# IDEA-DAC: Integrity-Driven Editing for Accountable Decentralized Anonymous Credentials via ZK-JSON

Shuhao Zheng*
McGill University & Nullus Labs
Montreal, Canada
shuhao.zheng@mail.mcgill.ca

Zonglun Li*
McGill University & Nullus Labs
Montreal, Canada
zonglun.li@mail.mcgill.ca

Junliang Luo
McGill University
Montreal, Canada
junliang.luo@mail.mcgill.ca

Ziyue Xin
McGill University
Montreal, Canada
ziyue.xin@mail.mcgill.ca

Xue Liu
McGill University
Montreal, Canada
xueliu@cs.mcgill.ca

## ABSTRACT

Decentralized Anonymous Credential (DAC) systems are increasingly relevant, especially when enhancing revocation mechanisms in the face of complex traceability challenges. This paper introduces **IDEA-DAC**, a paradigm shift from the conventional revoke-and-reissue methods, promoting direct and **I**ntegrity-**D**riven **E**diting (IDE) for **A**ccountable **DAC**s, which results in better integrity accountability, traceability, and system simplicity. We further incorporate an Edit-bound Conformity Check that ensures tailored integrity standards during credential amendments using R1CS-based ZK-SNARKs. Delving deeper, we propose *ZK-JSON*, a unique R1CS circuit design tailored for IDE over generic JSON documents. This design imposes strictly $O(N)$ rank-1 constraints for variable-length JSON documents of up to $N$ bytes in length, encompassing serialization, encryption, and edit-bound conformity checks. Additionally, our circuits only necessitate a one-time compilation, setup, and smart contract deployment for homogeneous JSON documents up to a specified size. While preserving core DAC features such as selective disclosure, anonymity, and predicate provability, IDEA-DAC achieves precise data modification checks without revealing private content, ensuring only authorized edits are permitted. In summary, IDEA-DAC offers an enhanced methodology for large-scale JSON-formatted credential systems, setting a new standard in decentralized identity management efficiency and precision.

## CCS CONCEPTS

• **Security and privacy** → **Privacy-preserving protocols**; • **Applied computing** → **Version control**; • **Theory of computation** → *Circuit complexity*.

---

*Both authors contributed equally to this research.

## KEYWORDS

ZK-JSON, integrity-driven editing, decentralized anonymous credential, edit-bound conformity check

## 1 INTRODUCTION

Credentials serve as attestations, confirming an individual's identity or qualifications. Anonymous credentials emerged to fuse this verification process with user privacy preservation [6]. Decentralized anonymous credentials (DACs) [13] advance this concept, allowing users to verify specific attributes without revealing their complete identity, while operating independently of any central governing entity. Within a DAC framework, there are several pivotal algorithms, such as *request, issue, prove, verify*, and *revoke* [11].

The emergence of DACs and Decentralized Identifiers (DIDs) proffers a shift towards self-sovereign, privacy-centric authentication, standing in contrast to the predominant centralized models [4]. This paradigmatic transition towards Self-Sovereign Identity (SSI) through DAC is gaining momentum, especially with the advancements in blockchain technology [14, 20, 26]. Recognizing this evolution, the World Wide Web Consortium (W3C) is making strides towards standardizing DID documents [30] and the data model for verifiable credentials [31]. These standards seek to establish foundational metadata for identifiers and enhance the robustness of the verification mechanisms in digital transactions. Crucially, W3C endorses the expression of verifiable credentials in JSON-structured formats [31]. Such a credential can encapsulate facets ranging from issuer identity and subject attributes to specific conditions like validity periods. Leveraging embedded digital signatures, this JSON-oriented approach ensures data authenticity and resistance against tampering. To illustrate, W3C furnishes an exemplar[1] where an entire JSON document serves as a credential. This document comprises components like the issuer's details, issuance timestamp, and subject information, all enveloped using the JSON Web Signature to vouchsafe the data's integrity and authenticity.
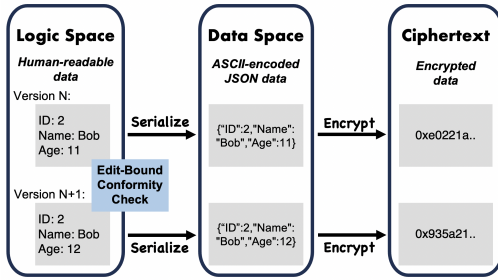
---

[1] https://www.w3.org/TR/vc-data-model/

**Figure 1: Edit-bound Conformity Check Framework**

Traditionally, updates to credentials necessitate a revoke-and-reissue approach to ensure system integrity and prevent unauthorized modifications [11, 23, 24]. This two-step approach not only introduces computational redundancy but also necessitates an additional revocation check during every credential verification, amplifying the overhead. Further compounding the inefficiency, these prevailing revoke-and-reissue paradigms frequently neglect the criticality of editing integrity. Without an integrity mechanism, systems cannot ensure that each data field consistently adheres to its established norms or criteria. Such oversight undermines system resilience and exacerbates risks, especially when data fields are adjusted to accommodate real-world scenarios.

Given the identified limitations in existing systems, it becomes beneficial to refine the credential update mechanism and enhance its integrity. To this end, we introduce **IDEA-DAC**, a novel methodology that facilitates **I**ntegrity-**D**riven **E**diting (IDE) designed for **A**ccountable **D**ecentralized **A**nonymous **C**redentials. Contrary to traditional systems, IDEA-DAC enables edit directly to a JSON credential document, ensuring that security and trustworthiness remain intact. Central to our approach is the application of zero-knowledge proofs that rigorously assess each edit against robust integrity standards, thereby presenting an optimized and holistic framework for the management of JSON credentials.

Furthermore, we introduced an edit-bound conformity check to ensure the integrity of credentials during modifications, utilizing ZK-SNARKs built upon Rank-1-Constraint-System (R1CS). The edit-bound conformity check introduces a framework that emphasizes integrity checks in the logic space rather than directly on the data space, which allows for fine-grained control over human-readable data, as illustrated in Figure 1. Subsequently, data from this logic space undergoes serialization into the data space, followed by encryption into ciphertext. This encrypted form is then suitable for publication. We formulated *ZK-JSON*, a distinctive R1CS circuit designed for IDE. This design mandates $O(N)$ rank-1 constraints for JSON documents with variable lengths, up to a maximum of $N$ bytes, covering serialization, encryption, and edit-bound conformity checks. Crucially, our circuits requires only a singular compilation, setup, and smart contract deployment, catering to homogeneous JSON documents up to a predefined size.

In our experimental assessment, we implemented IDEA-DAC and evaluated its performance using a real-world scenario: academic profiles of Ph.D. students, which demonstrates the potential data types and associated integrity rules in a JSON credential. We primarily gauged our algorithm's efficiency using three metrics: number of rank-1 constraints, proving time, and verification time. Our results suggest a linear correlation between the credential file

size and number of constraints. Additionally, the proving time stays pragmatically efficient, and the verification time remains constant.

While preserving DAC security and privacy properties, such as selective disclosure, anonymity, and predicate provability, IDEA-DAC ensures precise data modification checks. These checks operate without exposing sensitive data during credential edits, thereby strengthening the system against any unauthorized changes. The main contributions of IDEA-DAC can be delineated as:

- **Integrity-Driven Editing (IDE):** IDEA-DAC introduces a new mechanism to a DAC system named Integrity-Driven Editing (IDE). It strengthens editing integrity and traceability, harnessing the power of ZK-SNARKs built upon R1CS circuits. Moreover, IDEA-DAC introduces edit-bound conformity checks as specialized sub-circuits. These checks impose concrete integrity standards during credential amendments. As opposed to generic methods, our approach ensures that all credential adjustments strictly comply with established rules. Via our circuit design, these integrity checks seamlessly operate within the logic structure of the data, all the while upholding the privacy of credentials.
- **Advanced *ZK-JSON* Circuit:** IDEA-DAC introduces *ZK-JSON*, a pioneering R1CS circuit design specifically tailored for JSON serialization, achieving $O(N)$ rank-1 constraint complexity for JSON documents up to $N$ bytes in size. This innovative design not only lays the groundwork for deeper investigations into R1CS circuit architectures and JSON-compatible systems but also ensures a one-time setup with inherent scalability. By adeptly avoiding potential exponential complexities, it proves particularly robust for handling expansive JSON datasets, such as repositories detailing an academic's comprehensive publication history.

## 2 RELATED WORK

Decentralized identity research has had a multitude of projects from both academia and industry. Many Decentralized Identity (DID) initiatives commonly feature Predicate Provability, enabling users to demonstrate compliance without exposing personal data. The W3C DID standard acts as a key reference; systems are typically classified as compliant or non-compliant. Compliant systems such as Candid [21], SpruceID [28], and Verite [8] exist, but only Candid prioritizes privacy-preserving, granting data access solely upon user approval. Contrastingly, non-W3C compliant systems like zk-creds [24] and Coconut [27] also emphasize user privacy. However, a consistent challenge for these systems is the accurate tracking of Precise Data Modification Checks. Distinctly, IDEA-DAC, showcased in Table 1, amalgamates features such as Privacy-preserving, Predicate Probability, W3C Conformity, and precise data change monitoring. Notably, while Coconut deploys a decentralized strategy against malicious issuers, IDEA-DAC utilizes edit-bound conformity checks, optimizing the oversight of issuer activities.

