Institute of Information Security

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

OpenID for Verifiable Credentials: Formal Security Analysis using the Web Infrastructure Model

Fabian Hauck

Course of Study:	Informatik
Examiner:	Prof. Dr. Ralf Küsters
Supervisor:	Dr. Daniel Fett Pedram Hosseyni, M.Sc.
Commenced:	April 4, 2023
Completed:	October 2, 2023

Abstract

In our increasingly connected world, digital identities play a fundamental role in delivering secure online services around the globe. To enable the seamless exchange of identification data among various entities, the adoption of standardized protocols is essential. The protocol family OpenID for Verifiable Credentials (OID4VC) is ideally suited for exchanging identities. The two most important protocols in this family are OpenID for Verifiable Credential Issuance (OID4VCI) and OpenID for Verifiable Presentations (OID4VP) with a wide range of applications in e-government as well as in the private sector. A prominent example is the European Digital Identity Framework [20], which includes these two protocols, among others. This means that any future wallet in the European Union will implement OID4VCI and OID4VP. Therefore, it is extremely important to guarantee their security.

This thesis performs a rigorous formal security analysis of both the OpenID for Verifiable Credential Issuance [16] and the OpenID for Verifiable Presentations [19] protocols. In particular, we focus on analyzing the security of both protocols when they interact in an ecosystem. It is not sufficient to consider the two protocols separately, because the interaction between them may introduce new vulnerabilities. Therefore, the formal model created in this thesis models both protocols simultaneously. The model is based on the Web Infrastructure Model (WIM) [7], which closely follows existing web technologies. To describe what security means in this context, we define an authentication security property and a session integrity security property for OID4VCI and OID4VP. We prove that the model is secure with respect to the security properties under the assumption of a vigilant user. If this assumption is violated, we have discovered a number of attacks.

This work makes several contributions to the protocol specifications: First, the discovered vulnerabilities were brought to the attention of the working group. Second, several issues were filed to improve the quality and security of the specifications. Lastly, we engaged in ongoing discussions on related issues.

Kurzfassung

In unserer immer stärker vernetzten Welt spielen digitale Identitäten eine fundamentale Rolle bei der Bereitstellung von sicheren Online-Dienstleistungen auf der ganzen Welt. Um den nahtlosen Austausch von Identitätsdaten zwischen verschiedenen Organisationen zu ermöglichen, ist die Einführung von standardisierten Protokollen unerlässlich. Die Protokollfamilie OpenID for Verifiable Credentials (OID4VC) ist ideal geeignet für den Austausch von Identitäten. Die beiden wichtigsten Protokolle der Familie sind OpenID for Verifiable Credential Issuance (OID4VCI) und OpenID for Verifiable Presentations (OID4VP). Diese haben ein breites Anwendungsspektrum sowohl bei E-Government Services als auch in der Privatwirtschaft. Ein prominentes Beispiel ist das European Digital Identity Framework [20], das unter anderem die beiden Protokolle OID4VCI und OID4VP beinhaltet. Das bedeutet, dass jedes zukünftige Wallet in der Europäischen Union diese Protokolle implementieren wird. Daher ist es äußerst wichtig, ihre Sicherheit zu gewährleisten.

In dieser Arbeit wird eine gründliche formale Sicherheitsanalyse der Protokolle OpenID for Verifiable Credential Issuance [16] und OpenID for Verifiable Presentations [19] durchgeführt. Insbesondere konzentrieren wir uns auf die Analyse der Sicherheit beider Protokolle, wenn sie in einem Ökosystem interagieren. Es reicht nicht aus, die beiden Protokolle getrennt zu betrachten, da die Interaktion zwischen ihnen neue Schwachstellen hervorbringen kann. Daher modelliert das in dieser Arbeit erstellte formale Modell beide Protokolle gleichzeitig. Das Modell basiert auf dem Web Infrastructure Model (WIM) [7], das sich eng an bestehenden Webtechnologien orientiert. Um zu beschreiben, was Sicherheit in diesem Zusammenhang bedeutet, definieren wir die beiden Sicherheitseigenschaften Authentication und Session Integrity für OID4VCI und OID4VP. Dann zeigen wir, dass das Model in Bezug auf diese Sicherheitseigenschaften unter der Annahme eines aufmerksamen Benutzers sicher ist. Für den Fall, dass diese Annahme verletzt wird, haben wir eine Reihe von Angriffen entdeckt.

