WHEN ZKPS AREN'T ENOUGH

AND WHAT IT MEANS FOR AGE VERIFICATION POLICY

Sofía Celi (Brave, University of Bristol)

https://eprint.iacr.org/2026/227

Outline

A bit of a complex talk (so bear with me ;))

- We'll walk through what it actually takes to ship a complex system: the full architecture, not just the proof
  - Zero Knowledge Attestation Systems (ZKA)
- We want to know:
  - Why do we think on deploying them
    - The current policy landscape on this
    - Under which security model
    - Under which assumptions from users
  - How they can fail in practice
- *Cool cryptography* but is it **good as it is**?

Outline

The design of a complex/distributed system (ZKA systems)

- ZKA systems are ones that use ZKPs to *attest something* (that an statement is true) about an *external byte stream* (sometimes "committed to") so that we can *authorize actions* based on said attestations:
  - Attest a user is **not fraudulent** so they **access other services**

    OR

  - In the the case of *zkLogin*, **authorize transactions**
- In theory, they are "straightforward"; in practice, they are not

Mise-en-scène

- We want to be able to prove *something while preserving privacy* (*)
  - To give users the ability to accesses services or perform actions by:
    - Attesting humanity
    - Attesting “age”
    - Attesting state
    - Attesting authentication
- We cannot send that *something* in the clear → use *zero knowledge proofs (ZKP)*

Mise-en-scène

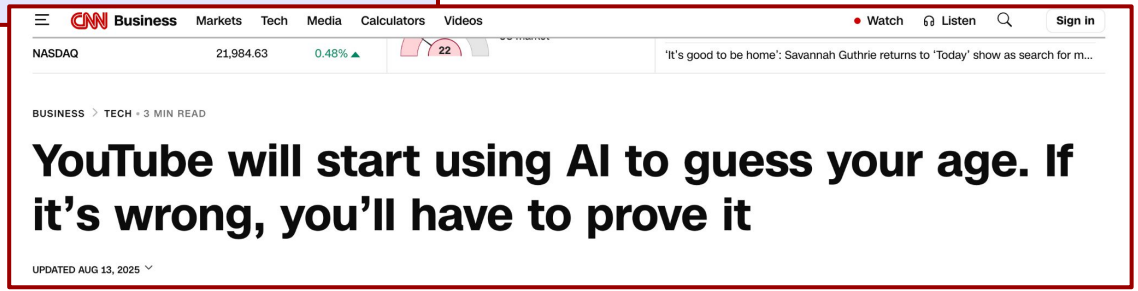
- Many designs…
- However, they follow the same “idea”:
  - Have an “authenticated” and “well-formed” crafted byte stream (optionally, private -so not hiding-)
  - Commit (to a degree) to said byte stream
  - Prove that is “well-formed”
  - Prove “facts” (or attributes) about the byte stream

- Examples: zkTLS, zkAuthorization, zkMiddleboxes, zkCompilation

Mise-en-scène

Why are we interested?

Update as of 3/9: Brazil's Digital Statute for Children and Adolescents (Digital ECA) goes into effect on March 17th. Learn more about how this impacts users in Brazil in our Help Center article here.



Mise-en-scène

- Instead of asking users to reveal their documents (sometimes, over and over) that state that they are “of age” (or any other statement: show your gender, that you are authorised), use a *ZKP*

- *This talk is the result of reverse-engineering, protocol evaluation, analysis and more*

For this talk

Keep in mind:

- We need an “authoritative” document:
  - Issued by an **authority** or it can be the result of an **authenticated interaction**
  - Said document should be **verifiable** (can have a signature, for example)
  - (Optional) Said verifiability can be publicly scrutinized
  - Said document contains **statements** that we can **walk over and reason about**
  - Said document **conforms to a grammar** and **semantic checks**
- The interaction is usually a standardised protocol that gives us assurances, at least, on authentication

For this talk

Keep in mind:

- If we want a ZKP over a document, we have to attest that:
  - The statement exists
  - The statement exists at *the correct place*
  - The statement *conforms to a grammar* (it is a string, for example, and follows the rules for strings)
- We call the two last points “prove parsing” in ZKP: currently costly

```
{
  "name": "Alice",
  "age": 30,
  "name": "Bob",
  "age": "25"
}
```