Another branch of research that aligns with our work delves into secure data modifications and privacy-preserving verification, especially when multiple parties are involved. For instance, in the area of image editing, Naveh et al. presented PhotoProof [22]. Their solution allows certain image alterations while ensuring the edited image remains traceable and authentic. Meanwhile, zkDocs [1] emphasizes the secure verification of documents in contexts like mortgage applications. While these approaches have demonstrated feasibility on real-world data, they pose implementation challenges. The

| | | Candid [21] | SpruceID [28] | Verite [8] | zk-creds [24] | Coconut [27] | **IDEA-DAC** | PhotoProof [22] | ZKDocs [1] |
|---|---|---|---|---|---|---|---|---|---|
| DAC Security & Privacy Properties | Privacy-preserving | ✔ | ✗ | ✗ | ✔ | ✔ | ✔ | | |
| | Predicate Provability | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | | |
| | W3C Conformity | ✔ | ✔ | ✔ | ✗ | ✗ | ✔ | | |
| | Precise Data Editing Checks | ✗ | ✗ | ✗ | ✗ | ✗ | ✔ | | |
| Multi-party Privacy-preserving Information Sharing & Verification | Verifiable Information Editing | | | | | | ✔ | ✔ | ✔ |
| | Fine-grained Editing Check | | | | | | ✔ | ✔ | ✗ |
| | One-time Setup ZK Circuit | | | | | | ✔ | ✗ | ✗ |
| | Programmable Editing Bound | | | | | | ✔ | ✔ | ✗ |

**Table 1: Comparison between IDEA-DAC and other related systems in terms of DAC Security & Privacy Properties and Multi-party Privacy-preserving Information Sharing & Verification**

underlying ZK circuits demand frequent adjustments: any change in the underlying data requires recompilation, setup, and redeployment of the circuit. To illustrate, even a minor image adjustment, such as cropping from $512 \times 512$ to $384 \times 384$, would require a brand new circuit specific to the modified size. Additionally, while PhotoProof can track pixel-wise transformations, zkDocs lacks the ability to examine fine-grained edits within the circuit, largely due to its representation of data that isn't interpretable by humans. Our proposed IDEA-DAC addresses these challenges. It features a circuit design that requires setup only once, suitable for variable-length JSON documents, and is adept at ensuring detailed edit integrity checks, leveraging an advanced R1CS circuit framework.

## 3 PRELIMINRIES

In this section, we delve into the foundational concepts necessary for constructing the IDEA-DAC system. Specifically, we introduce ZK-SNARKs, detail the structure and significance of R1CS, and explore the Hint strategy and its implications.

### 3.1 ZK-SNARKs

Zero-Knowledge Proofs (ZKPs) [15, 16] are a cryptographic method by which one party (the prover) can prove to another party (the verifier) that they know a value $x$, without conveying any information apart from the fact that they know the value $x$. A formal definition of ZKP can be found in Appendix B.

Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (ZK-SNARKs) [5, 12, 17, 33, 35] are a special form of ZKPs that have the properties of being non-interactive and succinct. Non-interactivity means that the proof consists of a single message from the prover to the verifier. Succinctness means that the size of the proof is small (polylogarithmic in the size of the statement being proven), and verification of the proof is fast (also polylogarithmic).

### 3.2 Rank-1 Constraint System (R1CS)

A Rank-1 Constraint System (R1CS) over a field $\mathbb{F}$ serves as a foundational arithmetic representation in the construction of ZKP protocols. Its structure provides a systematic way to express and verify computational statements without revealing any information about the inputs, other than the fact that they satisfy the given statement. Formally, an R1CS over a field $\mathbb{F}$ is described as a triple $(A, B, C)$, where each of $A$, $B$, and $C$ are $n \times m$ matrices. Here, $n$ denotes the number of constraints, while $m$ represents the number of wires. An assignment $\mathbf{x} \in \mathbb{F}^m$ is said to satisfy the R1CS if and only if the following equation is true for all $i$: $\langle \mathbf{a}_i, \mathbf{x} \rangle \cdot \langle \mathbf{b}_i, \mathbf{x} \rangle = \langle \mathbf{c}_i, \mathbf{x} \rangle$, where $\mathbf{a}_i$, $\mathbf{b}_i$, and $\mathbf{c}_i$ represent the $i$-th rows of matrices $A$, $B$, and $C$ respectively, and $\langle \cdot, \cdot \rangle$ denotes the dot product. In the realm of ZKP protocols, R1CS plays a pivotal role in constructions such as ZK-SNARKs (e.g., Groth16 [17], Marlin [7], Aurora [3], Orion [34])

and recent folding schemes (e.g., Nova [19], SuperNova [18]). These protocols exploit the succinctness and efficiency of R1CS to provide proofs that are both compact and quick to verify, thereby enabling a myriad of applications in privacy-preserving transactions, secure computations, and blockchain scalability solutions.

### 3.3 Hint

R1CS, inherently designed to verify constraints for inputs, extends its capabilities beyond mere value computations within a circuit. It introduces two primary methods for calculating values designated for other circuit sections. The first is a direct approach that utilizes circuit arithmetic for computation. However, when this direct method becomes inefficient or cumbersome, an alternative strategy, termed **Hint**, is preferred. The "Hint" method allows the value to be computed off-circuit, and validated in-circuit. This implies that the prover can input the value directly into the circuit, with constraints in place to ensure its accurate computation. To illustrate, consider the computation of $y = f(x)$. If constructing $f(\cdot)$ within the circuit is challenging, an alternative is to use a function $g(x, y)$ that verifies the correctness of this computation in a more efficient manner. For instance, to compute the inverse of $x \in \mathbb{Z}_p$, one might directly compute $y = x^{p-2}$ in-circuit with $\lceil \log p \rceil$ multiplications. However, with the "Hint" approach, by directly inputting $y$, only a single rank-1 constraint is needed to ensure $x \cdot y = 1$. Effectively deploying the "Hint" strategy in cases where direct computation is onerous can result in significant in-circuit computational savings, leading to enhanced proving and verification times.
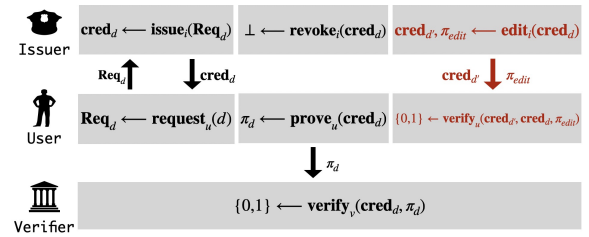


**Figure 2: IDEA-DAC Functionalities**

## 4 IDEA-DAC

Decentralized anonymous credentials (DACs) [13] are essential for decentralized identity systems. DACs allow users to get verified without showing their actual identities, balancing privacy with trust. In simple terms, DACs let users hold multiple credentials, each approved by the issuer's private key, eliminating intermediaries during the verification process.

In a DAC system, three main players exist: issuers, users, and verifiers. Users ask for credentials by sharing their data and any other needed information. Issuers, using their private key, grant these

credentials and remain the rights to revoke them. Users can then show these credentials to verifiers, who can check their authenticity without contacting the issuer directly.

However, existing DAC systems overlook the integrity of editing. Within the current DAC's method of revoking and reissuing as a form of editing, maintaining the integrity of edits is challenging. Furthermore, editing goes beyond mere content alterations; it necessitates setting distinct integrity rules for different editors in different use cases. In response to this challenge, we present IDEA-DAC: a system that guarantees editing integrity in JSON-formatted DACs through edit-bound conformity checks, as depicted in Figure 2. This section delves into the formal definitions of JSON credentials and the associated edit-bound conformity checks.

## 4.1 JSON Credentials

A credential $P$ is represented as a JSON document, comprised of various fundamental items. These items include strings, numbers, arrays, and dictionaries. For the purpose of subsequent definitions, we introduce a universal object set, $O$. It is formally defined as $O := S \cup N \cup A \cup D$, where $S, N, A$, and $D$ stand for the sets of all strings, numbers, arrays, and dictionaries, respectively. This definition of $O$ will be utilized in the ensuing sections.

**Definition 4.1 (String).** A *String* item is denoted by $r \in \Sigma^*$, where $\Sigma$ is the set of all the possible characters.

To encompass most practical scenarios, we assume all characters to be represented as ASCII-encoded byte values; thus, $\Sigma = [0, 255]$.

**Definition 4.2 (Number).** Define a *Number* item $r$ as $r \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^*$.

The definition for numbers can be generalized to encompass positive integers. Furthermore, it can be extended to accommodate various numerical representations, including negative integers (denoted by $-r$), floating-point numbers (denoted by $(-)r_1.r_2$), and scientific notations (denoted by $(-)r_1.r_2e(-)r_3$).

**Definition 4.3 (Array).** An array is $a \in O^*$, which means $a$ is an ordered tuple whose elements are all elements in $O$.

In an array, an element can be of types such as a string, integer, dictionary, or even another array. The length of the array can be adjusted by adding or removing items within it.

