Privacy-Preserving OpenID Connect

Sven Hammann
Department of Computer Science
ETH Zurich
Switzerland
sven.hammann@inf.ethz.ch

Ralf Sasse
Department of Computer Science
ETH Zurich
Switzerland
ralf.sasse@inf.ethz.ch

David Basin
Department of Computer Science
ETH Zurich
Switzerland
basin@inf.ethz.ch

ABSTRACT

OpenID Connect is the most widely used Internet protocol for delegated authentication today. It provides single sign-on functionality for users who use their account with an identity provider to authenticate to different services, called relying parties. Unfortunately OpenID Connect is not privacy-friendly: the identity provider learns with each use which relying party the user logs in to. This necessitates a high degree of trust in the identity provider, and is especially problematic when the relying parties' identity reveals sensitive information.

We present two extensions to OpenID Connect that address this privacy concern. We first present a simple extension that prevents the identity provider from learning to which relying parties its users log in, and we further extend this solution to also prevent colluding relying parties from tracking users. We give formal security proofs for both standard OpenID Connect and our extensions using the Tamarin security protocol verification tool.

KEYWORDS

OpenID Connect; single sign-on; privacy; protocol verification

ACM Reference Format:
Sven Hammann, Ralf Sasse, and David Basin. 2020. Privacy-Preserving OpenID Connect. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS '20), October 5–9, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3320269.3384724

1 INTRODUCTION

The Internet provides access to an ever increasing number of services, many of which require its users to have an account with credentials. This is a considerable cognitive burden for users and leads to password reuse and other poor security practices [21]. Delegated authentication protocols offer a way out of this dilemma. These protocols allow a user to use her account at an *identity provider* (IdP) to log in to other services, called *relying parties* (RP). The user's account at the IdP should then be protected with a password and preferably with a second authentication factor. This provides users with *single sign-on* functionality, where one account can be used to log in to many different services. The most widely used delegated authentication protocol today is OpenID Connect [30], supported by identity providers like Google and Microsoft.

The downside of OpenID Connect is that the IdP learns exactly which RPs a user accesses, and when and how often. This is problematic for user privacy. First, the very act of logging in to an RP reveals sensitive information about the user. Consider for example an RP hosting a medical forum for patients with an incurable disease, a chatroom for abuse victims, or a help site for people with substance abuse disorders. Second, the IdP often possesses user profiles containing verified personal user attributes. This allows the IdP to link the data about which RPs the user visits to a uniquely identified person rather than just a pseudonym. For example, the *VERIMI* IdP service [4] performs extensive verification procedures, such as video identification for ID cards and passports.

For these two reasons, allowing an IdP to observe and collect its users' login data requires trusting that the IdP will not use this data inappropriately. Moreover, the IdP must be trusted to protect this data against both malicious insiders and outsiders. Clearly, it is desirable to reduce the need for such a high degree of trust.

Providing IdPs with this login data is also problematic from the perspective of data protection regulations. In particular, the European General Data Protection Regulation (GDPR) mandates that the collection of user data is *limited to what is necessary in relation to the purposes for which the data is processed* [1]. We show how OpenID Connect can be extended so that the IdP no longer learns at which RP the user logs in. Thus, this information about the RP is, in fact, unnecessary, and hence should not be collected in the first place.

Contributions. We present two privacy-friendly extensions of OpenID Connect called *Privacy-preserving OpenID Connect* (POIDC) and *pairwise POIDC*. Both extensions achieve *login unlinkability* with respect to the IdP, which means that the IdP cannot distinguish, given two RPs, which RP the user logged in to.

In POIDC, we implement the following functionality without the IdP learning the RP's identity:

- The user can give consent to log in to a specific RP.
- The IdP signs *id_tokens* that are only valid for that one RP.
- The user's browser is redirected to the correct URL owned by that RP.

In pairwise POIDC, we address an additional privacy concern. Two or more RPs may collude, sharing information provided by the OpenID Connect protocol to link user accounts at different RPs to the same person. There is an existing protection mechanism in OpenID Connect that prevents this, called pairwise subject identifiers, where the IdP assigns different identifiers for the same user and different RPs. However, this mechanism requires the IdP to look up the correct pairwise identifier for the user at the RP she wishes to log in to. Thus, pairwise identifiers seem to conflict with hiding the RP's identity from the IdP.

We solve this conflict by presenting a new design for pairwise identifiers. Our pairwise POIDC protocol provides login unlinkability with respect to both the IdP and colluding RPs. This is accomplished by a zero-knowledge proof carried out between the user agent (the user's browser) and the IdP. Pairwise POIDC's improved privacy properties therefore come at the cost of higher complexity and increased size of the exchanged messages.

Due to this trade-off, the appropriate protocol choice depends on the situation. For example, many RPs require personal information about their users, such as an e-mail address or a physical address, for example, when the RP must ship physical goods to the user. When multiple RPs have access to this kind of personal information, they can already link their users' accounts. Thus, pairwise subject identifiers provide no privacy benefit here. Therefore, POIDC can be used with public subject identifiers as a strict privacy improvement over standard OpenID Connect.

When RPs do not receive personal information about their users, then pairwise subject identifiers prevent colluding RPs from linking their users' accounts. Thus, for these RPs, pairwise POIDC should be used to obtain login unlinkability with respect to both the IdP and the RPs. We summarize the provided privacy properties and our recommendations for the two situations described in Figure 1.

Another important feature of our proposed extensions is that they provide the same security guarantees as standard OpenID Connect. To show this, we formalize the security properties of standard OpenID Connect, POIDC, and pairwise POIDC as trace properties and prove them using the security protocol verification tool Tamarin [29, 31]. For all three protocols, we show that a user can only log in to an RP when that user has started the protocol and has given consent for logging in to that RP. Previous security proofs for OpenID Connect have been manual [20]. Hence we provide the first security proofs for OpenID Connect that use an automated security protocol verification tool. Furthermore, our models use a new and more precise way to symbolically model signatures, introduced recently by Jackson et al. [26]. Thus, our models capture a wider range of attacker capabilities with respect to signatures than has been traditionally considered for symbolic protocol verification.

Outline. In Section 2, we provide background on OpenID Connect. In Section 3, we define our attacker models, system model, and privacy properties. We describe the protocol details of POIDC in Section 4 and of pairwise POIDC in Section 5. We present the security properties we proved using Tamarin in Section 6. We compare with related work in Section 7 and draw conclusions in Section 8.

2 BACKGROUND ON OPENID CONNECT

A delegated authentication protocol involves three parties: identity providers (IdPs), relying parties (RPs), and users. Users have accounts at IdPs. An authenticated user can request a (short-lived) token from the IdP that she can use to log in to an RP. The most widely used protocol of this kind is OpenID Connect [30], built on top of the OAuth 2.0 standard [24].

Example 1. *A popular OpenID Connect implementation is* Google Sign-In. *A user can click on a* Sign In With Google *button, for example on* stackoverflow.com. *She is then redirected to Google, where she is asked to log in to her Google account, if she is not already logged in. The Google page then displays a message that asks the user to confirm that she wants to log in to* Stack Overflow. *She must also confirm that* Stack Overflow *may access her e-mail address. Afterwards, she is redirected to* stackoverflow.com, *where she is logged in and her user profile already contains her verified e-mail address.*

More generally, when a user wishes to log in to an RP using OpenID Connect, she first selects her IdP from a given list of alternatives. This is usually done by selecting the appropriate button on the RP's page. The user is then redirected to the IdP, where she authenticates herself (the IdP's authentication mechanism is not part of OpenID Connect). The IdP then asks the user for consent to log in to the RP and to release some of her data to the RP. The next steps depend on which OpenID Connect flow is used.

In *implicit flow*, when the user gives consent, the IdP sends a signed JSON Web Token (JWT) [27], called the *id_token*, to the RP via the user's browser. This token includes an *issuer identifier* identifying the IdP, a *subject identifier* identifying the user, and an *audience identifier* denoting the identity (called *client_id*) of the intended recipient RP. This ensures that the *id_token* will only be accepted by that RP. The *id_token* may contain additional information, such as the user's e-mail address, if requested by the RP.

In *authorization code* flow, a code is first forwarded via the user's browser to the RP, which is then exchanged for an *id_token* via a direct channel between the RP and the IdP. In this flow, the RP might additionally have to authenticate to the IdP. This direct channel between RP and IdP is incompatible with our privacy goals. We thus use implicit flow as the base for our extensions.

Throughout this paper, we denote standard OpenID Connect by OIDC, referring to the implicit flow, unless stated otherwise. We focus on the *id_token* functionality offered by OIDC only and we do not consider OAuth 2.0 access tokens.