zkLogin

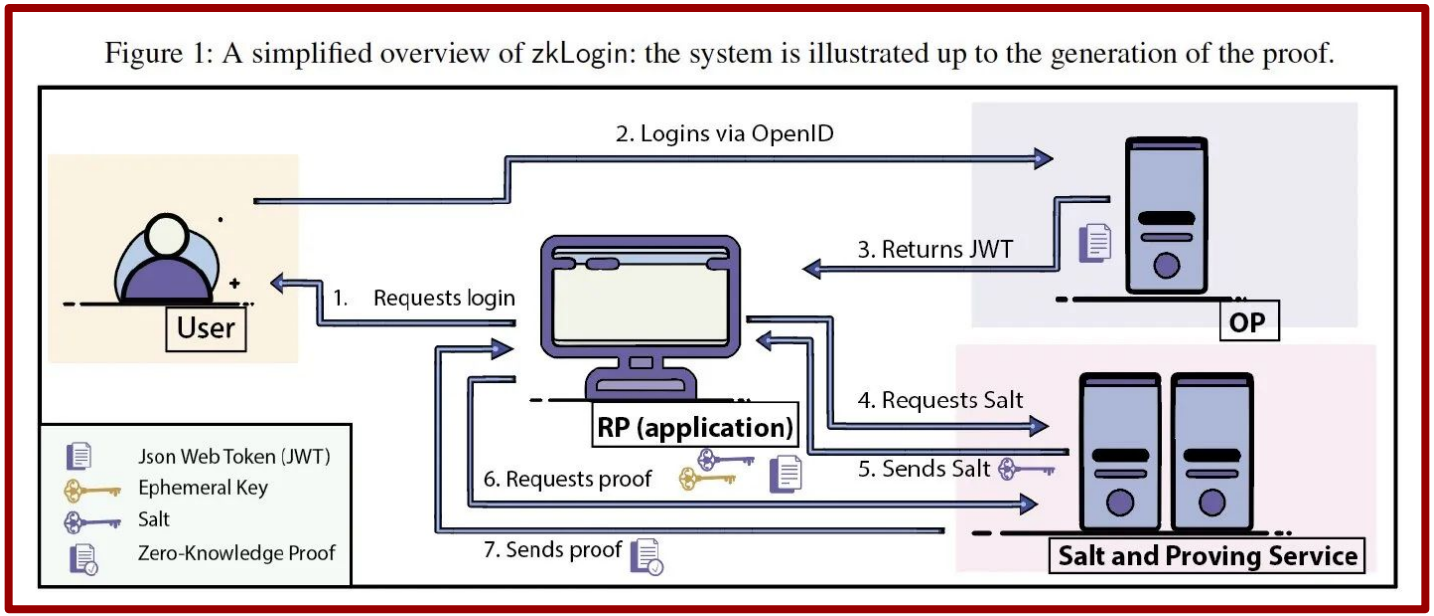
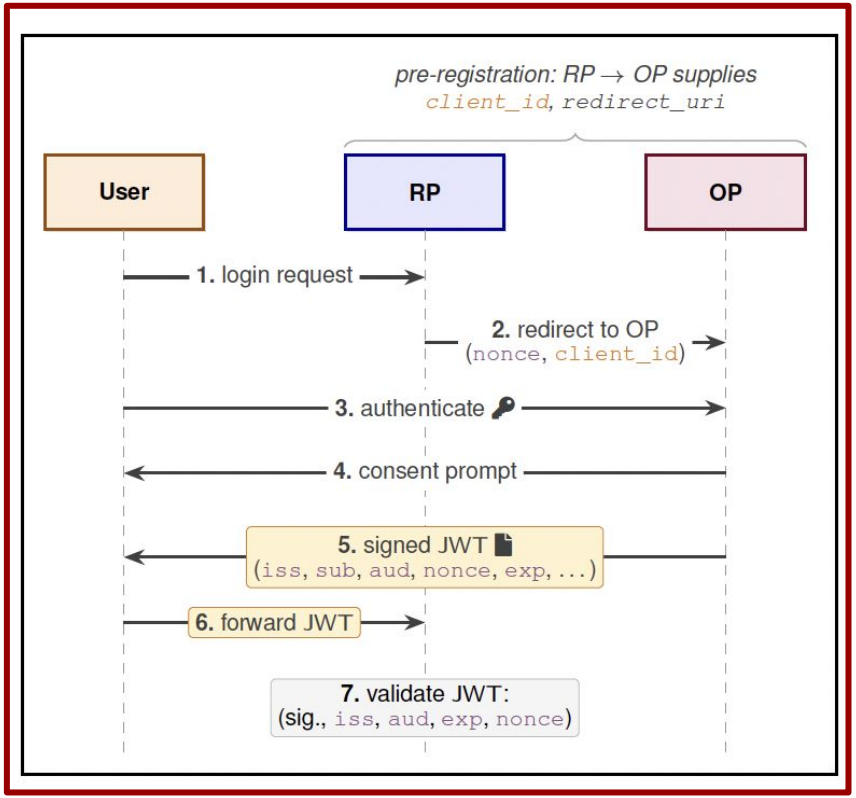


Figure 1: A simplified overview of zkLogin: the system is illustrated up to the generation of the proof.
2. Logins via OpenID
3. Returns JWT
1. Requests login
User
OP
RP (application)
4. Requests Salt
5. Sends Salt
6. Requests proof
7. Sends proof
Salt and Proving Service
Json Web Token (JWT)
Ephemeral Key
Salt
Zero-Knowledge Proof

zkLogin

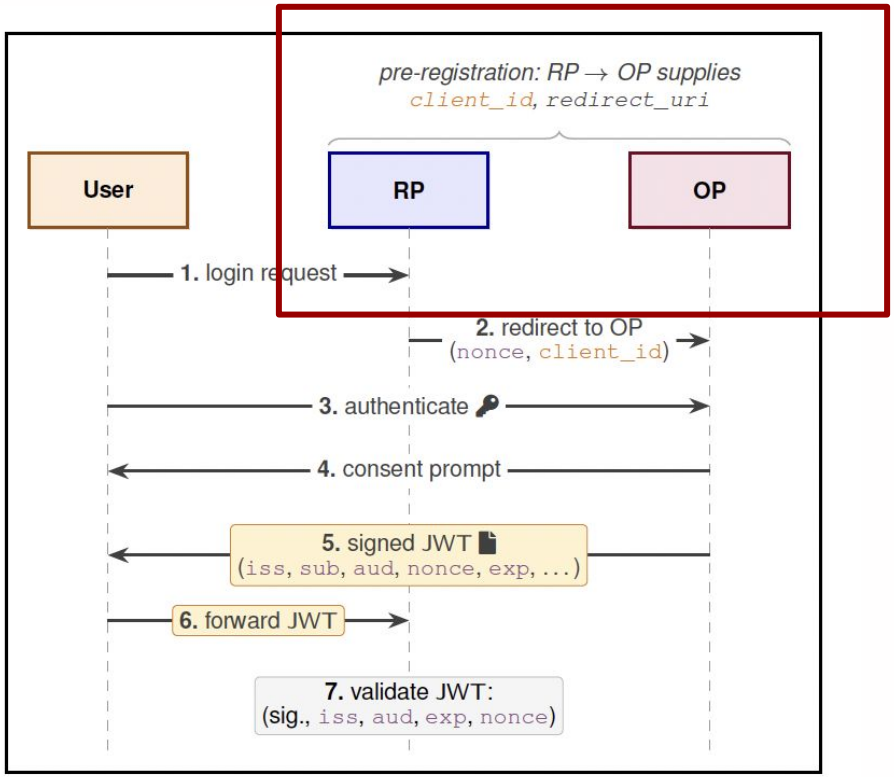
What is *zkLogin*?

