Nova Aadhaar: Privacy-Preserving Aadhaar-based Age Proofs Suitable for Resource-Constrained Environments

Saravanan Vijayakumaran
Department of Electrical Engineering
IIT Bombay
Mumbai, India
sarva@ee.iitb.ac.in

ABSTRACT

An age proof is a cryptographic proof that a government-issued digital ID contains a birth date of someone who is above a certain age. Such a proof is said to be privacy-preserving if it hides the other contents of the document. All Indian citizens are eligible to acquire an identity document called Aadhaar. Every physical and digital version of an Aadhaar document has a QR code containing digitally signed personal details of the holder including their birth date.

We present the design and implementation of Nova Aadhaar, a privacy-preserving age proof scheme based on Aadhaar QR codes which has lower resource (memory/download) requirements than the state-of-the-art. We have implemented and deployed our scheme in two ways: as a web application that can be accessed by a browser and as a mobile application that can be installed on an Android-based phone. Proof generation takes about a minute on a mid-range phone and a couple of minutes in a laptop browser. Proof sizes are about 16 kB.

KEYWORDS

Zero-knowledge proofs, age proofs, folding schemes, Aadhaar

1 INTRODUCTION

Age verification on the Internet is a challenging problem. But it is a problem in need of practical solutions because age-based Internet restrictions have become a reality in some countries [29].

One approach is to verify that a user is in possession of an authenticated digital document attesting to their age. There is currently no way to check that the document actually belongs to the user. Even if users have the necessary documents, they will be reluctant to share them with third-parties since the documents may contain other private information apart from their age (like gender and address).

If the document has a digital signature, zero-knowledge (ZK) proofs can be used to prove that the age mentioned in it is above a certain threshold. To preserve privacy, the ZK proofs need to be generated on the client (the user's computer or mobile phone). For this solution to be available to a large section of users, the computational resources required to generate such proofs should be as low as possible. A solution which works only on high-end phones will exclude users who cannot afford such phones.

Aadhaar is an identity document issued by the Indian government to all its citizens [46]. It contains a QR code which encodes the holder's personal details including their date of birth. The QR code also contains a digital signature attesting to this data, which was created by a government-owned private key. There already exists implementations of age proofs based on Aadhaar QR codes.

In this paper, we present the design and implementation of Nova Aadhaar, a privacy-preserving Aadhaar-based age proof that has lower resource requirements than the state-of-the-art schemes. Its name reflects the fact that it is based on the Nova folding scheme [32].

Paper Organization. In Section 2, we describe related work. Our contributions are summarized in Section 3. The Aadhaar QR code format is described in Section 4. In Section 5, we discuss desirable features of Aadhaar-based age proofs. In Section 6, we discuss the challenges in uniquely identifying an Aadhaar holder purely based on their Aadhaar QR code data and conclude with an assumption. In Section 7, we motivate the features of our design by first considering two simpler strawman designs and their flaws. The key notion of a nullifier is introduced here. In Section 8, we give a brief overview of the Nova proof system. We describe our proposed scheme in Section 9. We describe its implementation and performance in Section 10, followed by a few concluding remarks in Section 11.

2 RELATED WORK

The Anon Aadhaar project [6] by the Privacy and Scaling Explorations team at Ethereum Foundation is the pioneer in Aadhaar-based zero-knowledge proofs. They first developed a system to check the validity of an Aadhaar PDF file [38, 39] without revealing the file itself.

Later, they developed an age proof scheme which uses the Aadhaar QR code [15, 17, 18]. They use the Groth16 [21] proof system because it enables the proofs to be verified on the Ethereum blockchain. The proofs are only a few hundred bytes long.

But the Groth16 proof system requires a statement-specific trusted setup to generate a large structured reference string (SRS). The Anon Aadhaar v2 trusted setup involved 105 participants [40]. Provers and verifiers need to download the SRS before they can generate or verify proofs. The size of this string is approximately 300 MB, which makes it inconvenient for people with restricted internet access. Furthermore, the Anon Aadhaar web application [18] has high memory requirements [16] which can cause the proof generation to fail in some mobile browsers.

3 OUR CONTRIBUTIONS

We present a design and implementation of an Aadhaar-based age proof scheme based on the Nova folding scheme [32]. Since Nova has a transparent setup, there is no SRS download needed. All the

Proceedings of XXXX YYYY(X), 1–15
2025. https://doi.org/XXXXXXX.XXXXXXX

public parameters can be generated on-demand by the prover and verifier.

We have deployed our scheme in two ways: as a web application that can be accessed by a browser (https://age-proof.vercel.app/) and as a mobile application that can be installed on an Android-based phone [41–43]. Proof generation takes about a minute on a mid-range phone and a couple of minutes in a laptop browser. Proof sizes are about 16 kB. Our experiments show that our implementations have a lower memory footprint than Anon Aadhaar. This makes them usable on a wider range of user devices.

4 AADHAAR QR CODE DATA FORMAT

The Aadhaar QR code data consists of the following fields in sequence. The fields are separated by a delimiter byte which has the value `0xFF` [11], except that there is no delimiter byte before the masked email address and RSA signature fields (see Sections 4.10 and 4.11). The sequence of fields and the RSA signature generation procedure are illustrated in Figure 1.

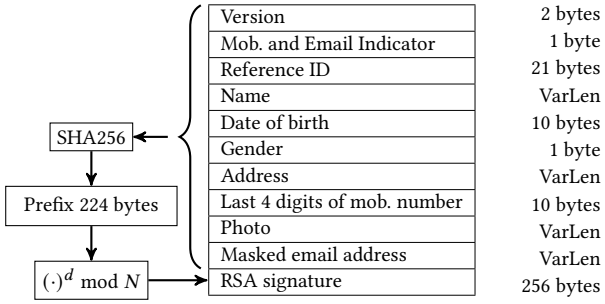


Figure 1: The fields in the Aadhaar QR code data and their lengths. VarLen indicates that the corresponding field is variable-length. The RSA signature is generated by raising the EMSA-PKCS1-v1_5 encoding of the other fields to the RSA decryption exponent d modulo the RSA modulus N.

4.1 Version

The version field is a two-byte ASCII encoding [9] of the string "`Vn`" where `n` can be either 2, 3, or 4. So the first two bytes can be `0x5632`, `0x5633`, or `0x5634`.

4.2 Mobile Number and Email Indicator

This one-byte field is the ASCII encoding of an integer in the range 0 to 3. Version `V1` of the Aadhaar QR code had the facility to store SHA256 hashes of the holder's mobile phone number and/or email address if they were registered by the holder at the time of QR code creation [45]. From version `V2` onwards, instead of SHA256 hashes the QR code stores partially masked versions of the mobile number and email address (see Sections 4.8 and 4.10). The value of the mobile number and email indicator byte indicates the following.

- **0** No mobile number or email information present in QR code
- **1** Only email information present in QR code
- **2** Only mobile number information present in QR code
- **3** Both mobile number and email information present in QR code

4.3 Reference ID

This 21-byte field is the ASCII encoding of the following data.

- The first 4 bytes contain the last four digits of the holder's Aadhaar number. Note that the remaining digits of the holder's Aadhaar number do not appear in the Aadhaar QR code.
- The remaining 17 bytes contain the QR code creation timestamp in the `ddMMyyyyHHmmssSSS` format [13].

An Aadhar holder with a registered mobile number can download a new QR code using the mAadhaar mobile app [36] or a new Aadhaar pdf from the UIDAI website [37]. For every download, the timestamp in the QR code is updated to the current date and time.

4.4 Name

This is a variable-length field encoding the name of the Aadhaar holder. Since the length of this field varies across Aadhaar holders, the byte location of the birth date (the next field) in the QR code data is not fixed. This variation necessitates additional logic in the arithmetization of the age proof.

4.5 Date of Birth

This 10-byte field is the ASCII encoding of the Aadhaar holder's date of birth in the `DD-MM-YYYY` format.

4.6 Gender

This is a one byte field which contains the ASCII encoding of one of the characters `M`, `F`, or `T`. It specifies that holder's gender as male, female, or transgender.

4.7 Address

The address of the Aadhaar holder is specified using 11 consecutive fields (separated by delimiter bytes) in the following order. All of them except the pin code are variable-length fields. The fields are ordered in alphabetical order of their names (except for the landmark field appearing before the house number field).

(1) **Care of**: This field specifies the name of the Aadhaar holder's head of family. It is an optional field and can be empty.
(2) **District**: The district the address is located in.
(3) **Landmark**: A landmark (if any) associated with the address.
(4) **House No.**: The house number.
(5) **Location**: The name of the area in the village/town/city the address is located in.
(6) **PIN code**: The six digit PIN code of the address.
(7) **Post office**: The post office nearest to the address.
(8) **State**: The state the address is located in.
(9) **Street**: The street name.
(10) **Sub-district**: The sub-district the address is located in.
(11) **VTC**: The village/town/city the address is located in.

4.8 Last Four Digits of Mobile Number

If the value of the mobile number and email indicator byte is 0 or 1 (see Section 4.2), this field is empty. Otherwise, it contains the 10-byte ASCII encoding of the string `XXXXXXABCD` where `ABCD` are the last four digits of the holder's mobile number. For example, if

the mobile number is 9123456789, then this field will be the ASCII encoding of the string `XXXXXX6789`.

4.9 Photo

This is a variable-length field containing a photo of the Aadhaar holder in JPEG2000 format [28]. The end of this field is indicated by a `0xD9` byte that corresponds to the End of Image (EOI) marker in the JPEG2000 format.

4.10 Masked Email Address

If the value of the mobile number and email indicator byte is 0 or 2 (see Section 4.2), this field is empty. Otherwise, it contains a variable-length ASCII encoding of a masked version of the holder's email address. We could not find an exact specification for the masking algorithm. From examples, we conjecture that if the holder's email address is `michael@gmail.com`, then this field is the ASCII encoding of the string `mxcxxxl@gxxxxxxxx`.

4.11 RSA Signature

The last 256 bytes of the Aadhaar QR code data contain an RSA signature on all the previous QR code data generated using the Aadhaar private key. The data is encoded for signing using the EMSA-PKCS1-v1_5 scheme [27] with SHA256 as the hash function. This is illustrated in Figure 1.

Let `qr` denote an array of n bytes containing the QR code data. We will use Python array notation to denote slices (subarrays) of `qr`. The notation `qr[start:stop]` for integers `start`, `stop` in the range $\{0, 1, \ldots, n\}$ with start $<$ stop will denote the bytes in `qr` with indices `start`, `start` + 1, ..., stop − 1.

The RSA signature is contained in the slice $\texttt{qr}[n-256:n]$. The data protected by the signature is in the slice $\texttt{qr}[0:n-256]$. Let `H` denote the 32-byte SHA256 hash of the bytes in $\texttt{qr}[0:n-256]$.

Let $\|$ denote the byte array concatenation operator. Then the EMSA-PKCS1-v1_5 encoding of the message $\texttt{qr}[0:n-256]$ is given by the 256-byte array

$$\texttt{EM} = \texttt{0x00} \,\|\, \texttt{0x01} \,\|\, \texttt{PS} \,\|\, \texttt{0x00} \,\|\, \texttt{T} \,\|\, \texttt{H}, \tag{1}$$

where `PS` is a 202-byte padding sequence of `0xFF` bytes and `T` is a 19-byte array indicating that the hash function used to compute the message digest `H` is SHA256. The value of `T` is given by

`0x3031300d060960864801650304020105000420`.

