::Combine($CircuitDir,"test_2", $InputFile)

# Test if the circuit file exists
if (-not (Test-Path $circomFilePath))
{
    Write-Error "The circuit file $circomFile does not exist on path $circomFilePath"
    exit 1
}

function Test-LastCommadExecution {
    param (
        [string] $message
    )
    if ($LASTEXITCODE -ne 0)
    {
        Write-Error $message
        exit 1
    }
}
```

Code A.2: Power Shell Execution Script: Input and Setup 2.

```
function Start-CompileCircuit {
    #Compile the circuit with circom

    Write-Host "Compiling the circuit..."
    circom $circomFilePath --r1cs --wasm --sym -o $fullOuputDir -l $NodeModulesDir
    Test-LastCommadExecution "Failed to compile the circuit."
}
```

Code A.3: Power Shell Execution Script: Compile function.

```
function Start-GenerateWitness {
    Write-Host "Generating the witness..."
    if($Test1 -or $Test2)
    {
        if($Test1)
        {
            node "$jsDir/generate_witness.js" "$jsDir/$wasmFile" $inputFilePathTest1
                $witnessFileFullPathTest1
        }
        if($Test2)
        {
            node "$jsDir/generate_witness.js" "$jsDir/$wasmFile" $inputFilePathTest2
                $witnessFileFullPathTest2
        }
    }
    else
    {
        node "$jsDir/generate_witness.js" "$jsDir/$wasmFile" $inputFilePath
            $witnessFileFullPath
    }
    Test-LastCommadExecution "Failed to generate the witness."
}
```

Code A.4: Power Shell Execution Script: Generate Witness Function.

```
function Start-GenerateProof{
    Write-Host "Generating the proof..."
    if($Test1 -or $Test2)
    {
        if($Test1)
        {
            snarkjs groth16 prove $fullOuputDir/circuit_final.zkey
                $witnessFileFullPathTest1 $fullOuputDirTest1/proof.json
                $fullOuputDirTest1/public.json
        }
        if($Test2)
        {
            snarkjs groth16 prove $fullOuputDir/circuit_final.zkey
                $witnessFileFullPathTest2 $fullOuputDirTest2/proof.json
                $fullOuputDirTest2/public.json
        }
    }
    else
    {
        snarkjs groth16 prove $fullOuputDir/circuit_final.zkey $witnessFileFullPath
            $fullOuputDir/proof.json $fullOuputDir/public.json
    }
    Test-LastCommadExecution "Failed to generate the proof."
}
```

Code A.5: Power Shell Execution Script: Generate Proof Function.

```
function Start-Phase1(){
    # Extract the ceremony size

    # Run the snarkjs info command and capture its output
    $output = snarkjs r1cs info "$fullOuputDir/$r1csFile"
    Test-LastCommadExecution "Failed to get the number of constraints."

    # Extract the line containing the number of constraints
    $constraintsLine = $output | Select-String "# of Constraints:"

    # Use a regular expression to extract the number from the line
    if ($constraintsLine -match ": (\d+)")
    {
        $constraintsNumber = [int]$matches[1]
        Write-Host "Number of constraints: $constraintsNumber"
    }
    else
    {
        Write-Error "Could not find the number of constraints."
        exit 1
    }
    # Double the number of constraints
    $doubledConstraints = $constraintsNumber * 2

    # Calculate the exponent of the smallest power of two greater than or equal to the
        doubled number
    $cerimonySize = [Math]::Ceiling([Math]::Log($doubledConstraints, 2))
    Write-Host "Cerimony size: $cerimonySize"

    # Perform the trusted setup (Starting the Power of Tau ceremony)
    Write-Host "Performing the trusted setup..."
    snarkjs powersoftau new bn128 $cerimonySize $fullOuputDir/powersOfTau_0000.ptau
    Test-LastCommadExecution "Failed to perform the trusted setup."

    # Generate a random string to be used as a seed for the contribution
    $seed = [System.Guid]::NewGuid().ToString("N")
    $challengeFile = "$fullOuputDir/powersOfTau_0000.ptau"
    $responseFile = "$fullOuputDir/powersOfTau_0001.ptau"
    $contributorName = "First Contributor"

    # Making the first contribution
    Write-Host "Making the first contribution..."
    snarkjs powersoftau contribute $challengeFile $responseFile --name="$contributorName"
        --entropy="$seed"
    Test-LastCommadExecution "Failed to make the first contribution."
}
```

Code A.6: Power Shell Execution Script: Setup 1 Function.