- *zkLogin* is a protocol that allows to **authorise transactions** in a blockchain model by assuring a user is already authenticated via a ZKP of an OpenID Connect (OIDC) flow (signed JWT). It has several parties:
  - OIDC issuer or OP (*Issuer*)
  - An app/website where a user authenticates to using OIDC (*RP*)
  - *An external prover and/or salt service*
  - A user
  - A verifier (i.e., a blockchain)



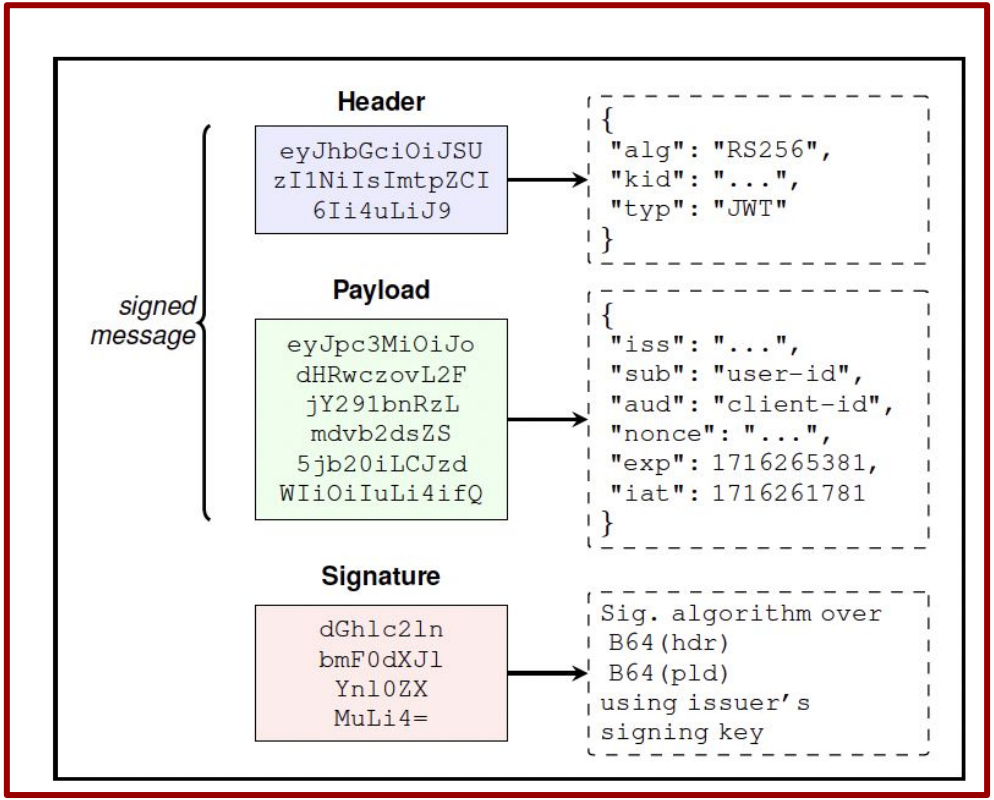
- OIDC issuer or OP (*Issuer*)
- An app/website where a user authenticates with using OIDC (*RP*)
- A user