The Aadhaar public key consists of a 2048-bit modulus $\texttt{N}_{\text{rsa}}$ and an encryption exponent $e = 65537 = 2^{16} + 1$. The value of $\texttt{N}_{\text{rsa}}$ is given in Appendix A. Let d be the corresponding decryption exponent. The RSA signature on the message $\texttt{qr}[0:n-256]$ is given by the following 256-byte value

$$\text{sig} = \texttt{EM}^d \bmod \texttt{N}_{\text{rsa}}. \tag{2}$$

5 AGE PROOF DESIDERATA

We discuss some desirable properties for Aadhaar-based age proofs to motivate our proposed design. By an *adult Aadhaar holder*, we mean that the birth date in the holder's QR code is 18 or more years in the past from the current date.

We envision that age proofs will be used in the following manner. An adult Aadhaar holder will generate and submit an age proof to an **application**. The application could be a social media website which wants to restrict access to users under the age of 18. Or it could be a website that collects and stores personal data about its visitors. Consequently, it requires consent from a visitor allowing it collect their data and an age proof proving that the visitor is an adult making them eligible to give consent.

We will assume that an application has a unique 128-bit **application identifier (AppID)**.[1] An Aadhaar holder can use the AppID as input while generating an age proof, making it application-specific.

We would like Aadhaar-based age proofs to satisfy the following properties.

- **Soundness**: Only adult Aadhaar holders should be able to generate a valid age proof.
- **Privacy**: The age proof should not reveal any information about the fields in the Aadhaar QR code, apart from the fact that the holder is an adult. For example, it should not reveal the exact birth date or even birth year of the holder.
- **Intra-application linkability**: An application should be able to link multiple age proofs created by the same Aadhaar holder.
  While an application cannot prevent an adult Aadhaar holder from generating multiple age proofs and sharing them with others who are not adults, if it can identify that these multiple proofs are from the same holder then it can take appropriate action (like banning all requests from the holder or allowing only the first request).
- **Inter-application unlinkability**: Age proofs generated by the same holder for different applications (having distinct AppIDs) should not be linkable.
  This is to prevent cross-application tracking of holders by applications. For example, if the age proofs submitted by a holder to a news website and an ecommerce website can be linked, then the news website could display targeted ads to the holder based on their purchase history in the ecommerce website.
- **Low computational requirements**: Due to privacy concerns, an Aadhaar holder may not want to outsource the age proof generation to a third party. Hence Aadhaar holders should be able to generate the age proof of their own devices which could have limited computational resources (CPU, memory, storage). Specifically, it should be possible to generate the age proof on mobile phones.

6 UNIQUELY IDENTIFYING AN AADHAAR HOLDER

Note that the holder's QR code does not contain their 12-digit Aadhaar number in its entirety. Only its last four digits of appear in the reference ID field (see Section 4.3). This raises the following question: *When can we say that two QR codes belong to the same holder without using their unique 12-digit Aadhaar number?* This question is relevant for evaluating whether an age proof design satisfies

[1] Our scheme encodes the AppID as an element in the scalar field of an elliptic curve. The elliptic curves used by Nova so far can accommodate at least 253 bits in their scalar fields. For simplicity, we chose the largest power of 2 less than 253 for the bit length of the AppID. The same design choice was made by Anon Aadhaar, which uses the name *nullifier seed* for the AppID.

the intra-application linkability and inter-application unlinkability properties.

The only field in the QR code data that can be changed without much effort from the holder is the 17-byte QR code creation timestamp in the reference ID field. Every new download of the QR code will contain a new timestamp.

Changing the name, gender, date of birth, mobile number, email address, or photo requires the Aadhaar holder to visit an Aadhaar enrolment centre with supporting documentation [47]. Changing the address field can be done online by holders with a registered mobile number, if they can upload a valid proof of address.

One possibility is to identify an Aadhaar holder based on the triple of name, date of birth, and last four digits of Aadhaar number. In a country as populous as India, there is a small chance of two holders having the same values in these three fields. If the age proof scheme satisfies the intra-application linkability property based of this triple, then an honest Aadhaar holder may be unfairly denied access to the application due to having the same values in this triple as another holder.

Suppose we define uniqueness based on the address field in addition to the name, date of birth, and last four digits of Aadhaar number. Then an Aadhaar holder who changes their address will be able to generate two age proofs which will be considered as belonging to distinct holders by an application, even if the age proof scheme satisfies intra-application linkability. They will use the QR code with the old address to generate the first proof and the QR code with the new address to generate the second proof. This is possible because the Aadhaar system does not currently have a mechanism to *revoke* stale data about an Aadhaar holder.

We could enforce revocation of stale data by requiring the age proof to additionally prove that the timestamp in the reference ID field is in the recent past, for example, one month or less from the current date. With this constraint in place, when an Aadhaar holder changes their address they will not be able to generate a valid age proof using the old QR code one month after the address change has been accepted by the Aadhaar system. This design has two issues:

- Aadhaar holders who do not have their mobile number registered cannot download a fresh QR code. They only have access to the QR code on the physical copy of their Aadhaar card, which could be several years old. Mandating that the timestamp in the QR code must be in the recent past will prevent such holders from generating age proofs.
- Even if an Aadhaar holder has their mobile number registered, they will have to download a fresh QR code every month to be able to generate a valid age proof. This is even if none of their Aadhaar details have changed. While this is not an insurmountable obstacle, it degrades the user experience of the Aadhaar holders.

Since our goal is to demonstrate the feasibility of generating Aadhaar-based age proofs in resource-constrained environments, we defer a more careful design of the uniqueness criteria to future work. We make the following assumption for demonstration purposes.

Assumption 1. *Two QR codes belong to different Aadhaar holders if they differ in any of the fields except the timestamp in the reference ID field.*

7 DESIGN CONSIDERATIONS

In this section, we motivate the design of the statement which is proved by Nova Aadhaar. It is identical to the statement proved by the Anon Aadhaar scheme [6, 17], except for some minor differences which are listed at the end. We are essentially motivating the statement chosen by Anon Aadhaar, by first presenting two strawman designs that do not meet our requirements.

A major difference between Nova Aadhaar and Anon Aadhaar is the choice of ZK proof system used to prove the statement. Anon Aadhaar uses the Groth16 proof system [21], while we use Nova [32] to lower the computational requirements for generating age proofs. Switching the proof system to Nova is not straightforward, as demonstrated in Section 9. The main challenge is to translate a regular statement, which does not naturally have the structure of an incremental verifiable computation (IVC), into an IVC form.

7.1 Strawman Design 1

An adult Aadhaar holder can use their Aadhaar QR code data itself as the age proof. To verify the age proof, the verifier will first check that the RSA signature is valid using the Aadhaar public key. Then the verifier will check that the date of birth is 18 or more years in the past.

This design satisfies the soundness requirement since valid age proofs can only be obtained by adult Aadhaar holders. It also has low computational requirements as the holder does not need to perform any extra computation. They only need to download the QR code and share it with the verifier.

But this design is bad for privacy as the verifier sees the entire QR code data. Also, while it does satisfy the intra-application linkability property, it fails to satisfy the inter-application unlinkability property.

7.2 Strawman Design 2

Let $\mathtt{H}$ be the SHA256 hash of the QR code data except for the last 256 bytes (as defined in Section 4.11). Let sig be the RSA signature defined in equation (2).

An adult Aadhaar holder could generate an age proof as the triple $(\mathtt{H}, \text{sig}, \pi)$ where π is a zero-knowledge (ZK) proof that $\mathtt{H}$ is the SHA256 hash of some Aadhaar QR code data (excluding RSA signature) containing a date of birth which is 18 or more years in the past.

The age proof verifier performs the following checks.

(1) It first verifies that sig is a valid RSA signature on $\mathtt{H}$ using the Aadhaar RSA public key $\langle \mathtt{N}_{\text{rsa}}, e \rangle$. This involves checking that the following equation holds where $\mathtt{EM}$ is calculated from $\mathtt{H}$ using equation (1).

$$\text{sig}^e = \mathtt{EM} \bmod \mathtt{N}_{\text{rsa}}$$

(2) It then verifies that the ZK proof π is valid for the instance $\mathtt{H}$.

The RSA signature verification ensures that $\mathtt{H}$ is the SHA256 hash of valid Aadhaar QR code data (excluding signature). Consequently,

this design satisfies the soundness property as long as the ZK proof scheme satisfies its corresponding soundness property, i.e. it should be computationally infeasible to generate a valid proof π using QR code data whose SHA256 hash is $\mathtt{H}$ but it contains a date of birth which less than 18 years in the past.

This design preserves privacy as only the SHA256 hash $\mathtt{H}$ of the QR code data is revealed. Since the photo field (which is more than 1000 bytes long) is also hashed to generate $\mathtt{H}$, it is infeasible for an observer to recover the values in the QR code fields from $\mathtt{H}$ using a brute-force search.

This design does not satisfy the intra-application linkability property since the SHA256 hash changes with every new download of the QR code data. An adult Aadhaar holder can use two of his QR codes with different timestamps to generate two age proofs having different values for $\mathtt{H}$. To address this issue, we need to modify the design to reveal a hash $\mathtt{H}'$ of the data excluding the timestamp bytes.

This design also does not satisfy the inter-application unlinkability property since the age proof is independent of the AppID. Either the hash $\mathtt{H}$ or the RSA signature sig can be used to link the age proofs submitted by the same Aadhaar holder to different applications. To address this issue, we need to consider designs that do not reveal the values of $\mathtt{H}$ and sig. Additionally, the hash $\mathtt{H}'$ needed to solve the intra-application linkability issue must be calculated with AppID as part of the input.

7.3 Nova Aadhaar Statement

Let $\mathtt{qr}$ be an n-byte array representing the data in an Aadhaar QR code. Let $\mathtt{qr}_\mathrm{d}$ be the byte array of length $n-256$ which corresponds to the bytes in $\mathtt{qr}[0 : n-256]$, the QR code data excluding the RSA signature. Let $\mathtt{qr}_\mathrm{m}$ be the byte array corresponding to the bytes in $\mathtt{qr}_\mathrm{d}$ with the 17 timestamp bytes replaced with zero bytes (the subscript m indicates that the data is masked). So for $i = 0, 1, 2, \ldots, n-256$, we have

$$\mathtt{qr}_\mathrm{m}[i] = \begin{cases} \mathtt{qr}_\mathrm{d}[i] & \text{if } i \notin \{7, 8, \ldots, 23\}, \\ 0 & \text{if } i \in \{7, 8, \ldots, 23\}. \end{cases} \tag{3}$$

Let $\mathtt{A}_\mathrm{id}$ be the AppID of the target application for the age proof. We will assume that $\mathtt{A}_\mathrm{id}$ is a 128-bit integer. Let $\mathtt{H}_\mathrm{pos}$ denote the Poseidon hash function [20].