```
function Start-Phase2{
    Write-Host "Preparing Phase2..."
    snarkjs powersoftau prepare phase2 $fullOuputDir/powersOfTau_0001.ptau $fullOuputDir/
        final.ptau
    Test-LastCommadExecution "Failed to prepare phase2."

    # Generate the .zkey file
    Write-Host "Generating the .zkey file..."
    snarkjs groth16 setup "$fullOuputDir/$r1csFile" $fullOuputDir/final.ptau
        $fullOuputDir/circuit_0000.zkey
    Test-LastCommadExecution "Failed to generate the .zkey file."

    # Make the contribution to phase 2
    Write-Host "Building the verification key..."
    $seed = [System.Guid]::NewGuid().ToString("N")
    snarkjs zkey contribute $fullOuputDir/circuit_0000.zkey $fullOuputDir/circuit_final.
        zkey --name="ZK_DIGSIG_KYC" --entropy="$seed"
    Test-LastCommadExecution "Failed to build the verification key."

    # Export the verification key
    Write-Host "Exporting the verification key..."
    snarkjs zkey export verificationkey $fullOuputDir/circuit_final.zkey $fullOuputDir/
        verification_key.json
    Test-LastCommadExecution "Failed to export the verification key."
}
```

Code A.7: Power Shell Execution Script: Setup 2 Function.

```
function Start-VerifyProof {
    Write-Host "Verifying the proof..."
    if($Test1 -or $Test2)
    {
        if($Test1)
        {
            snarkjs groth16 verify $fullOuputDir/verification_key.json $fullOuputDirTest1
                /public.json $fullOuputDirTest1/proof.json
        }
        if($Test2)
        {
            snarkjs groth16 verify $fullOuputDir/verification_key.json $fullOuputDirTest2
                /public.json $fullOuputDirTest2/proof.json
        }
    }
    else
    {
        snarkjs groth16 verify $fullOuputDir/verification_key.json $fullOuputDir/public.
            json $fullOuputDir/proof.json
    }
    Test-LastCommadExecution "Failed to verify the proof."
}
```

Code A.8: Power Shell Execution Script: Verify Proof Function.

```
function Start-GenerateSolidityVerifier {
    # Generate the solidity verifier
    Write-Host "Generating the solidity verifier..."
    snarkjs zkey export solidityverifier $fullOuputDir/circuit_final.zkey $fullOuputDir/
        verifier.sol
    Test-LastCommadExecution "Failed to generate the solidity verifier."
}
```

Code A.9: Power Shell Execution Script: Generate Solidity Contract Function.

```
function Start-Process{
    if($Compile){
        Start-CompileCircuit
        Write-Host "OK"
    }
    if($Setup){
        Start-Phase1
        Start-Phase2
        Write-Host "OK"
    }
    if($Solidity){
        Start-GenerateSolidityVerifier
        Write-Host "OK"
    }
    if($Proof){
        Start-GenerateWitness
        Start-GenerateProof
        Write-Host "OK"
    }
    if($Verify){
        Start-VerifyProof
        Write-Host "OK"
    }
}

Start-Process
```

Code A.10: Power Shell Execution Script: Start Process.

The script can be executed by either running the tests outlined in the Implementation chapter or by executing the following command from the command line. Various execution options are available.

```
.\<ScriptFileName>.ps1 -CircuitName <name> -CircuitDir <dir> -Compile -Setup -Solidity -Proof -Verify -Test1
```

Command A.1: Execute the ZKP script (Real Data case).

Next, the complete code of the second Power Shell script is reported. The execution command is omitted, as the input crafting script is intended solely for testing and should

be executed within the designated testing environment.

