zkLogin: Privacy-Preserving Blockchain Authentication with Existing Credentials

Foteini Baldimtsi
Mysten Labs, George Mason University

Konstantinos Kryptos Chalkias
Mysten Labs

Yan Ji
Cornell University

Jonas Lindstrøm
Mysten Labs

Deepak Maram
Mysten Labs

Ben Riva
Mysten Labs

Arnab Roy
Mysten Labs

Mahdi Sedaghat
imec-COSIC, KU Leuven

Joy Wang
Mysten Labs

***Abstract*—For many users, a private key based wallet serves as the primary entry point to blockchains. Commonly recommended wallet authentication methods, such as mnemonics or hardware wallets, can be cumbersome. This difficulty in user onboarding has significantly hindered the adoption of blockchain-based applications.**

We develop zkLogin, a novel technique that leverages identity tokens issued by popular platforms (any OpenID Connect enabled platform e.g. Google, Facebook, etc.) to authenticate transactions. At the heart of zkLogin lies a signature scheme allowing the signer to *sign using their existing OpenID accounts* and nothing else. This improves the user experience significantly as users do not need to remember a new secret and can reuse their existing accounts.

zkLogin provides strong security and privacy guarantees. By design, zkLogin builds on top of the underlying platform's authentication mechanisms, and derives its security from there. Unlike prior related works however, zkLogin avoids the use of additional trusted parties (e.g., trusted hardware or oracles) for its security guarantees. zkLogin leverages zero-knowledge proofs (ZKP) to ensure that the link between a user's off-chain and on-chain identities is hidden, even from the platform itself.

The signature scheme at the heart of zkLogin enables a number of important applications outside blockchains. Most fundamentally, it allows billions of users to produce *verifiable digital content leveraging their existing digital identities*, e.g., email address. For example, a journalist can use zkLogin to sign a news article with their email address, allowing verification of the article's authorship by any party.

We have implemented and deployed zkLogin on the Sui blockchain as an alternative to traditional digital signature-based addresses. Due to the ease of web3 on-boarding just with social login, without requiring mnemonics, many hundreds of thousands zkLogin accounts have already been generated in various industries such as gaming, DeFi, direct payments, NFT collections, ride sharing, sports racing, cultural heritage, construction and electricity sectors and many more.

1. Introduction

Blockchains are decentralized ledgers maintained by a network of validators or miners. The blockchain ledger functions as an append-only record, logging transactions in a secure and immutable manner. In existing designs, each user is equipped with a unique pair of cryptographic keys: a private key and a public key. The private key of a user essential holds the user's assets and is used to execute transactions. To initiate a transaction, a user digitally signs it using their private key, and validators can confirm the validity of the signed transaction using the corresponding public key. Once verified, transactions are permanently added to the blockchain.

Users can opt to store their blockchain secret keys in a self-managed, or else non-custodial wallet. While this option gives full control to users, it also comes with the responsibility to store, manage, and secure their private keys. If a private key is lost, the associated assets are no longer retrievable. As a result, users often resort to custodial services for private key management. While these platforms offer a more intuitive user experience reminiscent of traditional online platforms, their reliability is contentious. The downfall of notable custodial firms [1], [2], [3], [4], whether due to mismanagement, security hacks or fraud, has made it difficult for users to place faith in emerging entities.

A potential resolution to this predicament is to leverage the existing trust that users have in globally recognized platforms, e.g. Google, Apple etc. The ubiquity and acceptance of standards like OAuth 2.0, which allow for the use of an existing account from one platform to authenticate on another, could serve as a direct gateway for integrating users of their platforms into the blockchain ecosystem.

However, a direct use of OAuth requires the introduction of a new trusted party for authentication purposes. Specifically, the OAuth protocol requires a web server (an OAuth client) to retrieve user-specific details from the OAuth Provider, like Google. However, since a blockchain cannot function as an OAuth client, this model would necessitate the introduction of a trusted web server, functioning as an oracle, to relay pertinent information to the blockchain.

This scenario naturally leads to a pivotal question: Can we harness existing authentication systems to oversee a cryptocurrency wallet, *without necessitating reliance on additional trusted entities*?

We answer the above question in the affirmative. Our approach relies on a crucial detail of prevalent OAuth 2.0 implementations. A substantial number of recognized providers employ OpenID Connect [5], a fundamental identity layer

```
{
  "sub": "1234567890", # User ID
  "iss": "google.com", # Issuer ID
  "aud": "4074087", # Client or App ID
  "iat": 1676415809, # Issuance time
  "exp": 1676419409, # Expiry time
  "name": "John Doe",
  "email": "john.doe@gmail.com",
  "nonce": "7-VU9fuWeWtgDLHmVJ2UtRrine8"
}
```

Listing 1: JWT Payload.

built upon the OAuth 2.0 protocol. OpenID providers (OP) issue a signed statement, referred to as a JSON Web Token (JWT). An example JWT is in Fig. 1. The JWT's payload contains basic user information, as shown in the example payload of Listing 1.

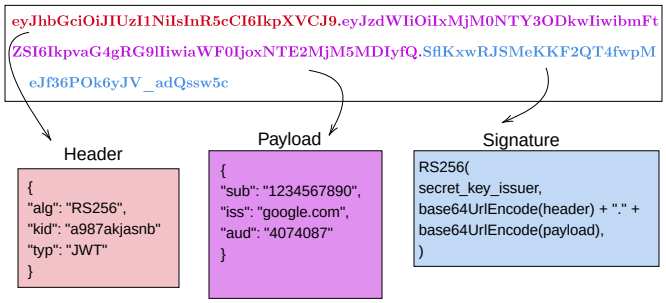

Figure 1: JSON Web Token.

The main idea in zkLogin is to utilize the JWT's signature to directly authenticate the user with a blockchain, thus eliminating the need for any middlemen.

A strawman way of realizing this would be as follows.

1) The user logs in to their existing account (on some OP like Google), leveraging OpenID Connect to obtain a JWT.
2) The JWT is sent on the blockchain, e.g., to a smart contract.
3) The embedded signature within the JWT facilitates its verification. Note that the smart contract would need to store the public keys of the said OpenID provider to be able to verify the JWT.
4) The smart contract can employ the persistent subject identifier (sub) present in the JWT to be able to identify the same user across different sessions.

A similar approach was previously proposed in [6]. While this can work[1], the main problem is that it reveals the entire JWT payload publicly, including sensitive claims such as name, email, profile image, etc. The above solution also does not show how the user can authorize a blockchain transaction, which is needed to truly realize a wallet.

1. It's vital to acknowledge that this preliminary solution is susceptible to frontrunning attacks as outlined in [6].

1.1. The zkLogin Approach

A natural way to avoid revealing the entire JWT is to leverage Zero-Knowledge Proofs (ZKP) [7]. In particular, the user can input the JWT, which we define by J, as a private witness and prove that J contains a valid signature issued by the OpenID provider among other things.

A few practical challenges arise when realizing the above idea. Existing JWTs use traditional cryptographic primitives like SHA-2 and RSA signatures which are not ZK-friendly. In addition, most state-of-the-art Zero-Knowledge Proving systems incur high computational-overhead for proving (focusing more on reducing the verification complexity). This state of affairs implies that we will likely need to employ powerful hardware to be able to generate proofs efficiently. But in our setting, the proving entity is the user – which means that the ZKP would need to be generated in resource-constrained environments, e.g., poor hardware / browsers, thus making it impractical.

If naïvely done, we'd have to generate a new ZKP for every transaction that the user signs (an approach taken by a recent work [8]). This further compounds the previous issue.

A simple trick helps us overcome the above challenges. Before getting a JWT, the user generates an ephemeral key pair (sk_u, vk_u) and implants the public key vk_u into the nonce during the OpenID Connect protocol (OpenID uses nonce to prevent replay attacks). The signed JWT J thus acts as a certificate for the ephemeral public key, and we can reuse the corresponding private key sk_u to sign any number of transactions.

Implanting public key into the nonce allows authorizing transactions. The zkLogin-signature on a transaction tx contains two things: (1) a ZKP proving validity of a JWT J and showing that it contains vk_u in its nonce, and (2) a traditional digital signature on tx with sk_u.

Moreover, the above trick also allows us to reuse the same ZKP to sign any number of transactions – thus amortizing the cost of a ZKP generation across many transactions. The ephemeral key pair can be deleted at an appropriate time, e.g., after a browsing session ends.

While the above can help reduce the number of times a ZKP needs to be generated, the user still needs to run the costly proof generation process once in a while. This is not ideal because we find that proving moderately complex ZKPs (e.g., around 1M constraints in Groth16) is not practical on a browser, leading to either crashes or delays in the order of minutes which is unacceptable for a good UX.

We solve this by *offloading the proof generation* to a different server in a way that this entity *need not be trusted*. In particular, notice that without knowledge of the ephemeral private key sk_u, an untrusted entity cannot generate a complete zkLogin signature. Therefore, we offload the Zero-Knowledge Proof generation to a server, and once the server returns the proof, the user can verify it efficiently (since most general-purpose ZKPs have fast verification) and complete the zkLogin signature.

To summarize, the idea of embedding data into the nonce helps us solve three challenges, namely, (a) authorize trans-

actions, (b) reuse a single ZKP across many transactions, and (c) offload ZKP generation securely.

Identifying the user on-chain: While using a ZKP can hide most of the sensitive information in a JWT, one more challenge remains. Any authentication system needs a way to persistently identify a user across multiple sessions. In today's private-key based wallets, this role is fulfilled by the public key which gets used to derive a user's blockchain address.

In zkLogin, a unique and persistent user identifier from the JWT can be used to generate a user address. We call such an identifier as a "stable identifier". A few possible options for a stable identifier include the subject identifier (sub), email address or username.

1.2. zkLogin features

Using a widely used identifier like email as the stable identifier makes the zkLogin account *easily discoverable*. This can be useful for entities wanting to maintain a public blockchain profile, e.g., a journalist or a financial organization such as an exchange for transparency reasons. Prior to zkLogin, this was only possible through the use of a trusted oracle.

Discoverability, however, comes with an inherent privacy problem as the link between the user's stable identifier and a blockchain address is forever public. So it must be used carefully.

Therefore, we do not make zkLogin accounts discoverable by default. Instead, we use an additional randomizer in the form of a "salt" to hide the user's off-chain identity. A user's address is a hash of the stable identifier, salt and a few other fields (e.g., the OpenID Provider's and the application's unique IDs). The use of a salt helps *unlink* a zkLogin address from the corresponding off-chain identity enhancing privacy. A key consideration is who manages the salt: the two main choices are the application (no unlinkability from the app but better UX) or the user (relatively worse UX but strong unlinkability).

zkLogin can also be used to create *claimable blockchain accounts*, i.e., safely sending claimable assets to new users. A sender can derive the receiver's zkLogin address using the receiver's email address and a randomly chosen salt. The salt can be sent to the receiver over a personal channel, e.g., over their email. This feature was not possible before without revealing the receiver's private key to the sender, which is undesirable.

1.3. Technical Challenges

Key rotation: Most OpenID Providers frequently rotate the keys used to sign JWTs for security reasons, e.g., Google rotates their keys once a week. So in zkLogin, we require that blockchain validators periodically fetch the latest verification key (or alternatively rely on an oracle).

Expiring the ephemeral keys: In practice, it is prudent to set a short expiry time for the ephemeral key pair vk_u for security reasons. A first idea is to use the JWT expiry time, e.g., the "exp" claim in Listing 1. However, this is not ideal because applications may want more control over its expiry, e.g., many JWTs expire 1hr after issuance which may be too small. Moreover, it is challenging to use real time in blockchains that skip consensus for certain transactions [9].

zkLogin facilitates setting an arbitrary expiry time exp by embedding it into the nonce. For example, if a blockchain publishes a block once every 10 mins and the current block number is cur, and we'd like to set an expiry of two hours, then set $exp = cur + 12$ and compute $nonce = H(vk_u, exp)$. Note that it is convenient to use the chain's local notion of time, e.g., block or epoch numbers, than the real time. More broadly, arbitrary policy information governing the use of the ephemeral key vk_u can be embedded into the nonce, e.g., permissions on what can be signed with vk_u.

Formalization: zkLogin closely resembles Signatures of Knowledge [10] where the knowledge of a witness is enough to produce a valid signature. The key difference in zkLogin is that witnesses (JWT and ephemeral key pair) expire.

We capture this property by proposing a novel cryptographic primitive called *Tagged Witness Signature*. The Tagged Witness Signature syntax provides an interface for a user to sign a message by demonstrating that it can obtain a secret witness, namely a JWT, to a public "tag", namely the OpenID Provider's public key. A Tagged Witness Signature has two main properties: unforgeability and privacy. Unforgeability states that it is hard to adversarially forge a signature, even if witnesses to other tags get leaked (e.g., expired JWTs). Privacy states that it is hard for an adversary to learn non-public components of the witness, e.g., the JWT and the salt, from the signature.