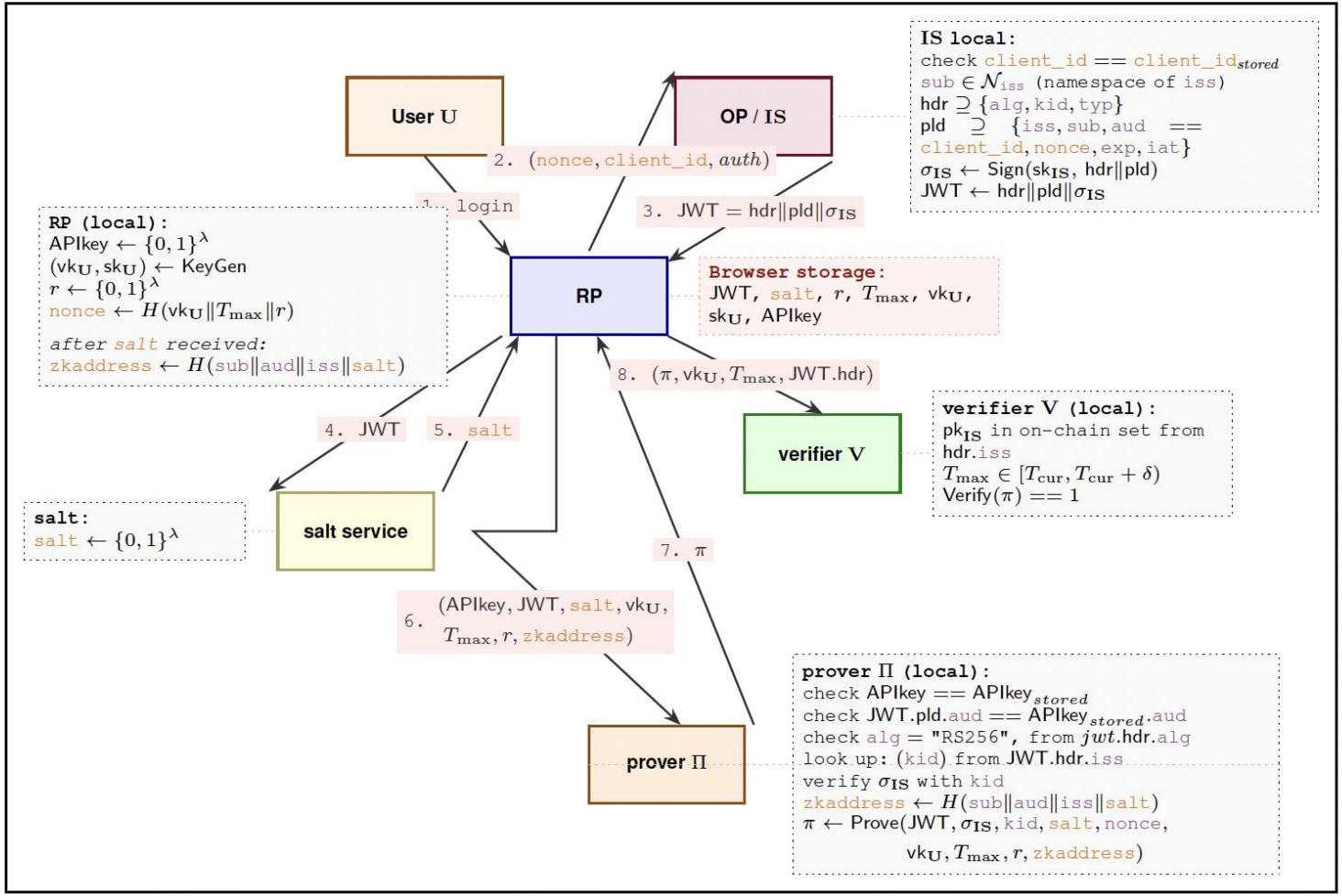
- *An external prover and/or salt service*
- A verifier (i.e, a blockchain)



- Note that there is a registration procedure here:
  - The IS/OP generates a client_id, which will act as *aud* for RP

- The system (a ZKA one) heavily relies on **JWT** and **OIDC**
- A signed JWT is:
  a. Header
  b. Payload:
     - With claims: aud, sub, iss, exp
  c. Signature