In Nova Aadhaar, an adult Aadhaar holder will generate an age proof as the tuple $(\mathtt{A}_\mathrm{id}, \sigma, \pi)$ where π is a ZK proof attesting to the following claims:

- $\sigma = \mathtt{H}_\mathrm{pos}(\mathtt{A}_\mathrm{id}, \mathtt{qr}_\mathrm{m})$ for some byte array $\mathtt{qr}_\mathrm{m}$.
- There exists a byte array $\mathtt{qr}_\mathrm{d}$ having the same length as $\mathtt{qr}_\mathrm{m}$ that satisfies the constraint in equation (3).
- The SHA256 hash of the byte array $\mathtt{qr}_\mathrm{d}$ is $\mathtt{H}$.
- The prover knows a signature sig such that the value $\mathtt{EM}$ obtained by substituting H into equation (1) satisfies the equation

$$\mathrm{sig}^e = \mathtt{EM} \bmod \mathtt{N}_\mathrm{rsa}. \tag{4}$$

- The date of birth in $\mathtt{qr}_\mathrm{m}$ is 18 or more years in the past.

This design satisfies the soundness property as long as the ZK proof scheme satisfies its corresponding soundness property. The proof π does not reveal any information beyond the validity of the claims. If we model the Poseidon hash function as a random oracle [7], the value of $\sigma = \mathtt{H}_\mathrm{pos}(\mathtt{A}_\mathrm{id}, \mathtt{qr}_\mathrm{m})$ is uniformly distributed in a large field and does not reveal information about $\mathtt{qr}_\mathrm{m}$. Hence this age proof design satisfies the privacy property.

The value σ will act as a **nullifier** of the Aadhaar holder's age proof. Age proofs generated for the same AppID using two QR codes which differ only in their timestamp bytes will result in the same nullifier value σ. This enables this design to satisfy the intra-application linkability property.

If we model the Poseidon hash function as a random oracle, the values of $\mathtt{H}_\mathrm{pos}(\mathtt{A}_\mathrm{id}, \mathtt{qr}_\mathrm{m})$ and $\mathtt{H}_\mathrm{pos}(\mathtt{A}'_\mathrm{id}, \mathtt{qr}_\mathrm{m})$ will be uniformly distributed and independent for $\mathtt{A}_\mathrm{id} \neq \mathtt{A}'_\mathrm{id}$. This will ensure that this design satisfies the inter-application unlinkability property. We make the following assumption.

Assumption 2. *The Poseidon hash function can be modeled as a random oracle.*

The above design is identical to the Anon Aadhaar scheme [3] except for a few minor differences.

- The Anon Aadhaar scheme calculates the nullifier as the Poseidon hash of a nullifier seed (which could be the application ID) and the photo bytes of the user [1]. In our design, we hash all the non-timestamp bytes in the QR code data (excluding RSA signature).
- The Anon Aadhaar scheme reveals the date and hour of the timestamp in the QR code data. The minutes, seconds, and milliseconds fields are excluded to avoid identifying the user [2].
- If requested by the user, the Anon Aadhaar scheme reveals the gender, state, and PIN code fields in the QR code.

8 NOVA

In this section, we cover aspects of the Nova proof system [32, 35] needed to describe our proposed age proof scheme. Nova is a recursive *zero-knowledge succinct non-interactive argument of knowledge (zkSNARK)* for statements that can be expressed as *incrementally verifiable computation (IVC)* instances [48].

8.1 Incrementally Verifiable Computation

Let $\mathbb{F}$ be a finite field. An *IVC instance* is given by (F, n, z_0, z_n) where $n \in \mathbb{N}$, $z_0, z_n \in \mathbb{F}^l$ for some $l \in \mathbb{N}$, and $F : \mathbb{F}^k \to \mathbb{F}^k$ for $k \geq l$ is a function called the **step function**.

An *IVC scheme* allows a prover to prove that for some public step function F, public initial input z_0, public final output z_n, it knows auxiliary input values $w_0, w_1, \ldots, w_{n-1} \in \mathbb{F}^{k-l}$ such that

$$z_n = F\left(F\left(\ldots F\left(F\left(F\left(z_0, w_0\right), w_1\right), w_2\right), \ldots\right), w_{n-1}\right). \tag{5}$$

Such a proof is generated by proving the execution of a series of incremental computations of the form $z_{i+1} = F(z_i, w_i)$, for each $i \in \{0, 1, \ldots, n-1\}$, where z_i and z_{i+1} are the public input and output in the ith step, respectively. A sequence of incremental computations is illustrated in Figure 2.

The values in $\mathbf{w} = [w_0, w_1, \ldots, w_{n-1}]$ constitute a *witness* for the IVC instance (F, n, z_0, z_n). The Nova proof system does not reveal the values in the witness.

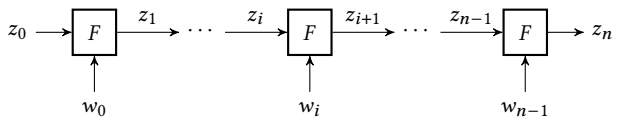


Figure 2: A sequence of incremental computations

8.2 Rank-1 Constraint Systems

To use the Nova proof system to prove an IVC instance, the step function F needs to be expressed as an instance of a *rank-1 constraint system (R1CS)* [44], [8, Appendix E]. For completeness, we give a definition of R1CS instances.

Definition 8.1. Let $\mathbb{F}$ be a finite field. A rank-1 constraint system (R1CS) instance is a tuple $(\mathbb{F}, A, B, C, io, m, n)$ where

- A, B, C are $m \times m$ matrices with entries from the field $\mathbb{F}$ with at most $n = \Omega(m)$ non-zero entries.
- io is a vector with entries from $\mathbb{F}$ representing the public input and output of the instance, whose length satisfies $|io| + 1 \leq m$.

Definition 8.2. An R1CS instance $(\mathbb{F}, A, B, C, io, m, n)$ is said to be *satisfiable* if there exists a witness $w \in \mathbb{F}^{m-|io|-1}$ such that

$$Au \circ Bu = Cu,$$

where $u = \begin{bmatrix} io & 1 & w \end{bmatrix}^T$ and $\circ$ is the Hadamard vector product operation.

To express the computation $z_{i+1} = F(z_i, w_i)$ as an R1CS instance, we need to find matrices A, B, C such that the equation $Au \circ Bu = Cu$ is satisfied for $io = \begin{bmatrix} z_i, z_{i+1} \end{bmatrix}$ and some $w \in \mathbb{F}^{m-|io|-1}$ if and only if there exists a witness w_i such that $z_{i+1} = F(z_i, w_i)$.

The equation in Definition 8.2 encodes m R1CS constraints in the field $\mathbb{F}$. Each constraint is a quadratic expression in the entries of u. Let $a_{i,j}, b_{i,j}, c_{i,j}$ denote the entries of the A, B, C matrices, respectively. Let u_j denote the jth entry of u. Then for $i = 1, 2, \ldots, m$, the ith R1CS constraint is given by

$$\left(\sum_{j=1}^{m} a_{i,j} u_j\right)\left(\sum_{j=1}^{m} b_{i,j} u_j\right) = \sum_{j=1}^{m} c_{i,j} u_j.$$

8.3 Nova Proof Costs

Let $|F|$ denote the number of R1CS constraints needed to express the computation of the step function F.

Prior to proof generation and verification, Nova requires a one-time generation of public parameters. These parameters are points from an elliptic curve group that are used to generate commitments to vectors of length $O(|F|)$. For the Pedersen commitment scheme (which is what we use in our implementation), the public parameters generation involves $O(|F|)$ steps. The generated parameters have a total size which is $O(|F|)$.

For n steps, the Nova proof generation time is $O(n|F|)$. At each step, the prover's computation is dominated by two multi-scalar multiplications (MSMs) of size $O(|F|)$ in the elliptic curve group. The memory required for these MSMs is $O(|F|)$.

The generated proof has size $O(\log |F|)$. If the Pedersen commitment scheme is used in Nova, the proof verification time is $O(|F|)$. Note that the proof size and verification time are independent of the number of steps n.

8.4 Nova Security Guarantees

The Nova proof system satisfies completeness, knowledge-soundness and zero-knowledge properties. The definitions of these properties are presented in Appendix B.

Informally speaking, the completeness property states that a prover who knows a valid witness $\mathbf{w}$ for an IVC instance (F, n, z_0, z_n) can always generate a valid proof. The knowledge-soundness property states that if a polynomial-time prover can generate a valid proof for an IVC instance (F, n, z_0, z_n) then it knows a valid witness $\mathbf{w}$ except with a negligible probability.

The zero-knowledge property states that a Nova proof π for an IVC instance (F, n, z_0, z_n) does not leak any information about the witness value $\mathbf{w}$. However, the instance contains the number of steps n which is revealed to the verifier. If the values in $\mathbf{w}$ are obtained by breaking a larger witness into fixed-size pieces, then n can reveal the size of the larger witness.

9 NOVA-BASED AGE PROOF

Recall the Nova Aadhaar statement presented in Section 7.3. The claims to be proved involve calculating the SHA256 hash of a byte array and performing a modular exponentiation. The Anon Aadhaar implementation [4] requires 1,115,080 R1CS constraints to express the age proof circuit (statement). Such a large number of constraints translates to high memory requirements for the prover device.

Our motivation for using Nova is to reduce the memory footprint of the prover, so that the age proof can be generated on a wider range of devices (specifically low-end mobile phones).

To represent the age proof as an IVC instance, we have to specify the tuple (F, n, z_0, z_n) where F is the step function, n is the number of IVC steps, and z_0, z_n are the initial input and final output of F respectively.

9.1 Key Insights

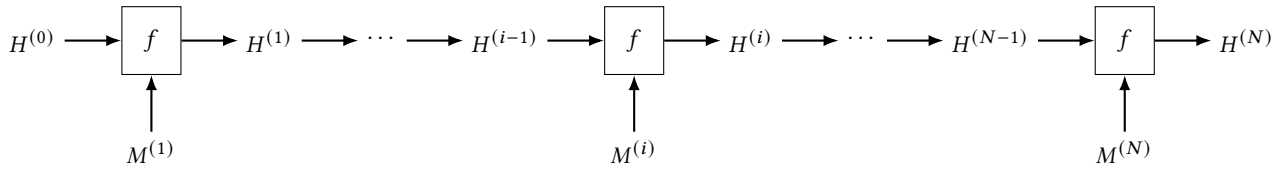
The first insight is that the SHA256 hash function [34] already has an incremental structure. At a high level, the SHA256 hash of a bitstring $M \in \{0,1\}^*$ having length less than 2^{64} bits is calculated as follows.