Implementation: We instantiate the Zero-Knowledge Proof using Groth16 [11] as the proving system and circom DSL [12] as the circuit specification language (Sec. 5).

The main circuit operations are RSA signature verification and parsing the JWT to read relevant claims (e.g., "sub", "nonce") from the payload. We use previously optimized circuits for RSA verification and write our own for parsing the JWT.

Naïvely parsing the JWT would've needed to Base64 decode the entire JWT and fully parse the resulting JSON. The latter would've also required implementing a complete JSON parser in R1CS. We manage to optimize significantly due to the observation that JSONs used in JWTs follow a simpler grammar. In particular, all the claims of interest, e.g., "sub" are simple JSON key-value pairs. This observation allows us to *only parse specific parts of the JWT payload*, namely the JSON key-value pairs, without looking at rest of its contents to a large extent.

Our final circuit has around a million constraints. SHA-2 is the most expensive taking 74% of the constraints and RSA big integer operations are the second-most expensive at 15%. Thanks to above optimizations, the JWT parsing circuit only takes the remaining 11% (115k constraints) whereas a naïve implementation would've resulted in significantly more.

1.4. Other applications: Content Credentials

The core primitive in zkLogin can be viewed as an Identity-Based signature (IBS) [13]. In an IBS, a key distribution authority issues a signing key over a user's identity id, e.g., their email address, such that a user-generated signature can be verified using just their identity id, thereby eliminating the necessity for a traditional Public Key Infrastructure (PKI).

zkLogin can be viewed as an IBS where the *OpenID provider implicitly functions as the key distribution authority*.[2] To our knowledge, this represents the first large-scale realization of an IBS completely avoiding the cost of setting up a new key distribution authority.

This enables a number of critical applications. With the rise of generative AI, knowing the authenticity of content, e.g., emails, documents or text messages, has become challenging [14]. A recent proposal by major technology firms attempts to establish provenance via *content credentials* [15], a cryptographic signature attached to a piece of digital content, e.g., news article, photos or videos. Issuing content credentials requires setting up a new PKI.

Our IBS scheme facilitates creating content credentials without having to setup a new PKI. For instance, a journalist could digitally sign a news article using their existing email address or a photographer may sign a photo using their existing Facebook account. Thus zkLogin makes it easy to onboard content creators. Another important benefit is that zkLogin-derived content credentials are tied directly to the creator's existing digital identity.

A variation of the IBS scheme allows for ring signatures, useful when additional privacy for the content creator is desired. Assuming email address is the stable identifier, the idea is to use only the email address domain as the user's identity. For example, the domain of the email "ram@example.com" is "example.com". Such signatures attest to the individual's affiliation with a specific organization, like a news outlet or educational institution, while maintaining their anonymity. This feature is useful in applications ranging from credential verification and whistleblowing to reputation systems.

1.5. Contributions

In summary, our contributions are as follows:

1) We propose zkLogin, a novel approach to the design of a blockchain wallet that offers significantly better user experience than traditional wallets, thanks to its use of well-established authentication methods.
2) We introduce the notion of tagged witness signatures to formally capture zkLogin, and prove their security.
3) We introduce novel features of zkLogin like unlinkability, discoverability, partial reveal, anonymous accounts and claimability.
4) We implement zkLogin using Groth16 as the NIZK in just 1M R1CS constraints, thanks to several circuit optimizations, e.g., efficient JSON parsing, string slicing, that maybe of independent interest. Generating a zkLogin signature only takes about 2.5-3s.

Structure of the Paper: We start off the rest of the paper with an overview of OpenID, Blockchains, and the cryptographic primitives we need for the paper, in Sec. 2. In Sec. 3, we define Tagged Witness Signature along with its security and privacy properties. In Sec. 4, we describe the zkLogin system. In Sec. 5, we describe our production deployment of zkLogin and document its performance. Finally, in Sec. 6, we review existing works that are similar to us and in some cases, inspired our work, and conclude in Sec. 7.

2. Preliminaries

Basic Notations: Throughout the paper, we denote the security parameter by λ and its unary representation by 1^λ. A function $\mathsf{negl} : \mathbb{N} \to \mathbb{R}^+$ is called *negligible* if there exists $k_0 \in \mathbb{N}$ such that for all $k > k_0$ and $c > 0$ it holds that $\mathsf{negl}(\lambda) < 1/k^c$.

2.1. OpenID Connect

OpenID Connect (OIDC) is a modern authentication protocol built on top of the OAuth 2.0 framework. It allows third-party applications to verify the identity of end users based on the authentication performed by an OpenID Provider (OP), e.g., Google, as well as to obtain basic profile information about the end user. OIDC introduces the concept of an ID token, which is a JSON Web Token (JWT) that contains *claims* about the authenticated user.

JSON Web Tokens (JWTs) are a versatile tool for securely transmitting information between parties using a compact and self-contained JSON format. A JWT consists of three components: a header, a payload, and a signature (see Fig. 1). All the three components are encoded in base64. Decoding the header and payload results in JSON structures. Sticking to JWT terminology, we refer to a JSON key as a *claim name* and the corresponding value as the *claim value*.

JWT header: Fig. 1 also shows a decoded JWT header. The "alg" claim specifies the signing algorithm used to create the signature. The JSON Web Algorithms spec recommends the use of two algorithms: RS256 and ES256 [16] for this purpose. Of the two, we found that RS256 is the most widely used, hence we only support that currently.

The "kid" claim helps identify the key used for signature verification. Let $(\mathsf{sk}_{OP}, \mathsf{pk}_{OP})$ generate the actual key pair where there is a one-to-one mapping between the "kid" value and pk_{OP}. The public key pk_{OP} is posted at a public URI, e.g., Google posts its keys at https://www.googleapis.com/oauth2/v3/certs. Furthermore, many OpenID providers rotate keys frequently, e.g., once every few weeks – so a JWT verifier needs to periodically fetch keys from the OP's website.

2. In addition, zkLogin also requires the existence of an app implementing the OAuth flows. The app may be viewed as another component of the key distribution authority.

JWT Payload: Listing 1 shows an example JWT payload. Any OIDC-compliant JWT contains the following claims:

1) "iss" (issuer): Identifies the entity that issued the JWT, typically an OpenID Provider (OP). This claim is fixed per OP, i.e., it's value is same for all tokens issued by the same OpenID Provider.
2) "sub" (subject): Represents the subject of the JWT, often the user or entity the token pertains to. This claim is fixed per-user. The OIDC spec defines two approaches an OP can take to generate the subject identifier:
   a) *Public identifier*: Assign the same subject identifier across all apps. A majority of current providers choose public identifiers, e.g., Google, Twitch, Slack, Kakao.
   b) *Pairwise identifier*: Assign a unique subject identifier for each app, so that apps do not correlate the end-user's activities. E.g., Apple, Facebook, Microsoft.
3) "aud" (audience): Defines the intended recipient(s) of the token, ensuring it's only used where it's meant to be. This claim is fixed per-app. The aud value is assigned to an app after it registers with the OP.
4) "nonce": A unique value generated by the client to prevent replay attacks, particularly useful in authentication and authorization flows.

Apart from the above, OIDC allows providers to include some optional claims like emails or set some custom claims.

The notion of a public (vs) pairwise identifier can also be used to refer to other OP-issued identifiers in the JWT. For example, Apple offers the feature of "Hide my email", where a randomized email address is sent in the JWT. This private email can be viewed as a pairwise identifier.

JWT API: We model the process of issuing and verifying a JWT as follows:

- $\mathsf{jwt} \leftarrow JWT.\mathsf{Issue}(\mathsf{sk}_{OP}, \mathcal{C})$: After the user successfully authenticates, the OpenID Provider signs the claim set $Sig.Sign(\mathsf{sk}_{OP}, \mathcal{C})$ and returns a Base64-encoded JWT as shown in Fig. 1. The *claim set* $\mathcal{C} = \{\mathsf{sub}, \mathsf{aud}, \mathsf{iss}, \mathsf{nonce}, \ldots\}$ contains the list of claims. The OIDC spec mandates the presence of certain claims like $\mathsf{sub}, \mathsf{aud}, \mathsf{iss}, \mathsf{nonce}$[3] in a JWT. An OpenID provider can optionally include additional claims, e.g., email, profile image. Note that in practice the signature is actually over the Base64 encoding of header and payload. $JWT.\mathsf{Issue}$ inherits the standard unforgeability property of Sig.
- $0/1 \leftarrow JWT.\mathsf{Verify}(\mathsf{pk}_{OP}, \mathsf{jwt})$: Verifies that the JWT was signed by the OpenID Provider.

We use the notation jwt.claimName to refer to the value of a particular claim in the JWT. For example, if jwt refers to the example in Fig. 1, then jwt.sub = "1234567890" is the subject identifier.

3. For nonce, the spec mandates that a nonce claim be present if the request contains it.

2.2. Cryptographic Primitives

To design zkLogin, we rely on a number of cryptographic primitives including non-interactive zero-knowledge proof systems, digital signatures and collision-resistance hash functions. In what follows, we briefly summarize the main properties of these primitives while the formal definitions can be found in App. A.

Zk-SNARK: Zero-Knowledge (ZK) proofs allow a prover to convince a verifier about the knowledge of a (hidden) witness to a (public) instance belonging to an NP-language, $\mathcal{L}$, in several rounds of communication, without revealing any extra information beyond the validity of the statement itself. Non-Interactive Zero-Knowledge (NIZK) arguments remove the interaction between the prover and verifier either in the Random Oracle Model (ROM) or Common Reference String (CRS) model. Zk-SNARK (Succinct Non-Interactive ARgument of Knowledge) is essentially a NIZK in the CRS model with a short proof and efficient verification time. The primary security properties of this fundamental primitive are: completeness, zero-knowledge and knowledge soundness. Informally, completeness ensures any honest prover can successfully convince any third party about the validity of a correct statement. Zero-knowledge guarantees the verifier does not learn anything beyond the validity of the proof, while the knowledge soundness ensures no malicious prover can form verifiable proofs on wrong statements.

Digital Signatures: Digital Signatures (DS) as a well-known cryptographic primitive emulate the traditional handwriting signatures. A signer with the possession of a secret signing key can generate a signature on a message that persuades another party, called verifier, of the message's authenticity and origin from the signer. The correctness property ensures that any properly generated signature satisfies the verification conditions successfully. On the other hand, the unforgeability guarantees no one except the secret signing key holder, i.e., the signer can form a valid signature for a fresh message.

Cryptographic Hash functions: Hash functions generate unpredictable outputs of a certain size within a given domain by processing input data. Collision resistance property guarantees that different data inputs do not produce the same output. zkLogin makes specific use of Poseidon [17], a recent ZK-friendly hash function.

3. Tagged Witness Signature

In traditional digital signature schemes, a signer needs to maintain a long-lived secret key which can be burdensome. The goal of Tagged Witness Signatures (TWS) is to slightly relax this requirement by replacing the secret signing key with a valid *witness* to a public statement. The statement comprises of a tag t, with respect to a public predicate P, which is fixed for the scheme. At a high level, a TWS should satisfy the properties of completeness, unforgeability and witness-hiding. The completeness property ensures that signatures produced with a valid witness w, i.e., $P(t, w) = 1$

verify. The unforgeability property ensures that an adversary cannot produce a valid signature without knowing a valid witness. The witness-hiding property guarantees that a signature does not reveal any information about the witness, essentially capturing zero-knowledge or privacy.

Like Signatures of Knowledge (SoK) [10], there is no explicit secret key required for signing - the "secret" is the ability to obtain a witness. However, a crucial difference is that unforgeability holds even if witnesses corresponding to different tags are leaked to the adversary. Modeling witness leakage is crucial in practical settings where the chances of an old witness leaking over a long-enough duration of time are high, such as in zkLogin. In this sense, Tagged Witness Signature can be thought of as SoK with forward secrecy. We also achieve a few more desirable properties compared to Krawczyk [18] who defined forward secure signatures. In contrast to the construction of [18], our construction allows arbitrary numbers of dynamically defined timestamps.

In addition, in contrast to SoK, we employ a $\mathsf{Gen}(\cdot)$ algorithm that preprocesses the predicate and generates specific public parameters for it. This design eliminates the need for the verifier to know or read the predicate, enabling significant optimizations in the blockchain environment.