**Definition 4.4 (Dictionary).** A dictionary $d$ is a set of key-value pairs with the following properties:

$$d \in D \iff \exists n \in \mathbb{Z}^+, d = \{(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\},$$
$$k_i \in S, \quad v_i \in O \quad \text{for each } i \in [n],$$

A dictionary within a credential comprises multiple key-value pairs; the key must always be a string, whereas the value can be a string, integer, array, or even another dictionary. Conventionally, a JSON credential $P$ is represented as a dictionary, i.e., $P \in D$.

## 4.2 Edit-bound Conformity Check

Every legitimate editing activity must adhere to several *integrity rules*, each represented as a function $\sigma$. These rules ensure that the editor follows the appropriate rules to undertake such edits. The integrity verification comes in two variants: the target-only check and the source-target differential check.

**Definition 4.5 (Target-only check).** A target-only check is a function $\sigma_\theta(r) \to \{0, 1\}$, where $r$ is the edited credential item and $\theta$ is some pre-defined parameters.

The combination of $\sigma$ and $\theta$ defines an *integrity rule* specific to an editor. As an example, an editor might be constrained to an *integrity rule* for a *String* credential item, termed as "One of the Set". This rule restricts the editor to only modify the item to a new string $r'$ such that $r' \in \theta$. Here, $\theta$ denotes a pre-defined set of strings allowable for this particular editor. Thus, the pairing of $\sigma$ with $\theta$ characterizes a distinct *integrity rule* for that editor.

At times, merely examining the target credential is inadequate for regulating an editing activity. A *source-target differential check* assesses the disparity between the pre- and post-edited credentials.

**Definition 4.6 (Source-target differential check).** A source-target differential check is a function $\sigma'_\theta(r, r') \to \{0, 1\}$, where $r$ and $r'$ stand for pre- and post-edited credential items.

Just as with target-only checks, an *integrity rule* is defined by the combination of $\sigma'$ and $\theta$. For instance, an editor might be subject to an integrity rule for an array item, termed "Append Only". This rule dictates that the editor is only permitted to append elements to the array item $r$. If the augmented array is denoted as $r'$, $\sigma'_\theta(r, r') = True \iff r_i = r'_i, \forall 0 \leq i < |r|$, where $|r|$ is the length of the array item $r$. In such a situation, $\theta$ is always $\perp$, meaning that this rule does not need additional parameters.

These two checks solely verify if the editing activity adheres to the appropriate integrity rules. However, if an editor intends to publish the edited credential item as a public record, additional checks must be passed. We denote the checks that need to be conducted prior to publishing content as the *encoding and encryption check* $\omega$, which assesses the correctness of the serialization and encryption procedures. It will grant approval if and only if both the serialization and encryption processes are executed correctly.

**Definition 4.7 (Encoding and Encryption check).** An encoding and encryption check is a function: $\omega(r, c, k) \to \{0, 1\}$ that verifies if $c = Enc_k(Serialize(r))$, where $r$ is the credential and $Enc, Serialize$ is the symmetric encryption algorithm and the serialization algorithm respectively.

Finally, we define the *edit-bound conformity check*, which encompasses all the checks above.

**Definition 4.8 (Edit-bound Conformity Check).** Denote $\lambda$ as the security parameter. The *edit-bound conformity check* is represented by the function $\phi(r, r', c, c', k, \Sigma, \Sigma') \to \{0, 1\}$, where $r$ and $r'$ denote the pre- and post-edited items, respectively, and $c$ and $c'$ are their corresponding ciphertexts. Additionally, $k$ is the encryption key, known to both the user and editors, and $\Sigma, \Sigma'$ stands for set of all the applied target-only checks and the source-target differential checks. Note that both $c$ and $c'$ are encrypted using the same key $k$. The function $\phi$ satisfies completeness, soundness, and privacy-preserving properties, formally defined in Appendix C.

## 5 INTEGRITY-DRIVEN EDITING VIA *ZK-JSON*

In this section, we delve into *ZK-JSON*, the intricate R1CS circuit design behind Integrity-Driven Editing (IDE) for IDEA-DAC. With a focus on modularity and efficiency, the IDE circuit is crafted to

validate every edit made to a JSON document against strict integrity rules, all while requiring setup only once. Through advanced serialization, encryption, and efficient batch merging, it guarantees secure and optimal in-circuit document processing. Coupled with stringent conformity checks, this design assures both the accuracy and integrity of document edits, making it a cornerstone for the security of large-scale JSON credentials in real-world scenarios.

## 5.1 R1CS Primitives

The Rank-1 Constraint System (R1CS) is essential for converting certain logic structures. To effectively harness the power of R1CS, we introduce several foundational primitives. A critical element in this process is the **linear combination**, which underpins the necessary composability for circuit logic.

**Definition 5.1 (Linear combination).** A linear combination, denoted as $l = [(c_i, x_i) \mid i = 1, 2, \ldots, n]$, comprises an array of 2-tuples $(c_i, x_i)$ with arbitrary length $n \in \mathbb{Z}^+$. Here, $c_i \in \mathbb{F}$ represents the coefficient, while $x_i \in \mathbb{Z}_{\geq 0}$ is the wire ID in R1CS, denoting the $x_i$-th element in the assignment vector $\mathbf{x}$.

Notably, the variable ID 0 signifies the constant 1. Following this definition, the conversion of a linear combination into rows within the matrices $A$, $B$, and $C$ is intuitive. To streamline our discussions, we'll use **linear combination** and **variable** interchangeably henceforth. Based on the foundation of linear combinations, we present the subsequent R1CS primitives:

- $a_{[N]} \leftarrow$ **NewArray**($N$): Initializes an array of size $N$.
- $c \leftarrow$ **Add**($a$, $b$): Combines the terms of $a$ and $b$ to produce a new linear combination. No new wires or constraints are added.
- $c \leftarrow$ **Sub**($a$, $b$): Produces a linear combination by subtracting $b$ from $a$. To achieve this, all coefficients in $b$ are negated in $\mathbb{F}$. No additional wires or constraints are introduced.
- $c \leftarrow$ **Mul**($a$, $b$): Represents the product of two variables. If either variable is constant, the coefficients of the other are scaled. Otherwise, it creates $c = [(1, \text{newID})]$ and a constraint $a \cdot b = c$.
- $c \leftarrow$ **Or**($a$, $b$): Computes the logical OR between two Boolean-constrained variables. If both are Boolean-constrained, it results in the constraint $a \cdot b = a + b - c$.
- $c \leftarrow$ **And**($a$, $b$): Computes the logical AND for two Boolean variables, which is synonymous with **Mul**($a$, $b$).
- $b \leftarrow$ **IsZero**($a$): Determines if variable $a$ is zero. Instead of using $\lceil \log |\mathbb{F}| \rceil$ constraints for direct computation, it uses three constraints based on hint. Specifically, the constraints are: $b \cdot (1-b) = 0$, $a \cdot b = 0$, and, given the inverse $c$ of $a + b$, $(a + b) \cdot c = 1$.
- $b \leftarrow$ **IsNotZero**($a$): Checks if variable $a$ is non-zero. This is derived from **IsZero**($a$) using the operation $b \leftarrow$ **Sub**(1, **IsZero**($a$)).
- $c \leftarrow$ **IsEqual**($a$, $b$): Checks if $a$ and $b$ are equal, implemented using **IsZero**(**Sub**($a$, $b$)) with three constraints.
- $c \leftarrow$ **Select**($s$, $r_0$, $r_1$): Outputs $r_0$ if $s = 1$ and $r_1$ otherwise. It introduces $c = [(1, \text{newID})]$ and the constraint $s \cdot (r_0 - r_1) = c - r_1$.
- **Assert**($a == b$): Ensures $a$ and $b$ are equal in the constraint system using the constraint $a \cdot 1 = b$.
- $b \leftarrow$ **WithinBinary**($a$, $N$): Verifies if variable $a$ has a bit-size of at most $N$. It inputs the bit decomposition $bit_{[N]}$ of $a$ and uses $N + 3$ constraints, specifically: $bit[i] \cdot (1 - bit[i]) = 0, \forall i \in [N]$, and checks the decomposition with **IsEqual**($(\sum_{i=0}^{n-1} 2^i bit[i])$, $a$).

- $r \leftarrow$ **RO**($\ldots$): Utilizes an in-circuit random oracle to generate a pseudo-random variable derived from existing circuit variables. This is enforced using collision-resistant ZK-friendly hash functions like MiMC, resulting in efficient R1CS construction.

## 5.2 Length-prepadded String (LPS)

To standardize a one-time setup for homogeneous documents, the circuit must accommodate variable-length inputs up to a maximal threshold. This flexibility allows for efficient version control when documents of the same kind undergo changes, such as increased string lengths or added array items. By introducing redundancy into the circuit structure, we can mitigate the need for frequent re-compilations, setups, and deployments whenever data undergoes modifications. Thus, we present a length-prepadded representation for strings. This encoding embeds the actual string length directly into the circuit, facilitating efficient string processing. Given that ASCII characters are byte-encodable, each byte is treated as an individual variable for easier string operations.

**Definition 5.2 (Length-prepadded String (LPS)).** An LPS, $s$, with a maximum byte-length $N$, is an array of size $N + 1$ described as $s = [l, s_0, s_1, \ldots, s_{l-1}, D, \ldots, D]$. Here, $l, s_i \in \mathbb{F}$ with $s_i \in [0, 255]$ and $l < N$. The symbol $D$ signifies a dummy constant excluded from valid string characters, such that $D \in \mathbb{F}$ and $D \notin [0, 255]$. For notation simplicity, we use $l || s_{[N]}$ to represent the string.