```
param(
    [string]$FiscalCode,
    [string]$FilesPath,
    [string]$DocumentPath
)


function Test-LastCommadExecution {
    param (
        [string] $message
    )
    if ($LASTEXITCODE -ne 0)
    {
        Write-Error $message
        exit 1
    }
}


function Generate-RootCA {
    $rootCAKeyFile = Join-Path $FilesPath "RootCA.key"
    $rootCAPemFile = Join-Path $FilesPath "RootCA.pem"
    $rootCACerFile = Join-Path $FilesPath "RootCA.cer"

    if (Test-Path $rootCACerFile) {
        Write-Host "Root CA already exists. Skipping generation."
        return
    }

    Write-Host "Generating new Root CA..."

    openssl genpkey -out $rootCAKeyFile -algorithm RSA -pkeyopt rsa_keygen_bits:2048
    Test-LastCommadExecution "Error generating Root CA key."

    openssl req -x509 -new -nodes -key $rootCAKeyFile -sha256 -days 3650 -subj "/C=IT/ST=
        Lazio/L=Rome/O=CustomRootCA/OU=RootCADept/CN=CustomRootCA" -out $rootCAPemFile
    Test-LastCommadExecution "Error generating Root CA certificate."

    openssl x509 -in $rootCAPemFile -outform DER -out $rootCACerFile
    Test-LastCommadExecution "Error converting Root CA certificate to DER format."

    Write-Host "Root CA generation complete."
}
```

Code A.11: Power Shell Crafting Script: Inputs, Setup and Generate root CA certificate.

```
function Generate-UserCertAndSign {
    Write-Host "Generating user certificate and signing file..."

    $userKeyFile = Join-Path $FilesPath "User.key"
    $userCsrFile = Join-Path $FilesPath "User.csr"
    $userCertFile = Join-Path $FilesPath "User.crt"

    $rootCAKeyFile = Join-Path $FilesPath "RootCA.key"
    $rootCAPemFile = Join-Path $FilesPath "RootCA.pem"

    openssl genpkey -out $userKeyFile -algorithm RSA -pkeyopt rsa_keygen_bits:2048
    Test-LastCommadExecution "Error generating user key."

    $subject = "/C=IT/ST=Lazio/L=Rome/O=MyCompany/OU=Dept/CN=RealUser/serialNumber=TINIT-
        $FiscalCode"

    openssl req -new -key $userKeyFile -subj $subject -out $userCsrFile
    Test-LastCommadExecution "Error generating user CSR."

    openssl x509 -req -in $userCsrFile -CA $rootCAPemFile -CAkey $rootCAKeyFile -
        CAcreateserial -out $userCertFile -days 1825 -sha256
    Test-LastCommadExecution "Error generating user certificate."

    $signedFilePath = "$DocumentPath.p7m"

    openssl cms -sign `
        -cades `
        -in $DocumentPath `
        -signer $userCertFile `
        -inkey $userKeyFile `
        -outform DER `
        -md sha256 `
        -binary `
        -nodetach `
        -out $signedFilePath
    Test-LastCommadExecution "Error signing file."

    Write-Host "File signed successfully -> $signedFilePath"
}
```

Code A.12: Power Shell Crafting Script: Generate user certificate and sign file.

```
function Start-Process {
    Generate-RootCA
    Generate-UserCertAndSign
    Write-Host "OK"
}

Start-Process
```

Code A.13: Power Shell Crafting Script: Start Process.

A.1.2. ZKP Additional Circuits

The main ZKP circuit calls several auxiliary circuits and their complete source code is presented in this subsection. These supplementary circuits leverage a wide set of functions provided by [24].

```
template ExtractMessageDigestFromSignedAttributes(maxSignedAttributesLength,
    maxMessageDigestLength) {
    signal input SignedAttributes[maxSignedAttributesLength];
    signal input SignedAttributesLength;
    signal input MessageDigestPatternStartingIndex;
    signal output MessageDigest[maxMessageDigestLength];

    //ASN1 pattern for the message digest
    //6 => 0x06 (OBJECT IDENTIFIER)
    //9 => 0x09 (length of the OID)
    //42 134 72 134 247 13 1 9 4 => 0x2A 86 48 86 F7 0D 01 09 04 (OID for message digest)
    //49 => 0x31 (Asn1 SET tag)
    //34 => 0x22 (length of the SET)
    //4 => 0x04 (OCTET STRING tag) specify the format of the message digest
    //32 => 0x20 (length of the message digest) == 32 bytes == 256 bits
    var messageDigestPatternLength = 15;
    var messageDigestPattern[messageDigestPatternLength] =
        [6,9,42,134,72,134,247,13,1,9,4,49,34,4,32];

    assert(maxSignedAttributesLength > SignedAttributesLength);
    assert(SignedAttributesLength > 0);
    assert(maxMessageDigestLength > 0);
    assert(MessageDigestPatternStartingIndex + messageDigestPatternLength +
        maxMessageDigestLength <= maxSignedAttributesLength);