Definition 1 (Tagged Witness Signature)**.** *A tagged witness signature scheme, for a predicate P, is a tuple of algorithms $\mathsf{TWS} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ defined as follows:*

$\mathsf{Gen}(1^\lambda) \to \mathsf{pk}$ *: The Gen algorithm takes the predicate $P : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$ and a security parameter λ as inputs, and outputs a public key pk. The input to the predicate is a public tag t and a secret witness w.*

$\mathsf{Sign}(t, \mathsf{pk}, w, m) \to \sigma$ *: The Sign algorithm takes as input a tag t, the public key pk, a witness w and a message m, and outputs a signature σ.*

$\mathsf{Verify}(t, \mathsf{pk}, m, \sigma) \to 0/1$ *: The Verify algorithm takes a tag t, the public key pk, a message m and a signature σ as inputs, and outputs a bit either 0 (reject) or 1 (accept).*

Definition 2 (Completeness)**.** *A Tagged Witness Signature for a predicate $P : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$, achieves completeness if for all tag and witnesses t, w such that $P(t,w) = 1$ and message m, and sufficiently large security parameter λ, we have:*

$$\Pr\left[\begin{array}{c}\mathsf{pk} \leftarrow \mathsf{Gen}(1^\lambda), \\ \sigma \leftarrow \mathsf{Sign}(t, \mathsf{pk}, w, M) : \\ \mathsf{Verify}(t, \mathsf{pk}, m, \sigma) = 1\end{array}\right] \geq 1 - \mathsf{negl}(\lambda) \cdot$$

Definition 3 (Unforgeability)**.** *Let $\mathsf{TWS} := (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ be a Tagged Witness Signature for a predicate P. The advantage of a PPT adversary $\mathcal{A}$ playing the described security game in Fig. 2, is defined as:*

$$Adv^{\mathsf{EUF\text{-}CTMA}}_{\mathsf{TWS},\mathcal{A}}(\lambda) := \Pr[\mathbf{Game}^{\mathsf{EUF\text{-}CTMA}}_{\mathcal{A},\mathsf{TWS}}(1^\lambda) = 1]$$

A TWS achieves unforgeability against chosen tag and message attack if we have $Adv^{\mathsf{EUF\text{-}CTMA}}_{\mathsf{TWS},\mathcal{A}}(\lambda) \leq \mathsf{negl}(\lambda)$.

$\mathbf{Game}^{\mathsf{EUF\text{-}CTMA}}_{\mathcal{A},\mathsf{TWS}}(1^\lambda)$:
$\mathsf{pk} \leftarrow \mathsf{Gen}(1^\lambda)$
$(t^*, m^*, \sigma^*) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}^{\mathsf{Sign}}(\cdot),\mathcal{O}^{\mathsf{Wit}}(\cdot)}(\mathsf{pk})$
return $((t^*, m^*) \notin \mathcal{Q}_s \ \wedge \ t^* \notin \mathcal{Q}_w \wedge \mathsf{Verify}(t^*, \mathsf{pk}, m^*, \sigma^*))$

$\mathcal{O}^{\mathsf{Wit}}(t)$:
Obtain w, s.t. $P(t,w) = 1$
$\mathcal{Q}_w \leftarrow \mathcal{Q}_w \cup \{t\}$
return w

$\mathcal{O}^{\mathsf{Sign}}(t, m)$:
Obtain w, s.t. $P(t,w) = 1$
$\sigma \leftarrow \mathsf{Sign}(t, \mathsf{pk}, w, m)$
$\mathcal{Q}_s \leftarrow \mathcal{Q}_s \cup \{(t, m)\}$
return σ

Figure 2: The unforgeability security game.

In simple terms, this definition addresses the situation in which the witnesses used to generate signatures might become known to an adversary. Note in particular how we model witness leakage through a witness oracle. If it is computationally hard for the adversary to obtain a witness for a new tag, the property of unforgeability ensures that the adversary cannot create a signature for this fresh tag.

As noted before, zkLogin requires modeling a notion of "tag freshness". This can be done by simply setting time to be one of the components of a tag. Since an adversary can request witnesses corresponding to any tag of its choosing, it can request witnesses for old tags, thus modeling leakage of old witnesses. Note how the unforgeability definition is agnostic to how a higher-level protocol defines what it means for a tag to be "fresh".

Definition 4 (Witness Hiding)**.** *A Tagged Witness Signature for a predicate P, $\mathsf{TWS} := (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$, achieves Witness-Hiding property if for all PPT adversaries $\mathcal{A}$, there exist simulators $(\mathsf{SimGen}, \mathsf{SimSign})$, playing the described security games in Fig. 3 and we have:*

$$\left|\begin{array}{l}\Pr[\textbf{\textit{Expt-Real}}_{\mathcal{A}}(1^\lambda) = 1] \\ -\Pr[\textbf{\textit{Expt-Sim}}_{\mathcal{A}}(1^\lambda) = 1]\end{array}\right| \leq \mathsf{negl}(\lambda) \ .$$

$\mathbf{Expt\text{-}Real}_{\mathcal{A}}(1^\lambda)$:
$\mathsf{pk} \leftarrow \mathsf{Gen}(1^\lambda)$
$b \xleftarrow{\$} \mathcal{A}^{\mathcal{O}^{\mathsf{Sign}}(\cdot,\cdot)}(\mathsf{pk})$
return b

$\mathcal{O}^{\mathsf{Sign}}(t, m)$:
Obtain w, s.t. $P(t,w) = 1$
$\sigma \leftarrow \mathsf{Sign}(t, \mathsf{pk}, w, m)$
return σ

$\mathbf{Expt\text{-}Sim}_{\mathcal{A}}(1^\lambda)$:
$(\mathsf{pk}, \mathsf{trap}) \leftarrow \mathsf{SimGen}(1^\lambda)$
$b \xleftarrow{\$} \mathcal{A}^{\mathcal{O}^{\mathsf{SimSign}}(\cdot,\cdot)}(\mathsf{pk})$
return b

$\mathcal{O}^{\mathsf{SimSign}}(t, m)$:
$\sigma \leftarrow \mathsf{SimSign}(t, \mathsf{pk}, \mathsf{trap}, m)$
return σ

Figure 3: The Witness Hiding security game.

This characterizes the idea that an adversary gains no additional information about the witness associated with tags by observing the signatures. Essentially, this defines a privacy property for witnesses.

We will build the core cryptographic component of zkLogin as a Tagged Witness Signature. We also develop

a generic construction in App. C.

We can also construct an Identity-based Signature (IBS) scheme [13] from a TWS. The tag can be identified with the identity, and the witness could a signature on the tag along with any additional data. Key expiry can be handled by including timestamps in the signature. The exposure of one tag's witness doesn't affect the security of signature with other tags, modeling security under key leakage.

4. The zkLogin system

The primary goal of zkLogin is to allow users to maintain blockchain accounts leveraging their existing OpenID Provider accounts.

4.1. Model

There are four principal interacting *entities* in zkLogin:

1) **OpenID Provider (OP)**: This refers to any provider that supports OpenID Connect, such as Google. A key aspect of these providers is their ability to issue a signed JWT containing a set of claims during the login process. For more details, see Sec. 2.1.
2) **User**: End users who own the zkLogin address and should have the capability to sign and monitor transactions. They are assumed to hold an account with the OP and typically possess limited computational resources.
3) **Application**: The application coordinates the user's authentication process. It comprises two components: the Front-End (FE), which can be an extension, a mobile or a web app, and the Back-End (BE). As we will discover shortly, the implementation of certain BE services plays a crucial role in significantly enhancing the user experience within zkLogin.
4) **Blockchain**: The blockchain is composed of validators who execute transactions. zkLogin requires blockchain support for on-chain ZKP verification.

Adversarial model: We assume that the app's backend is untrusted whereas its frontend is trusted. This is reasonable because the frontend code of an app is typically public as it gets deployed on user's devices, and is thus subject to greater public scrutiny.

We assume that the OpenID Provider (OP) is trusted. This is reasonable because the main goal of our system is to design a user-friendly wallet. This does not make the OP a custodian since zkLogin works with existing unmodified API and the OP is not even required to know about the existence of zkLogin.

4.1.1. Properties. We require zkLogin to guarantee that unintended entities should not be able to perform certain actions or gain undesired visibility.

Security: This property captures the notion that transactions are *unforgeable*. Like in any secure signature scheme, an adversary should not be able to sign messages on behalf of the user. In addition, we also want to prevent signatures on transactions based on expired JWTs. We formally model zkLogin in the presence of such leakages as a Tagged Witness Signature (Sec. 3) and prove unforgeability.

Unlinkability: This property captures the inability of any party (except the app) to link a user's off-chain and on-chain identities. That is, no one can link a user's OP-issued identifier, the app they used, or any other sensitive field in the JWT, with their zkLogin-derived blockchain account. The only exception is the issuer iss claim, i.e., the unlinkability property does not mandate that the issuer be unlinkable.

We formalize this below. At a high level, given 2 adversarially indicated claim sets $\mathcal{C}_0$ and $\mathcal{C}_1$ (recall that a claim set is the list of claims present in a JWT), the adversary cannot link which one corresponds to a given zkLogin address zkaddr, even given access to several zkLogin signatures for any address of its choosing. If either of the claim sets $\mathcal{C}_0$ or $\mathcal{C}_1$ belong to a user controlled by the adversary, an adversary can win the game trivially – so both $\mathcal{C}_0$ and $\mathcal{C}_1$ must correspond to honest users.

We relax the unlinkability property w.r.t. the application (in addition to the OP) in order to create a user-friendly wallet experience. So the adversary below refers to any party except the app or the OP.

$\mathbf{Game}^{\mathrm{UL}}_{\mathcal{A},\mathsf{TWS}}(1^\lambda)$:	$\mathcal{O}^{\mathsf{Sign}}(T, m, \mathsf{zkaddr}')$:
$\mathsf{pk} \leftarrow \mathsf{Gen}(P, 1^\lambda)$	Let $t = (\mathsf{pk}^{cur}_{OP}, \mathsf{iss}, \mathsf{zkaddr}', T)$
Sample $b \xleftarrow{\$} \{0,1\}$	Obtain w, s.t. $P(t, w) = 1$
$(\mathcal{C}_0, \mathcal{C}_1, st) \leftarrow \mathcal{A}_1(\mathsf{pk})$	$\sigma \leftarrow \mathsf{Sign}(t, \mathsf{pk}, w, m)$
Ensure $\mathcal{C}_0.\mathsf{iss} == \mathcal{C}_1.\mathsf{iss}$	return σ
Construct zkaddr from $\mathcal{C}_b$	
$b' \leftarrow \mathcal{A}_2^{\mathcal{O}^{\mathsf{Sign}}(\cdot,\cdot,\cdot)}(st, \mathsf{zkaddr})$	
return $b' == b$	

Figure 4: The unlinkability game.

4.2. System details

We begin by explaining how we derive addresses in zkLogin and then explain how zkLogin works.

Fix an OpenID Provider (iss). Typically, each application that wants to use the provider's sign-in flows needs to register with the provider manually. At the end, the app receives a unique audience identifier (aud).

Address derivation: A simple way to define a user's blockchain address is by hashing the user's subject identifier (sub), app's audience (aud) and the OP's identifier (iss).

More generally, zkLogin addresses can be generated from any identifier given by the OpenID Provider, as long as it is unique for each user (meaning no two users have the same identifier) and permanent (meaning the user can't change it). We call such an identifier a "Stable Identifier", denoted by stid.

A good example of a Stable Identifier is the Subject Identifier (sub). The OpenID Connect spec requires that the subject identifier be stable [19].

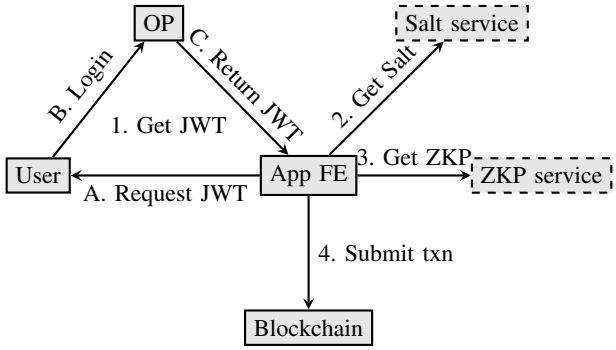

Figure 5: The zkLogin system overview. OP and FE stand for OpenID Provider and Front-End, respectively. The App's Back-End components, i.e., salt service and ZKP service, are showed with a dashed border.

Besides the subject identifier, other identifiers like email addresses, usernames or phone numbers might also meet these criteria. However, whether an identifier is considered stable can differ from one provider to another. For instance, some providers like Google don't allow changing email addresses, but others might.

The necessity of Salt: An important privacy concern arises whenever the stable identifier is sensitive, such as an email address or a username. Note that the subject identifier is also sensitive if the provider uses public subject identifiers, meaning if a user logs into two different apps, the same sub value is returned. To address privacy concerns and prevent the stable identifier from being easily linked to a user's blockchain address, we introduce a concept called "salt" – a type of persistent randomness.

With this approach, a user's zkLogin address is

$$\mathsf{zkaddr} = H(\mathsf{stid}, \mathsf{aud}, \mathsf{iss}, \mathsf{salt}) \ . \tag{1}$$

Adding a salt is not necessary if the stable identifier is already private, like in the case of providers that support pairwise identifiers. For our discussion, we will assume we are using a public stable identifier and therefore include a salt in the process.