Diese Arbeit liefert mehrere Beiträge zu den Protokollspezifikationen. Erstens wurden die entdeckten Schwachstellen der Arbeitsgruppe zur Kenntnis gebracht. Zweitens wurden mehrere Issues zur Verbesserung der Qualität und Sicherheit der Spezifikationen eingereicht. Zudem haben wir aktiv an den laufenden Diskussionen zu den Themen teilgenommen.

Acknowledgment

First of all, I would like to thank Prof. Ralf Küsters for providing me the opportunity to write a master's thesis in the field of formal security analysis. The analysis of protocols from the OpenID for Verifiable Credentials family was especially interesting for me because I have practical experience with them and they are important for the European Digital Identity Framework. Thanks also to my supervisor Pedram Hosseyni at the University of Stuttgart for his constructive feedback during the entire process.

This master's thesis was generously supported by Verimi GmbH through the IDunion project. I am especially grateful to my supervisor Daniel Fett for always answering my questions and giving me valuable advice throughout the process. Furthermore, I would like to thank my family for supporting me during the time of writing this thesis and my girlfriend Vanessa for her help in correcting the language and grammar.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

DY Dolev-Yao. 29

IdP Identity Provider. 21

OID4VC OpenID for Verifiable Credentials. 3

OID4VCI OpenID for Verifiable Credential Issuance. 3

OID4VP OpenID for Verifiable Presentations. 3

PAR OAuth 2.0 Pushed Authorization Requests. 24

PKCE Proof Key for Code Exchange. 24

SIOPv2 Self-Issued OpenID Provider v2. 18

WIM Web Infrastructure Model. 3

1 Introduction

Every few years, the United Nations measures the level of digitization in each of its 193 member states with the E-Government Development Index. This index comprises the dimensions of online services, human capacity, and telecommunication connectivity. The global average of this index has risen from 0.36 in 2003 to 0.61 in 2022, with values ranging from 0 to 1.[1] This shows how important digitization has become for societies around the world.

The fundamental building block for secure digital services, especially e-government services, are tamper-proof digital identities. Most current systems use a federated identity model, which means that a single authority controls all identities. This has major privacy and data sovereignty drawbacks. For these reasons, the three-party model[2] using verifiable credentials has been developed. The

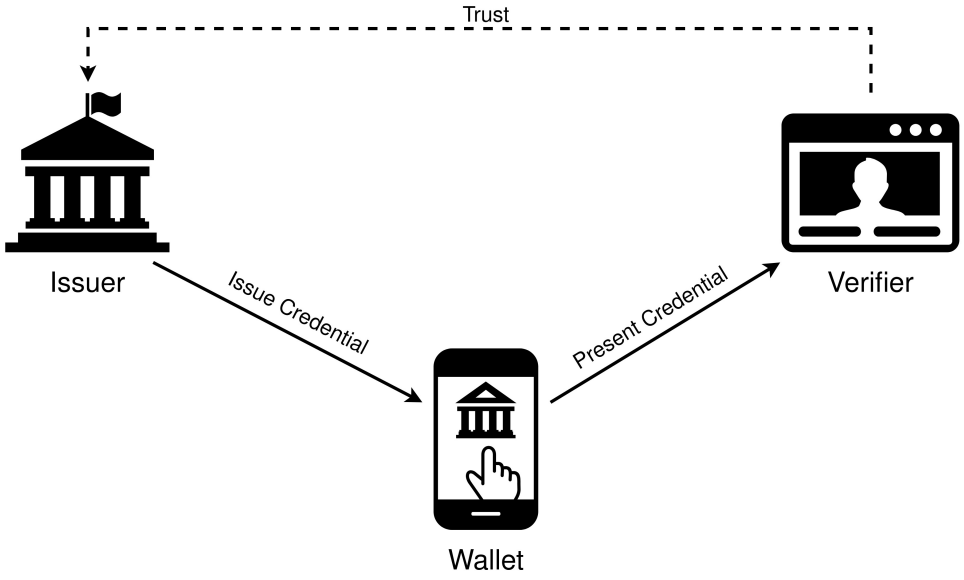

Figure 1.1: Three-party model.

three-party model consists of issuers, wallets, and verifiers, as seen in Figure 1.1. The verifiable credentials are issued by an issuer and stored in the user's wallet. The user can then use the verifiable credentials, more or less independently of the issuer, with any verifier that wants to know verified data about the user.