To further streamline string operations, we introduce two primitives for handling dummy values:

- $b \leftarrow$ **IsDummy**($a$): Determines if variable $a$ represents a dummy character in a string, achieved using **IsEqual**($a$, $D$).
- $b \leftarrow$ **IsNotDummy(IND)**($a$): Verifies if variable $a$ is not a dummy character, computed via **Sub**(1, **IsDummy**($a$)).

## 5.3 LPS Operations

*5.3.1 Range Check.* To ensure that string inputs to the circuit are valid, every string variable must either be a byte or the dummy value $D$. Specifically, $s[i] < 256$ or $s[i] = D$ for all $i = 0, 1, \ldots, N-1$. For this purpose, we propose an $O(N)$-sized range check circuit: **Assert**$\big(\sum_{i=0}^{N-1}(\textbf{WithinBinary}(s[i], 8) + \textbf{IsDummy}(s[i])) == N\big)$, which necessitates $14N + 1$ constraints. The correctness of this construction is evident when observing that both **WithinBinary** and **IsDummy** cannot simultaneously be true. If any $s[i]$ falsifies both, the sum will be less than $N$. Therefore, the assertion mandates that at least one of these values be true for each $s[i]$.

*5.3.2 Legitimacy Check.* For operations to be correctly executed later, strings input to the circuit must adhere to the LPS format. To verify this for all strings up to a maximum length $N$, we propose an $O(N)$ legitimacy check. Given an LPS $l || s_{[N]}$, the legitimacy check validates three aspects: 1) $s[i] \neq D$ for $0 \leq i < l$; 2) $s[i] = D$ for $l \leq i < N$; 3) $l < N$. While this may seem straightforward, accounting for variable $l$ that might assume any value in $\mathbb{F}$ complicates matters. A naive approach involves $O(N)$ in-circuit variable comparisons, resulting in $O(N \log N)$ constraints, which is sub-optimal. We present an optimized legitimacy check that needs only $O(N)$ constraints. The circuit computes a tag $c[i]$ for each variable: $c[i] = l - i - 1$ if $s[i] \neq D$, and $i + 1 - l$ otherwise. This requires $4N$ constraints. Then, we establish the following constraints based on $c[i]$, also taking into account if the end of the

actual string has been reached: $c[i] == c[i-1] - 1$ if the end hasn't been reached, and $c[i] == c[i-1] + 1$ otherwise. This implementation requires $7N - 6$ constraints. The string ends when the first index $i$ has $c[i]$ as a dummy variable. This is tracked using a Boolean variable, requiring $N$ constraints. Then, considering: $0 < l < N \iff \exists 0 \le i < N, c[i] = 0$. We need to ensure either: 1) only one 0 exists in $c[i]$; or 2) $l = 0$. This demands $3N+4$ constraints. An optimized legitimacy check algorithm is shown in Algorithm 2.

*5.3.3 Merge Check.* During JSON credential serialization, two LPS strings must be concatenated while retaining format integrity. Constructing a generic circuit capable of merging strings of various lengths poses challenges. It's essential to design a multipurpose circuit that merges any two strings efficiently and correctly.

The naive approach requires $O((N_a+N_b)^2)$ constraints to merge two LPS strings of maximal lengths $N_a$ and $N_b$, as shown in Algorithm 6. We propose a hint-based optimization that needs only $O(N_a + N_b)$ constraints. Instead of using a multiplexer, the prover inputs the merged LPS directly. The circuit then verifies its correctness via various constraints. Given two LPS structures, $s_a =$ aLen$||$a$_{[N_a]}$ and $s_b =$ bLen$||$b$_{[N_b]}$, the prover inputs the result LPS $s_c =$ cLen$||$c$_{[N_a+N_b]}$ directly into the circuit. We validate $s_c$ by ensuring it's correctly formatted and is the exact merged LPS of $s_a$ and $s_b$ through the following checks:

(1) Assert that the length of the merged string is the sum of the lengths of both strings: **Assert**(cLen == **Add**(aLen, bLen)).
(2) Ensure the range check of the merged string: **RangeCheck(** c$_{[N_a+N_b]}$), referencing the same circuit as in Section 5.3.1.
(3) Invoke an in-circuit random oracle to produce a pseudo-random value $r$ and validate the equation:

$$\prod_{i=0}^{N_a-1} [r - (256(i+1) + a[i])\mathbf{IND}(a[i])]$$
$$\times \prod_{i=0}^{N_b-1} [r - (256(i+1+\text{aLen}) + b[i])\mathbf{IND}(b[i])] \quad (1)$$
$$= \prod_{i=0}^{N_a+N_b-1} [r - (256(i+1) + c[i])\mathbf{IND}(c[i])],$$

Define sets $S_1 = \{(256(i+1) + a[i])\mathbf{IND}(a[i])\} \cup \{(256(i+1 + \text{aLen})+b[i])\mathbf{IND}(b[i])\}$ and $S_2 = \{(256(i+1)+c[i])\mathbf{IND}(c[i])\}$. Each variable in a$_{[N_a]}$ and b$_{[N_b]}$ is coupled with its target index in the merged string. The lowest 8 bits store the byte value, while the subsequent bits indicate the target index starting from 1. This ensures the uniqueness and non-zero nature of each non-dummy element in sets $S_1$ and $S_2$. Due to the range check constraints and unique index bits of the variables, we ensure that the first aLen + bLen variables in $S_2$ match those in $S_1$. Given $s_a$ and $s_b$ are both legitimate LPS as checked initially, and there are exactly aLen+bLen non-zero terms in $S_1$, we infer that all the other variables in $S_2$ are dummy variables, ensuring the legitimacy of $s_c$. Finally, according to Schwartz–Zippel lemma [25], this check passes with a negligible soundness error of $\frac{N_a+N_b}{|\mathbb{F}|}$. This final check utilizes $|\mathbf{RO}| + 10(N_a + N_b) - 2$ constraints, a great improvement over the naive method when employing ZK-friendly hash functions as the random oracle.

## 5.4 Serialization

Drawing from the foundational LPS operations, we design a JSON serialization circuit, aligning with the definition in Section 4.1. By predetermining maximal length parameters for Number, String, and Array, we eliminate repetitive re-compilations, setups, and deployments. While changing value types within a Dictionary poses challenges, we address this by presuming fixed value types given the typical static nature of JSON document structures. This strategy allows flexibility with arbitrary keys and values, ensuring a one-time setup for homogeneous documents. We subsequently detail the serialization circuit **Encode** for each JSON data type.

*5.4.1 Encoding Number.* The process of encoding a number mandates its decimal decomposition within the circuit. This operation, when approached traditionally, can be computationally costly. To ameliorate this, we employ a strategy similar to the one in **Within-Binary** for bit-decomposition using hints. This strategy involves computing the decomposition outside the circuit and subsequently verifying its accuracy within the circuit. For a given number $x$ with a maximum of $N$ digits, the prover inputs a length-prepadded decimal decomposition, represented as len$||$dec$_{[N]}$ in big-endian format, with trailing 0s serving as dummy digits. The decomposition's accuracy is confirmed using three dedicated sub-circuits:

(1) **RangeCheckDecimal**(dec$_{[N]}$): This validates that each dec$[i]$ lies in $[0, 9]$ for all $i \in [N]$: **Assert(** $\sum_{i=0}^{N-1}(\mathbf{IsEqual}(s[i], 8) +$ **IsEqual**$(s[i], 9) + \mathbf{WithinBinary}(s[i], 3)) == N)$, necessitating $12N + 1$ constraints. The correctness of this method resembles that for the LPS range check in Section 5.3.1.
(2) **Decomposition Check**: Utilizing a remLen variable to track the residual length, and an isEnd variable to signify the termination of the current digit sequence, we reconstruct the digits in big-endian format to match the original number using: 1) sum = **Select**(isEnd[$i$], sum, **Mul**(sum, 10)); 2) sum = **Add**(sum, dec[$i$]). Then, we use an assertion, **Assert**(sum = $x$), to validate the correctness of the decomposition. It's noteworthy that this mechanism inherently ensures $x < 10^N$, given the nonexistence of an $N$-digit decomposition for $x$ when $x \ge 10^N$.
(3) **Legitimacy Check**: This ascertains that the decomposition is in valid format, ensuring all trailing digits are zeroes, represented by: **Assert**(**And**(isEnd[$i$], **IsNotZero**(dec[$i$])) == 0).

The final step involves translating the digits into an ASCII-encoded LPS as: out[$i$] = **Select**(isEnd[$i$], $D$, **Add**(dec[$i$], 48)). Collectively, these checks and conversions demand $O(N)$ rank-1 constraints. A complete algorithm is provided in Algorithm 1.

*5.4.2 Encoding String.* Before we initiate the encoding of strings, it's essential to validate that the strings adhere to the expected LPS format. This validation is achieved through the range and legitimacy checks as described in Sections 5.3.1 and 5.3.2. After ensuring they are in the correct LPS format, we then merge a double quotation mark at both the beginning and the end of the LPS.