Fig. 5 depicts the system's workflow including four parts, explained in follows. The first two parts ① Get JWT, and ② Get Salt, describe the protocol flows for implementing the $\mathcal{O}^{\mathsf{Wit}}(\cdot)$ oracle. The next two parts ③ Compute ZKP, and ④ Submit Transactions, informally describe how the $\mathsf{Gen}(\cdot)$, $\mathsf{Sign}(\cdot)$, and $\mathsf{Verify}(\cdot)$ functions are deployed. The construction is formalized as a Tagged Witness Signature scheme, $\Sigma_{\mathsf{zkLogin}}$, in Fig. 6, over the predicate $P_{zkLogin}$.

① Get JWT: One of the key ideas in zkLogin is to treat the OpenID Provider as a certificate authority by embedding data into the nonce during the OpenID flow [20].

The application generates an ephemeral key pair (vk_u, sk_u), sets the key pair's expiry time T_{max}, generates a randomness r and computes the nonce via

$$\mathsf{nonce} \leftarrow H(vk_u, T_{max}, r) \ .$$

$P_{zkLogin}\left(\begin{matrix}\mathsf{tag} = (\mathsf{pk}_{OP}^{cur}, \mathsf{iss}, \mathsf{zkaddr}, T), \\ w = (\mathsf{jwt}, \mathsf{salt}, r, vk_u, sk_u)\end{matrix}\right)$:
$\mathsf{zkaddr} = H(\mathsf{jwt.stid}, \mathsf{jwt.aud}, \mathsf{jwt.iss}, \mathsf{salt})$ and $\mathsf{jwt.iss} = \mathsf{iss}$ and $\mathsf{jwt.nonce} = H(vk_u, T, r)$ and $JWT.Verify(\mathsf{pk}_{OP}^{cur}, \mathsf{jwt})$ and (sk_u, vk_u) is a valid sig-key-pair.

$\mathsf{Ckt}\left(\begin{matrix}\mathsf{zkx} = (\mathsf{pk}_{OP}^{cur}, \mathsf{iss}, \mathsf{zkaddr}, T, vk_u), \\ \mathsf{zkw} = (\mathsf{jwt}, \mathsf{salt}, r)\end{matrix}\right)$:
$\mathsf{zkaddr} = H(\mathsf{jwt.stid}, \mathsf{jwt.aud}, \mathsf{jwt.iss}, \mathsf{salt})$ and $\mathsf{jwt.iss} = \mathsf{iss}$ and $\mathsf{jwt.nonce} = H(vk_u, T, r)$ and $JWT.Verify(\mathsf{pk}_{OP}^{cur}, \mathsf{jwt})$.

$\mathsf{Gen}(1^\lambda)$:
- Let $\Pi = (\mathsf{Gen}, \mathsf{Prove}, \mathsf{Verify})$ be a NIZK scheme.
- Sample $zkcrs \leftarrow \Pi.\mathsf{Gen}(1^\lambda, \mathsf{Ckt})$.
- Output $pk = zkcrs$.

$\mathsf{Sign}(\mathsf{tag}, \mathsf{pk}, w, M)$:
- Parse tag as $(\mathsf{pk}_{OP}^{cur}, \mathsf{iss}, \mathsf{zkaddr}, T)$.
- Parse pk as $zkcrs$.
- Parse w as $(\mathsf{jwt}, \mathsf{salt}, r, vk_u, \mathsf{sk}_u)$.
- Set $\sigma_u \leftarrow Sig.Sign(\mathsf{sk}_u, M)$.
- Set $\mathsf{zkx} \leftarrow (\mathsf{pk}_{OP}^{cur}, \mathsf{iss}, \mathsf{zkaddr}, T, vk_u)$.
- Set $\mathsf{zkw} \leftarrow (\mathsf{jwt}, \mathsf{salt}, r)$.
- Set $\pi \leftarrow \Pi.\mathsf{Prove}(zkcrs, \mathsf{zkx}, \mathsf{zkw})$.
- Output $\sigma = (vk_u, T, \sigma_u, \pi)$.

$\mathsf{Verify}(\mathsf{tag}, \mathsf{pk}, M, \sigma)$:
- Parse tag as $(\mathsf{pk}_{OP}^{cur}, \mathsf{iss}, \mathsf{zkaddr}, T)$.
- Parse σ as (vk_u, T, σ_u, π).
- Set $\mathsf{zkx} \leftarrow (\mathsf{pk}_{OP}^{cur}, \mathsf{iss}, \mathsf{zkaddr}, T, vk_u)$.
- Verify $Sig.Verify(vk_u, \sigma_u, M)$.
- Verify $\Pi.\mathsf{Verify}(zkcrs, \pi, \mathsf{zkx})$.

Figure 6: Tagged Witness Signature, $\Sigma_{\mathsf{zkLogin}}$. Address is derived from the stable identifier, e.g., $\mathsf{stid} = \mathsf{sub}$.

Note that the expiration time, T_{max}, must use a denomination that can be understood by the blockchain validators, e.g., "T_{max} = epoch #100" if the blockchain operates in epochs. The randomness r helps to achieve unlinkability as it prevents the OP from learning the ephemeral public key.

Next the app initiates an OAuth flow where the user logs in to the OP. After the user successfully authenticates, the app receives a JWT from the OP, $\mathsf{jwt} \leftarrow JWT.\mathsf{Issue}(\mathsf{sk}_{OP}, \{\mathsf{stid}, \mathsf{aud}, \mathsf{iss}, \mathsf{nonce}, \ldots\})$. In essence, the JWT acts as a certificate over vk_u, i.e., the JWT asserts that the owner of sk_u is indeed the same as the user identified by the OP-issued sub.

② Get Salt: As noted before, we employ an additional salt to unlink a user's OP-issued identifier and blockchain address. Managing the salt, however, poses an operational challenge as losing the salt implies that the assets will be permanently locked.

Our main approach is to employ a salt backup service that returns a user's salt upon authentication. Different authentication polices can be used: the approach we take is to return a user's salt only if they submit a valid JWT that

$\mathsf{zkLoginSign}(\mathsf{zkaddr}, \mathsf{iss}, M)$**:**

- Obtain current time T_{cur}.
- Let $T_{max} = T_{cur} + max\_offset$.
- Obtain pk_{OP}^{cur} from JWK of iss.
- Obtain $\mathsf{pk} = zkcrs$ from the blockchain.
- Let $\mathsf{tag} \leftarrow (\mathsf{pk}_{OP}^{cur}, \mathsf{iss}, \mathsf{zkaddr}, T_{max})$.
- Obtain $w \leftarrow GetWitness(\mathsf{tag})$.
- Output $\sigma \leftarrow \Sigma_{\mathsf{zkLogin}}.\mathsf{Sign}(\mathsf{tag}, \mathsf{pk}, w, M)$.

$\mathsf{GetWitness}(\mathsf{tag})$**:**

- Parse tag as $(\mathsf{pk}_{OP}^{cur}, \mathsf{iss}, \mathsf{zkaddr}, T)$.
- Sample $(vk_u, \mathsf{sk}_u) \leftarrow Sig.Gen(1^\lambda)$.
- Sample $r \leftarrow \{0,1\}^\lambda$.
- Set $\mathsf{nonce} \leftarrow H(vk_u, T, r)$.
- Obtain $\mathsf{stid}, \mathsf{aud}, \mathsf{salt}$ from the User/App.
- Obtain $\mathsf{jwt} \leftarrow JWT.\mathsf{Issue}(\mathsf{sk}_{OP}^{cur}, \{\mathsf{stid}, \mathsf{aud}, \mathsf{iss}, \mathsf{nonce}\})$ from the OP.
- Output $w = (\mathsf{jwt}, \mathsf{salt}, r, vk_u, \mathsf{sk}_u)$.

$\mathsf{zkLoginVerify}(\mathsf{pk}, \mathsf{zkaddr}, \mathsf{iss}, M, \sigma)$**:**

- Obtain associated time T_{cur}.
- Output 0, if $T_{cur} > \sigma.T$.
- Obtain pk_{OP}^{cur} from JWK of iss.
- Let $\mathsf{tag} \leftarrow (\mathsf{pk}_{OP}^{cur}, \mathsf{iss}, \mathsf{zkaddr}, T_{cur})$.
- Output $\Sigma_{\mathsf{zkLogin}}.\mathsf{Verify}(\mathsf{tag}, \mathsf{pk}, M, \sigma)$.

Figure 7: The complete signature scheme of zkLogin leveraging $\Sigma_{\mathsf{zkLogin}}$.

verifies, where the salt is deterministically derived from a seed, k_{seed}, as $\mathsf{salt} = H(k_{seed}, \mathsf{sub}, \mathsf{aud}, \mathsf{iss})$ (cf. Eq. (1)). This approach has the benefit that the size of the salt service storage is very small, and is independent of the number of users.

A downside of using a backup service is that it can break unlinkability, i.e., it can link the user's OP-issued identifier with the blockchain address. One way to mitigate this privacy risk is to employ trusted enclaves, as we do so in our implementation (Sec. 5). Minimizing the storage comes in handy here as the trusted enclave only needs to manage a small seed.

However, trusted enclaves come with their own set of technical challenges [21]. Therefore, our system can support other salt strategies targeting a different usability (vs) security trade-off, for example, let the user maintain the salt. Allowing the user to self-maintain can offer additional security as it can prevent even the OP from stealing a user's assets. It effectively becomes a 2-out-of-2 wallet between salt and the user's OP account.

③ **Compute ZKP:** The next step is to use the salt and the JWT to compute a Zero-Knowledge Proof, denoted by π, proving the association between the ephemeral public key, vk_u, and the address, zkaddr. The ZKP's public inputs and witnesses are:

- Public inputs: OP's public key, pk_{OP}, user's address, zkaddr, the ephemeral public key, vk_u, and its expiration time T_{max}, i.e., $P = (\mathsf{pk}_{OP}, \mathsf{iss}, \mathsf{zkaddr}, vk_u, T_{max})$.
- Witnesses: jwt, salt and the nonce, randomness r.

The ZKP formally proves the predicate, Ckt, depicted in Fig. 6. The setup process, $\Pi.\mathsf{Gen}$, for the NIZK system, Π, is run at the beginning to generate the $zkcrs$. This process only needs to be done once and the generated $zkcrs$ can used for all users and OPs. Formally, this is part of the Gen function of the Tagged Witness Signature, $\Sigma_{\mathsf{zkLogin}}$. Informally, Ckt captures that the following steps:

1) Hashing the claims $\mathsf{stid}, \mathsf{aud}, \mathsf{iss}$ (extracted from the JWT) with the salt gives the expected address zkaddr.
2) Hashing the ephemeral public key, vk_u, expiry time, T_{max}, and the randomness r gives the expected nonce (extracted from the JWT).
3) The JWT verifies, i.e., $JWT.\mathsf{Verify}(\mathsf{pk}_{OP}, \mathsf{jwt})$.

We delegate the computation of the Zero-Knowledge Proof to a different ZKP service, as shown in Fig. 5. The motivation stems from the fact that we need to work with non-ZK-friendly cryptographic primitives such as SHA-2 and modular exponentiation, making the ZKP computation impractical on users devices. We discuss more details about the implementation of the ZK circuit in Sec. 5.

④ **Submit transaction:** Say that the transaction data is tx. The app uses the ephemeral private key to sign the transaction, i.e., set $\sigma_u \leftarrow Sig.\mathsf{Sign}(\mathsf{sk}_u, tx)$. The final zkLogin signature on the transaction tx is $(vk_u, T_{max}, \sigma_u, \pi)$. Each validator can verify the signature by:

1) Verifying the ZKP, π, with the public inputs $P = (\mathsf{pk}_{OP}, \mathsf{iss}, \mathsf{zkaddr}, T_{max}, vk_u)$.
2) Verify that pk_{OP} is indeed the public key of the OpenID Provider iss. This step requires an oracle posting the public keys periodically as the OP may rotate the keys.
3) Verify $Sig.\mathsf{Verify}(vk_u, \sigma_u, tx)$.
4) Verify $T_{max} \geq T_{cur}$ where T_{cur} is the current time (known to all validators).

Finally, if all the conditions hold then the validators can execute the transaction tx sent by the address zkaddr.

The steps above of accessing the current time and the public key of the OP are part of the zkLogin system that are not captured by the Tagged Witness Signature formalism. We assume that they are obtained correctly to ensure unforgeability and enforce freshness of the tag.

4.3. Security Analysis

We prove that the proposed Tagged Witness Signature, $\Sigma_{\mathsf{zkLogin}}$, achieves the unforgeability and unlinkability properties, with formal proofs given in App. B.