Example 2. *In Example 1, the id_token contains:* www.google.com *as the issuer identifier; the* identifier for stackoverflow *(called client_id) chosen by Google as the audience identifier; the identifier Google assigned to the* user *as the subject identifier; the user's e-mail address; and a nonce that was generated by the RP and sent to the IdP.*

There are two kinds of subject identifiers in OIDC, called *public* and *pairwise* subject identifiers. A public subject identifier is a unique identifier for a user at an IdP. Pairwise subject identifiers are specific to the RP, so the IdP looks up the user's pairwise identifier for a specific RP when creating an *id_token* for that RP.

OIDC has the following setup: The RP must register with the IdP and provide a set of URIs called *redirect_uris* that belong to the RP and to which *id_tokens* may be sent. The RP provides additional metadata, such as its name, which the IdP shows the user when asking for login consent.

OIDC (implicit flow) consists of the following nine steps, illustrated in Figure 2.

(1) The user, using her user agent (typically a web browser), initiates the protocol by requesting to log in to the RP. She specifies which IdP she wants to use.
(2) The RP redirects the user agent to the IdP, sending its *client_id* and a freshly generated *nonce* as query parameters.

	Login-unlinkability w.r.t. the *IdP*	Login-unlinkability w.r.t. *colluding RPs*
Standard OIDC with pairwise identifiers		
POIDC	✓	
Pairwise POIDC	✓	

(a) If RPs require additional user-identifying information, then use POIDC.

	Login-unlinkability w.r.t. the *IdP*	Login-unlinkability w.r.t. *colluding RPs*
Standard OIDC with pairwise identifiers		✓
POIDC	✓	
Pairwise POIDC	✓	✓

(b) If RPs require only subject identifiers, then use pairwise POIDC.

Figure 1: Privacy properties fulfilled by standard OIDC, POIDC, and pairwise POIDC in two different situations.

(3) The user agent is redirected to the IdP, forwarding these query parameters to the IdP (the user also logs in at the IdP, if necessary, which is not part of OIDC).
(4) The IdP opens a dialogue to be displayed in the user agent, asking the user to confirm that she wishes to log in to the RP belonging to the *client_id*. For this, the IdP looks up a human-readable *client_name* that belongs to the *client_id*.
(5) The user clicks the confirm button in the dialogue.
(6) The IdP returns an *id_token* as described earlier. In particular, the *id_token* contains the *client_id* in its *aud* (audience) field, and the *nonce* that was chosen by the RP.
(7) The user agent forwards the *id_token* to the RP (namely, to an endpoint called the *redirect_uri*, which must be one of the *redirect_uris* that are registered at the IdP for the RP).
(8) The RP validates the IdP's signature on the *id_token* and compares the *aud* field with its own *client_id*. The RP accepts the *id_token* only if these values are equal (and signature validation succeeds).
(9) The RP notifies the user agent whether the login was successful.

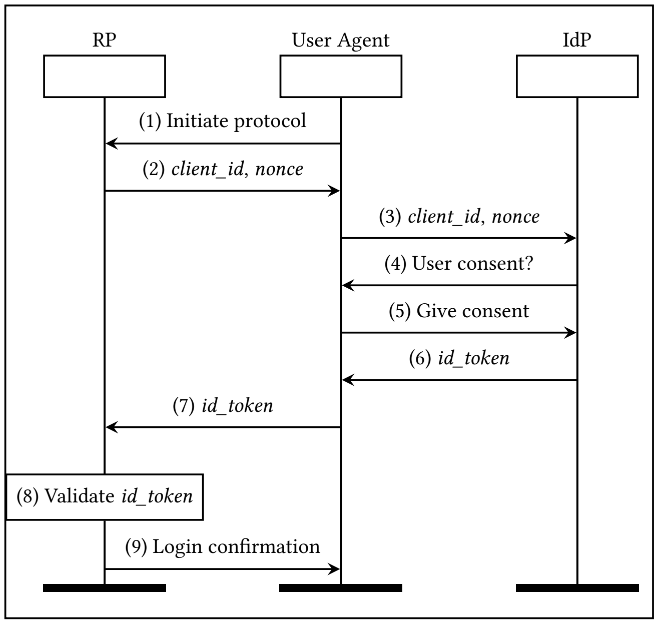

Figure 2: OpenID Connect implicit flow.

3 SETUP, ASSUMPTIONS, AND PROPERTIES

In this section, we explain our setup, including the attacker models that we consider when analyzing privacy and security.

3.1 System model

The following parties are involved in our protocols.

The RP exposes an interface, such as a button, which allows the user to send login requests from its user agent. It also contains endpoints for requests sent to a *redirect_uri* registered for the RP.

The user agent is a standard browser that can perform actions, such as cryptographic operations, by loading JavaScript code from web pages. This code is executed locally in the browser and is therefore part of the user agent. Messages that are sent to the user agent but are not forwarded are sent as URI fragment identifiers [6]. These identifiers are accessible to the JavaScript code and can thus be processed by the user agent. However, unlike query parameters, they are not sent to the IdP back-end.

The IdP corresponds to the *IdP back-end only*. Messages sent to the IdP in our protocol descriptions are sent as query parameters.

3.2 Attacker models

We employ different attacker models for the properties we analyze. We consider *security* with respect to a standard *Dolev-Yao network attacker*, which we formalize in Section 6. For *privacy*, we consider two different attacker models: An *honest-but-curious IdP*, and a set of *colluding RPs*.

Definition 1. An *honest-but-curious IdP* follows the protocol, but retains detailed transcripts of each protocol run that contain every message the IdP received during the run. The IdP analyzes these transcripts and correlates information from transcripts for different protocol runs.

We argue that this is the right adversary model for analyzing OpenID Connect. First, a malicious IdP can trivially violate its users' privacy and security. The IdP can log in under one of its users' account at any RP by forging *id_token*s. Thus, it can learn which

RPs the user visits, namely those where the user has an account. It can even obtain additional data from the user's account at the RP.

Our assumption that the user's IdP is not malicious is clearly a basic assumption underlying OpenID Connect. Moreover, this assumption is realistic for IdPs associated with major companies such as Google or Microsoft, which are concerned about their reputation. If an IdP's misbehavior were detected, then news of this would spread quickly, with disastrous effects for the IdP. While corporate IdPs are extremely sensitive about their reputation, they are also businesses and their business model often includes monetizing user data, for example through targeted ads. Thus, while the IdP has strong incentives to adhere to its protocol definition, it still wishes to gather as much user data as possible, provided this cannot be detected.

When analyzing *privacy*, in contrast to security, we do not consider a network attacker. A network attacker can observe the traffic between users and RPs to see where a user logs in. The problem of protecting user privacy against a network attacker is independent of protecting user privacy with respect to the IdP, and it cannot be solved by the delegated authentication protocol alone. To ensure privacy against a network attacker, a user must employ other methods such as mix networks [12] or Tor [15] to realize an anonymous communication channel.

Definition 2. A set of *colluding RPs* consists of (potentially malicious) RPs that share information obtained during protocol runs with each other, such as information contained in *id_token*s.

Colluding RPs try to link their users' accounts to the same person. When an *id_token* includes the user's e-mail address, then they can trivially link this information. However, when no such information is explicitly given, linking accounts should not be possible. In standard OIDC, there are two different kinds of subject identifiers: public and pairwise. If public identifiers are used, then the same identifier is used for the user at each RP, and thus colluding RPs can link two accounts to the same person. Pairwise identifiers prevent this by assigning a different identifier for the user for each RP.

We also make the standard assumption that the honest-but-curious IdP and the colluding RPs are computationally bounded by a polynomial with respect to an implicit security parameter. In particular, they cannot perform an arbitrary number of guesses with respect to a hashed value's pre-image.

3.3 Privacy properties

We now state the privacy properties to be achieved by our protocols.

Definition 3. Login unlinkability with respect to the IdP. A delegated authentication protocol P provides *login unlinkability with respect to the IdP* if, for any user u and any two honest RPs rp_0 and rp_1, an honest-but-curious IdP cannot distinguish between a transcript of a protocol run where the user u logs in to rp_0, and one where u logs in to rp_1, even if the IdP has access to additional transcripts from previous protocol runs (by any users). An *honest RP* follows its protocol description and does not collude with the IdP.

Note that this defines a stronger requirement than secrecy of the RP's identity. In particular, it also prevents the IdP from linking repeated logins of the same user to the same RP.

Definition 4. Login unlinkability with respect to colluding RPs. A delegated authentication protocol P provides *login unlinkability with respect to a set of colluding RPs* if a set S of (potentially malicious) RPs cannot distinguish, given information learned from users' logins, whether any two logins to two different RPs in S were performed by the same honest user u or by two different honest users u and u'. An *honest user* is a user for which none of the user's credentials are known by any of the colluding RPs.