(1) The bitstring M is padded to a multiple of 512 bits to obtain M_{padded}.
(2) A 256-bit state variable $H^{(0)}$ is initialized using bits from the square roots of the first eight primes.
(3) M_{padded} is split into N blocks $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$ each having 512 bits (64 bytes).
(4) Using a compression function $f : \{0,1\}^{512} \times \{0,1\}^{256} \to \{0,1\}^{256}$, the values of $H^{(i)}$ are calculated as

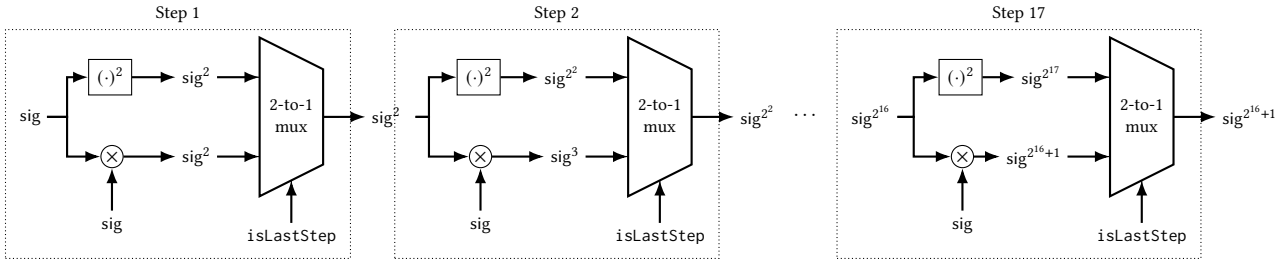
$$H^{(i)} = f\left(M^{(i)}, H^{(i-1)}\right), \tag{6}$$

for $i = 1, 2, \ldots, N$. This step is illustrated in Figure 3a.
(5) The 256-bit string $H^{(N)}$ is the SHA256 hash of M.

So it is possible to spread the SHA256 hash computation of the QR code data over multiple steps where each step invokes the compression function f.



(a) Computation of the SHA256 hash of a paddded message from its 512-bit blocks.



(b) Computation of RSA signature exponentiation using 17 identical steps.

Figure 3: Expressing SHA256 hashing and RSA signature exponentiation as incremental computations.

The second insight stems from the observation that the RSA exponent e in the Aadhaar public key has the value $65537 = 2^{16} + 1$. This means that the exponentiation sig^e in equation (4) can be spread over 17 steps where the first 16 steps perform a squaring operation and the last step performs a multiplication. Since the step function F in Nova has to be the same in every step, we can use a multiplexer to choose between the outputs of the squaring operation and product operations. This is illustrated in Figure 3b where `isLastStep` is a selector bit which selects the first or second input of the multiplexer depending on whether it is 0 or 1. For an example of a multiplexer implementation using R1CS constraints, see the `Mux1` circuit in `circomlib` [23].

The third insight is that the calculation of the SHA256 hash `H` and the eth power of the signature sig^e can proceed in parallel across the Nova steps. Once both these values are available, the value of `H` can be used to obtain the encoded message `EM` as in equation (1) and the equality in equation (4) can be checked.

9.2 Splitting the QR Code Data

In Nova Aadhaar, the age proof IVC instance will have $n = 17$ steps. All n steps involve the RSA signature modular exponentiation as illustrated in Figure 3b.

We could have computed one instance of the SHA256 compression function f as shown in equation (6) in each step, consuming 64 bytes of the QR code data per step. But then, depending on the length of the name field, the date of birth field would lie either in the first 64-byte block or the second 64-byte block, or span the boundary between these two blocks (see Figure 1). This complicates the logic needed to read the date of birth field.

While this is not an insurmountable obstacle, an additional disadvantage of processing 64 bytes per step is that the number of steps needed to compute the SHA256 hash of the QR code data can exceed 17. We empirically observed two instances of real-world Aadhaar QR code data (excluding the 256-byte RSA signature) having lengths 1064 and 1114 bytes. After the SHA256 padding, the first one would require 17 steps while the second one would require 18 steps to compute the SHA256 hash (if each step consumed 64 bytes). Thus the number of steps would leak information about the length of the QR code data.

Instead, we split the QR code data after SHA256 padding into 128-byte blocks and apply the SHA256 compression function f twice in each step. If the number of 64-byte blocks in the data after SHA256 padding is odd, we append a 64-byte block of zero bytes. Now the date of birth field lies in the first 128-byte block as long as the name field occupies 90 or fewer bytes (see Figure 1 with the caveat that there is a 1-byte delimiter after each of the first four fields). This approach does not leak the length of the QR code data as long as it has 17 or fewer 128-byte blocks after padding (data length after padding needs to be 2176 or fewer bytes). Empirically, we observed the number of 128-byte blocks to be 9 (pre-padding data length was between 1050 and 1150 bytes).

Thus, our proposed age proof scheme relies on the following assumptions.

Assumption 3. *The number of bytes in the QR code data after SHA256 padding is at most 2176 bytes.*

Assumption 4. *The number of bytes in the name field of the QR code data is at most 90.*

We have not been able to validate these assumptions. We hope to share our scheme with UIDAI authorities to get their feedback.

9.3 Opcodes

As the number of 128-byte blocks can be less than 17, SHA256 hashing need not be performed in all the 17 steps. We need to pass a flag across the Nova steps that controls whether SHA256 hashing needs to be performed at the current step. We define a 1-bit SHA256

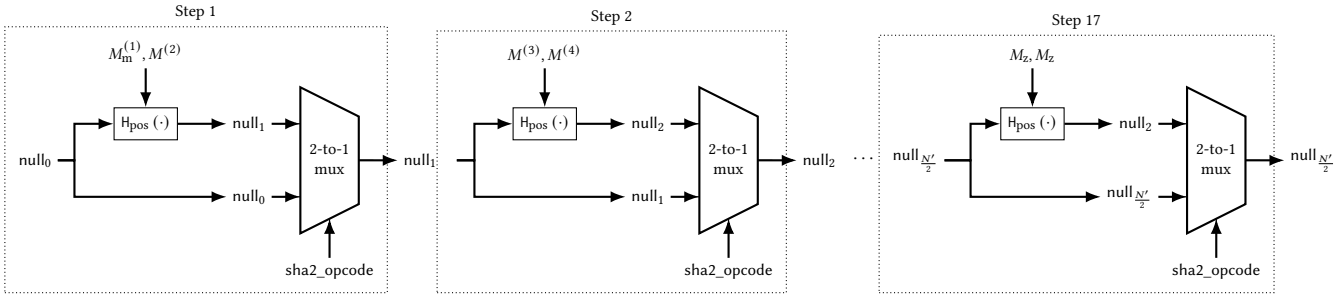


Figure 4: Illustration of the incremental calculation of a nullifier of the QR code data.

opcode as follows.

$$\text{sha2\_opcode} = \begin{cases} 0 & \text{if SHA256 hash needed in current step,} \\ 1 & \text{if SHA256 hash not needed in current step.} \end{cases}$$

Similarly, we need an opcode to control whether RSA signature verification needs to be performed in the current step. This opcode will also be used to decide which input to the 2-to-1 multiplexer is sent to the output in Figure 3b. We defined a 5-bit RSA opcode rsa_opcode which takes the value $i-1$ in step i. Since $i \in \{1, 2, \ldots, 17\}$, rsa_opcode can take values $0, 1, \ldots, 16$.

For efficiency, we combine these two opcodes into a single opcode as follows.

$$\text{opcode} = \text{rsa\_opcode} + 2^5 \times \text{sha2\_opcode}. \tag{7}$$

The constituent opcodes can be recovered from the combined opcode via a bit decomposition R1CS gadget [24].

9.4 Public Inputs/Outputs

The public input/output values z_i belong to $\mathbb{F}^2$ where $\mathbb{F}$ is a 255-bit prime field. The initial input is given by

$$z_0 = \begin{bmatrix} \text{initial\_opcode} & \text{current\_date} \end{bmatrix},$$

where initial_opcode has the value 0 corresponding to rsa_opcode = sha2_opcode = 0 in equation (7) and current_date is a serialized form of the date from which the age of the holder will be calculated.

In our scheme, if current_date is set to the present date, we obtain a proof that the Aadhaar holder is 18 years or older. By setting current_date to n years in the future, we obtain a proof that the Aadhaar holder is $18-n$ years or older. By setting current_date to n years in the past, we obtain a proof that the Aadhaar holder is $18+n$ years or older.

Let null_0 be the initial value of the nullifier which is set to the 128-bit AppID. The AppID could occupy 254 bits (as $\mathbb{F}$ is a 255-bit prime field) but we restrict it to 128 bits for simplicity.

Suppose the QR code data after padding has N blocks of 64 bytes each. If N is odd, then a 64-byte block of zero bytes is appended to this list. Let $M^{(1)}, M^{(2)}, \ldots, M^{(N')}$ be the list of blocks after the append step, where $N' = N+1$ if N is odd and $N' = N$ if N is even. Let $M_m^{(1)}$ be the first block with the timestamp bytes masked (they are replaced with zero bytes). Recall that null_0 is equal to the AppID. For $i = 1, 2, \ldots, 17$, the nullifier value after the ith step is given by

$$\text{null}_i = \begin{cases} H_{\text{pos}}\left(\text{null}_{i-1}, M_m^{(1)}, M^{(2)}\right) & \text{if } i = 1, \\ H_{\text{pos}}\left(\text{null}_{i-1}, M^{(2i-1)}, M^{(2i)}\right) & \text{if } 1 < i \leq \frac{N'}{2}, \\ \text{null}_{\frac{N'}{2}} & \text{otherwise.} \end{cases} \tag{8}$$

In words, the nullifier calculation proceeds as follows. In the first step, the nullifier value is the Poseidon hash of the AppID and the first two 64-byte message blocks with timestamp bytes masked. In the subsequent steps until step $\frac{N'}{2}$, the nullifier value is the Poseidon hash of the previous step's nullifier value and a pair of 64-byte message blocks. In step $\frac{N'}{2}$, if N is odd then the block $M^{(N')}$ will contain all zero bytes. In the steps after step $\frac{N'}{2}$, the nullifier value from step $\frac{N'}{2}$ will be output. Thus $\text{null}_{\frac{N'}{2}}$ will be the final nullifier of the QR code data. This illustrated in Figure 4 where the M_z inputs to the Poseidon hash function in step 17 are 64-byte blocks containing all zero bytes. These blocks represent dummy inputs to the Poseidon hash function after the message blocks have been exhausted.

Suppose that an honest prover is generating the proof. We will consider malicious provers later. For $i = 1, 2, \ldots, 16$, the output of the ith step will be given by

$$z_i = \begin{bmatrix} \text{opcode}_i & \text{io\_hash}_i \end{bmatrix},$$

where $\text{opcode}_i = i + 2^5 \times \text{sha2\_opcode}$ and the value of io_hash_i is given by

$$\text{io\_hash}_i = \begin{cases} H_{\text{pos}}\left(H^{(2i)}, \text{sig}, \text{sig}^{2^i}, \text{null}_i\right) & \text{if } i < \frac{N'}{2}, \\ H_{\text{pos}}\left(H^{(N)}, \text{sig}, \text{sig}^{2^i}, \text{null}_i\right) & \text{if } i \geq \frac{N'}{2}, \end{cases}$$

where $H^{(j)}$ is the output of the SHA256 compression function just after block $M^{(j)}$ has been processed in equation (6).

The role of io_hash_i is to carry forward the values of $H^{(j)}$, sig, sig^{2^i}, and null_i from step i to step $i+1$. These values are recovered from io_hash_i by the step function F by providing them as auxiliary inputs and checking that their hash matches io_hash_i.