    //Extract the message digest with its ASN1 pattern in front
    signal MessageDigestWithPattern[maxMessageDigestLength + messageDigestPatternLength];
    MessageDigestWithPattern <== VarShiftLeft(maxSignedAttributesLength,
        maxMessageDigestLength + messageDigestPatternLength)(SignedAttributes,
        MessageDigestPatternStartingIndex);

    //Check if the first bytes match the known pattern
    component patternMatchCheck = CheckSubstringMatch(messageDigestPatternLength);
    for (var i = 0; i < messageDigestPatternLength; i++) {
        patternMatchCheck.in[i] <== MessageDigestWithPattern[i];
        patternMatchCheck.substring[i] <== messageDigestPattern[i];
    }
    patternMatchCheck.isMatch === 1;

    //Fill the message digest with the bytes following the pattern
    for (var i = 0; i < maxMessageDigestLength; i++) {
        MessageDigest[i] <== MessageDigestWithPattern[messageDigestPatternLength + i];
    }
}
```

Code A.14: Circuit ExtractMessageDigestFromSignedAttributes.

```
template FormatterAndSignatureVerifier(maxDataLength, keyLength, chunksBitLength,
    totalChunksNumber) {
    assert(maxDataLength % 64 == 0);
    assert(chunksBitLength * totalChunksNumber > keyLength);
    assert(chunksBitLength < (((keyLength \8)-1) \ 2));
    signal input data[maxDataLength];
    signal input dataLength;
    signal input signature[totalChunksNumber];
    signal input publicKey[totalChunksNumber];
    //Compute hash of data
    signal output sha[256] <== HashPadded(maxDataLength)(data, dataLength);
    //Convert the hash to chunks
    var rsaMessageSize = (256 + chunksBitLength) \ chunksBitLength;
    component rsaMessage[rsaMessageSize];
    for (var i = 0; i < rsaMessageSize; i++) {
        rsaMessage[i] = Bits2Num(chunksBitLength);
    }
    for (var i = 0; i < 256; i++) {
        rsaMessage[i \ chunksBitLength].in[i % chunksBitLength] <== sha[255 - i];
    }
    for (var i = 256; i < chunksBitLength * rsaMessageSize; i++) {
        rsaMessage[i \ chunksBitLength].in[i % chunksBitLength] <== 0;
    }
    //Verify the RSA signature
    component rsaVerifier = RSAVerifier65537(chunksBitLength, totalChunksNumber);
    for (var i = 0; i < rsaMessageSize; i++) {
        rsaVerifier.message[i] <== rsaMessage[i].out;
    }
    for (var i = rsaMessageSize; i < totalChunksNumber; i++) {
        rsaVerifier.message[i] <== 0;
    }
    rsaVerifier.modulus <== publicKey;
    rsaVerifier.signature <== signature;
}
```

Code A.15: Circuit FormatterAndSignatureVerifier.

```
template HashPadded(maxBytesLength) {
    signal input paddedBytes[maxBytesLength];
    signal input paddedBytesLength;
    //Assertions
    component n2bBytesLength = Num2Bits(log2Ceil(maxBytesLength));
    n2bBytesLength.in <== paddedBytesLength;
    assert(maxBytesLength > paddedBytesLength);
    AssertZeroPadding(maxBytesLength)(paddedBytes, paddedBytesLength);
    //Compute hash
    signal output sha[256] <== Sha256Bytes(maxBytesLength)(paddedBytes, paddedBytesLength
        );
}
```

Code A.16: Circuit HashPadded.

```
template VerifyFiscalCodeAndPubkeyFromCertTbs(maxCertificateTbsLength,maxFiscalCodeLength
    ,chunksBitLength,totalChunksNumber) {

    signal input CertificateTbs[maxCertificateTbsLength];

    signal input CertificateTbsLength;

    signal input FiscalCodePatternStartingIndex;

    signal input PublicKeyModulusPatternStartingIndex;

    signal input FiscalCode[maxFiscalCodeLength];

    signal input PublicKeyModulus[totalChunksNumber];

    //ASN1 pattern for fiscal code
    var fiscalCodePatternLength = 13;

    var fiscalCodePattern[fiscalCodePatternLength] = [
        // (1) OID: 06 03 55 04 05
        6, 3, 85, 4, 5,
        // (2) Tag for PrintableString (0x13):
        19,
        // (3) Length (6 bytes 'TINIT-' + 16 bytes FISCAL CODE):
        22,
        // (4) First 6 bytes for "TINIT-" (Modify IT for other countries)
        84, 73, 78, 73, 84, 45
    ];