We describe next our two protocols. The first protocol, POIDC, achieves the first privacy property. The second protocol, pairwise POIDC, achieves both, but at the cost of higher complexity.

4 POIDC: PROTECTING AGAINST THE IDP

The IdP learns to which RP the user logs in using OIDC because the *client_id*, which identifies the RP, is sent to the IdP. The *client_id* serves two purposes. First, it is included in the *aud* (audience) field of the *id_token*, so that the token is only valid for that one RP. Second, the IdP looks up the following data for the RP based on its *client_id*:

(1) a human-readable name for the RP (called the *client_name*), which it displays when asking the user for consent to log in to that RP; and
(2) a set of *redirect_uris* that belong to the RP and can be used for sending an *id_token* to that RP.

Note that the trivial solution for privacy where the *client_id* is not sent to the IdP, and thus not included in the *id_token* either, would not be secure. Such an *id_token* would be valid for any RP, and a malicious RP could use any *id_token* it was sent to log in under the user's account to any other RP.

We now show how POIDC achieves privacy by hiding the RP's identity from the IdP while preserving security. That is, our goal is to provide the following properties:

(1) Login-unlinkability with respect to an honest-but-curious IdP; and
(2) security with respect to a Dolev-Yao network attacker.

POIDC relies on two main features. First, POIDC replaces the *client_id* with a one-time pseudonym for the RP, used as the audience of the *id_token*. Second, POIDC enables the user agent to ask the user for consent to log in to an RP, since this can no longer be done by the IdP.

4.1 Masking the client_id

We replace the *client_id* that is sent to the IdP with a hashed value $H(\mathit{client\_id}||\mathit{rp\_nonce}||\mathit{u\_nonce})$. In this expression, H is a cryptographic hash function, *u_nonce* is a cryptographically secure, unpredictable, random value generated in the user agent, *rp_nonce* is the nonce generated by the RP for replay protection, and || denotes concatenation. It is critical that the *u_nonce* is generated by the user agent so that the IdP back-end does not learn its value.

Including the *rp_nonce* in the hash means that it does not have to be sent in plaintext, and thus prevents the IdP from potentially obtaining information about the RP from the *rp_nonce*, e.g., if RPs use different nonce formats. Note that adding the *u_nonce* is still necessary since we do not want to depend on the RP to provide sufficient randomization.

We include $H(client\_id||rp\_nonce||u\_nonce)$ in the *id_token*, in a field called *private_aud*. This field replaces OIDC's *aud* (audience) field, and we call a token that has a *private_aud* field a *private_id_token*. Such a token is different from a usual *id_token* (the otherwise mandatory *aud* field is missing). An RP that runs OIDC would therefore not accept a *private_id_token*. Thus, POIDC can be run in parallel with standard OIDC with no risk of interference.

4.2 User consent in user agent

In OIDC, the IdP presents a user consent dialogue asking her if she wishes to log in to a particular RP. In POIDC, the IdP cannot look up the required information since it does not know the RP's identity. Thus, a user consent dialogue is instead shown locally by the user agent without looking up information in the IdP back-end. For this, we must enable the user agent to map a *client_id* to a user-readable *client_name*. This mapping must be authentic, since an attacker could otherwise impersonate any RP to the user to obtain an *id_token* of the user for that RP.

We therefore introduce an additional step when the RP originally registers at the IdP (or when the IdP enables support for POIDC), where the IdP sends a signed token called a *client_id_binding* to the RP. The *client_id_binding* contains:

- the RP's *client_id*;
- the RP's *client_name*, the name of the RP as it can be understood and checked by the user; and
- the RP's set of registered *redirect_uris* that are valid endpoints to send the *private_id_token* to. The IdP must verify that these endpoints belong to the RP.

During a POIDC protocol run, the RP includes its *client_id_binding* token in a URI fragment so that it is sent to the user agent, but not to the IdP back-end. With this information, the user agent can construct a user consent dialogue displaying the *client_name*. The user agent will further only forward the *private_id_token* to one of the *redirect_uris* listed in the *client_id_binding*.

4.3 POIDC Protocol steps

The protocol steps of POIDC are illustrated in Figure 3.

(1) The user u initiates the protocol by requesting to log in to the RP. She specifies which IdP she wants to use, e.g., by clicking on the corresponding button on the RP's web page. This triggers a request sent from the user agent to the RP.
(2) The RP redirects the user agent to the IdP, sending the *client_id_binding* (containing its *client_id*) and its *rp_nonce* to the user agent as URI fragments. These URI fragments are accessible to JavaScript, but are *not* sent to the IdP back-end. This redirect must *not* contain an HTTP referer header.
(3) The user agent is redirected to the IdP to load its website, but this request does not contain any query parameters.
(4) The IdP sends its website to the user agent. The website contains JavaScript code that allows the user agent to perform the actions in Steps 5–9, as well as the IdP's public verification key $\mathrm{pk}_{\mathrm{IdP}}$ that is used in the next step.
(5) The user agent executes the JavaScript code to verify the IdP's signature on the *client_id_binding* using $\mathrm{pk}_{\mathrm{IdP}}$. If this verification succeeds, the user agent then opens a dialogue for the user to confirm that she wishes to log in to the RP with the *client_name* in the *client_id_binding*.
(6) The user agent generates a cryptographically secure random value *u_nonce* and computes $masked\_aud := H(client\_id||rp\_nonce||u\_nonce)$.
(7) It sends *masked_aud* to the IdP back-end, using, for example, an XMLHttpRequest.
(8) The IdP back-end returns a *private_id_token* to the user agent, which contains $H(client\_id||rp\_nonce||u\_nonce)$ in its *private_aud* field.
(9) The user agent verifies that the *private_aud* field of the received *private_id_token* contains the value for $H(client\_id||rp\_nonce||u\_nonce)$ it computed earlier. If this verification succeeds, the user agent sends this token and its *u_nonce* to the *redirect_uri* given in the *client_id_binding*, using, for example, *window.open(redirect_uri)*.

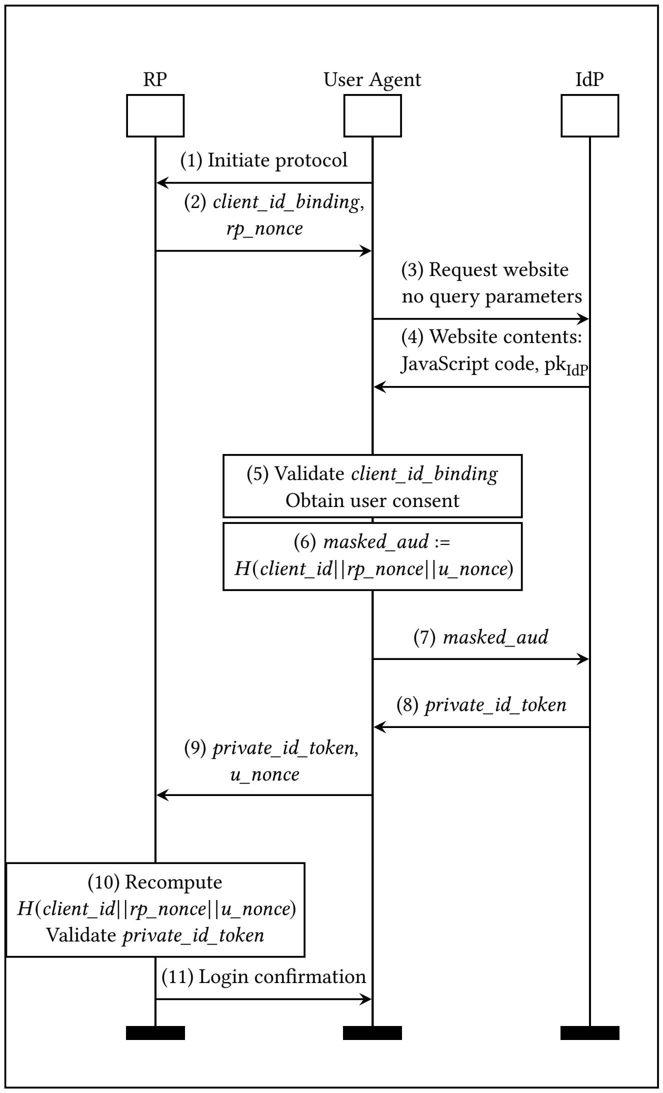
RP
User Agent
IdP
(1) Initiate protocol
(2) client_id_binding, rp_nonce
(3) Request website no query parameters
(4) Website contents: JavaScript code, pk_IdP
(5) Validate client_id_binding Obtain user consent
(6) masked_aud := H(client_id||rp_nonce||u_nonce)
(7) masked_aud
(8) private_id_token
(9) private_id_token, u_nonce
(10) Recompute H(client_id||rp_nonce||u_nonce) Validate private_id_token
(11) Login confirmation

Figure 3: Privacy-preserving OpenID Connect.