The values $H^{(j)}$, sig, sig^{2^i}, null_i themselves could have been directly included in z_i. Hashing them and including a hash instead in z_i reduced the number of R1CS constraints.[2]

[2]Nova constructs an augmented step function F' from the step function F [32]. The number of constraints required to express F' increases with the size of the public IO vector z_i.

The final output, i.e. the output of the 17th step, is given by

$$z_{17} = \left[\text{final\_opcode} \quad \text{final\_nullifier}\right], \tag{9}$$

where final_nullifier equals the value $\text{null}_{\frac{N'}{2}}$ calculated in equation (8) and final_opcode = 49. The opcode in the 17th step has the value 48 corresponding to rsa_opcode = 16 and sha2_opcode = 1 in equation (7). The value of final_opcode is 1 more than the value of the opcode in the last step.

9.5 Auxiliary Inputs

For $i = 1, 2, \ldots, 17$, the step function receives the following as auxiliary inputs w_{i-1} in step i (see Figure 2). For each input, we indicate their values when an honest prover generates them. In the next subsection, we describe the step function F which will force malicious provers to set the same values as an honest prover.

(1) next_opcode: The value of the next step's opcode, i.e. the opcode in step $i + 1$.
(2) num_msg_blocks_odd: A boolean value which is set to true if the number of 64-byte blocks N in the QR code data after SHA256 padding is odd. Otherwise, it is set to false.
(3) dob_byte_index: An integer in the range $\{0, 1, 2, \ldots, 127\}$ that is equal to the location of the first byte of the date of birth field in the first 128-byte block of the QR code data.
(4) M_1, M_2 : A pair of 64-byte blocks.
Let $M^{(1)}, M^{(2)}, \ldots, M^{(N')}$ be the list of 64-byte blocks described in Section 9.4. This list is obtained by splitting the QR code data after SHA256 padding and appending a 64-byte block M_z of all zero bytes if num_msg_blocks_odd is true. So N' is always even and $M^{(N')} = M_z$ if num_msg_blocks_odd is true. The values of M_1, M_2 are given by

$$(M_1, M_2) = \begin{cases} \left(M^{(2i-1)}, M^{(2i)}\right) & \text{if } i \leq \frac{N'}{2}, \\ (M_z, M_z) & \text{if } \frac{N'}{2} < i \leq 17. \end{cases}$$

(5) current_sha256_digest: A 256-bit value which represents the SHA256 compression function output before the current step. For $j = 1, 2, \ldots, N$, let $H^{(j)}$ be the block $M^{(j)}$ has been processed in equation (6). Then we have

$$\text{current\_sha256\_digest} = \begin{cases} H^{(2i-2)} & \text{if } i \leq \frac{N'}{2}, \\ H^{(N)} & \text{if } \frac{N'}{2} < i \leq 17. \end{cases}$$

(6) sig: A 256-byte value corresponding to the RSA signature in the QR code data.
(7) sig_power: A 256-byte value corresponding to the current power of the RSA signature sig. For an honest power, it will have the value $\text{sig}^{2^{i-1}}$.
(8) current_nullifier: A field element corresponding to the current value of the nullifier, i.e. the value of the nullifier after step $i - 1$ given by null_{i-1}.

Note that the values of num_msg_blocks_odd, dob_byte_index, and sig do not vary with step index i.

9.6 Step Function Specification

The Nova step function for age proof verification is described in Algorithm 1. The description involves the following helper functions.

(1) `range_check`(x, m): Let $\mathbb{F}$ be a prime field whose cardinality occupies l bits. For $x \in \mathbb{F}$ and $1 \leq m < l$, this function checks that the binary representation of x can fit in m bits. An R1CS implementation of the range check gadget can be seen in `circomlib` [24].
(2) `decompose_opcode` (opcode): This function takes a 6-bit opcode as input and returns a pair $(\text{op}_1, \text{op}_2)$ where op_1 is a boolean value and op_2 is a 5-bit value.
(3) `assert` (predicate): This function adds an R1CS constraint that is satisfied if the predicate is true. The predicate is either an equality constraint or a boolean operation, both of can be asserted to be true using R1CS constraints [10].
(4) `create_flag` (predicate): This function returns a boolean variable whose value is true if and only if predicate is true.
(5) `cond_select`$(\text{flag}, \text{output}_1, \text{output}_2)$: This function returns output_1 if flag is true. It returns output_2 if flag is false. This is nothing but a 2-to-1 multiplexer.
(6) H_{pos} is the Poseidon hash function [20].
(7) `mask_timestamp_bytes`(M_1): This function takes a 64-byte block M_1 as input and outputs a 64-byte block $M_{1,\text{m}}$ which has the bytes in locations 8 to 24 zeroed out (like in equation (3)). If M_1 is the first 64 bytes of the Aadhaar QR code data, then the timestamp bytes are zeroed out.
(8) `emsa_pkcs1_v1_5`(H): This function takes a 256-bit value H as input and returns a 2048-bit value EM calculated as per equation (1).
(9) `check_age` (M, dob_byte_index, current_date): This function takes three inputs. The first input M is a 128-byte block. The second input dob_byte_index is equal to the index of the starting byte of the the date of birth field in the QR code data. It is an integer in the range $\{0, 1, 2, \ldots, 118\}$ if Assumption 4 holds. The third input is the current date. This function returns a boolean variable which is true if and only if the following conditions hold.
   (a) The number of delimiter bytes in M prior to dob_byte_index is exactly equal to 4 (see Section 4). These delimiter bytes are present after the version, mobile/email indicator, reference ID, and name fields.
   (b) The number of delimiter bytes in M in the range of indices from dob_byte_index to dob_byte_index + 9 is zero. Recall that the date of birth field is in the `DD-MM-YYYY` format (which occupies 10 bytes).
   (c) The bytes in M in the range of indices from dob_byte_index to dob_byte_index +9 correspond to a date in `DD-MM-YYYY` format that is 18 or more years in the past from current_date.

   An implementation of the `check_age` function can be found in our code repository.

9.7 Soundness of the Construction

A valid Nova proof for an IVC instance (F, n, z_0, z_n) only guarantees that there exist there exist auxiliary variables $w_0, w_1, \ldots, w_{n-1}$ such that equation (5) is satisfied. A malicious prover can supply arbitrary values for these variables in an attempt to forge a proof without having access to a valid Aadhaar QR code with an adult date of birth.

Algorithm 1: Step function algorithm for $z_i = F(z_{i-1}, w_{i-1})$ in step i

Input : If $i > 1$, public input $z_{i-1} = [\text{opcode}_{i-1}, \text{io\_hash}_{i-1}]$. If $i = 1$, then $z_{i-1} = z_0 = [\text{initial\_opcode}, \text{current\_date}]$.
Auxiliary input $w_{i-1} = \begin{bmatrix} \text{next\_opcode}, \text{num\_msg\_blocks\_odd}, \text{dob\_byte\_index}, \\ M_1, M_2, \text{current\_sha256\_digest}, \text{sig}, \text{sig\_power}, \text{current\_nullifier} \end{bmatrix}$

Output : If $i < 17$, public output $z_i = [\text{next\_opcode}, \text{next\_io\_hash}]$. If $i = 17$, then $z_i = [\text{next\_opcode}, \text{next\_nullifier}]$.

1 *// Check that current and next opcodes fit in 6 bits*;
2 range_check($opcode_{i-1}$, 6), range_check(*next_opcode*, 6)
3 *// Decompose current and next opcodes into constituent SHA256 and RSA opcodes. Note that SHA256 opcodes take boolean values.*;
4 (current_sha2_opcode, current_rsa_opcode) ← decompose_opcode($opcode_{i-1}$)
5 (next_sha2_opcode, next_rsa_opcode) ← decompose_opcode(*next_opcode*)
6 *// Next RSA opcode is always 1 more than current RSA opcode*;
7 assert(*next_rsa_opcode* == *current_rsa_opcode* + 1)
8 *// Either next_sha2_opcode == current_sha2_opcode OR next_sha2_opcode = current_sha2_opcode + 1*;
9 assert(¬*current_sha2_opcode* ∨ *next_sha2_opcode*)
10 *// Create flags*;
11 is_first_step ← create_flag(*current_rsa_opcode == 0*)
12 is_sha256_active ← create_flag(¬*current_sha2_opcode*)
13 is_opcode_last_sha256 ← create_flag(¬*current_sha2_opcode* ∧ *next_sha2_opcode*)
14 is_opcode_last_rsa ← create_flag(*current_rsa_opcode == 16*)
15 *// Check that the non-deterministic inputs hash to the expected value*;
16 calculated_io_hash ← H_{pos}(current_sha256_digest, sig, sig_power, current_nullifier)
17 assert(*is_first_step* ∨ (*calculated_io_hash* == io_hash_{i-1})) *// Hash equality does not hold in first step;*
18 *// Compute the next SHA256 digest using the compression function f*;
19 current_sha256_digest ← cond_select(is_first_step, $H^{(0)}$, current_sha256_digest) *// Overwrite SHA256 digest in first step with $H^{(0)}$;*
20 $H_1 \leftarrow f(M_1, \text{current\_sha256\_digest})$
21 $H_2 \leftarrow f(M_2, H_1)$
22 calculated_sha256_digest ← cond_select (is_opcode_last_sha256 ∧ num_msg_blocks_odd, H_1, H_2)
23 next_sha256_digest ← cond_select (is_sha256_active, calculated_sha256_digest, current_sha256_digest)
24 *// Compute the next nullifier*;
25 $M_{1,\text{m}} \leftarrow$ mask_timestamp_bytes(M_1)
26 $\text{temp\_nullifier}_1 \leftarrow \mathsf{H}_{\text{pos}}(\text{current\_nullifier}, M_{1,m}, M_2)$
27 $\text{temp\_nullifier}_2 \leftarrow \mathsf{H}_{\text{pos}}(\text{current\_nullifier}, M_1, M_2)$
28 calculated_nullifier ← cond_select $\left(\text{is\_first\_step}, \text{temp\_nullifier}_1, \text{temp\_nullifier}_2\right)$
29 next_nullifier ← cond_select (is_sha256_active, calculated_nullifier, current_nullifier)
30 *// Check validity of RSA signature if $i = 17$*;
31 sig_power ← cond_select (is_first_step, sig, sig_power) *// Overwrite sig_power in first step with sig;*
32 $\text{sig\_power\_square} \leftarrow (\text{sig\_power})^2 \bmod \mathsf{N}_{\text{rsa}}$
33 $\text{sig\_power\_times\_sig} \leftarrow (\text{sig\_power} \times \text{sig}) \bmod \mathsf{N}_{\text{rsa}}$
34 next_sig_power ← cond_select (is_opcode_last_rsa, sig_power_times_sig, sig_power_square)
35 EM ← emsa_pkcs1_v1_5(*next_sha256_digest*)
36 assert(¬*is_opcode_last_rsa* ∨ (*next_sig_power* == EM)) *// Signature power equals encoded message in last step;*
37 *// Check age in first step*;
38 is_age_above_eighteen ← check_age($M_1 \| M_2$, *dob_byte_index*, *current_date*)
39 assert(¬*is_first_step* ∨ *is_age_above_eighteen*) *// In the first step, age check must pass;*
40 *// Calculate output z_i*;
41 next_io_hash ← H_{pos}(next_sha256_digest, sig, next_sig_power, next_nullifier)
42 temp_output ← cond_select (is_opcode_last_rsa, next_nullifier, next_io_hash)
43 $z_i \leftarrow$ [next_opcode, temp_output]

We have to check that the constraints in the step function F ensure that a valid Nova proof implies that the prover used a valid Aadhaar QR code with an adult date of birth to generate it. In other words, we have to check that the soundness of Nova for this particular instance (F, n, z_0, z_n) implies the soundness of the age proof. We show that this implication holds under the following assumptions.