    //ASN1 pattern for public key modulus
    var publicKeyModulusPatternLength = 27;

    var publicKeyModulusPattern[publicKeyModulusPatternLength] = [
        // (1) RSA OID:  06 09 2A 86 48 86 F7 0D 01 01 01
        6,   9,   42,  134,  72,  134,  247,  13,   1,   1,   1,
        // (2) NULL parameters: 05 00
        5,   0,
        // (3) BIT STRING tag & length: 03 82 01 0F
        3,  130,   1,   15,
        // (4) Unused bits in BIT STRING: 00
        0,
        // (5) SEQUENCE (RSAPublicKey) tag & length: 30 82 01 0A
        48, 130,   1,   10,
        // (6) INTEGER (modulus) tag & length: 02 82 01 01
        2,  130,   1,   1,
        // (7) Leading 0 byte for the INTEGER
        0
    ];
```

Code A.17: Circuit VerifyFiscalCodeAndPubkeyFromCertTbs 1.

```
    //Extract the fiscal code with its ASN1 pattern in front
    signal FiscalCodeWithPattern[maxFiscalCodeLength + fiscalCodePatternLength];
    FiscalCodeWithPattern <== VarShiftLeft(maxCertificateTbsLength, maxFiscalCodeLength +
        fiscalCodePatternLength)(CertificateTbs, FiscalCodePatternStartingIndex);

    //Extract the public key modulus with its ASN1 pattern in front
    var maxPublicKeyModulusLength = 256; //byte length of a 2048 bit key
    signal PublicKeyModulusWithPattern[maxPublicKeyModulusLength+
        publicKeyModulusPatternLength];
    PublicKeyModulusWithPattern <== VarShiftLeft(maxCertificateTbsLength,
        maxPublicKeyModulusLength + publicKeyModulusPatternLength)(CertificateTbs,
        PublicKeyModulusPatternStartingIndex);

    //Check if the first bytes match the known pattern for fiscal code
    component patternMatchCheckFiscalCode = CheckSubstringMatch(fiscalCodePatternLength+
        maxFiscalCodeLength);
    for (var i = 0; i < fiscalCodePatternLength; i++) {
        patternMatchCheckFiscalCode.in[i] <== FiscalCodeWithPattern[i];
        patternMatchCheckFiscalCode.substring[i] <== fiscalCodePattern[i];
    }

    //Verify the fiscal code correspondence with the one passed as input
    for(var i = 0; i < maxFiscalCodeLength; i++){
        patternMatchCheckFiscalCode.in[fiscalCodePatternLength+i] <==
            FiscalCodeWithPattern[fiscalCodePatternLength+i];
        patternMatchCheckFiscalCode.substring[fiscalCodePatternLength+i] <== FiscalCode[i
            ];
    }
    patternMatchCheckFiscalCode.isMatch === 1;

    //Check if the first bytes match the known pattern for public key modulus
    component patternMatchCheckModulus = CheckSubstringMatch(
        publicKeyModulusPatternLength);
    for (var i = 0; i < publicKeyModulusPatternLength; i++) {
        patternMatchCheckModulus.in[i] <== PublicKeyModulusWithPattern[i];
        patternMatchCheckModulus.substring[i] <== publicKeyModulusPattern[i];
    }
    patternMatchCheckModulus.isMatch === 1;

    //Extract the public key modulus and chunk it to verify its correspondence with the
        one passed as input
    signal ExtractedPublicKeyModulus[maxPublicKeyModulusLength] <== VarShiftLeft(
        publicKeyModulusPatternLength+maxPublicKeyModulusLength,
        maxPublicKeyModulusLength)(PublicKeyModulusWithPattern,
        publicKeyModulusPatternLength);
    signal ChunkedPublicKeyModulus[totalChunksNumber] <== SplitBytesToWords(
        maxPublicKeyModulusLength,chunksBitLength,totalChunksNumber)(
        ExtractedPublicKeyModulus);

    for(var i = 0; i < totalChunksNumber; i++){
        ChunkedPublicKeyModulus[i] === PublicKeyModulus[i];
    }
}
```

Code A.18: Circuit VerifyFiscalCodeAndPubkeyFromCertTbs 2.