[1] https://publicadministration.un.org/egovkb/en-us/Data-Center

[2] Also known as the issuer-wallet-verifier model, decentralized identity, or self-sovereign identity (SSI).

To enable the exchange of verifiable credentials between different parties, standardized protocols are needed. Today, many different credential formats exist, such as SD-JWT [8], ISO mDL [12], W3C Verifiable Credentials [18], and AnonCreds [2]. The challenge is that most of these credential formats use their own transport protocol, which makes interoperability very difficult. This has led to the need for a credential format independent transport protocol.

Since OAuth 2.0 and OpenID Connect are already battle-tested protocols for resource authorization and identity transfer, it makes sense to build on top of them to create protocols for credential exchange. With this in mind, the protocol family OpenID for Verifiable Credentials (OID4VC) was developed. The OID4VC family is unique because, to our knowledge, it is the only format independent, standardized credential exchange protocol. The OID4VC protocol family includes OpenID for Verifiable Credential Issuance (OID4VCI), OpenID for Verifiable Presentations (OID4VP), Self-Issued OpenID Provider v2 (SIOPv2), OpenID for Verifiable Presentations over BLE, and OpenID Connect UserInfo Verifiable Credentials.

The OpenID for Verifiable Credential Issuance [16] and OpenID for Verifiable Presentations [19] protocols are the most important because there are already implementation plans and prototypes based on these specifications. For example, OID4VP will be implemented by the NIST National Cybersecurity Center of Excellence and is included in the DIF JWT VC Presentation Profile.[3] Furthermore, OID4VCI and OID4VP are part of the European Digital Identity Framework [20], which implies that any future wallet in the European Union will support them. This means that it is extremely important to ensure the security of these two protocols.

This thesis performs a rigorous formal security analysis of the OpenID for Verifiable Credential Issuance and OpenID for Verifiable Presentations protocols using the Web Infrastructure Model (WIM) [7]. The focus of this analysis is to analyze the interaction of OID4VCI and OID4VP in an ecosystem. It is not sufficient to analyze the protocols in isolation, as the interaction may create new vulnerabilities. Such a detailed formal security analysis of three-party model protocols used in practice is a novelty since, to the best of our knowledge, it has never been done for similar protocols.

A formal security analysis with the WIM is different from a penetration test. With the WIM specifications are analyzed, whereas in a penetration test implementations are tested. This means that the WIM cannot find vulnerabilities in implementations, whereas a penetration test can. The benefit of the WIM is that it can find unknown types of attacks, while a penetration test usually only finds known types of attacks. The advantage of a formal security analysis with the WIM over a penetration test is that the formal approach proves the security of the protocols with respect to the definition of security. This means that there is a much higher security guarantee than with a penetration test.

A formal analysis with the WIM is done in three steps, as shown in Figure 1.2. The first step is to create a formal model from the OID4VCI specification and the OID4VP specification. The underlying protocols are described in Chapter 2 and the formal model is described in Chapter 3. The second step is to define what security means in this context, for example, that an attacker cannot log in as an honest user. The definition of security is formalized in the security properties found in Chapter 4. We have defined two types of security properties: One is the authentication security property for the OID4VCI and OID4VP protocols, and the other one is the session integrity