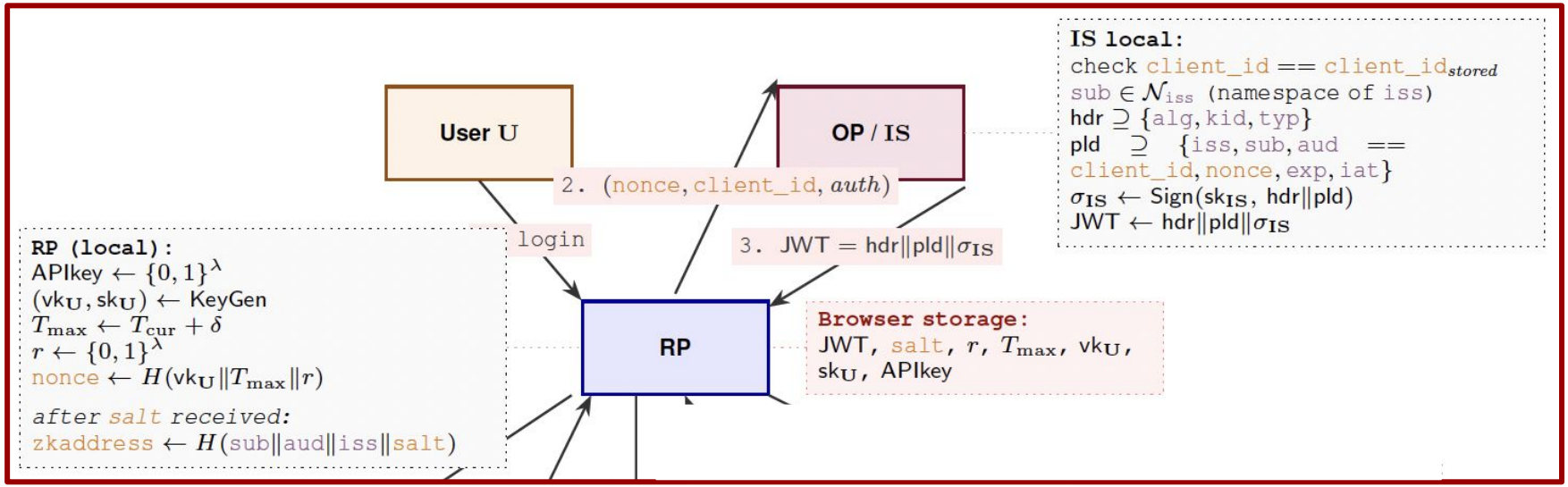

User U
OP / IS
2. (nonce, client_id, auth)
1. login
3. JWT = hdr||pld||σIS
RP
RP (local):
APIkey ← {0,1}^λ
(vkU, skU) ← KeyGen
r ← {0,1}^λ
nonce ← H(vkU||Tmax||r)
after salt received:
zkaddress ← H(sub||aud||iss||salt)
IS local:
check client_id == client_idstored
sub ∈ Niss (namespace of iss)
hdr ⊇ {alg, kid, typ}
pld ⊇ {iss, sub, aud == client_id, nonce, exp, iat}
σIS ← Sign(skIS, hdr||pld)
JWT ← hdr||pld||σIS
Browser storage:
JWT, salt, r, Tmax, vkU, skU, APIkey
8. (π, vkU, Tmax, JWT.hdr)
4. JWT
5. salt
verifier V
verifier V (local):
pkIS in on-chain set from hdr.iss
Tmax ∈ [Tcur, Tcur + δ)
Verify(π) == 1
salt service
salt:
salt ← {0,1}^λ
7. π
6. (APIkey, JWT, salt, vkU, Tmax, r, zkaddress)
prover Π
prover Π (local):
check APIkey == APIkeystored
check JWT.pld.aud == APIkeystored.aud
check alg = "RS256", from jwt.hdr.alg
look up: (kid) from JWT.hdr.iss
verify σIS with kid
zkaddress ← H(sub||aud||iss||salt)
π ← Prove(JWT, σIS, kid, salt, nonce, vkU, Tmax, r, zkaddress)

zkLogin

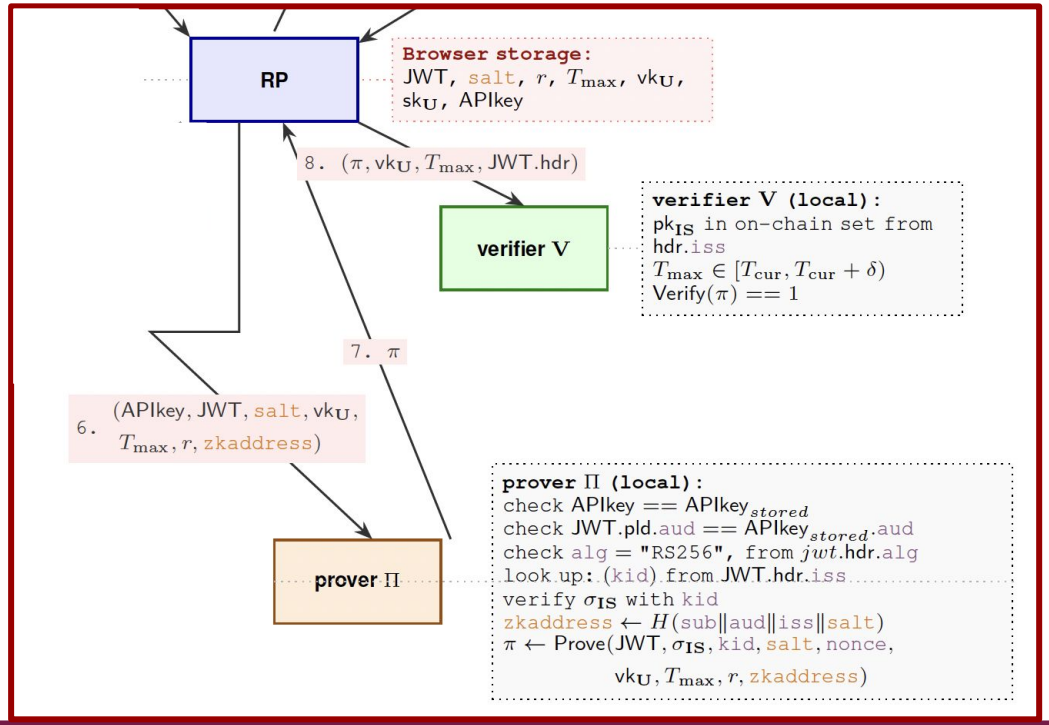
How does it work?

a. A **RP** generates ephemeral keys (*sk, vk*), and sets an expiration (*T_max*). It generates a *nonce* as *H(vk // T_max // r)*, and sends an authentication request via OIDC with this *nonce*
b. A **user**, via the RP, authenticates with an **OIDC provider (OP)** (they are forwarded to it). If successful, the OP returns a **signed JWT** (with the *nonce* in it)
c. The RP takes the JWT and asks for a salt (to generate an unlinkable wallet address: *zkaddress = H(sub // aud // iss // salt)*) to an external **salt service**
d. The RP takes the JWT, *nonce, vk, T_max, r, salt* and submits it **fully** to an external **proving service**

zkLogin


User U
OP / IS
RP
2. (nonce, client_id, auth)
login
3. JWT = hdr||pld||σIS
RP (local):
APIkey ← {0,1}^λ
(vkU, skU) ← KeyGen
Tmax ← Tcur + δ
r ← {0,1}^λ
nonce ← H(vkU||Tmax||r)
after salt received:
zkaddress ← H(sub||aud||iss||salt)
IS local:
check client_id == client_id_stored
sub ∈ Niss (namespace of iss)
hdr ⊇ {alg, kid, typ}
pld ⊇ {iss, sub, aud == client_id, nonce, exp, iat}
σIS ← Sign(skIS, hdr||pld)
JWT ← hdr||pld||σIS
Browser storage:
JWT, salt, r, Tmax, vkU, skU, APIkey