```
template VerifySimpleRsaEncryptionBase64AndExtractSubstring(maxBytesLength,
    maxBase64Length, maxSubstringLength, chunksBitLength, totalChunksNumber) {

    signal input SignatureBase64[maxBase64Length];
    signal input PublicKeyModulus[totalChunksNumber];
    signal input Message[maxBytesLength];
    signal input IndexOfPartialMessage;
    signal input MessageLength;
    signal output Substring[maxSubstringLength];

    assert(MessageLength <= maxBytesLength);

    //Decode the base64 signature and split it in chunks
    signal Signature[maxBytesLength] <== Base64Decode(maxBytesLength)(SignatureBase64);
    signal ChunkedSignature[totalChunksNumber] <== SplitBytesToWords(maxBytesLength,
        chunksBitLength,totalChunksNumber)(Signature);

    //Split data in chunks
    signal ChunkedMessage[totalChunksNumber] <== SplitBytesToWords(maxBytesLength,
        chunksBitLength,totalChunksNumber)(Message);

    //Check that the signature is in proper form and reduced mod modulus.
    component signatureRangeCheck[totalChunksNumber];
    component bigLessThan = BigLessThan(chunksBitLength, totalChunksNumber);
    for (var i = 0; i < totalChunksNumber; i++) {
        signatureRangeCheck[i] = Num2Bits(chunksBitLength);
        signatureRangeCheck[i].in <== ChunkedSignature[i];
        bigLessThan.a[i] <== ChunkedSignature[i];
        bigLessThan.b[i] <== PublicKeyModulus[i];
    }
    bigLessThan.out === 1;

    //Compute the rsa ciphertext
    component bigPow = FpPow65537Mod(chunksBitLength, totalChunksNumber);
    bigPow.base <== ChunkedMessage;
    bigPow.modulus <== PublicKeyModulus;

    //Verify the rsa signature
    for (var i = 0; i < totalChunksNumber; i++) {
        bigPow.out[i] === ChunkedSignature[i];
    }

    //Extract the substring
    Substring <== VarShiftLeft(maxBytesLength, maxSubstringLength)(Message,
        IndexOfPartialMessage);
}
```

Code A.19: Circuit VerifySimpleRsaEncryptionBase64AndExtractSubstring.

```
template VerifyHash(maxBytesLength) {
    var shaBitLength = 256;
    var shaByteLength = 32;
    //byte array
    signal input bytes[maxBytesLength];
    //byte array hash
    signal input expectedSha[shaByteLength];

    //convert expectedSha to bits
    component expectedSha2bits[shaByteLength];
    signal expectedShaBits[shaBitLength];
    for (var i = 0; i < shaByteLength; i++) {
        assert(expectedSha[i] >= 0 && expectedSha[i] < 256);
        expectedSha2bits[i] = Num2Bits(8);
        expectedSha2bits[i].in <== expectedSha[i];
        for (var j = 0; j < 8; j++) {
            expectedShaBits[i * 8 + j] <== expectedSha2bits[i].out[7-j];
        }
    }

    //Convert bytes to bits
    component byte2bits[maxBytesLength];
    signal bytesBits[maxBytesLength*8];
    for (var i = 0; i < maxBytesLength; i++) {
        assert(bytes[i] >= 0 && bytes[i] < 256);
        byte2bits[i] = Num2Bits(8);
        byte2bits[i].in <== bytes[i];
        for (var j = 0; j < 8; j++) {
            bytesBits[i * 8 + j] <== byte2bits[i].out[7-j];
        }
    }

    //Compute the hash
    signal computedShaBits[shaBitLength] <== Sha256(maxBytesLength*8)(bytesBits);