Theorem 1. *Given that* Π *satisfies knowledge-soundness, and JWT and Sig are EUF-CMA secure, and* $H(\cdot)$ *is a collision-resistant hash function, the proposed Tagged Witness Signature,* $\Sigma_{\mathsf{zkLogin}}$*, achieves unforgeability, defined in Def. 3.*

Theorem 2. *Given that* Π *satisfies zero-knowledge, the Tagged Witness Signature,* $\Sigma_{\mathsf{zkLogin}}$*, achieves witness hiding, defined in Def. 4.*

Security of zkLogin: The unforgeability of $\Sigma_{\mathsf{zkLogin}}$ implies that an adversary cannot forge a signature for a given tag, even if it gets access to witnesses corresponding to other tags. In zkLogin, the system layer ensures that tags time out on a defined cadence, and hence witnesses need to be refreshed for new tags. Hence older tags are no longer useful for creating signatures. In particular, this means that a new JWT is needed for a zkLogin signature, once the ephemeral public key expires.

The security of zkLogin holds even if the app's backend acts maliciously. With regards to the ZKP service, this is because the ZKP only proves the association with an ephemeral public key. An attacker would also need the corresponding ephemeral private key to sign transactions.

A similar reason also applies to the salt service. However, there is a reliance on the app's liveness. For example, the loss of salt implies that the users funds are locked forever. We discuss ways to mitigate this risk in Section 4.4.

Users may leverage the "sign-in with X" feature in different websites or apps. So it is important that a malicious app cannot steal a user's zkLogin assets. zkLogin achieves unforgeability against such malicious apps by including the app's unique identifier, aud, in the address derivation (cf. Eq. (1)).

Unlinkability of zkLogin: zkLogin achieves unlinkability based on the witness-hiding property of $\Sigma_{\mathsf{zkLogin}}$ and on zkaddr being a hiding commitment to stid and aud.

In addition, zkLogin also provides unlinkability against honest-but-curious OpenID Providers (OPs). This is because the use of randomness in nonce generation hides the user's ephemeral public key from the OP. Consequently, the OP cannot link the user's stable identifier with their zkLogin address nor track their transactions without resorting to network-based side-channel analysis.

The ZKP service (part of the app's backend) receives the user's JWT and salt – so it is able to link the user's stable identifier with the zkLogin address. We specify a way to hide the salt from the ZKP service to attain unlinkability in Sec. 4.4.

The salt service, by design, maintains users' salts. But our use of an enclave provides protection against a malicious salt service operator. The blockchain records the transactions along with the zkLogin signatures. These records are publicly visible and contain information about the zkLogin address that signed it. However, the only information visible in a transaction is the OP identifier.

4.4. Design extensions

Avoiding reliance on the app's liveness: It is desirable to ensure that a user can access their zkLogin account even if an application completely disappears (loss of liveness). Users can hedge against this risk by setting up a zkLogin Multi-sig between two (or more) apps. This can ensure that the user can access their zkLogin account through either of the two apps.

In some cases, it is possible to allow users to access their zkLogin account even if it was created normally, i.e., without a Multi-sig. However, it requires that the stable ID is a public identifier (see Sec. 2 for an explanation) and that users have previously backed up their salt (even if they use a salt service normally). If these conditions are met, we can create a slightly modified ZKP to allow users to access their funds: if a special flag is set in the ZKP inputs, then set the audience to the disappeared app's audience instead of the new app's one when deriving the address.

Unlinkability from the ZKP service: As noted above, the ZKP service can link a user's OP-issued identifier with their blockchain address as it receives both the JWT and salt as part of the witness. We can avoid this by splitting the ZKP into two parts: one proving that $\mathsf{zkaddr} = H(\mathsf{stid}, \mathsf{aud}, \mathsf{iss}, \mathsf{salt})$ and another proving that the JWT verifies. We'd also need to use hiding commitments to each of $(\mathsf{stid}, \mathsf{aud}, \mathsf{iss})$, which act as the link between the two proofs. In this mode, the salt is only a witness for the first ZKP and not for the second. Also note that the heavy computation of verifying the JWT is happening in the second ZKP. So we can compute the first ZKP locally on the user device and delegate computation of the second ZKP to a backend service.

Attestations without nonce: Even though the OpenID Connect spec requires providers to include a nonce when the request contains it, some providers do not. Moreover, outside the realm of OpenID, several prominent identity documents, e.g., e-Passports [22], already contain a digital signature over user's biographic information [23].

The main idea is to bind the ephemeral public key by directly hashing it with the JWT, in line with the generic construction in App. C. The ZKP proves the JWT validity like before and in addition computes the hash of JWT and the ephemeral public key. The main drawback with this solution is that it is not clear how to offload the ZKP generation securely – note that a malicious backend can hash the JWT with its own ephemeral key pair (privacy is also a concern for sensitive credentials like identity documents). We leave how to make zkLogin practical for nonce-less providers for future work.

Portability for cross-platform applications: Some applications might have apps across multiple platforms, e.g., Android and iOS. For the best user experience, they'd want users to be able to login to the same account across the different platforms. But OPs typically assign different audience IDs for apps on different platforms even though they belong to to the same entity. Again, we can solve this issue by using a zkLogin Multi-sig assuming that the application can gather all the audience IDs beforehand.

4.5. Novel features

Apart from easy onboarding, zkLogin offers a few novel features that were not seen before to the best of our knowledge.

Discoverability: This means that a user's existing digital identifier with which they are prominently identified (email, username, etc.) is publicly linked to their blockchain address. While this obviously breaks unlinkability, this feature can be extremely useful in certain contexts where users want to maintain public profiles. For example, content creators may want to establish provenance by digitally signing their content [14]. Users with an existing zkLogin account can make their account discoverable by simply revealing the stable ID, audience ID and salt. New users can create a discoverable account by avoiding the use of a salt.

Partial reveal: It can also be useful to make an existing zkLogin account partially discoverable, e.g., revealing only the audience ID or just a portion of their stable ID. The latter would allow employees of an organization to reveal that they belong to an organization without revealing their identity, e.g., if email is the stable ID, Alice ("alice@nyu.edu") can reveal her university affiliation by revealing the email's domain name, i.e., "@nyu.edu". Revealing just the TLD of an email can also have interesting applications, e.g., reveal you are a student (".edu") or belong to a particular country (".uk" implies a UK based email).

Anonymous blockchain accounts: zkLogin allows creation a blockchain account that hides the identity of the account owner within an anonymity set. Two different approaches are possible. If the stable identifier has some structure, e.g., with emails, we can derive a blockchain address from a portion of the email address (either the domain name or the TLD), $\mathsf{zkaddr} = H(P, \mathsf{aud}, \mathsf{iss})$ where P is the relevant portion of the email (no salt).

A different approach is needed for other identifiers that do not have such a structure or if greater control is needed over the anonymity set. First decide the anonymity set, e.g., a list of Google Subject Identifiers L. The zkLogin address is derived from the entire anonymity set, $\mathsf{zkaddr} = H(L, \mathsf{aud}, \mathsf{iss})$. To sign a transaction, the user needs to prove that their Google ID belongs to the list L.

Claimability: This implies the ability to send assets to a person even before they have a blockchain account. A sender can derive the receiver's zkLogin address using the receiver's email (or a similar identifier) and randomly chosen salt. The salt can either be sent to the sender over a personal channel or simply set to a default value, e.g., zero. The former approach places a burden on the receiver to manage the salt whereas the latter makes the address discoverable.

5. Implementation and Evaluation

We now present the implementation details of zkLogin. A crucial implementation choice is the underlying proving system. In theory, zkLogin can be instantiated with any currently available proving system [24], [25], [26]. However, we chose Groth16 [11] due to its mature tooling ecosystem and compact proof size. In particular, we leverage the circom DSL [12] to efficiently write up the R1CS circuit.

Sec. 5.1 provides an in-depth look at the circuit details. It discusses various optimizations that have helped reduce the number of R1CS constraints by at least an order of magnitude for some components. Groth16 necessitates a circuit-specific trusted setup, and to this end, we have orchestrated a ceremony with the participation of over 100 external contributors to generate the Common Reference String (CRS). Lastly, the performance of all the critical components of zkLogin is evaluated in Sec. 5.2.

5.1. ZKP details

Despite offloading proof generation to an untrusted server, optimizing our R1CS circuit remains crucial for several reasons. First, a larger circuit translates to increased operational complexity during the ceremony, requiring the transmission of significantly larger files. Second, a more complex circuit incurs greater proving costs, both in terms of the time required to generate a proof and the expense of maintaining powerful servers.

Recall that the circuit takes a JWT J as input and parses certain claims, e.g., "sub" from it. Accordingly, it has two main components: (i) validating the JWT, and (ii) parsing the JWT. In total, our R1CS circuit has around 2^{20} constraints. Notably, the circuit utilizes the Poseidon hash function [17] for hashing in the circuit where possible.

5.1.1. JWT validation. One set of circuit operations is to verify the JWT signature. The IETF spec recommends the use of two algorithms for signing JWTs: RS256 and ES256 [16]. Of the two, we found that RS256 is the most widely used, hence we only support that currently. Our implementation can be extended to support ES256 in the future.

RS256 is short for RSASSA-PKCS1-v1_5 using SHA-256. Verifying a message under RS256 involves two steps:

1) hash the JWT's header and payload with SHA-2 to obtain a hash h and
2) perform a modular exponentiation over a 2048-bit modulus: if the signature is σ and modulus is p (both 2048-bit integers), check if $\sigma^{65537} \% p = pad(h)$ where $pad()$ is the PKCS1-v1_5 padding function.

We have largely reused existing code for RS256 signature verification. SHA-2 is the most expensive operation in the overall circuit taking up 74% of the constraints. Big integer operations needed to perform modular exponentiation are the second most expensive operation taking around 15% of the constraints. For bigint operations, we leverage existing code[4] which in turn implement efficient modular multiplication techniques from [27].

4. We used code from https://github.com/doubleblind-xyz/double-blind, https://github.com/zkp-application/circom-rsa-verify for the RSA circuit.

5.1.2. JWT parsing. Parsing the JWT for relevant claims takes approximately 115k constraints (10%) and is where most of our optimization efforts lie.

A naïve approach to JWT parsing would involve:

1) Base64 decoding the entire JWT header and payload.
2) Parsing the complete header and payload JSONs to extract all the relevant claims, e.g., "sub", "aud", "iss" and "nonce" from payload and "kid" from the header.

Decoding a single Base64 character takes 73 constraints, so decoding the entire JWT of maximum possible length $L_{max} = 1600$ bytes (we set L_{max} to 1600 based on empirical data) would incur $73L_{max} \approx 116k$ constraints. Likewise, fully parsing the a JSON inside a ZK circuit involves encoding every rule in the JSON grammar, which is likely to be very inefficient.

To address these challenges, we optimize the circuit to only parse and decode relevant parts of the JWT. We achieve this by:

- *Public Input Header*: Instead of decoding the JWT header in the circuit, we reveal it as a public input. This public header is then parsed and validated as part of the zkLogin signature verification process. This approach is possible because the JWT header (cf. Fig. 1) does not contain any sensitive claims.
- *Selective Payload Decoding*: We only decode and parse relevant portions of the JWT payload, avoiding unnecessary decoding overhead. We achieve this by leveraging the following observations about JWT payloads:
  1) Provider follows the JSON spec. In particular, it only returns valid JSONs and properly escapes all user-input strings in the JSON.
  2) All the claims of interest are in the top-level JSON and the JSON values are either strings or boolean.
  3) Escaped quotes do not appear inside a JSON key. See App. E for an attack example.

Our strategy then is to slice the portion of the JWT Payload containing a JSON key-value pair (a claim name and value) together with the ensuing delimiter, i.e., either a comma "," or a right brace "}". It is important to include the delimiter as it indicates the end of the value. In more detail, the JSON key-value pair parsing component of the circuit takes a (unparsed JSON) string S and does the following for every claim to be parsed:

Listing 2: Decoded JWT payload

```
{"sub":"123","aud":"mywallet","nonce":"ajshda"}
```

1) Given a start index i and length l, use string slicing techniques (see below) to extract the substring $S' = S[i : i+l]$. For example, if S is as shown in Listing 2, $i = 1$ and $l = 12$, then $S' =$`"sub":"123",`.
2) Check that the last character of S' is either a comma "," or a close brace "}".
3) Given a colon index j, check that $S'[j]$ is a colon ":".
4) Output $key = S'[0 : j] =$ `"sub"` and $value = S'[j+1 : -1] =$ `"123"`. In addition, while we do not explain here, our circuit can also tolerate some JSON whitespaces in the string S using similar techniques.
5) Check that the key and value are JSON strings by ensuring that the first and last characters of key (resp., $value$) are the start and end quote respectively.

The above strategy is used to parse four claims in the circuit, namely, the stable ID (e.g., sub or email), nonce, audience and email verified[5] in around 115k constraints. Following this, the extracted claim values are processed, e.g., the stable ID is fed into the address derivation, the nonce value is checked against a hash of the ephemeral public key, its expiry and randomness, etc. In addition, we also check that all the claims appear at the top-level, i.e., not at a nested level.