(10) The RP validates the IdP's signature on the *private_id_token* and recomputes $H(\mathit{client\_id}||\mathit{rp\_nonce}||\mathit{u\_nonce})$ from its own *client_id*, the *rp_nonce* that it generated earlier, and the *u_nonce* that it received from the user agent.
It accepts the *private_id_token* only if its *private_aud* field matches this recomputed value (and signature validation succeeds).
(11) The RP notifies the user agent whether the login was successful.

Recall that the IdP is honest-but-curious and thus, in Step 4, provides JavaScript code that correctly performs Steps 5–9.

Note that the user may have to log in to her account at the IdP if she does not have an existing session. This authentication step is, however, not explicitly part of the OIDC protocol, and thus is not part of POIDC either. A user authenticating to the IdP is also independent of the RP, so there is no risk of leaking information about the RP's identity during that process.

4.4 Privacy proof

We show that POIDC provides login-unlinkability with respect to the IdP. We show that this only depends on the secrecy of the *u_nonce*, but not on the secrecy of the *rp_nonce*. Thus, an RP who uses a predictable *rp_nonce* would not break the user's privacy.

Theorem 1. *POIDC provides login-unlinkability with respect to an honest-but-curious IdP in the random oracle model, even if that IdP knows the value of any rp_nonce that was used during protocol runs.*

Proof. First, note that any transcript of a protocol run from the IdP's viewpoint only contains one message that the IdP receives, containing a value of the form $H(\mathit{client\_id}||\mathit{rp\_nonce}||\mathit{u\_nonce})$.

To distinguish the setting where a user logs in to RP rp_0 with *client_id* cid_0 from the setting where the user logs in to RP rp_1 with *client_id* cid_1, the IdP would have to distinguish
$H(\mathit{cid}_0||\mathit{rp\_nonce}_0||\mathit{u\_nonce})$ from $H(\mathit{cid}_1||\mathit{rp\_nonce}_1||\mathit{u\_nonce})$. In the random oracle model, these are both random values drawn from the same distribution from H's domain, and are thus indistinguishable to the IdP.

Furthermore, since *u_nonce* is a cryptographically secure, unpredictable, random value, the IdP cannot efficiently compute a pre-image of the hash, even if the IdP has access to the *client_id* and *rp_nonce*. □

5 PAIRWISE POIDC: PROTECTING ALSO AGAINST COLLUDING RPS

The previous version of POIDC requires public subject identifiers. Namely, there is a single subject identifier for each user at the IdP, and this identifier is included in each *id_token* for the user, regardless of the intended audience RP. Thus, it does not provide privacy with respect to colluding RPs, who can link accounts to the same user.

In standard OIDC with pairwise identifiers, the IdP can simply look up the respective identifier when the user wishes to log in to an RP. When the IdP should no longer learn to which RP the user logs in, designing pairwise identifiers becomes more challenging.

We outline next the requirements for implementing pairwise identifiers in POIDC. Our goal is to provide the following properties:

(1) Login-unlinkability with respect to an honest-but-curious IdP;
(2) login-unlinkability with respect to colluding RPs; and
(3) security with respect to a Dolev-Yao network attacker.

Each requirement follows from one of these goals, which we list next to the requirement.

User Agent Computable Identifiers *(Login-unlinkability with respect to the IdP)*
The *user agent* must be able to compute the pairwise identifier for the user at a specific RP, since the IdP does not know to which RP the user wishes to log in.

No Identifier Registration *(Login-unlinkability with respect to the IdP)*
There must be no difference between the first login (registration) of a user at an RP and to an RP where the user has logged in before. Thus, the IdP cannot generate new pairwise identifiers or even keep any sort of pairwise identifier list for the user.

Pairwise Identifier Unlinkability *(Login-unlinkability with respect to colluding RPs)*
Comparing pairwise identifiers must not provide the RPs with any information that enables linking them to the same user. This is the main reason to have pairwise identifiers, and is easy to solve in isolation, but is challenging when combined with the other requirements.

IdP-hidden Identifiers *(Login-unlinkability with respect to the IdP)*
While the pairwise identifier for a specific user at an RP must be persistent for functionality reasons, the IdP must not be able to link protocol runs where the user uses the same pairwise identifier.

User-bound Identifiers *(Security)*
An IdP must only sign an *id_token* that contains a subject identifier that belongs to the user who is currently logged in at the IdP. While this requirement is again simple in isolation, it rules out many potential solution for the other requirements.

We first explain our design for *User Agent Computable Identifiers* that have *No Identifier Registration*. First, we introduce a global user identifier *user_id*, which is unique to each user at the IdP, and is not shared with RPs. Then, we define the following mapping from a *user_id* to pairwise identifiers for RPs. The pairwise identifier for an RP with *client_id* *cid* is given as *pairwise_sub* := $H(\mathit{user\_id}||\mathit{cid})$, where H is a cryptographic hash function. Thus, the user agent can compute the pairwise identifier, and it is not necessary to keep a list of identifiers for each user.

We ensure *Pairwise Identifier Unlinkability* by requiring that the *user_id* is an unpredictable random value from a large domain so that an RP with *client_id* *cid* cannot feasibly obtain a user's *user_id* by brute force from a pairwise identifier $H(\mathit{user\_id}||\mathit{cid})$. That is, the RP should not be able to efficiently compute an x such that $H(x||\mathit{cid})$ matches the pairwise identifier. In particular, the *user_id* must not be a username chosen by the user.

For *IdP-hidden Identifiers*, the user agent must mask the identifier similarly to how it masks the audience field. For this, it generates a cryptographically secure, unpredictable, random value *u_nonce_2* and computes $H(pairwise\_sub||u\_nonce\_2) = H(H(user\_id||cid)||u\_nonce\_2)$. This masked value is then sent to the IdP to be used as a masked subject identifier in a run of pairwise POIDC.

Finally, to ensure *User-bound Identifiers*, the IdP must be certain that the masked subject identifier provided by the user agent has been computed from the current user's *user_id*, i.e., that it is of the form $H(H(user\_id||x)||y)$ for some x and y. To solve this, the user agent proves this statement in zero-knowledge, e.g., using ZKBoo [22]. ZKBoo provides an efficient way to prove this kind of statement. In particular, ZKBoo is well suited for proving statements of the form $y = H(x)$ and has been evaluated for SHA-256 to require 55ms of time and 836 KB of space [22]. While our statement uses a nested hash and would thus result in a larger circuit, the scale-up is linear in the circuit size. Thus, ZKBoo supports a practical implementation of pairwise POIDC.

We next give the protocol description for pairwise POIDC, as illustrated in Figure 4.

(1) The user u initiates the protocol by requesting to log in to the RP. She specifies which IdP she wants to use, e.g., by clicking on the corresponding button on the RP's web page. This triggers a request sent from the user agent to the RP.
(2) The RP redirects the user agent to the IdP, sending the *client_id_binding* (containing its *client_id*) and its *rp_nonce* to the user agent as URI fragments. These URI fragments are accessible to JavaScript, but are *not* sent to the IdP back-end. This redirect must *not* contain an HTTP referer header.
(3) The user agent is redirected to the IdP to load its website, but this request does not contain any query parameters.
(4) The IdP sends its website to the user agent. The website contains JavaScript code that allows the user agent to perform the actions in Steps 5–10, as well as the IdP's public verification key pk_{IdP} and the user's unique identifier *user_id*.
(5) The user agent executes the JavaScript code loaded from the IdP's side to verify the IdP's signature using pk_{IdP} on the *client_id_binding*. If this verification succeeds, the user agent then opens a dialogue for the user to confirm that she wishes to log in to the RP with the *client_name* in the *client_id_binding*.
(6) The user agent generates *two* cryptographically secure random values, *u_nonce* and *u_nonce_2*, then computes the following values:
   - The masked audience identifier for the RP, $masked\_aud := H(client\_id||rp\_nonce||u\_nonce)$.
   - The pairwise subject identifier for the RP to which the user wishes to log in, $pairwise\_sub := H(user\_id||client\_id)$.
   - The masked version thereof, $masked\_sub := H(pairwise\_sub||u\_nonce\_2)$.
(7) The user agent sends *masked_aud* and *masked_sub* to the IdP back-end, using, for example, an XMLHttpRequest.
(8) The user agent provides a zero-knowledge proof to the IdP back-end showing that *masked_sub* is a value of the form $H(H(user\_id||x)||y)$ for some x and y, where *user_id* is the user's unique identifier.
(9) If the IdP back-end accepts the zero-knowledge proof, it then returns a *private_id_token* to the user agent, which contains