    //Verify equality
    for (var i = 0; i < shaBitLength; i++) {
        computedShaBits[i] === expectedShaBits[i];
    }
}
```

Code A.20: Circuit VerifyHash.

A.2. Additional on chain Testing within the Hardhat Environment

Additional on chain testing of the identity verification process was conducted using the Hardhat environment. It is an Ethereum development framework optimized for testing and debugging smart contracts and is well suited for TypeScript projects such as ZK-KYC-DSIG. Hardhat offers a robust local testing setup that simulates on chain interactions

effectively and serves as an excellent complement to Remix. Figure A.1 displays the outcome of the Hardhat test.

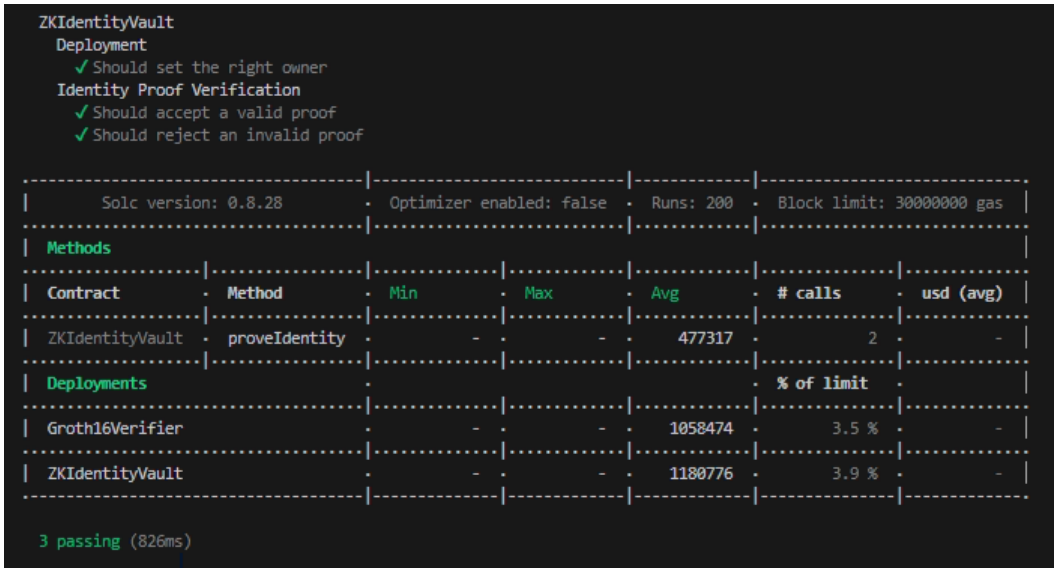

Figure A.1: Hardhat environment on chain proof verification test.

The on chain tests were performed using the Small Input version of the circuit. The gas fees computed with Hardhat (477,317 gas) are nearly identical to those previously obtained with Remix (477,236 gas), confirming the validity of the results reported in Table 6.3.

List of Figures

List of Tables

List of Codes

List of Commands

List of Acronyms

Acronym	Description
ARF	Architecture and Reference Framework
ASN	Abstract Syntax Notation
BES	Basic Electronic Signature
BER	Basic Encoding Rules
CA	Certificate Authority
CAdES	CMS Advanced Electronic Signatures
CMS	Cryptographic Message Syntax
CPU	Central Processing Unit
CRL	Certificate Revocation List
CRS	Common Reference String
DAPP	Decentralized Application
DEFI	Decentralized Finance
DER	Distinguished Encoding Rules
DID	Decentralized Identifier
DL(T)	Distributed Ledger (Technology)
DSL	Domain Specific Language
EAA	Electronic Attestation of Attributes
eIDAS	electronic Identification, Authentication and Trust Services
EU	European Union
EUDI(W)	European Digital Identity (Wallet)
EVM	Ethereum Virtual Machine
IPFS	Interplanetary File System
JWT	JSON Web Tokens

Acronym	Description
KYC	Know Your Customer
MPCitH	Multi Party Computation in the head
OID	Object Identifier
PEM	Privacy Enhanced EMail
PII	Personally Identifiable Information
PKCS#7	Public Key Cryptography Standards #7
PKI	Public Key Infrastructure
PoS	Proof of Stake
PoW	Proof of Work
QAP	Quadratic Arithmetic Program
RAM	Random Access Memory
R1CS	Rank 1 Constraint System
RSA	Rivest–Shamir–Adleman (Algorithm)
SHA	Secure Hash Algorithm
SNARK	Succinct Non-interactive Arguments of Knowledge
SOTA	State of the Art
SSI	Self Sovereign Identity
STARK	Scalable Transparent Argument of Knowledge
TBS	To Be Signed
VC	Verifiable Claim
VOLE	Vector Oblivious Linear Evaluation
ZK(P)	Zero Knowledge (Proof)

Acknowledgements

In this final segment, I would like to express my earnest gratitude to everyone who supported and encouraged me throughout the academic career, especially those who actively contributed and made it possible for me to succeed. Last but not least, I want to thank my professor and tutor for their guidance throughout the completion of this thesis, as well as my family for their constant support.