zkLogin


RP
Browser storage:
JWT, salt, r, $T_{\max}$, $\mathsf{vk_U}$, $\mathsf{sk_U}$, APIkey
8. $(\pi, \mathsf{vk_U}, T_{\max}, \mathsf{JWT.hdr})$
verifier V
verifier V (local):
$\mathsf{pk_{IS}}$ in on-chain set from hdr.iss
$T_{\max} \in [T_{\mathrm{cur}}, T_{\mathrm{cur}} + \delta)$
$\mathsf{Verify}(\pi) == 1$
7. π
6. (APIkey, JWT, salt, $\mathsf{vk_U}$, $T_{\max}$, r, zkaddress)
prover Π
prover Π (local):
check APIkey == APIkey_{stored}
check JWT.pld.aud == APIkey_{stored}.aud
check alg = "RS256", from jwt.hdr.alg
look up: (kid) from JWT.hdr.iss
verify $\sigma_{\mathbf{IS}}$ with kid
zkaddress $\leftarrow H(\mathsf{sub} \| \mathsf{aud} \| \mathsf{iss} \| \mathsf{salt})$
$\pi \leftarrow \mathsf{Prove}(\mathsf{JWT}, \sigma_{\mathbf{IS}}, \mathsf{kid}, \mathsf{salt}, \mathsf{nonce}, \mathsf{vk_U}, T_{\max}, r, \mathsf{zkaddress})$

zkLogin

- The proving service **generates a ZKP**. For this, it:
  - Performs an *ad-hoc selective parsing procedure*
- We call this *ad-hoc selective parsing* (note that this is **not a JWT or JSON compliant parser**)
  - Executed only for a single instance of some claims (“iss”, “aud”, “sub”, “nonce”)

zkLogin

```
{
  "name": "Alice",
  "age": 30,
  "name": "Bob",
  "age": "25"
}
```

Ad-hoc selective parsing procedure:

1. Checks that each substring ends in either a "," or a "}"
1. Checks that in the middle of the substring there is a ":"
2. Interprets
   a. everything prior to ":" as a *key*,
   b. everything after as a *value*,
   c. checks that they are strings by checking that they use ""

zkLogin

How does it work?

- The proving service **generates a ZKP**. The ZKP attests that:
  - The *ad-hoc selective parsing* was executed correctly
  - That the *nonce* value was computed correctly
  - The JWT's signature is signed by an issuer who publishes their keys independently
  - That the address uses the *salt*

The proof is returned to the RP, who then signs (σ) a transaction (the action) with *sk,* and posts for verification *vk, T_max,* σ*,* π*, JWT's header, iss*

zkLogin Threat Model

- *zkLogin* assumes that:
  - The backend services (prover and salt service) are untrusted
  - The OIDC issuer is trusted
  - The frontend is trusted: the claim is that it is “subject to public scrutiny”

However, *zkLogin* is executed in a system where:

- The trust boundary of OIDC is outside their control: is federated and OIDC issuers are continuously misconfigured
- The frontend can have injected scripts, malicious extensions, compromised dependencies, or brief device access, as it is in the browser

zkLogin

- **We don’t think this is a realistic threat model, and it is the culprit of vulnerabilities**
  - The system further uses JWTs, which have been shown to have many security problems, and repurposes them for longer periods

zkLogin Vulnerabilities

Semantic confusion

- Remember: we need to prove parsing, but
- *zkLogin* admits signed JWTs that are malformed:
  - claims are duplicated
  - claims are not strings or contain malformed characters
  - claims have injected Javascript
  - are expired
- This generates parser confusion where one entity can interpret it in one way while a proof binds to another

```
{
  // ----- Header -----
  "alg": "none", // bogus algorithm
  "alg ": "RS256", // trailing space
  "alg": "RS256",
  "kid": "992475",
  "typ": "JWT",

  // ----- Payload -----
  "iss": "https://accounts.example/", // honest OP
  "\/iss": "https://evil/", // injected
  "iss ": "https://evil2/", // trailing space
  "\u0069ss": "https://ex/", // unicode
  "sub": "C37900",
  "sub ": "MALLORY", // duplicate
  "sub ": "<script>alert('hi')</script>",
  "aud ": ["5731200", "other.."], // not allowed
  "nonce": "gIFt7xtjGLZq5cC0-TgIEeIcuJM",
  "nonce ": "AAAAAA", // last-wins
  "exp": 1716265381,
  "nbf": "not-a-number" // wrong type
}
```

zkLogin Vulnerabilities

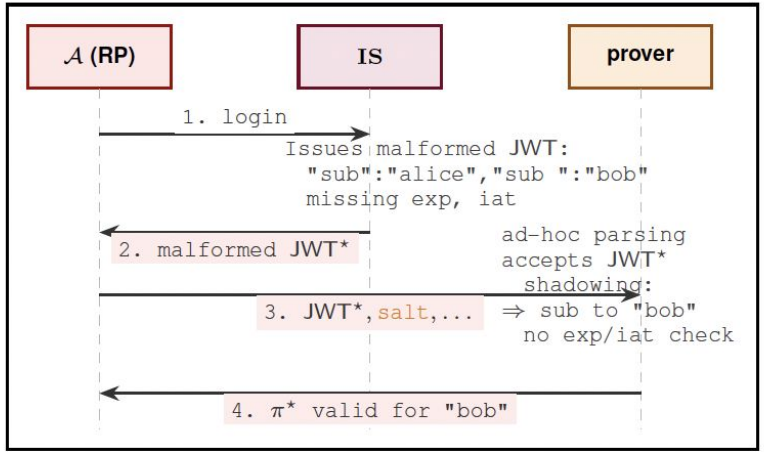


Fig. 4. Attack via malformed-JWT acceptance (V2a). A non-compliant or malicious **IS** issues a JWT with shadowed claims and missing temporal fields. The prover's ad-hoc parsing accepts it and binds the shadowed `sub` into the proof (the check JWT.pld.aud == APIkey$_{stored}$.aud passes as it uses the first field), despite the JWT* failing standards-compliant validation.