*5.4.3 Encoding Array.* When encoding an array that contains various objects, it's crucial to exclude any empty elements from the serialized LPS. To this end, we implement the **IsEmpty** function to ascertain the status of each object within the array. For a Number $x$, we designate a value $D'$, such that $D' > 10^{\text{maxDigit}}$, to represent its emptiness, with a slight modification of the number encoding circuit. For an LPS structure, $l||s_{[n]}$, the emptiness is determined by the condition $l = 0$. Both Arrays and Dictionaries

are assessed recursively to determine their emptiness. Given these mechanisms, the encoding process for an Array $a_{[n]}$ can be outlined as: 1) tmpLPS = **Merge**(oldLPS, **Encode**(a[i])); 2) newLPS = **SelectArray**(**IsEmpty**(a[i]), oldLPS, tmpLPS), where **SelectArray** is **Select** over each individual variable.

*5.4.4 Encoding Dictionary.* Encoding a Dictionary closely mirrors the Array encoding process, but there's a key distinction to note: Dictionary values are typified from the beginning. Therefore, when invoking the **Encode** function, both the key and its paired value are encoded. This approach offers flexibility, allowing documents to omit certain predefined Dictionary fields, while still adhering to a one-time setup. To sum up, given every JSON document inherently represents a Dictionary, the serialization process begins with this primary Dictionary, and then recursively proceeds to ensure accurate serialization of the entire JSON document.

## 5.5 Encryption

Upon obtaining the serialized JSON string $l||s_N$ as an LPS, we proceed to encrypt it within the circuit using an encryption key. This encryption conceals the internal data before it is published. Although ZK-friendly encryption algorithms like MiMC can adeptly encrypt field elements using minimal rank-1 constraints, encrypting the LPS byte-by-byte could result in a vast ciphertext. This arises since every byte is encrypted into a separate field element, potentially consuming up to 254 bits in the BN254 elliptic curve which is supported by native EVM pre-compiles. To address this, we emphasize compressing the LPS bytes before encryption. Each byte occupies a maximum of 8 bits, making it efficient to merge every $\lfloor \frac{254}{8} \rfloor = 31$ bytes into a single variable, achieving lossless compression. Additionally, it's crucial to encrypt only the actual length bytes, which necessitates additional constraints to manage the dummy values. The **Compress** procedure constructs compressed variables $m_{[M]}$, where $M = \lceil \frac{N}{31} \rceil$, and a dummy indicator isDummy$_{[M]}$ as: 1) $m[i] = \sum_{j=31i}^{\min(N, 31(i+1))} 2^{8(j-31i)} s[j] \cdot \textbf{IND}(s[j])$; 2) isDummy$[i] = $ **IsZero** $\left( \sum_{j=31i}^{\min(N, 31(i+1))} \textbf{IND}(s[j]) \right)$. Then, the **Encrypt** operation generates the ciphertext $c_{[M]}$ using the secret key $k \in \mathbb{F}$: $c[i] = $ **Select**(isDummy$[i]$, 0, **MiMC**(k, m[i])). We assign 0 to dummy variables to simplify the array comparison process in Section 5.8.

## 5.6 Edit-bound Conformity Check Circuit

Once all the data in the JSON document is logically represented in the circuit inputs, incorporating checks on this data becomes relatively straightforward, requiring only a modest number of rank-1 constraints. For instance, to ascertain whether a Number lies within a specific range, merely two in-circuit variable comparisons suffice. Verifying the correctness of a string's format can be accomplished via character-by-character comparison or a range check.

Nonetheless, each edit-bound check circuit is intrinsically tailored to its specific use cases. Attempting to design a universal circuit capable of managing every conceivable check isn't pragmatic. To retain the one-time setup characteristic of the circuit, we define two sets of sub-circuits: $\mathbb{C}_t$ and $\mathbb{C}_{st}$. These sets include all feasible target-only checks $\sigma_{\theta,s}(r)$ and source-target differential checks $\sigma'_{\theta,s}(r, r')$ relevant to the particular use case. Here, $\theta$ is also variables fed into the circuit to ensure utmost flexibility, while $s$

acts as a selection bit, indicating the applicability of this edit-bound check. Consequently, the final edit-bound check circuit, denoted **EditCheck**$(r, r', \theta, s)$, operating on an old credential $r$ and its new counterpart $r'$, is formulated as: **Assert** $\left( \sum_{\sigma_{\theta,s} \in \mathbb{C}_t} (1 - \sigma_{\theta,s}(r)) \cdot s + \sum_{\sigma'_{\theta,s} \in \mathbb{C}_{st}} (1 - \sigma'_{\theta,s}(r, r')) \cdot s == 0 \right)$. Note that both the parameters $\theta$ and the selection bits $s$ should be accessible to the verifier, ensuring the edit-bound check adheres to the correct protocol.

## 5.7 Achieving Strictly Linear Circuit Size

While the **Merge** technique delineated in Section 5.3.3 produces constraints linear in relation to the input size, consecutively merging LPS in a serial manner could culminate in a complexity of $O(N^2)$ constraints for documents with a maximum length of $N$ bytes. To enhance circuit efficiency, we introduce the **BatchMerge** strategy. This approach guarantees a strictly linear $O(N)$ constraint complexity for documents that span up to $N$ bytes in length. The main idea behind **BatchMerge** is to first gather all the LPSs that need to be merged in sequence. After collecting them, we then merge them all together using an enhanced version of the merge check process, detailed in Algorithm 4.

The transition from a 2-LPS merge check to a more comprehensive $n$-LPS merge check is similar to what we discussed in Section 5.3.3. We combine the target index with the byte value for each byte in the input LPS. Afterwards, we ensure the two batch multiplications are equal using randomness generated by an in-circuit random oracle. With minor alterations to the serialization procedure presented in Section 5.4, the entire IDE circuit can attain a linear circuit complexity in relation to the maximal document size. This optimization proves pivotal, especially when contemplating the scaling of JSON documents to vast dimensions, such as cataloging a scholar's complete publications within a singular document.

## 5.8 Put Everything Together

Using the modules we've described earlier, we now present a comprehensive circuit for Integrity-driven Editing (IDE) of JSON documents. The circuit begins with legitimacy and range checks for all inputted strings. Next, both the old and new credentials are serialized and encrypted. The circuit then compares these two versions to ensure they adhere to specified editing integrity rules. It's important to emphasize that the IDE circuit is designed to intrinsically encrypt both the old and new credentials using the same key to ensure that editors cannot modify the encryption key.

An essential step in finalizing the circuit is to validate that the in-circuit ciphertext matches the one computed externally. This ensures both serialization and encryption processes are accurate. Given the presence of dummy variables in our ciphertext variable array, we introduce a circuit, **AssertArray**$(a_{N_a}, b_{N_b})$, to validate the equality of two arrays, even if their lengths differ. It achieves this by padding the shorter array with zeros until both arrays reach a common length, $N = \max(N_a, N_b)$. The equality of the arrays is then verified using: **Assert** $\left( \sum_{i=0}^{N-1} \textbf{IsEqual}(a[i], b[i]) == N \right)$. This wraps up our discussion on the IDE circuit design. Each component has been meticulously refined for optimal performance, ensuring advanced capabilities with a reasonable constraint count. The complete IDE circuit can be found in Algorithm 5.

## 6  USE CASE

IDEA-DAC transforms credential management by emphasizing integrity-driven verification. It offers a more efficient approach than the traditional revoke-and-reissue method, guaranteeing that credential alterations uphold system integrity. DACs have diverse applications, from employment to government services and medical records, underlining their practical significance.

According to the W3C's production rule [29] and verifiable credentials data model [31], credentials are serialized into JSON format. EXAMPLE 33 [32] displays a verifiable credential in a DID document linked to a resident card. Organizations can validate this by examining its JSON Web Signature and the issuer's public key. Technically aligned with the W3C standard, IDEA-DAC explores JSON serialization through a provable circuit, supporting multi-party edits. Specific entities, like past employers, can add details about job titles, employment spans, and roles. Professional organizations can input data on memberships, certifications, or accolades. Educational bodies can list degrees, grades, and academic milestones, while peers or clients might contribute testimonials or skill endorsements. Though individuals can't modify these entries in their DAC, they can control their disclosure, like showcasing recent testimonials while withholding older ones. This offers a balance: while individuals control their data's visibility, their reputation also relies on externally verified, immutable inputs.

IDEA-DAC, as elaborated in Appendix ??, illustrates a multi-party editable credential system for DID. This protocol delineates the creation, reading, and updating of a credential document, akin to the *ethr-did* method on Ethereum [10]. While authorized parties can modify designated fields in the document, the permissions vary amongst different entities within these parties. In a university scenario, imagine the following Ph.D. profile acting both as a verifier of student status within educational institutions and as evidence of the educational level for job contexts:

```
{"program_status":"Ongoing","program_years":5,
"publications":[{"title":"XXX","year":2022}],
"student_id":"UNI421",
"duration":{"start":"08/01/18","end":"05/31/23"}}
```

The `program_status` field uses a specified set of strings to denote the student's current stage, and the `program_years` reflects the research duration. New articles can be added to the append-only `publications` without altering previous entries. Meanwhile, `student_id` provides a uniquely formatted identifier, and `duration` denotes the program's duration in a timeframe format.