Assumption 5. *The RSA signature scheme with SHA256 hashing and EMSA-PKCS1-v1_5 encoding is unforgeable.*

Assumption 6. *The Poseidon hash function is collision-resistant.*

We consider the evolution of the state variables in Algorithm 1 and argue the soundness of the construction as a consequence. Line numbers in the following discussion refer to those in Algorithm 1.

9.7.1 RSA Opcode. First, let us consider the evolution of current_rsa_opcode and next_rsa_opcode.

- As per line 4, in step i the value of current_rsa_opcode is derived from opcode_{i-1} as

  $$(\text{current\_sha2\_opcode}, \text{current\_rsa\_opcode}) \leftarrow \text{decompose\_opcode}(\text{opcode}_{i-1}).$$

  From equation (7), it follows that current_rsa_opcode consists of the 5 least significant bits of opcode_{i-1}.
- As discussed in Section 9.4, $\text{opcode}_0 = \text{initial\_opcode} = 0$. This implies that the value of current_rsa_opcode in the first step ($i = 1$) is zero.
- As per line 7, we have

  $$\text{next\_rsa\_opcode} = \text{current\_rsa\_opcode} + 1,$$

  in all the steps. Specifically, the value of next_rsa_opcode in the first step is 1.
- As per line 5, in step i the 5 least significant bits of next_opcode are equal to next_rsa_opcode since

  $$(\text{next\_sha2\_opcode}, \text{next\_rsa\_opcode}) \leftarrow \text{decompose\_opcode}(\text{next\_opcode}).$$

- As per line 43, in every step the first coordinate of the public output z_i is set to next_opcode. Thus, as per line 4, the value of current_rsa_opcode in step $i + 1$ will equal the value of next_rsa_opcode from step i.
- In conclusion, the value of current_rsa_opcode in step i is equal to $i - 1$ for $i = 1, 2, 3, \ldots, 17$.

Consequently, we have the following implications on the is_first_step and is_opcode_last_rsa flags.

- As per line 11, the flag is_first_step is true only in the first step.
- As per line 14, the flag is_opcode_last_rsa is true only in step 17 (the last step).

9.7.2 Signature Powers. Let us consider the evolution of the values of sig_power and next_sig_power.

- As per line 31, the value of sig_power is replaced with sig if the flag is_first_step is true. The implication is that in the first step the value of sig_power provided by the prover will be replaced by the value it provided for sig.
- As per lines 32 and 34, the value of next_sig_power in the first step will be sig^2. This value is hashed in line 41 to get next_io_hash. In the first step, in line 42 temp_output gets this value of next_io_hash. By line 43, the second coordinate of the output of the first step is

  $$\text{io\_hash}_1 = \mathsf{H}_{\text{pos}}(\text{next\_sha256\_digest}, \text{sig}, \text{sig}^2, \text{next\_nullifier}).$$

- In the second step ($i = 2$), on line 17 the hash calculated in line 16 is checked for equality with io_hash_1. By the collision-resistance of the Poseidon hash function (Assumption 6), these hashes will be equal only if the value sig_power in the second step is sig^2. Furthermore, the value of sig provided by the prover must be equal to the value it provided in the first step. The prover cannot use different values of sig as inputs to the first and second steps.
- In the second step, on line 31 the value of sig_power is not overwritten. It retains the value sig^2. By lines 32 and 34, the value of next_sig_power becomes $\text{sig}^4 = \text{sig}^{2^2}$. This value is hashed into $\text{next\_io\_hash} = \text{io\_hash}_2$ on line 41 and is passed as output in temp_output in line 43.
- In the third step ($i = 3$), once again the hash calculated in line 16 is check for equality with io_hash_2. By the same argument as above, the value of sig_power in the third step must be sig^{2^2} and the value of sig is identical to its value in step 2.
- Hence for $i = 1, 2, 3, \ldots, 16$, in step i we have $\text{sig\_power} = \text{sig}^{2^{i-1}}$ and $\text{next\_sig\_power} = \text{sig}^{2^i}$.
- In step 17, the value of $\text{sig\_power} = \text{sig}^{2^{16}}$. But in line 34 the flag is_opcode_last_rsa is true. This implies that the value of next_sig_power is $\text{sig}^{2^{16}+1}$ in step 17.

In step 17, the assertion in line 36 requires that $\text{sig}^{2^{16}+1} = \mathsf{EM}$. By line 35, EM is the EMSA-PKCS1-v1_5 encoding of next_sha256_digest. If the RSA signature is unforgeable (Assumption 5), then a message whose SHA256 hash is the value of next_sha256_digest in step 17 must have been signed by the Aadhaar private key.

9.7.3 SHA256 Opcode. Let us consider the evolution of the values of current_sha2_opcode and next_sha2_opcode. Recall that decompose_opcode takes a 6-bit input and returns a pair of values, the first of which is a 1-bit value.

- As per lines 4 and 5, the variables current_sha2_opcode and next_sha2_opcode are boolean variables that can only take values 0 or 1.
- As discussed in Section 9.4, $\text{opcode}_0 = \text{initial\_opcode} = 0$. This implies that the value of current_sha2_opcode in the first step ($i = 1$) is 0.
- As per line 12, the flag is_sha256_active is set to true if and only if current_sha2_opcode is 0.
- As per line 9, if the value of current_sha2_opcode is 0, then next_sha2_opcode can be 0 or 1. But if current_sha2_opcode is 1, then the value of next_sha2_opcode must be 1.
  The rationale behind this constraint is that the SHA256 operation begins in an active state where the 64-byte blocks need to be hashed. Once the message blocks are exhausted, the SHA256 operation goes into an inactive state. Once it

has reached the inactive state (current_sha2_opcode = 1), it cannot go back to an active state (next_sha2_opcode = 0).
- As per line 13, the flag is_opcode_last_sha256 is set to true if and only if current_sha2_opcode is 0 and next_sha2_opcode is 1.

9.7.4 SHA256 Digests. Now, let us consider the computation of the intermediate and final SHA256 digests.

- In line 19, the value of current_sha256_digest is replaced with $H^{(0)}$ if the flag is_first_step is true, where $H^{(0)}$ is the initial state of the SHA256 hash function (see Section 9.1). The implication is that in the first step the value of current_sha256_digest provided by the prover will always be replaced. This ensures that the SHA256 hash computation always begins from the prescribed initial state.
- In the first step, in lines 20, 21 the compression function is applied to the provided blocks M_1, M_2. In line 22, the flag in the `cond_select` will be false until the last 128-byte block is reached. Assuming that the number of 128-byte blocks in the QR code data is more than one, calculated_sha256_digest is set to H_2. In line 23, next_sha256_digest is set to calculated_sha256_digest.
  In line 41, next_sha256_digest is hashed along with other variables to get next_io_hash. In the first step, in line 42 temp_output gets this value of next_io_hash. Let us call this value io_hash_1 as before.
- In the second step, on line 18 the hash calculated in line 17 is checked for equality with io_hash_1. By the collision-resistance of the Poseidon hash function, these hashes will be equal only if the value current_sha256_digest in the second step is equal to the next_sha256_digest from the first step. Thus the SHA256 digest computed in the first step is correctly recovered as the current SHA256 digest prior to SHA256 hashing in the second step. In a similar manner, the digest computed in the previous step is recovered in the current step.
- Suppose the current step corresponds to the last 128-byte block. There are two cases to consider. If the number of 64-byte blocks in the QR code data was odd, then a dummy 64-byte block of zero bytes would have been appended as the last block. If this is the case, then the value of next_sha256_digest should be H_1. This is handled by the `cond_select` in line 22. If the number of 64-byte blocks is even, then the value of next_sha256_digest is H_2.
- If the number of 128-byte blocks is exhausted, i.e. the flag is_sha256_active is equal to zero, then next_sha256_digest is simply set to current_sha256_digest as per line 23.

If a malicious prover attempts to cheat by providing incorrect values for M_1, M_2, or next_opcode, then the next_sha256_digest value in step 17 will deviate from the value which was signed by the Aadhaar private key. This will lead to an incorrect value of `EM` in line 35. As a consequence, the signature verification in line 36 will fail in step 17.

9.7.5 Nullifier Values. Finally, let us consider the evolution of the nullifier values current_nullifier and next_nullifier.

- In the first step, the value of current_nullifier is unconstrained. It can be set to any value by the prover. Hence it can be used to accommodate the AppID.
- As in the SHA256 digest calculation, the calculated_nullifier from a previous step is correctly recovered as the value of current_nullifier in the current step using the assertion in line 17.
- In line 25, the first 64-byte message block M_1 is masked at byte locations 8 to 24 to obtain $M_{1,m}$ as in equation (3). While this operation occurs in every step, in the first step the $M_{1,m}$ block corresponds to first 64 bytes of the Aadhaar QR code data with the timestamp bytes zeroed out.
- In lines 26, 27, two candidates for next_nullifier are calculated.
  - temp_nullifier_1 is the Poseidon hash of current_nullifier, the masked 64-byte block $M_{1,m}$, and M_2.
  - temp_nullifier_2 is the Poseidon hash of current_nullifier, the unmasked 64-byte block M_1, and M_2.
- As per line 28, in the first step, calculated_nullifier is set to temp_output_1. In subsequent steps, it is set to temp_output_2.
- If the number of 128-byte blocks is exhausted, the value of next_nullifier is simply set to current_nullifier in line 29.

If a malicious prover attempts to generate a different nullifier by providing incorrect values for M_1, M_2, or next_opcode, then the next_sha256_digest value in step 17 will again deviate and the signature verification in line 36 will fail.

9.7.6 Age Proof. In line 38, the dob_byte_index and current_date are used to check that the date of birth field in the concatenation $M_1 \| M_2$ of the blocks M_1, M_2 is 18 or more years in the past. Since these blocks will contain other data after the first step, the age is asserted to be above 18 only in the first step as per line 39.

9.7.7 Final Output. In step 17 (last step), in line 42 the value of temp_output is set to next_nullifier instead of next_io_hash. For the signature verification to pass, the SHA256 hash of the QR code must be complete. Hence the value of next_sha2_opcode must be 1. Since next_rsa_opcode has value 17, next_opcode = $32 \times 1 + 17 = 49$. These values are consistent with the value of z_{17} in equation (9).

10 IMPLEMENTATION AND PERFORMANCE

We used the reference implementation of Nova [19] to implement the Aadhaar-based age proof in about 2100 lines of Rust. Our code is released on GitHub [42] under MIT/Apache licenses. The step function F in our implementation requires about 104k R1CS constraints. On a laptop with an Intel i5-11320H processor [12] and 16 GB RAM, the public parameter generation (PPG), proof generation (PG), and proof verification (PV) times were 2.2 seconds, 13.8 seconds, and 0.2 seconds, respectively. The peak memory usage was 360 MB.