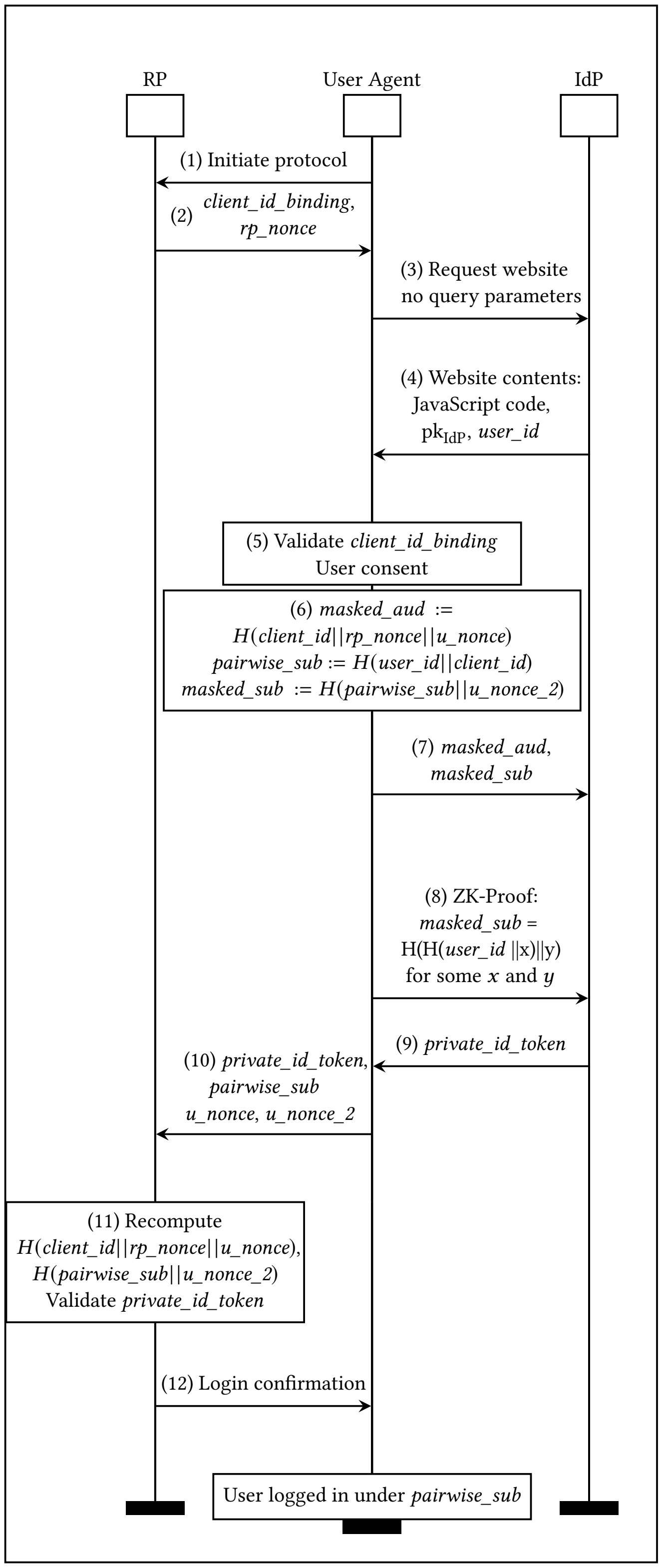
RP
User Agent
IdP
(1) Initiate protocol
(2) client_id_binding, rp_nonce
(3) Request website no query parameters
(4) Website contents: JavaScript code, pk_IdP, user_id
(5) Validate client_id_binding User consent
(6) masked_aud := H(client_id||rp_nonce||u_nonce) pairwise_sub := H(user_id||client_id) masked_sub := H(pairwise_sub||u_nonce_2)
(7) masked_aud, masked_sub
(8) ZK-Proof: masked_sub = H(H(user_id ||x)||y) for some x and y
(9) private_id_token
(10) private_id_token, pairwise_sub u_nonce, u_nonce_2
(11) Recompute H(client_id||rp_nonce||u_nonce), H(pairwise_sub||u_nonce_2) Validate private_id_token
(12) Login confirmation
User logged in under pairwise_sub

Figure 4: Privacy-preserving OpenID Connect with pairwise subject identifiers.

$H(client\_id||rp\_nonce||u\_nonce)$ in its *private_aud* field and *masked_sub* in its subject field.

(10) The user agent verifies that the received *private_id_token* contains the expected values in the *private_aud* and subject fields. If this verification succeeds, the user agent sends this token, as well as *pairwise_sub*, *u_nonce*, and *u_nonce_2* to the *redirect_uri* given in the *client_id_binding*, using, for example, *window.open(redirect_uri)*.

(11) The RP validates the IdP's signature on the *private_id_token*. It then recomputes $H(client\_id||rp\_nonce||u\_nonce)$ from its own *client_id*, the *rp_nonce* that it generated earlier, and the *u_nonce* that it received from the user agent. It also recomputes the *masked_sub* from *pairwise_sub* and *u_nonce_2* that the user provided. It accepts the *private_id_token* and considers the user logged in to the account under the identifier *pairwise_sub* only if the *id_token*'s *private_aud* and subject field match the expected, recomputed, values (and signature validation succeeds).

(12) The RP notifies the user agent whether the login was successful.

5.1 Transitioning to pairwise POIDC

To transition to pairwise POIDC from standard OIDC using pairwise subject identifiers, it is necessary to switch to the new kind of subject identifiers of the form $H(user\_id||client\_id)$.

If standard OIDC with pairwise subject identifiers created by the IdP were used before, then these can be converted to the subject identifiers introduced by pairwise POIDC as follows. The user requests an identifier switch for a specific RP, and the IdP creates a signed JWT called a *transition_token*. A *transition_token* contains the user's old pairwise identifier for that RP and the new one of the form $H(user\_id||client\_id)$. Upon the next login to the RP, the user presents a *transition_token* in addition to an *id_token*, and the RP will transfer the user's account to the new subject identifier.

Note that the login where the switch happens can still be observed by the IdP. Login unlinkability holds between logins to all RPs for which the switch has already been made and thus pairwise POIDC login is used.

5.2 Privacy proofs

We show that pairwise POIDC provides login-unlinkability with respect to the IdP. In particular, this only depends on the secrecy of *u_nonce* and *u_nonce_2*, but not on the secrecy of any other values used to construct the hash functions.

Theorem 2. *Pairwise POIDC is login-unlinkable with respect to an honest-but-curious IdP in the random oracle model, even if that IdP knows the value of any rp_nonce or pairwise_sub of the form* $H(user\_id||client\_id)$ *that were used during protocol runs.*

Proof. The messages that the IdP receives during a protocol run contain one value of the form $H(client\_id||rp\_nonce||u\_nonce)$, one value of the form $H(H(user\_id||client\_id)||u\_nonce\_2)$, and the messages exchanged during the zero-knowledge proof.

Consider an IdP that tries to distinguish the two settings where a user logs in to RP rp_0 with *client_id* cid_0 and one where the user logs in to RP rp_1 with *client_id* cid_1. In the first setting, the IdP receives $H(cid_0||rp\_nonce_0||u\_nonce)$ and $H(H(user\_id||cid_0)||u\_nonce\_2)$. In the second setting, he receives $H(cid_1||rp\_nonce_1||u\_nonce)$ and $H(H(user\_id||cid_1)||u\_nonce\_2)$.

In the random oracle model, all of these values are picked randomly using the same distribution over the hash function's domain. Furthermore, both *u_nonce* and *u_nonce_2* are unpredictable random values, so the IdP cannot efficiently compute a pre-image of the hash functions, not even with access to any *rp_nonce* or *pairwise_sub* of the form $H(user\_id||client\_id)$.

The messages exchanged in the zero-knowledge proof do not provide any information about the *client_id* by the definition of a zero-knowledge proof. □

We next also show that pairwise POIDC provides login unlinkability with respect to colluding RPs. Note that we assume that *id_token*s contain no user-identifying information in addition to the subject identifiers.