## 7  EXPERIMENTS

We implemented the IDEA-DAC circuit for the PhD Profile use case described in Section 6 as our benchmark, and conducted experiments to assess our circuit's performance. Our algorithmic design is encapsulated within a comprehensive circuit, developed using the Gnark [9] framework. All experiments were conducted on a standard AWS EC2 `r5a.8xlarge` instance, equipped with 32 vCPUs and 256GB of memory. To evaluate the scalability of our circuit, we varied the document size by incrementing the maximum number of publications and then measured three key metrics across these sizes: the number of rank-1 constraints, the proving time, and verification time. Beyond the Encoding and Encryption checks, we incorporates five distinct integrity rules:

(1) **One of the Set:** Suitable for various data types, this rule limits modifications to a well-defined set of values. For example, within the `program_status` field, the permissible values are limited to "Approved", "Ongoing", "Graduated", and "Withdrawn".
(2) **Number Within Range:** Tailored for numeric entries, this rule ensures that a number remains within a specified interval. For example, the `program_year` should lie between 3 and 8.
(3) **Append Only:** This rule, apt for list or dictionary, facilitates the addition of new entries while disallowing the removal of existing ones. In a scholar's publication history context, while new publications can be appended, previous ones are immutable.
(4) **Meet Certain Format:** Targeted at strings, this rule mandates edits to align with certain format. For example, the student ID may necessitate an "AAA111" pattern for a specified institution.
(5) **Time Sensitive:** Time can be encoded by Unix timestamps. This rule ensures, for example, that a project's start date precedes its end date, and the end date precedes the current date.
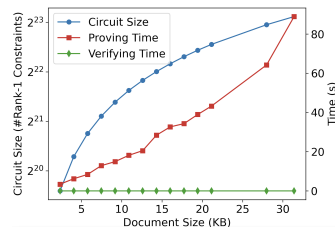


**Figure 3: Experimental results of IDEA-DAC, showcasing key metrics across varying credential document sizes.**

As depicted in Figure 3, the results highlight the algorithm's performance across document sizes ranging from 1KB to 33KB. We observe a clear linear correlation between the document size and the number of rank-1 constraints. Specifically, a document size of 1KB requires around $3 \times 10^5$ constraints, and there is an increase of around $5 \times 10^5$ constraints for every 2KB increment in the document size. Conversely, the proving time, which encompasses both constraint solving and proof generation, does not scale linearly. This non-linear behavior can be attributed to the use of ZK-friendly random oracles, such as MiMC [2], that are not amenable to parallel constraint solving. However, the time required for proof generation remains reasonable; for instance, a 32KB document takes just about 80 seconds. We underscore that our evaluations utilized the Groth16 [17] protocol over BN254 curve on a 32-vCPU machine—a basic configuration. Employing servers with enhanced CPU capabilities and leveraging advanced ZKP protocols will further optimize proving times, especially for large-scale applications. For example, a R1CS with $2^{22}$ constraints only requires 15$s$ to prove using Orion [34] with a single CPU core. In terms of verification, Groth16 performs well, with the time lying between 1ms and 2ms.

## 8  CONCLUSION

In the field of decentralized anonymous credentials (DACs) and Decentralized Identifiers (DIDs), updating credentials efficiently and with integrity is challenging. We present IDEA-DAC, a method for managing JSON credentials differently. IDEA-DAC utilizes Integrity-Driven Editing (IDE) with ZK-SNARKs and R1CS circuits for the editing process. Our R1CS design for JSON serialization, *ZK-JSON*, optimizes efficiency. Tests indicate a linear connection between credential file size and constraint count. Overall, IDEA-DAC advances editing integrity in DACs, addressing large JSON dataset complexities. Future work can further build on these insights.

## ACKNOWLEDGMENTS

We sincerely thank Yongzheng Jia for participating in the discussion of the initial idea.

## REFERENCES

[1] a16z crypto. 2023. ZkDocs: Zero-knowledge Information Sharing. https://a16zcrypto.com/posts/article/zkdocs-zero-knowledge-information-sharing/.

[2] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. 2016. MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity. In *Advances in Cryptology – ASIACRYPT 2016 (Lecture Notes in Computer Science)*, Jung Hee Cheon and Tsuyoshi Takagi (Eds.). Springer, Berlin, Heidelberg, 191–219. https://doi.org/10.1007/978-3-662-53887-6_7

[3] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P Ward. 2019. Aurora: Transparent succinct arguments for R1CS. In *Advances in Cryptology–EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part I 38*. Springer, 103–128.

[4] Clemens Brunner, Ulrich Gallersdörfer, Fabian Knirsch, Dominik Engel, and Florian Matthes. 2021. DID and VC:Untangling Decentralized Identifiers and Verifiable Credentials for the Web of Trust. In *Proceedings of the 2020 3rd International Conference on Blockchain Technology and Applications* (Xi'an, China) *(ICBTA '20)*. Association for Computing Machinery, New York, NY, USA, 61–66. https://doi.org/10.1145/3446983.3446992

[5] B. Bunz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. 2018. Bulletproofs: Short Proofs for Confidential Transactions and More. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 315–334. https://doi.org/10.1109/SP.2018.00020

[6] David Chaum. 1985. Security without identification: Transaction systems to make big brother obsolete. *Commun. ACM* 28, 10 (1985), 1030–1044.

[7] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. 2020. Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS. In *Advances in Cryptology – EUROCRYPT 2020 (Lecture Notes in Computer Science)*, Anne Canteaut and Yuval Ishai (Eds.). Springer International Publishing, Cham, 738–768. https://doi.org/10.1007/978-3-030-45721-1_26

[8] Circle. 2023. Verite. https://www.circle.com/en/verite.

[9] Consensys. 2023. Gnark. https://docs.gnark.consensys.net/overview.

[10] Veramo core development. 2023. Ethr-DID Library. https://github.com/uport-project/ethr-did.

[11] Jens Ernstberger, Jan Lauinger, Fatima Elsheimy, Liyi Zhou, Sebastian Steinhorst, Ran Canetti, Andrew Miller, Arthur Gervais, and Dawn Song. 2023. SoK: Data Sovereignty. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. IEEE Computer Society, 122–143. https://doi.org/10.1109/EuroSP57164.2023.00017

[12] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. 2019. PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge. Cryptology ePrint Archive, Paper 2019/953. https://eprint.iacr.org/2019/953

[13] Christina Garman, Matthew Green, and Ian Miers. 2014. Decentralized Anonymous Credentials. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society. https://www.ndss-symposium.org/ndss2014/decentralized-anonymous-credentials

[14] Sandro Rodriguez Garzon, Hakan Yildiz, and Axel Küpper. 2022. Decentralized Identifiers and Self-sovereign Identity in 6G. *IEEE Network* 36, 4 (2022), 142–148.

[15] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. 2008. Delegating Computation: Interactive Proofs for Muggles. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing (STOC '08)*. Association for Computing Machinery, New York, NY, USA, 113–122. https://doi.org/10.1145/1374376.1374396

[16] S Goldwasser, S Micali, and C Rackoff. 1985. The Knowledge Complexity of Interactive Proof-Systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing (STOC '85)*. Association for Computing Machinery, New York, NY, USA, 291–304. https://doi.org/10.1145/22145.22178

[17] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *Advances in Cryptology – EUROCRYPT 2016*, Marc Fischlin and Jean-Sébastien Coron (Eds.). Vol. 9666. Springer Berlin Heidelberg, Berlin, Heidelberg, 305–326. https://doi.org/10.1007/978-3-662-49896-5_11

[18] Abhiram Kothapalli and Srinath Setty. 2022. SuperNova: Proving universal machine executions without universal circuits. Cryptology ePrint Archive, Paper 2022/1758. https://eprint.iacr.org/2022/1758

[19] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. 2022. Nova: Recursive zero-knowledge arguments from folding schemes. In *Annual International Cryptology Conference*. Springer, 359–388.

[20] Shu Yun Lim, Omar Bin Musa, Bander Ali Saleh Al-Rimy, and Abdullah Almasri. 2022. Trust models for blockchain-based self-sovereign identity management: A survey and research directions. *Advances in Blockchain Technology for Cyber Physical Systems* (2022), 277–302.

[21] Deepak Maram, Harjasleen Malvai, Fan Zhang, Nerla Jean-Louis, Alexander Frolov, Tyler Kell, Tyrone Lobban, Christine Moy, Ari Juels, and Andrew Miller. 2021. Candid: Can-do decentralized identity with legacy compatibility, sybil-resistance, and accountability. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1348–1366.

[22] Assa Naveh and Eran Tromer. 2016. Photoproof: Cryptographic image authentication for any set of permissible transformations. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 255–271.

[23] Deevashwer Rathee, Guru Vamsi Policharla, Tiancheng Xie, Ryan Cottone, and Dawn Song. 2022. Zebra: Anonymous credentials with practical on-chain verification and applications to defi. *Cryptology ePrint Archive* (2022).

[24] M. Rosenberg, J. White, C. Garman, and I. Miers. 2023. zk-creds: Flexible Anonymous Credentials from zkSNARKs and Existing Identity Infrastructure. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 790–808. https://doi.org/10.1109/SP46215.2023.10179430

[25] J. T. Schwartz. 1980. Fast Probabilistic Algorithms for Verification of Polynomial Identities. *J. ACM* 27, 4 (Oct. 1980), 701–717. https://doi.org/10.1145/322217.322225

[26] Mohammed Shuaib, Noor Hafizah Hassan, Sahnius Usman, Shadab Alam, Surbhi Bhatia, Parul Agarwal, and Sheikh Mohammad Idrees. 2022. Land registry framework based on self-sovereign identity (SSI) for environmental sustainability. *Sustainability* 14, 9 (2022), 5400.