Note that it is possible for an attacker to over-extend the end index. Continuing the above example, an attacker could set $S' =$ `"sub":"1320606","aud":"mywallet",`. But this would have the effect of obtaining the JSON value of `"1320606","aud":"mywallet"`. And, crucially, this is not a valid JSON string as per the JSON grammar because a JSON string would escape the double quotes. This implies that no honest user will have this subject identifier. Therefore, the above attempt does not lead to a security break. Similarly, certain portions of the JWT allow user-chosen input, e.g., in nonce – but this does not lead to a security threat as you cannot inject unescaped key-value pairs into a JSON string.

The main security threat is of an attacker who inputs maliciously crafted inputs during the OpenID sign-in flow to sign transactions on behalf of a honest user (without having to steal the user's OpenID credentials). We have argued that this is not possible under reasonable assumptions.

Slicing arrays: Given an input array S of length n, index i and length m, we would like to compute the subarray $S[i : i+m]$. This primitive is used heavily: at least once for each JSON claim parsed.

A naïve approach to slicing is to do the following. For each output index j, do a dot product between S and a n-length vector O s.t. $O[j] = 1$. Since the dot product involves n multiplications, this takes n constraints per output index, and a total of $n * m$ constraints. Concretely, the value of n is $L_{max} = 1600$ and the value of m ranges between 50 and 200 (depending on the claim value length), so if $m = 100$, slicing once would incur 160k constraints. Since we slice around 4-5 times, this is quite costly.

We observe that the default input width of elements in the JWT is only 8 bits, which is much smaller than the allowed width of a field element in BN254 (253 bits). We pack 16 elements together (so the packed element is 128 bits), apply the naïve slicing algorithm over the packed elements, and finally unpack back to the original 8-bit width, while taking care of boundary conditions. With this optimization, slicing an array costs about $18m + (n * m)/32$ constraints. Using the above values of $n = 1600$ and $m = 100$, the number of constraints is only $33k$, i.e., a 4.8x reduction

5. If the stable ID is email, we have to additionally check that the "email_verified" claim is true in order to only accept verified emails [5].

per slice operation. Overall, this optimization reduces the number of constraints used to slice by more than an order of magnitude.

5.2. Performance

We have evaluated the performance of all the important components of zkLogin: salt service, ZK service and signature verification.

We have deployed the salt service within an AWS Nitro enclave running on an m5.xlarge[6] instance. This instance type boasts 4 vCPUs and 16 GiB RAM. The average response time for retrieving salt is 0.2s.

The ZK service is built around rapidsnark [28], a C++ and Assembly-based Groth16 prover. Before calling rapidsnark, the service converts user inputs (e.g., JWT, salt) to a witness using a combination of TypeScript code and the circom-generated witness calculator [12].

We have deployed the ZK service on a Google cloud n2d-standard-16[7] instance. This instance type boasts 16 vCPUs and 64 GB RAM. The average end-to-end response time is 2.68s with a standard deviation of 0.07s over 100 runs (including the network time). The above number is for retrieving a single ZK proof assuming no contention. The ZK service can be scaled horizontally to support any desired amount of traffic.

The verification of a zkLogin signature takes 2.04 ms on an Apple M1 Pro with 8 cores and 16 GB RAM.

6. Related Works

Many existing wallets leverage OAuth to onboard users onto blockchains. We are primarily interested in non-custodial OAuth-based wallets. Prior approaches can be classified as:

- (Trusted hardware) Use OAuth to authenticate to HSMs/Enclaves to verify the user's authentication token and retrieve secrets/attestation that can be used to sign transactions. E.g., Magic [29], DAuth [30], Face Wallet [31].
- (MPC) Use OAuth to authenticate to a non-colluding set of servers that either directly sign the transaction on the user's behalf, e.g., Web3Auth [32] uses threshold crypto for signing (or) retrieve secrets that are later used on the client side to sign transactions, e.g., Privy [33] only stores 1 out of 3 shares on the server.

zkLogin may also be viewed as a kind of 2PC between the OpenID provider and the app. However, the main difference between zkLogin and MPC wallets is that (a) zkLogin relies on trusting an established OpenID provider for the wallet's security instead of relying on the app to store secret shares (zkLogin only relies on the app for liveness), and (b) its novel features like discoverability, partial reveal and claimability that were not possible in any prior solution (MPC or otherwise).

6. https://aws.amazon.com/ec2/instance-types/m5/
7. https://cloud.google.com/compute/docs/general-purpose-machines

The approach of embedding arbitrary data into the nonce in the OpenID Connect protocol draws inspiration from recent works [6], [34]. However, a problem with these works is that they require showing the sensitive ID token to the verifier. This is a big issue if the verifier is a public blockchain.

Finally, ZK Address Abstraction [8] is the closest to our work: like us, they also use a general-purpose ZKP over the JWT to authenticate blockchain transactions. However, their approach has a few drawbacks which makes it impractical: (a) they tie the transaction closely into the ZKP, so the user would have to generate a new proof for every transaction, (b) they assume providers use a ZK-friendly signature scheme EDDSA that is not currently used by any popular OpenID providers and (c) they require the user to generate the ZKP on their own. Instead, we leverage the nonce embedding trick to reuse a single proof across many transactions and offload proof generation in a trustless manner.

7. Conclusion

We have introduced zkLogin, a novel approach for authenticating blockchain users by leveraging the widely-adopted OpenID Connect authentication framework. Crucially, the security of a zkLogin-based blockchain account relies solely on the security of the underlying authentication method without the need for additional trusted third parties.

At the heart of zkLogin is a mechanism that utilizes a (signed) OpenID token to authorize arbitrary messages. We formalized zkLogin as a Tagged Witness Signature, an extension of Signatures of Knowledge capturing the leakage of old tokens. We have employed Zero-Knowledge Proofs to conceal all sensitive details in an OpenID token. Additionally, the inclusion of a salt effectively obscures any connection between an individual's off-chain and on-chain accounts.

We have validated zkLogin's real-world viability with a fully functional implementation that is also currently deployed in a live production environment.

References

[1] K. Huang, "Why did ftx collapse? here's what to know." The New York Times, Nov 2022. [Online]. Available: https://www.nytimes.com/2022/11/10/technology/ftx-binance-crypto-explained.html

[2] C. Harkin, "Mt. gox rehabilitation plan worth billions in compensation approved, finalization to follow," CoinDesk, Oct 2021.

[3] D. Z. Morris. (2021, Jun) Gerald cotten and quadriga: Unraveling crypto's biggest mystery. CoinDesk. Accessed: Oct 27, 2023. [Online]. Available: https://www.coindesk.com/markets/2021/06/29/gerald-cotten-and-quadriga-unraveling-cryptos-biggest-mystery/

[4] P. Rizzo. (2017, Jul) The big news behind the btc-e arrest and mt gox connection. CoinDesk. Accessed: Oct 27, 2023. [Online]. Available: https://www.coindesk.com/markets/2017/07/26/the-big-news-behind-the-btc-e-arrest-and-mt-gox-connection/

[5] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore. (2014) Openid connect core 1.0 incorporating errata set 1. https://openid.net/specs/openid-connect-core-1_0.html. OpenID Foundation. Accessed: Oct 27, 2023.

[6] S. Palladino, "Sign in with google to your identity contract," 2019. [Online]. Available: https://forum.openzeppelin.com/t/sign-in-with-google-to-your-identity-contract-for-fun-and-profit/1631

[7] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof-systems (extended abstract)," in *17th ACM STOC*. ACM Press, May 1985, pp. 291–304.

[8] S. Park, J. H. Lee, S. Lee, J. H. Chun, H. Cho, M. Kim, H. K. Cho, and S.-M. Moon, "Beyond the blockchain address: Zero-knowledge address abstraction," Cryptology ePrint Archive, Report 2023/191, 2023, https://eprint.iacr.org/2023/191.

[9] S. Blackshear, A. Chursin, G. Danezis, A. Kichidis, L. Kokoris-Kogias, X. Li, M. Logan, A. Menon, T. Nowacki, A. Sonnino *et al.*, "Sui lutris: A blockchain combining broadcast and consensus," *arXiv preprint arXiv:2310.18042*, 2023.

[10] M. Chase and A. Lysyanskaya, "On signatures of knowledge," in *CRYPTO 2006*, ser. LNCS, C. Dwork, Ed., vol. 4117. Springer, Heidelberg, Aug. 2006, pp. 78–96.

[11] J. Groth, "On the size of pairing-based non-interactive arguments," in *EUROCRYPT 2016, Part II*, ser. LNCS, M. Fischlin and J.-S. Coron, Eds., vol. 9666. Springer, Heidelberg, May 2016, pp. 305–326.

[12] M. Bellés-Muñoz, M. Isabel, J. L. Muñoz-Tapia, A. Rubio, and J. Baylina, "Circom: A circuit description language for building zero-knowledge applications," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[13] A. Shamir, "Identity-based cryptosystems and signature schemes," in *CRYPTO'84*, ser. LNCS, G. R. Blakley and D. Chaum, Eds., vol. 196. Springer, Heidelberg, Aug. 1984, pp. 47–53.

[14] Technology Review, "Cryptography, ai, and the labeling problem: How c2pa aims to establish digital provenance," *Technology Review*, 07 2023. [Online]. Available: https://www.technologyreview.com/2023/07/28/1076843/cryptography-ai-labeling-problem-c2pa-provenance/

[15] C2PA, "Content Credentials : C2PA Technical Specification," 2023, accessed: [Nov 29, 2023]. [Online]. Available: https://c2pa.org/specifications/specifications/1.4/specs/C2PA_Specification.html#_trust_model

[16] M. Jones, "JSON Web Algorithms (JWA) Section 3," Internet Requests for Comments, Internet Engineering Task Force, RFC 7518, May 2015, section 3. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7518#section-3

[17] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger, "Poseidon: A new hash function for {Zero-Knowledge} proof systems," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[18] H. Krawczyk, "Simple forward-secure signatures from any signature scheme," in *ACM CCS 2000*, D. Gritzalis, S. Jajodia, and P. Samarati, Eds. ACM Press, Nov. 2000, pp. 108–115.

[19] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore. (2014) Openid connect core 1.0 incorporating errata set 1: Subject identifier types. https://openid.net/specs/openid-connect-core-1_0.html#SubjectIDTypes. OpenID Foundation. Accessed: Oct 27, 2023.

[20] E. Heilman, L. Mugnier, A. Filippidis, S. Goldberg, S. Lipman, Y. Marcus, M. Milano, S. Premkumar, and C. Unrein, "Openpubkey: Augmenting openid connect with user held signing keys," *Cryptology ePrint Archive*, 2023.

[21] A. Nilsson, P. N. Bideh, and J. Brorsson, "A survey of published attacks on intel sgx," *arXiv preprint arXiv:2006.13598*, 2020.

[22] International Civil Aviation Organization (ICAO), "Machine readable travel documents. part 10: Logical data structure (lds) for storage of biometrics and other data in the contactless integrated circuit (ic)," ICAO, Tech. Rep., 2010, https://www.icao.int/publications/Documents/9303_p10_cons_en.pdf.

[23] M. Rosenberg, J. White, C. Garman, and I. Miers, "zk-creds: Flexible anonymous credentials from zksnarks and existing identity infrastructure," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 790–808.

[24] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable, transparent, and post-quantum secure computational integrity," Cryptology ePrint Archive, Report 2018/046, 2018, https://eprint.iacr.org/2018/046.

[25] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge," Cryptology ePrint Archive, Report 2019/953, 2019, https://eprint.iacr.org/2019/953.

[26] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. P. Ward, "Marlin: Preprocessing zkSNARKs with universal and updatable SRS," in *EUROCRYPT 2020, Part I*, ser. LNCS, A. Canteaut and Y. Ishai, Eds., vol. 12105. Springer, Heidelberg, May 2020, pp. 738–768.

[27] A. Kosba, C. Papamanthou, and E. Shi, "xjsnark: A framework for efficient verifiable computation," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 944–961.

[28] iden3, "rapidsnark," https://github.com/iden3/rapidsnark, 2023.

[29] "Magic," https://magic.link.

[30] "Dauth," https://dauth.gitbook.io.

[31] "Face wallet." [Online]. Available: https://facewallet.xyz/

[32] "Web3auth," https://web3auth.io.

[33] "Privy," https://docs.privy.io.

[34] E. Heilman, L. Mugnier, A. Filippidis, S. Goldberg, S. Lipman, Y. Marcus, M. Milano, S. Premkumar, and C. Unrein, "OpenPubkey: Augmenting OpenID connect with user held signing keys," Cryptology ePrint Archive, Report 2023/296, 2023, https://eprint.iacr.org/2023/296.