Theorem 3. *Pairwise POIDC is login-unlinkable with respect to colluding RPs in the random oracle model.*

Proof. Each pairwise subject identifier has the form $H(user\_id||client\_id)$. In the random oracle model, each such value with a different *client_id* is a different random value from the hash function's domain, and thus does not provide any information about the *user_id*. Furthermore, the *user_id* is an unpredictable random value, and thus no RP can efficiently compute a pre-image of the hash function. □

Note that leaking a user's *user_id* would break login unlinkability with respect to colluding RPs. Thus, pairwise POIDC implementations must ensure that the *user_id* is not persisted in the user agent's state in a way that could be accessible by malicious RPs.

6 SECURITY PROOFS WITH TAMARIN

We used the Tamarin prover [29, 31] to prove security properties for standard OIDC, both implicit flow and code flow, as well as POIDC and pairwise POIDC. All Tamarin theories and proofs can be found at [23].

6.1 The Tamarin Prover

Tamarin [29, 31] is a state-of-the-art automated tool for security protocol verification in the symbolic model of cryptography. In Tamarin, protocols are described by multiset rewriting rules, and trace properties, like secrecy and authentication, are specified in a first-order logic fragment. Tamarin provides a counterexample, representing an attack, if a verification attempt terminates without proof. In general, termination is not guaranteed due to the undecidability of the underlying protocol verification problem.

Tamarin works in the symbolic model, also called Dolev-Yao (DY) model, where messages are represented as terms. The DY attacker represents an attacker who controls the network. He sees all messages and can block, reorder, and manipulate them, but he cannot break cryptography, e.g., he cannot forge signatures.

Security protocol execution is represented by a labeled state transition system. Here, a *state* is a finite multiset of so-called facts, including terms as their arguments, and models the current snapshot of the protocol's execution. This includes the local state of all participants, as well as all messages sent over the network and the messages an attacker can construct from those. The steps performed by a protocol participant or the attacker are specified as *rules*, rewriting one state into another. Rules are of the form `name: l -[ a ]--> r` with `l`, `a`, and `r` all finite sequences of facts. `l` are called the premises, `a` the actions, and `r` the conclusions of the rule. A step of multiset rewriting starts in a given state s, uses a rule `name: l -[ a ]--> r`, and can be executed if there is a substitution σ such that $\sigma(l) \in s$. Triggering the rule results in a new state $s' = (s \setminus \sigma(l)) \cup \sigma(r)$. Note that we simply write `name: l --> r` if there are no actions, or we present a rule where actions are omitted for brevity.

An execution starts with the empty multiset and alternates states and rewriting rule instances. The rule instance between two states is a step between those states. The *trace* of the execution is the sequence of the used rules' actions. A *trace property* is defined over traces, and holds for a protocol if and only if it holds on all possible traces for that protocol, resulting from all possible interleavings of executions of multiset rewriting rules modeling the protocol. These rewriting rules model protocol execution steps from an unbounded number of concurrent protocol sessions and attacker actions.

6.2 Signature model

Symbolic models of cryptography have traditionally formalized signature verification by the equation

$$\text{verify}(\text{sign}(\text{msg}, \text{sk}), \text{msg}, \text{pk}(\text{sk})) = \text{true} \,.$$

Any input to verify that is not exactly of this form does not match the pattern and thus implicitly returns false. Our models instead use an improved, more precise, way to model signatures recently introduced by Jackson et al. [26]. They show that using the equation above does not accurately capture subtle behaviors that are present even in EUF-CMA (existential unforgeability under chosen message attack) secure signature schemes.

In particular, the prevailing definition for unforgeability does not impose any restrictions on signatures that are verified with respect to an attacker-generated public key. For example, it may be possible to generate a public key pk' and a signature sig' such that $\text{verify}(\text{sig}', \text{msg}, \text{pk}') = \text{true}$ for any msg. These kinds of behaviors are not modeled by the equation traditionally used in symbolic verification, and thus verification would not find any attacks that exploit such behavior. We use the *verification model* introduced in [26] that works as follows.

The verify check *must* return true if not doing so would violate the *correctness* of the signature scheme. The verify check must *not* return true if doing so would violate *EUF-CMA security*, i.e., the unforgeability, of the signature scheme. In all other scenarios, e.g., when verify is used with respect to a public key that was *not* generated using the prescribed generation algorithm, the attacker chooses the result returned by verify, while maintaining consistency between multiple checks with the same arguments.

This model provides a sound over-approximation of the subtle behaviors that may be possible even for an EUF-CMA secure signature scheme. In particular, a proof in this model holds for any EUF-CMA secure signature scheme, even if the attacker could exploit unexpected behavior that may arise when signatures are verified using attacker-generated public keys.

6.3 OpenID Connect model in Tamarin

We next explain our Tamarin models for OIDC and POIDC. These protocols differ from traditional cryptographic protocols, and the differences affect modeling.

The first important difference is that these protocols operate on the application layer. As such, they rely on the security of the secure channel, usually provided by TLS. Thus, it is insufficient to model an insecure network and cryptography: TLS channels with abstract security properties must also be modeled.

Another important difference is that the user directly participates in the protocol via her user agent (browser), while usually security protocols describe only the machines at both ends. The user's intent and consent to log in to an RP are critical for defining our main security property. Thus, the user must be modeled as a separate protocol role who interacts with her user agent.

We next describe our abstract TLS channel model and our user-browser communication model. These models provide the baseline for our protocol models and are included in each of them.

Abstract TLS model and agent compromise. We assume that TLS and the underlying public-key infrastructure are secure. A formal analysis of TLS is out of scope for this work, but can be found in [13]. We model a TLS connection abstractly as a channel between a browser endpoint, which is not authenticated, but is invariant during a session, and an authenticated server endpoint. This models the common scenario where server certificates are used, but no client certificates. The server endpoint is modeled as a public name `$Server` and the browser endpoint as a session identifier `~brID_TLS`, which is freshly generated for each session.

We next show in more detail how we model abstract TLS channels using rules in our Tamarin models. The rule for initiating a TLS session in our Tamarin models looks as follows.

```
rule Init_TLS_Session:
[ !St_Browser_Init(~brID, $User), Fr(~brID_TLS) ] -->
[ !St_Browser_Session(~brID, $Server, ~brID_TLS),
!St_Server_Session($Server, ~brID_TLS) ]
```

`Fr(~brID_TLS)` models `~brID_TLS` as a freshly generated, unique, and unpredictable value. The rule generates two facts that are used in protocol rules to exchange messages over the TLS channel. Note that facts starting with an exclamation mark, such as `!St_Browser_Init(...)` can be used multiple times, i.e., they are not consumed when used as the input of a rule. We show an example protocol rule to illustrate how these facts model received and sent messages. The rule models an authentication request to the IdP in OIDC implicit flow. We omit some details, since the purpose is just to illustrate the TLS channel model.

```
rule AuthRequest_IdP:
[ !Client_to_Server_TLS(~brID_TLS, $IdP
, <'authRequest', ...>)
, !St_Server_Session($IdP, ~brID_TLS)
, !St_RP_Registered($RP, $IdP, $client_id, ...) ] -->
[ St_IdP_1(...)
, !Server_to_Client_TLS($IdP, ~brID_TLS
, <'show', <'authDialog', $IdP>>) ]
```

We show the corresponding rules executed by the browser later. Note that the facts denoting messages on a channel can be used as input multiple times, modeling a channel that is not inherently replay-protected.

We add rules to Tamarin's built-in Dolev-Yao attacker that allow him also to create TLS channels. We also provide rules that allow the attacker to compromise RPs and IdPs. A compromised RP or IdP leaks its secrets to the attacker. Furthermore, an attacker can send arbitrary messages over TLS channels controlled by a compromised RP or IdP as well as learn any message received on such channels.

Modeling the user and user agent. We differentiate between the user (person) and the user agent (browser). We model communication between the user and her user agent as a secure, replay-protected channel. Malware on the user's device would make this assumption invalid, but is an orthogonal problem to the one we address in this work.

We assume that a single user agent is only used by one user at a time, but that a user can have multiple user agents. The user performs checks with respect to human-readable names and URIs. The user checks whether the RP's name in a consent dialogue is the one she intended to log in to. That is, the user remembers the RP's name to which she wants to log in, and only gives consent if the displayed name is the one she remembers.

The user performs actions, such as initiating the protocol (e.g., by clicking on a button on the RP page) or giving consent (by clicking on a button in a consent dialogue). The browser is reactive, and performs actions only if prompted to do so. The following three kinds of prompts can be sent to the browser.

(1) User actions, such as clicking on a link or entering a URI.
(2) A server sending content, triggering a web page to be displayed to the user, and/or the execution of JavaScript code.
(3) A server redirecting the browser to another URI.