[27] A. Sonnino, M. Al-Bassam, S. Bano, S. Meiklejohn, and G. Danezis. 2019. Coconut: threshold issuance selective disclosure credentials with applications to distributed ledgers. *Proceedings 2019 Network and Distributed System Security Symposium* (2019). https://doi.org/10.14722/ndss.2019.23272

[28] SpruceID. 2023. SpruceID. https://spruceid.com/.

[29] W3C. 2023. Decentralized Identifiers v1.0, Production and Consumption. https://www.w3.org/TR/did-core/#dfn-production.

[30] W3C. 2023. Decentralized Identifiers v1.0, Terminology. https://www.w3.org/TR/did-core/{#}dfn-did-documents.

[31] W3C. 2023. Verifiable Credentials Data Model v1.1. https://www.w3.org/TR/vc-data-model/#example-a-simple-example-of-a-verifiable-credential.

[32] W3C. 2023. W3C DID. https://www.w3.org/TR/did-core/.

[33] Tiacheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. 2019. Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation. In *Advances in Cryptology – CRYPTO 2019 (Lecture Notes in Computer Science)*, Alexandra Boldyreva and Daniele Micciancio (Eds.). Springer International Publishing, Cham, 733–764. https://doi.org/10.1007/978-3-030-26954-8_24

[34] Tiancheng Xie, Yupeng Zhang, and Dawn Song. 2022. Orion: Zero Knowledge Proof with Linear Prover Time. In *Advances in Cryptology – CRYPTO 2022*, Yevgeniy Dodis and Thomas Shrimpton (Eds.). Vol. 13510. Springer Nature Switzerland, Cham, 299–328. https://doi.org/10.1007/978-3-031-15985-5_11

[35] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. 2020. Transparent Polynomial Delegation and Its Applications to Zero Knowledge Proof. In *2020 IEEE Symposium on Security and Privacy (SP)*. 859–876. https://doi.org/10.1109/SP40000.2020.00052

## A  CIRCUIT PSEUDOCODES

Additionally define the following two R1CS primitives for the naive Merge algorithm:

- $c \leftarrow$ **Lookup2**$(s_0, s_1, r_0, r_1, r_2, r_3)$: 2-bit lookup table. Output $r_0, r_1, r_2, r_3$ if $(s_0, s_1) = (0, 0), (1, 0), (0, 1), (1, 1)$, separately. This functionality can be achieved via a low-degree extension of the binary function, i.e., $f(s_0, s_1) = (1-s_0)(1-s_1)r_0 + s_0(1-s_1)r_1 + (1-s_0)s_1r_2 + s_0s_1r_3$. Simplifying the polynomial gives us a solution of using only 3 constraints: 1) $(r_3 - r_2 - r_1 + r_0) \cdot s_1 = t_1 - r_1 + r_0$; 2) $t_1 \cdot s_0 = t_2$; 3) $(r_2 - r_0) \cdot s_1 = c - t_2 - r_0$, where $t_1$ and $t_2$ are newly-added wires.

- $c \leftarrow$ **M**$(a_{[N]}, b)$: A multiplexer that selects the $b$-th element in $a_{[N]}$ where $b$ is also an in-circuit variable. We show a linear multiplexer implementation in Algorithm 7.

**Algorithm 1** Encoding a Number in ASCII String Format

1: **function** ENCODENUMBER(x, maxDigit)
    ▷ Input the decomposed decimals computed off-circuit
2:    len$\|$dec$_{[\text{maxDigit}]}$ ← GetDecimal(x,maxDigit)
3:    **RangeCheckDecimal**$\left(\text{dec}_{[\text{maxDigit}]}\right)$     ▷ Check 1
    ▷ Check the correctness of decomposition
4:    sum ← 0, remLen ← len
5:    isEnd$_{[maxDigit]}$ ← **NewArray**(maxDigit)
6:    **for** i = 0; i < maxDigit; i++ **do**
7:        isEnd[i] ← **IsZero**(remLen)
8:        sum ← **Select**(isEnd, sum, **Mul**(sum, 10))
9:        **Assert**(**And**(isEnd, **IsNotZero**(dec[i])) == 0) ▷ Check 2
10:       sum ← **Add**(sum, dec[i])    ▷ No effect on dummy 0s
11:       remLen ← **Select**(isEnd, 0, **Sub**(remLen, 1))
12:    **Assert**(sum == x)     ▷ Check 3
    ▷ Get the ASCII-encoded LPS Representation
13:    res$_{[\text{maxDigit}]}$ ← **NewArray**(maxDigit)
14:    **for** i = 0; i < maxDigit; i++ **do**
15:        res[i] ← **Select**(isEnd[i], D, **Add**(dec[i], 48))
16:    **return** len$\|$res$_{[\text{maxDigit}]}$

---

**Algorithm 2** LPS Legitimacy Check

1: **function** LEGITIMACYCHECK(aLen$\|$a$_{[N]}$)
2:    isEnd ← 0, numZero ← 0, numValid ← 0
3:    c$_{[N]}$ ← **NewArray**(N)
4:    **for** i = 0; i < N; i++ **do**
5:        d ← **IsDummy**(a[i])
6:        isEnd ← **Or**(isEnd, d)
7:        c[i] ← **Select**(d, **Sub**(i+1, aLen), **Sub**(aLen, i+1))
8:        numZero ← **Add**(numZero, **IsZero**(c[i]))
9:        **if** i > 0 **then**
10:           v1 ← **IsEqual**(c[i], **Add**(c[i-1], 1))
11:           v2 ← **IsEqual**(c[i], **Sub**(c[i-1], 1))
12:           valid ← **Select**(isEnd, v1, v2)
13:           numValid ← **Add**(numValid, valid)
14:    **Assert**(numValid == N − 1)
15:    **Assert**(**IsEqual**(numZero, 1)+**IsZero**(aLen) == 1)

---

**Algorithm 3** Hint-based Linear LPS Merge

1: **function** MERGE(aLen$\|$a$_{[N_a]}$, bLen$\|$b$_{[N_b]}$)
    ▷ Input the merged string computed off-circuit
2:    cLen$\|$c$_{[N_a+N_b]}$ ← Merge(aLen$\|$a$_{[N_a]}$, bLen$\|$b$_{[N_b]}$)
3:    **Assert**(cLen == **Add**(aLen, bLen))    ▷ Check 1
4:    **RangeCheck**$\left(c_{[N_a+N_b]}\right)$    ▷ Check 2
    ▷ Check3: the correctness of LPS merge
5:    r ← **RO**(aLen$\|$a$_{[N_a]}$, bLen$\|$b$_{[N_b]}$, cLen$\|$c$_{[N_a+N_b]}$)
    ▷ IND stands for **IsNotDummy**
6:    aMul ← $\prod_{i=0}^{N_a-1}[r-(256(i+1)+a[i])\textbf{IND}(a[i])]$
7:    bMul ← $\prod_{i=0}^{N_b-1}[r-(256(i+1+aLen)+b[i])\textbf{IND}(b[i])]$
8:    cMul ← $\prod_{i=0}^{N_a+N_b-1}[r-(256(i+1)+c[i])\textbf{IND}(c[i])]$
9:    **Assert**(**Mul**(aMul, bMul) == cMul)
10:    **return** cLen$\|$c$_{[N_a+N_b]}$

---

**Algorithm 4** Hint-based Linear Batch LPS Merge

1: **function** BATCHMERGE($l^{(0)}\|s_{[N_0]}^{(0)}, \ldots, l^{(n-1)}\|s_{[N_{n-1}]}^{(n-1)}$)
    ▷ Input the merged string computed off-circuit
2:    MaxLen ← $\sum_{i=0}^{n-1} N_i$
3:    oLen$\|$o$_{[\text{MaxLen}]}$ ← Merge($l^{(0)}\|s_{N_0}^{(0)}, \ldots, l^{(n-1)}\|s_{N_{n-1}}^{(n-1)}$)
4:    **Assert**(oLen == **Add**($l^{(0)}, l^{(1)}, \ldots, l^{(n-1)}$))   ▷ Check 1
5:    **RangeCheck**$\left(o_{[\text{MaxLen}]}\right)$    ▷ Check 2
    ▷ Check3: the correctness of LPS merge
6:    r ← **RO**($l^{(0)}\|s_{[N_0]}^{(0)}, \ldots, l^{(n-1)}\|s_{[N_{n-1}]}^{(n-1)}$, oLen$\|$o$_{[\text{MaxLen}]}$)
    ▷ IND stands for **IsNotDummy**
7:    cumLen$_i$ ← $\sum_{j=0}^{i-1} l^{(j)}$
8:    sMul$_i$ ← $\prod_{j=0}^{N_i-1}[r-(256(j+1+\text{cumLen}_i)+s[j])\textbf{IND}(s[j])]$
9:    oMul ← $\prod_{i=0}^{\text{MaxLen}-1}[r-(256(i+1)+o[i])\textbf{IND}(o[i])]$
10:    **Assert**(**Mul**(sMul$_0$, sMul$_1$, \ldots, sMul$_{n-1}$) == oMul)
11:    **return** oLen$\|$o$_{[\text{MaxLen}]}$