We created a web application (code hosted at [41]) for generating age proofs in the browser by compiling our Rust implementation to WASM [22]. An anonymized version of the application is deployed at https://age-proof.vercel.app/. In the Firefox browser on the test laptop, the PPG, PG, and PV times were 12 seconds, 108 seconds, and 10 seconds respectively. The peak memory usage was approximately 600 MB. In a mobile browser of a mid-range phone (Samsung Galaxy M31s), the times were 39 seconds, 585 seconds, and 42 seconds,

Scheme (Environ.)	PPG (s)	PG (s)	PV (s)	MC (GB)	ID (MB)
AA (iPhone 15)	NA	7	NA	1.4	> 600
AA (Laptop, browser)	NA	40	NA	9.6	307
PS (Laptop, shell)	2	14	3	0.36	NA
PS (Gal. M31s, browser)	39	585	42	-	1.5
PS (Gal. M31s, app)	8	50	9	0.58	12
PS (Pixel 8, browser)	11	118	13	-	1.5
PS (Pixel 8, app)	2	22	3	0.58	12

Table 1: Performance comparison of our proposed scheme (PS) and Anon Aadhaar (AA). Here PPG = public parameter generation, PG = proof generation, PV = proof verification, MC = Peak memory consumption during PG, ID = initial download. The NA indicates that either the functionality is not implemented or not needed. The memory consumption during PG in mobile browsers could not be measured.

respectively. On a high-end phone (Pixel 8), these times reduced to 11 seconds, 118 seconds, and 13 seconds, respectively. The initial download of the application (including WASM file) is under 1.5 MB. The proofs have a size of 15.3 KB and can be downloaded as a JSON file.

In comparison, the Anon Aadhaar web application [18] requires an initial download of about 307 MB, of which about 296 MB corresponds to the gzipped Groth16 structured reference string (SRS) and about 10.4 MB corresponds to the WASM file. On the same laptop, proof generation takes about 40 seconds with a peak memory usage of about 9.6 GB. Due to the high memory requirement, the Anon Aadhaar web application was disabled on mobile browsers until recently [16]. The proof can be downloaded as a JSON file with size 2.4 KB.

We have also created an Android app (code hosted at [43]) by generating Kotlin bindings to our Rust implementation [33]. The app can either scan a Aadhaar QR code using the camera or read a QR code image file from storage. It is available for download from the Google Play Store to a set of users whitelisted by us. The app download size is about 12 MB. The peak memory usage was 586 MB. On a Samsung Galaxy M31s, the PPG, PG, and PV times were 8 seconds, 50 seconds, and 9 seconds, respectively. On a Pixel 8, these times reduce to 2 seconds, 22 seconds, and 3 seconds, respectively.

The Anon Aadhaar team has created an iOS app (available only for private testing) which has a peak memory usage of 1.4 GB. After installation, it requires an initial download of 600 MB to retrieve the uncompressed Groth16 SRS [14]. The wide gap in the peak memory usage compared to the browser version could be because the iOS version uses Rapidsnark [25] while the browser version uses SnarkJS [26].

11 CONCLUSION

We described a design for an age proof scheme based on Aadhar QR codes which has lower memory and initial download requirements than Anon Aadhaar. Our experiments showcase the potential of Nova (and similar folding schemes) for client-side proof generation. On mobile phones, embedding the prover in a native application results in faster proof generation when compared to a browser (however the latter needs a smaller initial download). We expect that new techniques for folding schemes like Nebula [5] will further reduce proof generation times.

ACKNOWLEDGMENTS

We thank Ayush Modi for his work on the Rust-WASM-SvelteKit workflow during his internship at IITB Trust Lab. We would also like to thank Kumar Appaiah for sharing his Aadhaar QR code for testing, Yanis Meziane for discussions about the Anon Aadhaar performance, and Varun Thakore for discussions about Nova.

REFERENCES

[1] Anon Aadhaar. 2024. Nullifier Helper. https://github.com/anon-aadhaar/anon-aadhaar/blob/main/packages/circuits/src/helpers/nullifier.circom. Accessed: 2024-11-23.
[2] Anon Aadhaar. 2024. Timestamp Extractor. https://github.com/anon-aadhaar/anon-aadhaar/blob/8d6d38a52ee67e36286c1d1fa42a8952f8752fa7/packages/circuits/src/helpers/extractor.circom#L106. Accessed: 2024-11-23.
[3] Anon Aadhaar Documentation Team. 2024. How Does It Work? https://documentation.anon-aadhaar.pse.dev/docs/how-does-it-work Accessed: 2024-11-23.
[4] Anon Aadhaar Team. 2024. Anon Aadhaar GitHub Repository. https://github.com/anon-aadhaar/anon-aadhaar. Accessed: 2024-11-23.
[5] Arasu Arun and Srinath Setty. 2024. Nebula: Efficient read-write memory and switchboard circuits for folding schemes. Cryptology ePrint Archive, Paper 2024/1605. https://eprint.iacr.org/2024/1605
[6] PSE Team at Ethereum Foundation. 2024. Anon Aadhaar Project. https://pse.dev/en/projects/anon-aadhaar Accessed: 2024-11-23.
[7] Mihir Bellare and Phillip Rogaway. 1993. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*. ACM, 62–73. https://doi.org/10.1145/168588.168596
[8] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. 2013. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. Cryptology ePrint Archive, Paper 2013/507. https://eprint.iacr.org/2013/507
[9] Vint Cerf. 1969. ASCII Format for Network Interchange. https://www.rfc-editor.org/rfc/rfc20. Request for Comments 20.
[10] Argument Computer. 2024. Gadget for handling boolean constraints in Bellpepper. https://github.com/argumentcomputer/bellpepper/blob/main/crates/bellpepper-core/src/gadgets/boolean.rs. Accessed: 2024-11-23.
[11] Anon Aadhaar Project Contributors. 2024. Extractor Circom Code. https://github.com/anon-aadhaar/anon-aadhaar/blob/main/packages/circuits/src/helpers/extractor.circom. Accessed: 2024-11-23.
[12] Intel Corporation. 2024. Intel Core i5-11320H Processor (8M Cache, up to 4.50 GHz). https://www.intel.com/content/www/us/en/products/sku/217183/intel-core-i511320h-processor-8m-cache-up-to-4-50-ghz-with-ipu/specifications.html. Accessed: 2024-11-23.
[13] Oracle Corporation. 2023. SimpleDateFormat Class Documentation. https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/text/SimpleDateFormat.html. Accessed: 2024-11-23.
[14] Anon Aadhaar Developers. 2024. personal communication.
[15] Anon Aadhaar Developers. 2024. Anon Aadhaar Github repository. https://github.com/anon-aadhaar/anon-aadhaar/. Accessed: 2024-11-23.
[16] Anon Aadhaar Developers. 2024. anon-aadhaar Issue #236. https://github.com/anon-aadhaar/anon-aadhaar/issues/236. Accessed: 2024-11-23.
[17] Anon Aadhaar Developers. 2024. Anon Aadhaar Specs. https://github.com/zkspecs/zkspecs/blob/main/specs/2/README.md. Accessed: 2025-02-05.
[18] Anon Aadhaar Developers. 2024. Anon Aadhaar web application. https://anon-aadhaar-quick-setup.vercel.app/. Accessed: 2024-11-23.
[19] Nova Developers. 2024. Nova Github repository. https://github.com/microsoft/Nova. Accessed: 2024-11-23.
[20] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. 2021. Poseidon: A New Hash Function for Zero-Knowledge Proof Systems. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 519–535. https://www.usenix.org/conference/usenixsecurity21/presentation/grassi
[21] Jens Groth. 2016. On the Size of Pairing-Based Non-Interactive Arguments. In *Advances in Cryptology – EUROCRYPT 2016 (Lecture Notes in Computer Science, Vol. 9665)*. Springer, 305–326.
[22] WebAssembly Community Group. 2024. WebAssembly Documentation. https://webassembly.org/. Accessed: 2024-11-23.
[23] iden3. 2024. Circomlib - Mux1 Circuit Implementation. https://github.com/iden3/circomlib/blob/master/circuits/mux1.circom Accessed: 2024-11-23.

[24] iden3. 2024. Circomlib - Num2Bits Circuit Implementation. https://github.com/iden3/circomlib/blob/master/circuits/bitify.circom Accessed: 2024-11-23.
[25] Iden3. 2024. rapidsnark. https://github.com/iden3/rapidsnark. Accessed: 2024-11-23.
[26] Iden3. 2024. SNARK.js. https://github.com/iden3/snarkjs. Accessed: 2024-11-23.
[27] J. Jonsson and B. Kaliski. 2016. PKCS #1: RSA Cryptography Specifications Version 2.2, Section 9.2. Request for Comments (RFC) 8017. https://www.rfc-editor.org/rfc/rfc8017.html#section-9.2 Accessed: 2024-11-23.
[28] Joint Photographic Experts Group (JPEG). 2024. JPEG 2000: Image Coding System. https://jpeg.org/jpeg2000/. Accessed: 2024-11-23.
[29] Byron Kaye and Praveen Menon. 2024. Australia passes social media ban for children under 16. https://www.reuters.com/technology/australia-passes-social-media-ban-children-under-16-2024-11-28/. Accessed: 2024-11-29.
[30] Abhiram Kothapalli and Srinath Setty. 2023. HyperNova: Recursive arguments for customizable constraint systems. Cryptology ePrint Archive, Paper 2023/573. https://eprint.iacr.org/2023/573 https://eprint.iacr.org/2023/573.
[31] Abhiram Kothapalli and Srinath Setty. 2024. personal communication.
[32] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. 2022. Nova: Recursive Zero-Knowledge Arguments from Folding Schemes. In *CRYPTO 2022: 42nd Annual International Cryptology Conference, Santa Barbara, CA, USA.*
[33] Mozilla. 2024. UniFFI: Rust Library for Building Cross-Language Bindings. https://mozilla.github.io/uniffi-rs/latest/. Accessed: 2024-11-23.
[34] National Institute of Standards and Technology (NIST). 2015. *FIPS PUB 180-4: Secure Hash Standard (SHS).* Technical Report FIPS PUB 180-4. U.S. Department of Commerce, National Institute of Standards and Technology. Available at: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf.
[35] Wilson Nguyen, Dan Boneh, and Srinath Setty. 2023. Revisiting the Nova Proof System on a Cycle of Curves. Cryptology ePrint Archive, Paper 2023/969. https://eprint.iacr.org/2023/969
[36] Unique Identification Authority of India (UIDAI). 2024. mAadhaar FAQs. https://uidai.gov.in/en/contact-support/have-any-question/285-english-uk/faqs/your-aadhaar/maadhaar-faqs.html Accessed: 2024-11-23.
[37] Unique Identification Authority of India (UIDAI). 2024. Official Website of UIDAI. https://uidai.gov.in/en/ Accessed: 2024-11-23.
[38] D. M. Pierre. 2023. Anonymous Adhaar PDF Validity. https://anon-adhaar.vercel.app/. Accessed: 2024-11-23.
[39] D. M. Pierre. 2023. Anonymous Adhaar PDF Validity Github Repository. https://github.com/dmpierre/anon-adhaar. Accessed: 2024-11-23.
[40] Privacy & Scaling Explorations (PSE). 2024. Anon Aadhaar V2 Trusted Setup Ceremony. https://ceremony.pse.dev/projects/Anon%20Aadhaar%20V2%20Trusted%20Setup%20Ceremony. Accessed: 2024-11-23.
[41] Saravanan Vijayakumaran. 2024. Aadhaar-based Age Proof. https://github.com/avras/aadhaar-age-proof. Accessed: 2024-12-16.
[42] Saravanan Vijayakumaran. 2024. Age Proof from Aadhaar QR code. https://github.com/avras/nova-aadhaar-qr. Accessed: 2024-12-16.
[43] Saravanan Vijayakumaran. 2024. Android App for Aadhaar-based Age Proof. https://github.com/avras/aadhaar-age-proof-android. Accessed: 2024-12-16.
[44] Srinath Setty, Benjamin Braun, Victor Vu, Andrew J. Blumberg, Bryan Parno, and Michael Walfish. 2013. Resolving the conflict between generality and plausibility in verified computation. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 71–84. https://doi.org/10.1145/2465351.2465359
[45] Unique Identification Authority of India (UIDAI). 2019. *Secure QR Code Specification.* https://uidai.gov.in/images/resource/User_manulal_QR_Code_15032019.pdf Accessed: 2024-11-23.
[46] Unique Identification Authority of India (UIDAI). 2019. UIDAI website. https://uidai.gov.in Accessed: 2024-11-23.
[47] Unique Identification Authority of India (UIDAI). 2022. Document Update - Aadhaar Online Services FAQs. https://uidai.gov.in/en/contact-support/have-any-question/1061-english-uk/faqs/aadhaar-online-services/document-update.html Accessed: 2024-11-23.
[48] Paul Valiant. 2008. Incrementally Verifiable Computation or Proofs of Knowledge Imply Time/Space Efficiency. In *Proceedings of the 5th Conference on Theory of Cryptography* (New York, USA) *(TCC'08)*. 1–18.