[35] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game or A completeness theorem for protocols with honest majority," in *19th ACM STOC*, A. Aho, Ed. ACM Press, May 1987, pp. 218–229.

[36] M. Chase, M. Kohlweiss, A. Lysyanskaya, and S. Meiklejohn, "Malleable proof systems and applications," in *EUROCRYPT 2012*, ser. LNCS, D. Pointcheval and T. Johansson, Eds., vol. 7237. Springer, Heidelberg, Apr. 2012, pp. 281–300.

[37] J. Groth and A. Sahai, "Efficient non-interactive proof systems for bilinear groups," in *EUROCRYPT 2008*, ser. LNCS, N. P. Smart, Ed., vol. 4965. Springer, Heidelberg, Apr. 2008, pp. 415–432.

[38] S. Bowe, A. Gabizon, and I. Miers, "Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model," Tech. Rep. 2017/1050, Oct. 26, 2017.

[39] B. Abdolmaleki, K. Baghery, H. Lipmaa, and M. Zajac, "A subversion-resistant SNARK," in *ASIACRYPT 2017, Part III*, ser. LNCS, T. Takagi and T. Peyrin, Eds., vol. 10626. Springer, Heidelberg, Dec. 2017, pp. 3–33.

[40] G. Fuchsbauer, "Subversion-zero-knowledge SNARKs," in *PKC 2018, Part I*, ser. LNCS, M. Abdalla and R. Dahab, Eds., vol. 10769. Springer, Heidelberg, Mar. 2018, pp. 315–347.

[41] E. Ben-Sasson, A. Chiesa, M. Green, E. Tromer, and M. Virza, "Secure sampling of public parameters for succinct zero knowledge proofs," in *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2015, pp. 287–304.

[42] M. Maller, S. Bowe, M. Kohlweiss, and S. Meiklejohn, "Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings," in *ACM CCS 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. ACM Press, Nov. 2019, pp. 2111–2128.

Appendix A. Omitted Definitions

Digital Signature Schemes: A signature scheme $\mathsf{S} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ with message space M consists of three algorithms, defined as follows:

$\mathsf{Gen}(1^\lambda) \to (\mathsf{sk}, \mathsf{pk})$: is a randomized algorithm that on input the security parameter λ, returns a pair of keys $(\mathsf{sk}, \mathsf{pk})$, where sk is the signing key and pk is the verification key.

$\mathsf{Sign}(\mathsf{sk}, m) \to \sigma$ takes as input the signing key sk, and a message m, and returns a signature σ.

$\mathsf{Verify}(\mathsf{pk}, m, \sigma) \to 0/1$ is a deterministic algorithm that takes as input the verification key pk, a message m, and a signature σ, and outputs 1 (accepts) if verification succeeds, and 0 (rejects) otherwise.

A signature scheme satisfies correctness if for all λ, $m \in \mathsf{M}$, and every signing-verification key pair $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$, we have, $\mathsf{Verify}(\mathsf{pk}, m, \mathsf{Sign}(\mathsf{sk}, m)) = 1$.

Definition 5 (EUF-CMA)**.** *Let $\mathsf{S} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ be a signature scheme. The advantage of a PPT adversary $\mathcal{A}$ playing the security game described in Fig. 8, is defined as:*

$$Adv^{\mathsf{EUF-CMA}}_{\mathsf{S},\mathcal{A}}(\lambda) := \Pr[\mathbf{Game}^{\textit{EUF-CMA}}_{\mathcal{A},\mathsf{TWS}}(1^\lambda) = 1]$$

S achieves Existential Unforgeability under Chosen Message Attacks (EUF-CMA) if we have $Adv^{\mathsf{EUF-CMA}}_{\mathsf{S},\mathcal{A}}(\lambda) \leq \mathsf{negl}(\lambda)$.

$\mathbf{Game}^{\text{EUF-CMA}}_{\mathcal{A},\mathsf{TWS}}(1^\lambda)$:	$\mathcal{O}^{\mathsf{Sign}}(m)$:
$(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda)$	$\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, m)$
$(m^*, \sigma^*) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}^{\mathsf{Sign}}(\cdot)}(\mathsf{pk})$	$\mathcal{Q}_s \leftarrow \mathcal{Q}_s \cup \{m\}$
return $\big(m^* \notin \mathcal{Q}_s \wedge$	return σ
$\mathsf{Verify}(\mathsf{pk}, m^*, \sigma^*)\big)$	

Figure 8: The EUF-CMA security game.

Zero-Knowledge Proofs: Non-interactive zero knowledge (NIZK) [35] proof as a strong cryptographic primitive enables a prover to convince a (sceptical) verifier about the truth of a statement without disclosing any additional information in one round of communication. A NIZK can be build in two possible settings: either in Random Oracle Model (ROM) or in the Common Reference String (CRS) model. Next we recall the definition of NIZK proofs in the CRS model and list their main security properties.

Definition 6 (Non-Interactive Zero-Knowledge Proofs)**.** *Let $\mathcal{R}$ be an NP-relation, the language $\mathcal{L}_\mathcal{R}$ can be defined as $\mathcal{L}_\mathcal{R} = \{x \,|\, \exists\ w \text{ s.t. } (x, w) \in \mathcal{R}\}$, where x and w denote public statement and secret witness, respectively. A NIZK, denoted by Π, for $\mathcal{R}$ consists of three main PPT algorithms $\Pi = (\mathsf{Gen}, \mathsf{Prove}, \mathsf{Verify})$ defined as follows:*

- $\Pi.\mathsf{Gen}(1^\lambda, \mathcal{R}) \to \mathsf{CRS}$*: The CRS generation algorithm takes the unary representation of the security parameter λ and relation $\mathcal{R}$ as inputs and returns a set of common reference string CRS as output.*
- $\mathsf{Prove}(\mathsf{CRS}, x, w) \to \pi$*: The prove algorithm takes CRS, a public statement x and a secret witness w as inputs, and it then returns a proof π as output.*
- $\mathsf{Verify}(\mathsf{CRS}, x, \pi) \to 0/1$*: The verify algorithm takes CRS, a public statement x and a proof π as input, and it then returns a bit indicating either the acceptance, 1, or rejection, 0, as output.*

Informally speaking, a NIZK proof has three main security properties: Completeness, Zero-Knowledge and soundness (extractability), which we formally recall them in below:

Definition 7 (Completeness)**.** *A NIZK proof, Π, is called complete, if for all security parameters, λ, and all pairs of valid $(x, w) \in R$ we have,*

$$\Pr\left[\begin{array}{c}\mathsf{CRS} \leftarrow \mathsf{Gen}(1^\lambda) : \\ \mathsf{Verify}(\mathsf{CRS}, x, \mathsf{Prove}(\mathsf{CRS}, x, w)) = 1\end{array}\right] \geq 1 - \mathsf{negl}(\lambda)\ .$$

Definition 8 (Zero-Knowledge)**.** *A NIZK proof system, Π, for a given relation $\mathcal{R}$ and its corresponding language $\mathcal{L}_\mathcal{R}$, we define a pair of algorithms $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2)$ as the simulator. The simulator operates such that $\mathsf{Sim}'(\mathsf{CRS}, \mathsf{tpd}, x, w) = \mathsf{Sim}_2(\mathsf{CRS}, \mathsf{tpd}, x)$ when $(x, w) \in \mathcal{R}$, and $\mathsf{Sim}'(\mathsf{CRS}, \mathsf{tpd}, x, w) = \perp$ when $(x, w) \notin \mathcal{R}$, where tpd is a trapdoor. For $b \in \{0, 1\}$, we define the experiment $\mathsf{ZK}^\Pi_{b,\mathsf{Sim}}(1^\lambda, \mathcal{A})$ in Fig. 9. The associated advantage of an adversary $\mathcal{A}$ is defined as*

$$Adv^{\mathsf{ZK}}_{\Pi,\mathcal{A},\mathsf{Sim}}(\lambda) := \left|\begin{array}{c}\Pr[\mathsf{ZK}^\Pi_{0,\mathsf{Sim}}(1^\lambda, \mathcal{A}) = 1] - \\ \Pr[\mathsf{ZK}^\Pi_{1,\mathsf{Sim}}(1^\lambda, \mathcal{A}) = 1]\end{array}\right| .$$

A NIZK proof system Π achieves perfect and computational zero-knowledge, w.r.t a simulator $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2)$, if for all PPT adversaries $\mathcal{A}$ we have $Adv^{\mathsf{ZK}}_{\Pi,\mathcal{A},\mathsf{Sim}}(\lambda) = 0$, and $Adv^{\mathsf{ZK}}_{\Pi,\mathcal{A},\mathsf{Sim}}(\lambda) \leq \mathsf{negl}(\lambda)$, respectively.

$\mathsf{ZK}^\Pi_{0,\mathsf{Sim}}(1^\lambda, \mathcal{A})$	$\mathsf{ZK}^\Pi_{1,\mathsf{Sim}}(1^\lambda, \mathcal{A})$
$\mathsf{CRS} \leftarrow \mathsf{Gen}(1^\lambda)$	$(\mathsf{CRS}, \mathsf{tpd}) \leftarrow \mathsf{Sim}_1(1^\lambda)$
$\alpha \leftarrow \mathcal{A}^{\mathsf{Prove}(\mathsf{CRS}, \cdot, \cdot)}(\mathsf{CRS})$	$\alpha \leftarrow \mathcal{A}^{\mathsf{Sim}'(\mathsf{CRS}, \mathsf{tpd}, \cdot, \cdot)}(\mathsf{CRS})$
return α	return α

Figure 9: Zero-knowledge security property of a NIZK, Π.

Definition 9 (Extractability [36])**.** *A NIZK proof system Π for a relation $\mathcal{R}$ and the language L is called extractable if there exists a pair of algorithms $\mathsf{Ext} := (\mathsf{Ext}_1, \mathsf{Ext}_2)$ called extractors with the following advantage for all PPT adversaries $\mathcal{A}$:*

$$\begin{aligned} Adv^{\mathrm{CRS}}_{\Pi,\mathcal{A}} :=\big|\, &\Pr[\mathsf{CRS} \leftarrow \mathsf{Gen}(1^\lambda); 1 \leftarrow \mathcal{A}(\mathsf{CRS})] - \\ &\Pr[(\mathsf{CRS}, \mathsf{st}) \leftarrow \mathsf{Ext}_1(1^\lambda); 1 \leftarrow \mathcal{A}(\mathsf{CRS})]\big| , \end{aligned}$$

and

$$Adv^{\mathsf{Ext}}_{\Pi,\mathcal{A}}(\lambda) \coloneqq \Pr\left[\begin{array}{l}(\mathsf{CRS}_{\mathsf{Ext}}, \mathsf{st}_{\mathsf{Ext}}) \leftarrow \mathsf{Ext}_1(1^\lambda)\\ (x,\pi) \leftarrow \mathcal{A}(\mathsf{CRS}_{\mathsf{Ext}}) :\\ \mathsf{Verify}(\mathsf{CRS}_{\mathsf{Ext}}, x, \pi) = 1 \wedge\\ (x, \mathsf{Ext}_2(\mathsf{CRS}_{\mathsf{Ext}}, \mathsf{st}_{\mathsf{Ext}}, x, \pi)) \notin \mathcal{R}\end{array}\right].$$

A NIZK proof system Π is called extractable, w.r.t an extractor $\mathsf{Ext} = (\mathsf{Ext}_1, \mathsf{Ext}_2)$, if $Adv^{\mathsf{CRS}}_{\Pi,\mathcal{A}} \leq \mathsf{negl}(\lambda)$ and $Adv^{\mathsf{Ext}}_{\Pi,\mathcal{A}}(\lambda) \leq \mathsf{negl}(\lambda)$. Additionally, we refer to an extractable NIZK proof as a non-interactive zero-knowledge proof of knowledge, or NIZKPoK in short.

***Succinctness**. Zero-Knowledge Succinct Non-Interactive Arguments of Knowledge, zkSNARK in short, are NIZKPoK proofs that adhere to succinctness requirements. These proofs maintain communication complexity (proof size) at sublinear levels, and in some cases, the verifier's computational workload remains sublinear, regardless of the size of the witness. In this paper, we primarily concentrate on zkSNARKs, ensuring that the proofs are short and verification cost is low while the mentioned security definitions for NIZK remain applicable for them.*

Appendix B. Omitted Proofs

Theorem 1. *Given that Π satisfies knowledge-soundness, and JWT and Sig are EUF-CMA secure, and $H(\cdot)$ is a collision-resistant hash function, the proposed Tagged Witness Signature, $\Sigma_{\mathsf{zkLogin}}$, achieves unforgeability, defined in Def. 3.*

Proof. We prove this theorem using a sequence of games.

Game 0. This game is the same as the one defined in Def. 3.