We next show how we formalize each of these actions in our Tamarin models. User actions are modeled as follows.

```
rule Browser_Performs_User_Action:
[ User_InputTo_Browser($User, ~brID,
<'userAction', $Server, message>)
, !St_Browser_Init(~brID, $User)
, !St_Browser_Session(~brID, $Server, ~brID_TLS)] -->
[ !Client_to_Server_TLS(~brID_TLS, $Server, message) ]
```

The `User_InputTo_Browser` fact models a secure message from the user to the browser. For example, when the user clicks on a *Login with IdP* idp button on RP rp, this is modeled as a secure message `<'userAction', $rp, <'loginWith', $idp>>` from the user to the browser. The browser then reacts to this input by sending a message to the server named. In the example, the browser sends a TLS message `<'loginWith', $idp>` to the RP where the user clicked on the button.

We next show the rule for displaying web pages to the user.

```
rule Browser_Shows:
[ !Server_to_Client_TLS($Server, ~brID_TLS,
<'show', message>)
, !St_Browser_Session(~brID,  $Server, ~brID_TLS)
, !St_Browser_Init(~brID, $User)] -->
[ Browser_Shows_User(~brID, $User, $Server, message) ]
```

When the browser receives a TLS message `<'show', message>` from a server, the browser then displays `message` to the user, modeled as a `Browser_Shows_User` fact.

For example, when the IdP requests the user to authenticate, the IdP sends `<'show', 'authDialog'>` to the browser. The browser then displays a dialogue asking the user to enter her credentials for IdP I, `Browser_Shows_User(~brID, $User, $I, 'authDialog')`. We next show the rule for redirection.

```
rule Browser_Redirects_To_URI:
[ !Server_to_Client_TLS($Server1, ~brID_TLS1,
<'redirectToURI', $uri, message>)
, !St_Browser_Session(brID, $Server1, ~brID_TLS1)
, !St_Browser_Session(brID, $Server2, ~brID_TLS2)
, !Uri_belongs_to($uri, $Server2)] -->
[ !Client_to_Server_TLS(~brID_TLS2, $Server2, message) ]
```

The browser receives `<'redirectToURI', $uri, message>` from server `$Server1`, requesting to redirect `message` to `$uri`. The fact `!Uri_belongs_to($uri, $Server2)` denotes that this URI belongs to server `$Server2`, so the browser redirects `message` to `$Server2`. Note that these rules require that the browser has active TLS sessions with one or more servers, which are modeled by the `!St_Browser_Session(...)` facts. If the browser does not yet have an active session with one of these servers, then the `Init_TLS_Session` rule must first be executed.

In general, the browser is stateless and unaware of any protocol logic. The one notable exception is the JavaScript code that the browser loads from the IdP website in POIDC and pairwise POIDC. In this case, the browser keeps state that models its progression through the code execution. In particular, the browser is in one of two possible states after loading the code and displaying the consent dialogue: (1) waiting for the user to give consent, or (2) waiting for a *private_id_token* to be returned from the IdP back-end. After the browser sends the *private_id_token* to the *redirect_uri* given in the *client_id_binding*, the JavaScript code execution is finished.

6.4 Security property and proofs

We formalize *user authentication* similarly to the authentication property given by Fett, Küsters, and Schmitz [20]: An attacker should not be able to log in under an honest user's account at an honest RP using an honest IdP. For our security property definition, an honest user is one where the user's account at the IdP has not been compromised, and the user does not use compromised browsers or platforms. An honest RP or IdP is an entity that is not controlled by the attacker.

We explicitly model the user's actions separate from the browser's, and our security property also refers to the user's actions. An honest user using an uncompromised user agent should only be logged in at an honest RP using an honest IdP if the user has both expressed her *intent* (by starting the protocol) and her *consent* (by confirming a consent dialogue). We formalize this as follows.

Definition 5. A delegated authentication protocol is *secure with respect to user authentication* if the following holds for any honest RP rp, honest IdP idp, honest user u, and browser b. Whenever u is logged in at RP rp with IdP idp on b as a result of running the protocol, then b belongs to u, u has initiated the protocol, and u has given consent to log in to RP rp with IdP idp.

Our formalization of this definition in Tamarin differs slightly for the different flows. For OIDC implicit flow and POIDC, we use the following formalization.

```
lemma Intent_Consent_and_Correct_Browser:
"All rp uid idp #finish.
RP_gets_IDToken(rp, uid, idp)@finish
& not (Ex #j. Is_Compromised(idp)@j)
& not (Ex #j. Is_Compromised(rp)@j)
& not (Ex #j. AdversaryRegisters(uid)@j) ==>
(Ex usr browserSession browser
#k #start #consent #m #n.
UserID_belongs_To(uid, usr, idp)@k // (1)
& UserGivesConsent(usr, rp, idp)@consent // (2)
& UserStartsSession(usr, rp, idp)@start // (3)
& BrowserUser(browser, usr)@n // (4)
& BrowserServerSession(browser, rp, session)@m // (4)
& RPgetsIDToken_FromBr(rp, session, idp)@finish // (4)
& consent < finish & start < consent)" // (5)
```

This lemma can be understood as follows. Whenever an honest RP `rp` receives an *id_token* that contains a subject identifier `uid` that belongs to an honest user and an issuer identifier `idp` that belongs to an honest IdP, then the following hold.

(1) The subject identifier `uid` belongs to the user `usr`.
(2) The user `usr` has given consent to log in to `rp` using `idp`.
(3) The user `usr` has started the protocol session intending to log in to `rp` using `idp`.
(4) The RP obtained the *id_token* in a session with a browser that belongs to `usr`.
(5) The protocol start happened before giving consent, and giving consent happened before the RP received the *id_token*.

We next explain the adjustments that must be made for the OIDC code flow models. When the code flow is used, the RP receives the *id_token* from the IdP directly rather than from the browser. Thus, in condition (4), we require that the RP receives the *code* instead of the *id_token* from the user's browser, denoted by `RPgetsCode(rp, browserSession, idp)@getCode` in that model.

For pairwise POIDC, the *id_token* does not directly contain the user's global user identifier `uid`, but rather a pairwise subject identifier of the form `h(<uid, cid>)`, and we write

```
(RP_gets_IDToken(rp, h(<uid, cid>), idp)@finish
```

in the left-hand side of the implication. Thus, the lemma ensures that this identifier was computed from the *user_id* `uid` that belongs to the user `usr` who gave consent and started the session.

We provide five Tamarin models, each with between 320 and 340 lines of code. The models are available at [23]. The proofs for the following theorems were computed automatically on a server with 2 CPUs of type Intel(R) Xeon(R) E5-2650 v4 @ 2.2 GHz, 256 GB of RAM, running Ubuntu 16.04.3 LTS. Each CPU has 12 cores, but we limited our computation to use 10 cores only. The proof runtimes are given in Figure 5. The proofs are also available at [23].

Theorem 4. *OIDC implicit flow and authorization code flow, both with and without client secret, are secure with respect to user authentication.*

Theorem 5. *POIDC and pairwise POIDC are secure with respect to user authentication.*

	Proof Runtime
OIDC implicit flow	14s
OIDC code flow without client secret	30m41s
OIDC code flow with client secret	3m20s
POIDC	28m50s
Pairwise POIDC	13m14s

Figure 5: Proof runtime for automatic proof of user authentication in five different Tamarin models.

7 RELATED WORK

We first compare POIDC and pairwise POIDC to other privacy-preserving single sign-on solutions. Afterwards, we compare our formal security analysis of OIDC, POIDC, and pairwise POIDC to work related to the formal analysis of OIDC and the underlying OAuth 2.0 protocol.

7.1 Privacy-preserving single sign-on

Fett, Küsters, and Schmitz have proposed the privacy-preserving single sign-on system SPRESSO [18], for which they prove that the IdP does not learn the RP where the user logs in. Their approach is different from ours: They develop an entirely new protocol whereas we propose changes to the existing OIDC standard. Furthermore, they do not protect privacy with respect to colluding RPs. Their proposal (and ours) are the only ones that we are aware of that actually prove the claimed security and privacy properties.

Other single sign-on systems with enhanced user privacy have been proposed. For example, Maheswaran et al. propose CryptoBook [28], which builds a privacy-preserving cryptographic layer on top of existing protocols such as OAuth 2.0. Their system requires a user to obtain credentials from multiple credential producers (IdPs), and while it makes use of OAuth 2.0, it is an entirely different credential system rather than an extension of an existing standard. Dey and Weis propose PseudoID [14], a privacy-preserving extension for OpenID, OIDC's predecessor, based on blind signatures. Their solution introduces a blind signing service where the user must first blind a token that she then presents to the IdP. The use of this service adds steps to the user login process. The Mozilla Persona (or BrowserID) single sign-on system [2], which is no longer supported, was also designed to provide privacy-preserving single sign-on based on a user's e-mail address. However, several attacks on its privacy properties were given by Fett, Küsters, and Schmitz [17].