zkLogin Vulnerabilities

Malicious issuers

zkLogin (the protocol) allows JWTs that are generated *by any OIDC issuer*. Some deployments do have a allowlist of issuers but this is a deployment choice, not rooted in a trust chain:

a. OIDC is a federated-friendly protocol
b. OIDC are constantly misconfigured

Deployments of *zkLogin* allow issuers of the form:

- https://cognito-idp.region.amazonaws.com/tenant_id

Security cannot be reduced to trust on an *external third-party issuer without any proper validation:*

ii. Which is even more serious in a federated-friendly system as OIDC

zkLogin Vulnerabilities

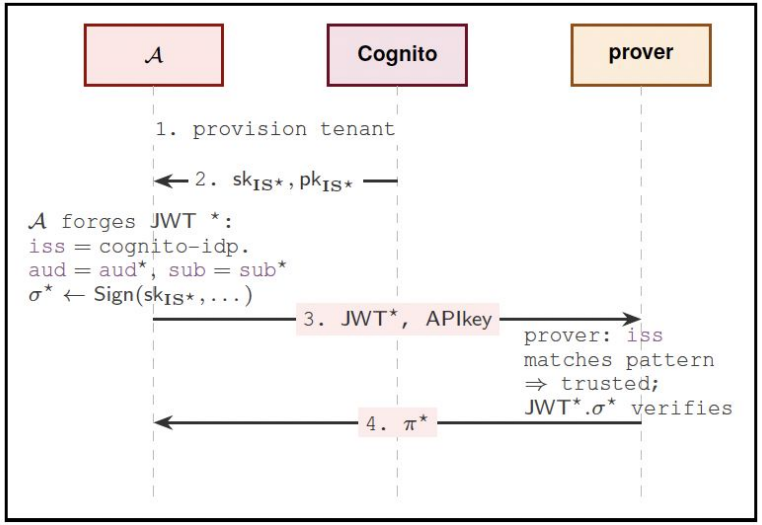


Fig. 6. Attack via pattern-based issuer trust (V1). The attacker provisions an AWS Cognito tenant via self-service, becoming a zkLogin-trusted issuer. They forge a JWT under their own signing key with a target `sub*`, obtain a valid proof from the prover, and submit it to the verifier. The cryptographic checks all pass; the trust failure is at the issuer-recognition step.

zkLogin Vulnerabilities

Cross aud (RP) and sub (user account) impersonation

- *aud* is a claim in JWT that loosely binds a JWT to an RP
- *sub* is the subject identifier claim and loosely binds a JWT to a user under the namespace of a issuer (think of an email)

- In zkLogin, this binding is not checked: a malicious or misconfigured issuer can issue JWTs for different/expired *aud* or *sub*, or another RP can present them as belonging to them
- In some *zkLogin* deployments:
  - Somewhat prevented by the usage of an **API key**
  - However, the protocol directly allows for cross-impersonation across *auds* and *subs*

zkLogin Vulnerabilities

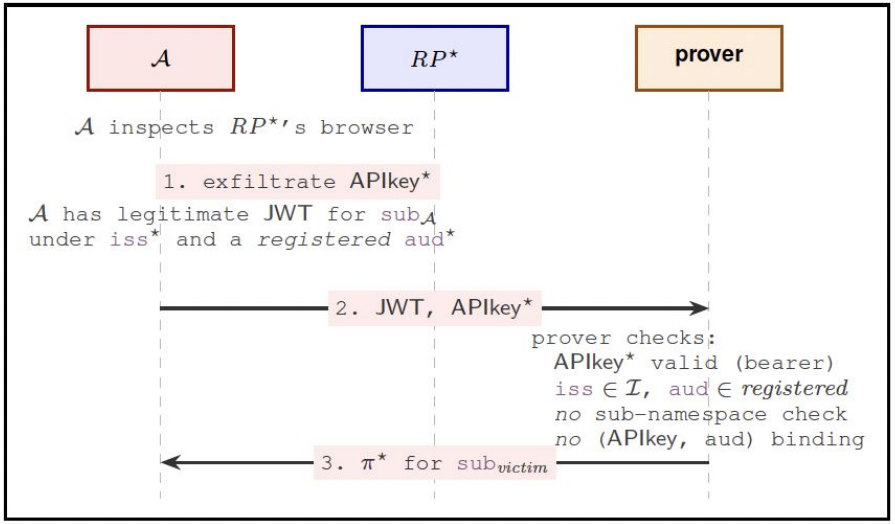


Fig. 5. Cross-RP / cross-subject impersonation (V2b, V2c). The attacker exfiltrates $RP^{\star}$'s API key from browser state, then submits a proof request with a substituted `sub`. The prover accepts because the API key is unbound to the (`iss`, `aud`, `sub`) tuple and `sub` is not validated against `iss`'s namespace.

zkLogin Vulnerabilities

Reliance on browser insecure storage

- The zkLogin documentation encourages storing keys (including secret keys), API keys, salts, JWTs, nonces and more in “regular” insecure browser storage:
  - They can be extracted by any injected script, malicious extensions, compromised dependencies, or brief device access

> For example, on browsers, use session storage instead of local storage to store the ephemeral key pair and the ZK proof. This is because session storage automatically clears its data when the browser session ends, while data in local storage persists indefinitely.

zkLogin Vulnerabilities

End-to-end vulnerability

By:

1. Registering an arbitrary *AWS Cognito tenant* as an issuer,
2. Extracting a RP's API key from the browser and using a public *aud* or public *sub*,
3. Constructing a JWT containing that *aud* and a *sub* under the attacker-controlled issuer
4. Signing it with the Cognito tenant's key and submitting it to the proving service as trusted RP, the attacker can obtain a valid zkLogin proof for another aud and sub.