A AADHAAR RSA PUBLIC KEY

The Aadhaar RSA public key consists of an encryption exponent $e = 65537$ and a 2048-bit modulus N_{rsa} given by the following bytes (in hexadecimal format).

$$\mathsf{N}_{\text{rsa}} =$$

```
a272 2c0e 5f2d e5ba 707e d93f 036e c2b1
f0b9 5593 2390 facc 0f0b cbf2 0c80 a15c
088b 4886 866b 72eb 25ab 0bf5 5a2f e06b
69ed bd5c c83b 74c7 709d b214 1e6c 07c6
8bcd f859 d3da f7f3 fced 241d 0720 55dc
1548 8474 b45c 2c98 2fa9 54aa 52aa 6fff
b248 1861 c650 85d8 4b64 158e 9f43 d9f3
5ca6 9b48 ce93 0052 2102 a4ca e093 bffd
474b 08d3 2d1d 0406 f687 f7e5 5bd2 26a2
384b f58a 41c3 84c3 974f 1c7c 5115 819c
926d e3ad f3ec bb99 04c4 86f1 f5d5 3039
77be a635 8436 7329 b5c1 68af dc95 1217
31f7 f48d 43af 7cf3 1f69 b1e3 bbe7 949d
c7a8 b10c 0bdd ebab abde bea0 76a1 cf81
66ad f06a 8a41 dedf 143b ff83 5dbd 2bd5
bb0b 61d2 472c 234a 6d44 11bc 9b53 6095
```

B ZKSNARK DEFINITIONS

In this section, we present the definition of a zero-knowledge succinct non-interactive argument of knowledge (zkSNARK) using notation from the Nova and HyperNova papers [30, 32]. Let $\lambda \in \mathbb{N}$ denote a security parameter. Let PPT and EPT denote probabilistic polynomial-time and expected polynomial-time, respectively.

Definition B.1 (Non-interactive Argument of Knowledge). Let $\mathcal{R}$ be a relation over public parameters pp, structure s, instance u, and witness w tuples. A non-interactive argument of knowledge for $\mathcal{R}$ consists of PPT algorithms $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ and deterministic $\mathcal{K}$, denoting the generator, the prover, the verifier, and the encoder, respectively, with the following interface:

- $\mathsf{pp} \leftarrow \mathcal{G}(1^\lambda)$: On input λ, $\mathcal{G}$ samples public parameters pp.
- $(\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{K}(\mathsf{pp}, \mathsf{s})$: On input s, representing common structure among instances, $\mathcal{K}$ outputs prover key pk and verifier key vk.
- $\pi \leftarrow \mathcal{P}(\mathsf{pk}, u, w)$: On input instance u and witness w, $\mathcal{P}$ outputs a proof π proving that $(\mathsf{pp}, \mathsf{s}, u, w) \in \mathcal{R}$.
- $1/0 \leftarrow \mathcal{V}(\mathsf{vk}, u, \pi)$: On input instance u and proof π, $\mathcal{V}$ verifies proof π for instance u. It outputs 1 if the proof verification succeeds and 0 otherwise.

Definition B.2 (Perfect Completeness). A non-interactive argument of knowledge $(\mathcal{G}, \mathcal{P}, \mathcal{V}, \mathcal{K})$ for relation $\mathcal{R}$ satisfies perfect completeness if for any PPT adversary $\mathcal{A}$ we have

$$\Pr\left[\begin{array}{l|l} \mathcal{V}\left(\mathsf{vk}, u, \pi\right) = 1 & \begin{array}{l}\mathsf{pp} \leftarrow \mathcal{G}(1^\lambda),\\ (\mathsf{s}, (u, w)) \leftarrow \mathcal{A}(\mathsf{pp}),\\ (\mathsf{pp}, \mathsf{s}, u, w) \in \mathcal{R},\\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{K}(\mathsf{pp}, \mathsf{s}),\\ \pi \leftarrow \mathcal{P}(\mathsf{pk}, u, w)\end{array} \end{array}\right] = 1.$$

In other words, if a non-interactive argument of knowledge for a relation $\mathcal{R}$ satisfies perfect completeness, then for every instance u in $\mathcal{R}$ the prover $\mathcal{P}$ can use the witness w to generate a proof π that will always be accepted by the verifier $\mathcal{V}$.

Definition B.3 (Knowledge-Soundness). A non-interactive argument of knowledge $(\mathcal{G}, \mathcal{P}, \mathcal{V}, \mathcal{K})$ for relation $\mathcal{R}$ satisfies knowledge-soundness if for all EPT adversaries $\mathcal{A}$ there exists an EPT extractor

$\mathcal{E}$ such that for all randomness ρ we have

$$\Pr\left[\begin{array}{l|l} \mathcal{V}\left(\mathsf{vk}, u, \pi\right) = 1, & \mathsf{pp} \leftarrow \mathcal{G}(1^\lambda), \\ (\mathsf{pp}, \mathsf{s}, u, w) \notin \mathcal{R} & (\mathsf{s}, u, \pi) \leftarrow \mathcal{A}(\mathsf{pp}; \rho), \\ & (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{K}(\mathsf{pp}, \mathsf{s}), \\ & w \leftarrow \mathcal{E}\left(\mathsf{pp}, \rho\right) \end{array}\right] \leq \mathsf{negl}(\lambda).$$

In other words, if a non-interactive argument of knowledge for a relation $\mathcal{R}$ satisfies knowledge-soundness, then if an adversary $\mathcal{A}$ can generate a valid proof π for an instance u then it must know a witness w such that (u, w) is a valid instance-witness pair in $\mathcal{R}$.

To define the notion of zero-knowledge, we need to first define computational indistinguishability.

Definition B.4 (Computational Indistinguishability). Let X_λ and Y_λ be two sequences of distributions ranging over $\{0,1\}^{p(\lambda)}$ for a polynomial p. We say that X_λ and Y_λ are computationally indistinguishable, denoted by $X_\lambda \approx Y_\lambda$, if for any PPT adversary $\mathcal{A}$ we have

$$\left|\Pr_{x \leftarrow X_\lambda}\left[\mathcal{A}(x) = 1\right] - \Pr_{y \leftarrow Y_\lambda}\left[\mathcal{A}(y) = 1\right]\right| \leq \mathsf{negl}(\lambda).$$

We adapt the definition of zero-knowledge for an interactive argument of knowledge given in the HyperNova paper [30, Definition 26] to the non-interactive setting to obtain the following definition. The Nova proof system satisfies this definition of zero-knowledge [31].

Definition B.5 (Zero-Knowledge). A non-interactive argument of knowledge $(\mathcal{G}, \mathcal{P}, \mathcal{V}, \mathcal{K})$ for relation $\mathcal{R}$ satisfies zero-knowledge if there exists an EPT simulator $\mathcal{S}$ such that for any PPT adversary $\mathcal{A}$ we have

$$\left\{\left(\mathsf{pp}, \mathsf{s}, u, \pi, \mathsf{st}\right) \middle| \begin{array}{l} \mathsf{pp} \leftarrow \mathcal{G}(1^\lambda), \\ (\mathsf{s}, (u, w), \mathsf{st}) \leftarrow \mathcal{A}(\mathsf{pp}), \\ (\mathsf{pp}, \mathsf{s}, u, w) \in \mathcal{R}, \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{K}(\mathsf{pp}, \mathsf{s}), \\ \pi \leftarrow \mathcal{P}\left(\mathsf{pk}, u, w\right) \end{array}\right\}$$

$$\approx \left\{\left(\mathsf{pp}, \mathsf{s}, u, \pi, \mathsf{st}\right) \middle| \begin{array}{l} \mathsf{pp} \leftarrow \mathcal{G}(1^\lambda), \\ (\mathsf{s}, (u, w), \mathsf{st}) \leftarrow \mathcal{A}(\mathsf{pp}), \\ (\mathsf{pp}, \mathsf{s}, u, w) \in \mathcal{R}, \\ \pi \leftarrow \mathcal{S}\left(\mathsf{pp}, \mathsf{s}, u, \mathsf{st}\right) \end{array}\right\}.$$

Here st is any auxiliary input available to the verifier.

In other words, if a non-interactive argument of knowledge for a relation $\mathcal{R}$ satisfies zero-knowledge, then for any PPT adversary $\mathcal{A}$ that generates an instance-witness pair (u, w) in $\mathcal{R}$ and auxiliary information st there exists an EPT simulator $\mathcal{S}$ that can generate a simulated proof π^{sim} using only the public parameters pp, structure s, instance u and auxiliary information st such that the joint distributions of $(\mathsf{pp}, \mathsf{s}, u, \pi^{\mathrm{act}}, \mathsf{st})$ and $(\mathsf{pp}, \mathsf{s}, u, \pi^{\mathrm{sim}}, \mathsf{st})$ are computationally indistinguishable, where π^{act} is the actual proof generated by an honest prover.

Definition B.6 (Succinctness). A non-interactive argument of knowledge $(\mathcal{G}, \mathcal{P}, \mathcal{V}, \mathcal{K})$ for relation $\mathcal{R}$ is succinct if the size of the proof π and verifier running time are at most polylogarithmic in the size of the structure s and witness w.