An alternative to systems based on an identity provider are user-centric systems that require the user to use a secret key when logging in. The most prominent line of work in this area is on anonymous credentials, introduced by Camenisch and Lysyanskaya [9] and implemented in idemix [11]. Camenisch and Pfitzmann explain the privacy issues of federated identity management and how idemix can solve them [10]. There are also other user-centric identity systems based on similar ideas, such as U-Prove [3]. However, these systems have seen little adoption and have not challenged the prominence of standards such as OIDC. One reason for this lack of adoption is the difficulty of key management. This problem also applies to proposals that combine delegated authentication protocols with secret keys held by users, such as the proposal by Camenisch, Gross, and Sommer to improve privacy in the WS-Security standard [8], or UnlimitID by Isaakidis et al. [25].

7.2 Formal analysis of OIDC or OAuth 2.0

Fett, Küsters, and Schmitz have used their web infrastructure model (WIM) [17] for the manual analysis of OIDC [20]. They also analyzed the OpenID Financial-grade API using the same model [16], as well as OAuth 2.0 [19]. In our OIDC analysis using Tamarin, we model the web infrastructure more abstractly to allow for proof automation and machine-checked proofs.

In the realm of automated analysis, Bansal et al. have used the WebSpi [5] framework for analyzing web applications based on ProVerif [7] to automatically discover attacks on OAuth 2.0. However, neither WebSpi nor WIM model the user as an entity separate from the browser, which makes it difficult to reason about explicit user consent. An interesting direction for future work in this area is to formally model a more detailed web infrastructure, closer to WIM, while still allowing for automation, and additionally model the user as a separate entity. Such a model could be used to obtain even stronger guarantees about the security of web-based protocols such as OIDC.

8 CONCLUSION

We have presented two extensions of OpenID Connect that prevent the IdP from learning which RPs a user visits. POIDC requires only few changes to OIDC and protects privacy with respect to the IdP, while pairwise POIDC additionally protects privacy with respect to colluding RPs at the cost of performing a zero-knowledge proof. We have given manual privacy proofs and machine-checked security proofs for our protocols. Ours are the first machine-checked proofs for OpenID Connect and include our extensions.

Users, IdPs, and RPs all benefit from our proposal. Users directly benefit since they have the guarantee that the IdP does not learn to which RPs they log in, reducing the amount of trust they must place in the IdP. RPs, especially those offering sensitive services, no longer need to trade-off features offered by OpenID Connect against their users' privacy. IdPs decrease the amount of critical data in their application logs, which helps them to conform to data minimization regulations, such as those mandated by the GDPR.

Due to the widespread deployment of OpenID Connect, our proposal has the potential to significantly improve privacy for most internet users. We therefore plan to work with the OpenID Connect standardization committee to adopt our extensions as new OpenID Connect flows in the standard. We are currently working on a reference implementation of our proposal.

9 ACKNOWLEDGEMENTS

We thank Xenia Hofmeier for constructing preliminary versions of the Tamarin models for OIDC implicit flow and authorization code flow. We also thank the members of the Information Security Group at ETH Zurich for their valuable feedback throughout different stages of this project. Finally, we thank the Zurich Information Security and Privacy Center (ZISC) for supporting this work.

REFERENCES

[1] GDPR, Art. 5, 1.c. https://gdpr-info.eu/art-5-gdpr/. Accessed: 2020-03-02.

[2] Mozilla Persona Website. https://developer.mozilla.org/en-US/docs/Archive/Mozilla/Persona. Accessed: 2020-03-02.

[3] U-Prove. https://www.microsoft.com/en-us/research/project/u-prove/. Accessed: 2020-03-02.

[4] verimi.de. https://verimi.de/. Accessed: 2020-03-02.

[5] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Discovering concrete attacks on website authorization by formal analysis. *Journal of Computer Security*, 22(4):601–657, 2014.

[6] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. Uniform resource identifier (uri): Generic syntax, section 3.5. STD 66, RFC Editor, January 2005. http://www.rfc-editor.org/rfc/rfc3986.txt.

[7] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Proceedings of the 14th IEEE workshop on Computer Security Foundations*, page 82, 2001.

[8] Jan Camenisch, Thomas Gross, and Dieter Sommer. Enhancing privacy of federated identity management protocols: anonymous credentials in ws-security. In *Proceedings of the 5th ACM workshop on Privacy in Electronic Society*, pages 67–72. ACM, 2006.

[9] Jan Camenisch and Anna Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. *Advances in Cryptology - EUROCRYPT 2001*, pages 93–118, 2001.

[10] Jan Camenisch and Birgit Pfitzmann. Federated Identity Management. In *Security, Privacy, and Trust in Modern Data Management*, pages 213–238. Springer, 2007.

[11] Jan Camenisch and Els Van Herreweghen. Design and implementation of the idemix anonymous credential system. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 21–30. ACM, 2002.

[12] David L Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[13] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017*, pages 1773–1788. ACM, 2017.

[14] Arkajit Dey and Stephen Weis. Pseudoid: Enhancing privacy in federated login. In *Hot Topics in Privacy Enhancing Technologies*, pages 95–107, 2010.

[15] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, Naval Research Lab Washington DC, 2004.

[16] Daniel Fett, Pedram Hosseyni, and Ralf Küsters. An extensive formal security analysis of the openid financial-grade api. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 453–471. IEEE, 2019.

[17] Daniel Fett, Ralf Küsters, and Guido Schmitz. An expressive model for the web infrastructure: Definition and application to the browser id sso system. In *2014 IEEE Symposium on Security and Privacy (SP)*, pages 673–688, 2014.

[18] Daniel Fett, Ralf Küsters, and Guido Schmitz. SPRESSO: A secure, privacy-respecting single sign-on system for the web. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1358–1369, 2015.

[19] Daniel Fett, Ralf Küsters, and Guido Schmitz. A comprehensive formal security analysis of oauth 2.0. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1204–1215. ACM, 2016.

[20] Daniel Fett, Ralf Küsters, and Guido Schmitz. The web SSO standard OpenID Connect: In-depth formal security analysis and security guidelines. *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pages 189–202, 2017.

[21] Dinei Florencio and Cormac Herley. A large-scale study of web password habits. In *Proceedings of the 16th international conference on World Wide Web*, pages 657–666. ACM, 2007.

[22] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. ZkBoo: Faster zero-knowledge for boolean circuits. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1069–1083, 2016.

[23] Sven Hammann, Ralf Sasse, and David Basin. Tamarin models for OpenID Connect and POIDC. https://github.com/tamarin-prover/tamarin-prover/tree/develop/examples/asiaccs20-POIDC, 2020.

[24] D. Hardt. The oauth 2.0 authorization framework. RFC 6749, RFC Editor, October 2012. http://www.rfc-editor.org/rfc/rfc6749.txt.

[25] Marios Isaakidis, Harry Halpin, and George Danezis. Unlimitid: Privacy-preserving federated identity management using algebraic macs. In *Proceedings of the 2016 ACM Workshop on Privacy in the Electronic Society*, pages 139–142. ACM, 2016.

[26] Dennis Jackson, Cas Cremers, Katriel Cohn-Gordon, and Ralf Sasse. Seems legit: Automated analysis of subtle attacks on protocols that use signatures. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 2165–2180, New York, NY, USA, 2019. ACM.

[27] M. Jones, J. Bradley, and N. Sakimura. Json web token (jwt). RFC 7519, RFC Editor, May 2015. http://www.rfc-editor.org/rfc/rfc7519.txt.

[28] John Maheswaran, Daniel Jackowitz, Ennan Zhai, David Isaac Wolinsky, and Bryan Ford. Building privacy-preserving cryptographic credentials from federated online identities. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 3–13. ACM, 2016.
[29] Simon Meier, Benedikt Schmidt, Cas J. F. Cremers, and David Basin. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *International Conference on Computer Aided Verification (CAV)*, volume 8044 of *LNCS*, pages 696–701. Springer, 2013.
[30] Nat Sakimura, John Bradley, Mike Jones, Breno de Medeiros, and Chuck Mortimore. OpenID Connect Core 1.0 incorporating errata set 1. 2014.
[31] Benedikt Schmidt, Simon Meier, Cas J. F. Cremers, and David Basin. Automated analysis of Diffie-Hellman protocols and advanced security properties. In *Computer Security Foundations Symposium (CSF)*, pages 78–94. IEEE, 2012.