The vulnerability does not depend on protocol deviations or cryptanalysis.

zkLogin Vulnerabilities

Privacy risks

- JWTs often contain private information such as:
  - Emails
  - Profile Photos
  - Full names
- They are sent in full to the salt and prover services without any consent from the user
  - It is unclear if they are aggregated
- The *iss* claim is sent to all services, which can allow for linking across them


```
{
  "iss": "https://accounts.google.com",
  "azp": "xxxx.apps.googleusercontent.com",
  "aud": "xxx",
  "sub": "x"xx,
  "email": "soficeli0@gmail.com",
  "email_verified": true,
  "nonce": "xx",
  "nbf": 1770336203,
  "name": "Sofi Celi",
  "picture": "https://lh3.googleusercontent.com/a/
ACg8ocJ3CCdi3fxTw5jpeWEckhneKpK8ZWMO5e6xO2k0tmhV61wCq4dw=s96-c",
  "given_name": "Sofi",
  "family_name": "Celi",
  "iat": 1770336503,
  "exp": 1770340103,
  "jti": "xxx"
}
```

When we report

We disclosed our findings on November, 2025 and (after pinging) received a reply on February 2026 and later a reply over X.

- It was argued that storing sensitive data in the browser was needed because it was needed to perform actions:
  - This is not needed, however, one can use session keys, cookies or a backend
- It was argued that issuers are assumed non-malicious:
  - More and more research shows that JWTs are issued misconfigured or issuers are misconfigured
  - But the developers will add full parsing in ZKP!
- A threat model that trusts external parties and local browser storage is not properly secure

What can we learn from this

Cryptography is sometimes not enough:

- Yes, ZKPs are cool and secure (mostly)
- But the architecture should preserve the security as well
- Integrating systems to higher-stakes checks as age-assurance can only aggravate security problems and introduce more attack vectors
- Authentication in the web is hard
- Systems should be open: our research was done with a lot of reverse engineering, as the comprete *protocol was unspecified*

However…

ZKPs often pitched as a privacy-preserving solution to **age or eligibility checks**

- Credential-based systems anchored in centralized authorities risk exclusion and censorship
- ZKPs do not guarantee meaningful privacy unless applied carefully and correctly. For example, a proof that a user's age lies in the range 20–21 may inadvertently disclose that the user is indeed 21:
  - **New risks:** when proofs are combined with other signals such as the site where the credential is used, the identity of the credential issuer (e.g., a specific government) or the user's location, the resulting tuple can uniquely identify individuals

However...

ZKPs often pitched as a privacy-preserving solution to **age or eligibility checks**.

- ZKP systems can devolve into a form of *client-side scanning*, where arbitrary attestations are made about the contents of data.
- Exclusion: if a party only accepts a limited set of credentials, such as residence permits, people without them are locked out of services
  - an estimated 850 million people worldwide lack formal identification

However…

Takeaway: ZKPs are not a silver bullet for age verification: they must be embedded within a broader system context (usability, revocation, interoperability, user autonomy, user verifiability)

However...

- Standardisation bodies and civil society are thinking about it
  - Workshops at the IETF and W3C
- The goal is to preserve privacy but not enhance exclusion or censorship in the process, but it also asks the question: do we want this?
- See: https://brave.com/blog/zkp-age-verification-limits/ , https://www.ietf.org/ietf-ftp/slides/slides-agews-paper-private-and-decentralized-age-verification-architecture-00.pdf
- Upcoming work on the risks of censorship that age assurance introduces in the digital world

Hints of future work

- We have explored one system that uses ZKPs to attest a document in order to execute an action. But there are others in production.
  - Discord's age verification:
    - Does not use ZKPs, but
    - They use(d) external API to manage age verification:
      - Said API made calls not only to attest that the document was issued correctly or was of the "correct age", but crucially it send calls to:

```
Chainalysis  – crypto address screening
Equifax      – credit/identity data
SentiLink    – synthetic identity fraud detection
Middesk      – business verification
Kyckr        – business registry lookups
TRM          – crypto compliance/investigation
MX           – financial data aggregation
OpenAI       – AI copilot (productivity)
Salesforce   – CRM
HubSpot      – CRM
Slack        – notifications
Zendesk      – support
```

Hints of future work

- We have started a reverse engineering effort to check the kind of signals that systems build for age-assurance perform
  - Mostly explored Persona-based systems
- They accept:
  - driver's licenses, passports, national IDs, residence permits, visas, green cards, social security cards, military IDs, voter IDs, MyKad/MyKid (Malaysia), PAN cards (India), tax IDs, weapon permits, border crossing cards, refugee IDs
  - It is unclear why it accepts refugee IDs or visa-related IDs
    - A refugee ID identifies someone who fled a state, often a state that may still be hostile to them

Hints of future work

- They extract:
  - full name, date of birth, sex, nationality, document number, issue/expiry dates, address, place of birth, MRZ (machine-readable zone for passports), barcode data (PDF417, QR, etc.), photo, signature presence, height/weight, eye/hair color, vehicle classes for DLs, and AAMVA-compliant fields specific to US/Canadian licenses.
  - The also check for ASYLUM_REQUEST, TRIBAL_ID

THANK YOU!

@claucece
https://sofiaceli.com/