Game 1. This proceeds as in the real construction, except that, while verifying the forged signature $(\mathsf{tag}^*, M^*, vk_u^*, T^*, \sigma_u^*, \pi^*)$, the challenger also checks whether vk_u^* was used in a previous $\mathcal{O}^{\mathsf{Wit}}(\cdot)$ or $\mathcal{O}^{\mathsf{Sign}}(\cdot)$ responses. If it was not present in any $\mathcal{O}^{\mathsf{Wit}}(\cdot)$ responses, but was used in an $\mathcal{O}^{\mathsf{Sign}}(\cdot)$ response for a different message, then the adversary loses.

This game is indistinguishable from the real protocol by the EUF-CMA security of Sig, i.e., Game 0.

Game 2. This proceeds as in Game 1, except that, while verifying the forged signature $(\mathsf{tag}^*, M^*, vk_u^*, T^*, \sigma_u^*, \pi^*)$, the challenger additionally uses the knowledge extractor for Π to extract witness $(\mathsf{jwt}^*, \mathsf{salt}^*, r^*)$ and the adversary loses if $P_{zk}((\mathsf{pk}_{OP}^{cur}, \mathsf{iss}, \mathsf{zkaddr}^*, T^*, vk_u^*), (\mathsf{jwt}^*, \mathsf{salt}^*, r^*))$ is false.

This game is indistinguishable from Game 1 by the knowledge-soundness of Π.

Game 3. This game runs identically to Game 2, except the adversary loses if there was no $\mathcal{O}^{\mathsf{Wit}}(\cdot)$ response of the form $(\mathsf{jwt}^*, \cdots)$.

This game is indistinguishable from Game 2 by the EUF-CMA security of $JWT.\mathsf{Issue}$.

Given the indistinguishability of the real protocol and Game 3, the unforgeability adversary succeeds in Game 3 with a probability that is negligibly away from that in the real game. Now observe that the Game 3 the adversary never wins. Hence the protocol is an unforgeable Tagged Witness Signature. □

Theorem 2. *Given that Π satisfies zero-knowledge, the Tagged Witness Signature, $\Sigma_{\mathsf{zkLogin}}$, achieves witness hiding, defined in Def. 4.*

Proof. The algorithm $\mathsf{SimGen}(\cdot)$ generates the $zkcrs$ in simulation mode with trapdoor trap. For any signing query $\mathsf{tag} = (\mathsf{pk}_{OP}^{cur}, \mathsf{iss}, \mathsf{zkaddr}, T)$, the oracle $\mathcal{O}^{\mathsf{SimSign}}(\cdot)$ generates a fresh key-pair $(vk_u, sk_u) \leftarrow Sig.\mathsf{Gen}(1^\lambda)$, sets $\mathsf{zkx} \leftarrow (\mathsf{pk}_{OP}^{cur}, \mathsf{iss}, \mathsf{zkaddr}, T, vk_u)$, computes $\sigma_u \leftarrow Sig.\mathsf{Sign}(sk_u, M)$, computes $\pi \leftarrow \Pi.Sim(zkcrs, \mathsf{trap}, \mathsf{zkx})$, and outputs signature (vk_u, T, σ_u, π).

This game is indistinguishable from real protocol by the ZK property of Π. Hence the proposed Tagged Witness Signature achieves witness hiding. □

Appendix C. A Generic Tagged Witness Signature Construction

We construct a Tagged Witness Signature for the signature verification predicate of a signature scheme $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$. This instantiation is built upon two primary components: a commitment scheme, $\mathsf{Com} : \{0,1\}^* \to \{0,1\}^\lambda$, and a non-interactive zero-knowledge (NIZK-PoK) scheme Π. It closely follows the proposed SoK construction in [10].

For a key-pair $(\mathsf{sk}, \mathsf{vk})$ sampled by $\Sigma.\mathsf{KeyGen}(\lambda)$, we define

$$P_{\mathsf{vk}}(t, w) \equiv \Sigma.\mathsf{Verify}(\mathsf{vk}, w, t)$$

Now we describe the Tagged Witness Signature Σ_{sig} for the predicate P_{vk}.

$\mathsf{Gen}(\lambda)$:

- Let language $\mathcal{L}_{\mathsf{vk}} = \{(t, m, c) \mid \exists w, r : c = \mathsf{Com}(t, m, w; r) \wedge \mathsf{Verify}(\mathsf{vk}, w, t) = 1\}$
- Sample $\mathsf{CRS} \leftarrow \Pi.\mathsf{Gen}(\lambda, \mathcal{L}_{\mathsf{vk}})$.
- Output $pk := \mathsf{CRS}$

$\mathsf{Sign}(t, pk, w, M)$:

- If $P_{\mathsf{vk}}(t, w)$ is false, then output $\perp$.
- Sample random r.
- Compute $c \leftarrow \mathsf{Com}(t, M, w; r)$.
- Compute $\pi \leftarrow \Pi.\mathsf{Prove}(\mathsf{CRS}, (t, M, c), (w, r))$.
- Output $\sigma := (c, \pi)$.

$\mathsf{Verify}(t, pk, M, \sigma)$:

- Parse $(c, \pi) := \sigma$
- Parse $\mathsf{CRS} := pk$
- Output $\Pi.\mathsf{Verify}(\mathsf{CRS}, \pi, (t, M, c))$

Theorem 3. *Σ_{sig} is an unforgeable tagged witness signature scheme, given that Π satisfies knowledge-soundness, the signature scheme is EUF-CMA secure, and Com is a binding commitment scheme.*

Proof. We construct an attack on the unforgeability of the signature scheme used for witness, given an attack on Tagged Witness Signature unforgeability.

The Tagged Witness Signature unforgeability challenger proceeds as in the real construction, except that it uses the knowledge extractor of Π, and has access to the signing oracle $\mathsf{Sign}(\mathsf{sk}, \cdot)$.

For an $\mathcal{O}^{\mathsf{Sign}}$ query (t, M), the challenger queries the signing oracle with t, and receives w. Then it samples r and computes $c \leftarrow \mathsf{Com}(t, M, w; r)$ and $\pi \leftarrow \Pi.\mathsf{Prove}(\mathsf{CRS}, (t, M, c), (w, r))$ and outputs $\sigma = (c, \pi)$.

For an $\mathcal{O}^{\mathsf{Wit}}$ query t, the challenger queries the signing oracle with t, and receives w.

When it receives a forgery $\sigma^* = (t^*, M^*, c^*, \pi^*)$, such that π^* verifies, it extracts witness w^*, r^* such that $c^* = \mathsf{Com}(t^*, M^*, w^*; r^*) \wedge \mathsf{Verify}(\mathsf{vk}, w^*, t^*) = 1$. Then it outputs (t^*, w^*) to the signature scheme challenger.

Due to the knowledge-soundness of Π, we should have, with high probability, $c^* = \mathsf{Com}(t^*, M^*, w^*; r^*) \wedge \mathsf{Verify}(\mathsf{vk}, w^*, t^*) = 1$.

Due to the unforgeagibility of the signature scheme, we should have that t^* was, with high probability, queried by the challenger to the signature oracle, either to respond to an $\mathcal{O}^{\mathsf{Sign}}$ query or an $\mathcal{O}^{\mathsf{Wit}}$ query. If t^* was queried to $\mathcal{O}^{\mathsf{Wit}}$ then the TWS adversary loses. Otherwise, if (t^*, M^*) was queried to $\mathcal{O}^{\mathsf{Sign}}$, then also the TWS adversary loses. We only have to consider the case that for all (t^*, M') queried to $\mathcal{O}^{\mathsf{Sign}}$, we have $M' \neq M^*$. Then we have $\mathsf{Com}(t^*, M^*, w^*; r^*) = \mathsf{Com}(t^*, M', w^*; r')$ for some $(M', r') \neq (M^*, r^*)$. Due to the binding property of Com, this only holds with negligible probability. □

Theorem 4. *The above NIZK-based construction is a witness hiding Tagged Witness Signature, given that Π satisfies ZK and Com is a hiding commitment scheme.*

Proof. We prove this using a sequence of games.

Game 1. In this game, the challenger generates zkcrs in the simulation mode and holds the trapdoor. Note that for some systems like [11] this is identical to the real mode, while for [37] these are distinct. The proofs are produced using the simulation mode. This game is indistinguishable from the real protocol by the zk property of Π.

Game 2. In this game, the challenger produces fake signatures using the simulation trapdoor. So it samples $c \leftarrow \mathsf{Com}(0)$ with a fake proof π over (t, m, c). This game is indistinguishable from Game 1 by the hiding property of Com.

Now observe that we can use the Game 2 challenger as the $\mathsf{SimSign}$ protocol. □

Appendix D. Groth16 Ceremony

zkLogin employs the Groth16 zkSNARK construction as the most efficient zkSNARK to date, to instantiate the zero-knowledge proofs. However, this construction requires a circuit-specific Common Reference String (CRS) setup by a trusted party, we use well-known trust mitigation techniques to relax this trust assumption. We ran a ceremony protocol to generate this CRS which bases its security on the assumed honesty of a single party out of a large number of parties.

The ceremony essentially entailed a cryptographic multi-party computation (MPC) conducted by a diverse group of participants to produce this Common Reference String (CRS). This process followed the MPC protocol for reusable parameters, referred to as MMORPG, as detailed by Bowe, Gabizon, and Miers in [38]. The protocol roughly proceeds in 2 phases. The first phase yields a sequence of monomials, which are essentially powers of a secret value τ in the exponent of a generator of a pairing-friendly elliptic curve. This sequence takes the form of $g, g^{\tau}, g^{\tau^2}, \ldots, g^{\tau^n}$, where n is an upper bound of circuit size and $\tau = \prod_{i=1}^{\ell} \tau_i$ s.t. ℓ denotes the total number of contributors. Thereby it enables reducing the trust level to 1 out of the total number of contributors, i.e., ℓ. As this phase is not specific to any particular circuit, we have adopted the outcome of the Perpetual Powers of Tau[8], which is contributed by a sufficiently large community. However, in order to fully implement the trust minimization process, we must establish a ceremony for the second phase of setup, which is tailored to the zkLogin circuit.

In the presence of a coordinator, the MMORPG protocol allows an indefinite number of parties to participate in sequence, without the need of any prior synchronization or ordering. Each party needs to download the output of the previous party, generate randomness of its own, and then layer it on top of the received result, producing its own contribution, which is then relayed to the next party. The protocol guarantees security, if at least one of the participants follows the protocol faithfully, generates strong randomness and discards it reliably.

Since the MPC is sequential, each contributor had to wait until the previous contributor finished in order to receive the previous contribution, follow the MPC steps, and produce their own contribution. Due to this structure, participants waited in a queue while those who joined before them finished. To authenticate participants, each participant received a unique activation code. The activation code was the secret key of a signing key pair, which had a dual purpose: it allowed the coordination server to associate the participant's email with the contribution, and it verified the contribution with the corresponding public key.

Participants had two ways to contribute: through a browser or a docker. The browser option was the more user-friendly as all parts of the process happened in the browser. The Docker option required Docker setup but was more transparent—the Dockerfile and contributor source code

8. https://github.com/privacy-scaling-explorations/perpetualpowersoftau

are open-sourced and the whole process is verifiable. The browser option utilized snarkjs[9] while the Docker option utilized a forked version of Gurkan's implementation[10]. This provided software variety so that contributors could choose whichever method they trust most. In addition, participants could generate entropy via entering random text or making random cursor movements.

The zkLogin circuit and the ceremony client code were made open source and the links were made available to the participants to review before the ceremony, if they chose to do so. In addition, developer docs and an audit report on the circuit were posted for review. Challenge number 81 was adopted (resulting from 80 community contributions) from perpetual powers of tau in phase 1, which is circuit agnostic. The output of the Drand random beacon was applied to remove bias. After the phase 2 ceremony, the output of the Drand random beacon was applied again to remove bias from contributions.

The final CRS along with the transcript of every participant's contribution is available in a public repository. Contributors received both the hash of the previous contribution they were working on and the resulting hash after their contribution, displayed on-screen and sent via email. They can compare these hashes with the transcripts publicly available on the ceremony site. In addition, anyone is able to check that the hashes are computed correctly and each contribution is properly incorporated in the finalized parameters.

We also note that various other trust mitigation techniques have also been defined, including subversion-resistant zkSNARKs [39], [40] and Multi-Party Computation (MPC) [41]. Additionally, we leave the implementation of zkLogin using universal and updatable zkSNARKs such as Plonk [25] and Sonic [42], which removes the complex circuit-dependent setups, as an interesting future extension.

Appendix E. Miscellaneous Details

E.1. Escaped quote in a JSON key

For example, the below JSON can be parsed in two ways as shown by the start and end index markers. The key is `"sub"` in both but the value is different.

Listing 3: Quote inside a JSON key

```
{
    "sub": "110463452167303598383",
    ^                             ^
    "\\"sub": "110463452167303598382",
       ^                          ^
}
```

9. https://github.com/iden3/snarkjs
10. https://github.com/kobigurk/phase2-bn254