---

**Algorithm 5** Integrity-driven Editing Circuit

1: **function** IDE($r_{old}, r_{new} : \mathcal{D}, k : \mathbb{F}, c_{old}, c_{new} : [\mathbb{F}], \theta, s : [\mathbb{F}]$)
2:    $c'_{old}$ ← **Encrypt**(k, **Compress**(**Encode**($r_{old}$)))
3:    $c'_{new}$ ← **Encrypt**(k, **Compress**(**Encode**($r_{new}$)))
    ▷ Assert the equivalence of two arrays with unequal length
4:    **AssertArray**($c'_{old} == c_{old}$)
5:    **AssertArray**($c'_{new} == c_{new}$)
6:    **EditCheck**($r_{old}, r_{new}, \theta, s$)

---

**Algorithm 6** Naive LPS Merge

1: **function** NAIVEMERGE(aLen$\|$a$_{[N_a]}$, bLen$\|$b$_{[N_b]}$)
2:    $l_1$ ← aLen, $l_2$ ← bLen, outLen ← **Add**($l_1, l_2$)
3:    out$_{[N_a+N_b]}$ ← **NewArray**($N_a + N_b$)
4:    a$_{[N_a+N_b]}$ ← pad(a$_{[N_a]}$, $\textbf{0}_{[N_b]}$)
5:    b$_{[N_a+N_b]}$ ← pad($\textbf{0}_{[N_a]}$, b$_{[N_b]}$)    ▷ Avoid index overflow
    ▷ Extract the output array
6:    **for** i = 0; i < $N_a + N_b$; i++ **do**
7:        $\alpha$ ← **IsZero**($l_1$), $\beta$ ← **IsZero**($l_2$)
8:        out[i] ← **Lookup2**($\alpha,\beta$,a[i],**b[i+$N_a$-aLen]**,a[i],0)
           ▷ **b[i+$N_a$-aLen]** computed by a linear multiplexer
9:        $l_1$ ← **Select**($\alpha$, 0, **Sub**($l_1$, 1))
10:       $l_2$ ← **Lookup2**($\alpha, \beta, l_2$, **Sub**($l_2$, 1), $l_2$, 0)
11:    **return** outLen$\|$out$_{[N_a+N_b]}$

---

**Algorithm 7** Linear Multiplexer

1: **function** M(x$_{[N]}$, idx: $|\mathbb{F}|$)
2:    logN ← logCeil(N)     ▷ $\lceil \log_2 N \rceil$
3:    res$_{[2^{\text{logN}}]}$ ← pad(x$_{[N]}$, $\textbf{0}_{[2^{\text{logN}}-N]}$)
4:    idxBin$_{[\text{logN}]}$ ← **ToBinary**(idx, logN)
5:    **for** i = 0; i < logN; i++ **do**
6:        **for** j = 0; j < $2^{(\text{logN}-i-1)}$; j++ **do**
7:           res[j] ← **Select**(idxBin[i], res[2j+1], res[2j])
8:    **return** res[0]

## B FORMAL DEFINITION OF ZERO KNOWLEDGE PROOF (ZKP)

Formally, a ZKP consists of three algorithms $(P, V, S)$, where $P$ is the prover, $V$ is the verifier, and $S$ is the simulator. A ZKP has the following properties:

- **Completeness:** If the statement is true, the honest verifier (that is, one following the protocol properly) will be convinced of this fact by an honest prover. Formally, for any $x, w$ such that $x \in L_w$ and any verifier strategy $V^*$,

$$\Pr[(P(x, w), V^*(x)) = 1] = 1.$$

- **Soundness:** If the statement is false, no cheating prover can convince the honest verifier that it is true, except with some small probability. Formally, for any $x \notin L_w$, any prover strategy $P^*$, and any $w'$,

$$\Pr[(P^*(x, w'), V(x)) = 1] \leq negl(\lambda),$$

where $negl(\lambda)$ is a negligible function.

- **Zero-knowledge:** If the statement is true, no verifier learns anything other than this fact. This is formalized by showing that every verifier has some simulator that, given only the statement to be proved (and no access to the prover), can produce a transcript that "looks like" an interaction between the honest prover and the verifier in question. Formally, for any $x, w$ such that $x \in L_w$, any verifier strategy $V^*$, and any $w'$,

$$\{(P(x, w), V^*(x))\} \approx \{(S(x, w'), V^*(x))\},$$

where the approximation symbol $\approx$ denotes computational indistinguishability.

## C FORMAL DEFINITION OF PROPERTIES IN EDIT-BOUND CONFORMITY CHECK

In this section, we show a formal definition of completeness, soundness, and privacy-preserving properties in edit-bound conformity check. Given the notations defined in Section 4.2, the three properties are defined as

- **Completeness:** If $(\forall \sigma_\theta, \sigma'_\theta \in \Sigma, \Sigma', \sigma_\theta(r') = 1, \sigma'_\theta(r, r') = 1) \wedge (\omega(r, c, k) = \omega(r', c', k) = 1) : \mathbf{Pr}\{\phi(r, r', c, c', k, \Sigma, \Sigma') = 1\} = 1$
- **Soundness:** If $(\exists \sigma_\theta \in \Sigma, \sigma_\theta(r') = 0) \vee (\exists \sigma'_\theta \in \Sigma', \sigma'_\theta(r, r') = 0) \vee (\omega(r, c, k) = 0) \vee (\omega(r', c', k) = 0) : \mathbf{Pr}\{\phi(r, r', c, c', k, \Sigma, \Sigma'\} = 1\} \leq negl(\lambda)$
- **Privacy-preserving:** for any Probabilistic Polynomial Time (PPT) distinguisher $\mathcal{D}$, the probability that $\mathcal{D}$ can correctly distinguish which of the two credentials has been randomly sampled, without knowledge of the key $k$, is bounded as: $\mathbf{Pr}\{\mathcal{D}(r, \hat{r}) = 1 \mid c, \hat{r} \xleftarrow{\$} R\} = \frac{1}{2} + negl(\lambda)$, where $R$ is the set of all possible credentials. This property applies to both the pre- and post-edited credentials.

## D SUPPLEMENTARY EXPLANATION OF EXPERIMENTS

We tested our algorithms by proving changes from

```
{"Status":"Approved","ProgramYear":5,
"student_id":"UNI42",
"publications":[{"Title": "ZK-Profile","Year": 2023},
                {"Title": "ZK-Cred","Year": 2022}],
"Duration": {
    "Start": 1561016554,
    "End": 1687275819
}}
```
to:
```
{"Status":"Approved","ProgramYear":5,
"student_id":"UNI42",
"publications":[{"Title": "ZK-Profile","Year": 2023},
                {"Title": "ZK-Cred","Year": 2022}
                {"Title": "ZK-Auth","Year": 2023}],
"Duration": {
    "Start": 1561016554,
    "End": 1781941354
}}
```
in our experiment.

To better define the minimum specifications for all users, we conducted an additional experiment using a standard personal laptop, specifically a MacBook Pro with an M1 Pro (10-core) processor and 32GB of RAM. The results, including circuit size, number of gates, and proving and verifying times, are detailed in Fig4.
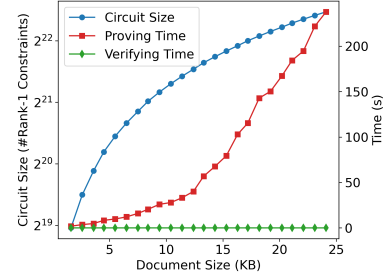


**Figure 4: Experimental results of IDEA-DAC, showcasing key metrics across varying credential document sizes on the local machine.**

Our findings indicate that 32GB of RAM is adequate for processing JSON certificates up to 24KB, a size that comfortably meets the requirements of most DAC use cases. Regarding proving times, they are not significantly longer than those reported in our paper. For instance, processing a certificate of around 10KB takes less than half a minute, which is approximately 1.3 times longer than the times achieved using more powerful machines, as documented in our research.

## E LIMITATIONS

IDEA-DAC encompasses primarily two limitations. The first limitation pertains to the need for pre-determining the types of values in dictionaries to leverage the one-time setup property of IDEA-DAC. This requirement can indeed pose constraints. However, a feasible workaround involves allowing for empty values in the dictionaries. By doing so, we can effectively cover all potential value types that might occur in real-world applications. This approach

provides a level of flexibility within the predefined structure, partially mitigating the limitation. The second limitation concerns the circuit design of IDEA-DAC, which is currently optimized for R1CS arithmetization, favoring linear operations. While IDEA-DAC is currently tailored for R1CS arithmetization, its adaptability to other arithmetization forms, such as the Plonkish representation, is straightforward in principle. The adaptation primarily involves constructing all the necessary primitives, a process that is relatively simple and direct. However, it's important to note that while adaptation for functional compatibility is straightforward, optimizing the circuit for Plonkish representation to enhance performance and minimize circuit size is a more involved process. This optimization would ensure that IDEA-DAC fully leverages the specific efficiencies of the Plonkish system, thereby delivering improved performance metrics.