[3] https://openid.net/sg/openid4vc/

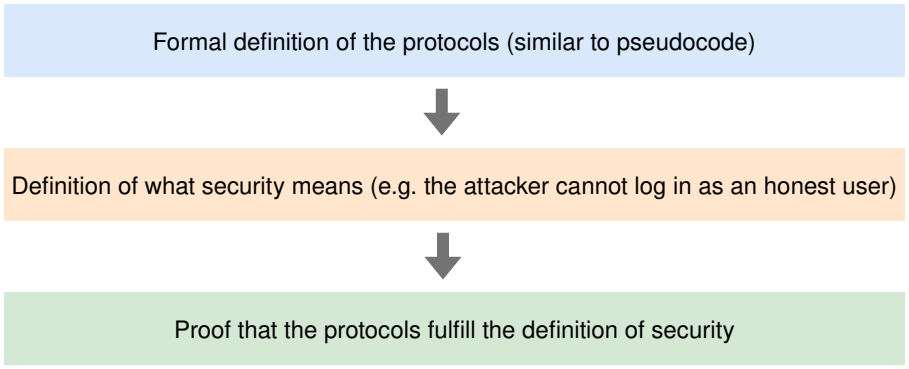

Figure 1.2: Formal security analysis using the WIM.

security property for the OID4VCI and OID4VP protocols. The final and central part of the analysis is the security proof, which shows that the formal model is secure with respect to the security properties. While trying to prove these security properties, we discovered attacks, which means that the model needs to be fixed and the proof repeated. These fixes are typically changes to the protocol or assumptions about user behavior, and they model real-world mitigations. The proof sketch presented in Chapter 5 and the formal proof in Appendix C show that the fixed model is secure with respect to the security properties.

This thesis makes substantial contributions to several standardization documents to improve their quality and security. The first contribution is the disclosure of the discovered attacks described in Chapter 6 to the working group. The second contribution is the submission of issues to improve the clarity of the specifications, which can be found in Chapter 7. The third and final contribution is active participation in ongoing discussions on issues that can also be found in Chapter 7.

2 OpenID for Verifiable Credentials

Moving from a federated identity model to a three-party model (Figure 2.1) requires the introduction of verifiable credentials. Verifiable credentials are typically a set of claims digitally signed by an issuer. The issuer (equivalent to an Identity Provider (IdP) in the federated identity model) holds user claims and issues the verifiable credentials. User claims are information about a person, such as name, birthday, or address. The verifiable credential is issued in a wallet that securely stores the credential and private keys. Private keys are required for credentials that are cryptographically bound to a private key (holder key) so that only the user can use them (holder binding). The stored credential can be used to prove the user's identity to the verifier. To present the credential to a verifier, also known as a relying party in the federated identity model, the credential must be embedded in a presentation. Such a presentation typically contains an identifier of the verifier (audience) and a transaction-specific nonce to prevent replay attacks. If the credential has a holder binding, the presentation must be signed with the holder key. The verifier then verifies the signature of the credential and decides whether or not to trust the issuer.

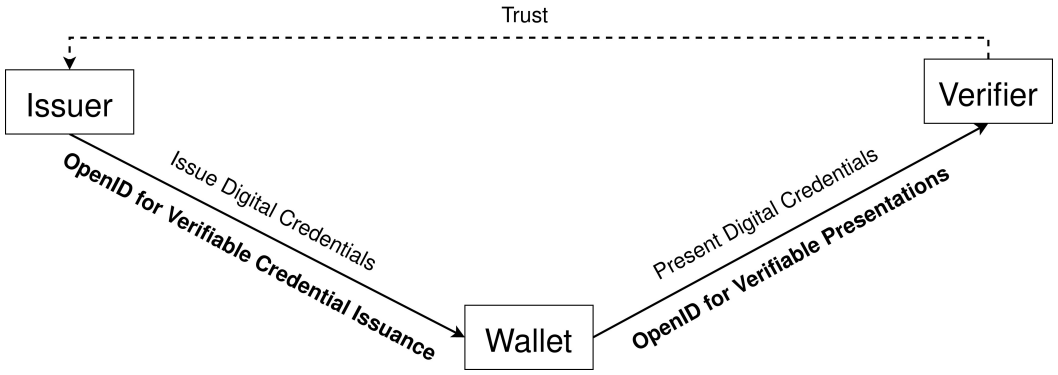

Figure 2.1: Three-party model.

To exchange verifiable credentials and presentations between entities, standardized protocols are needed. There are many different protocols, such as the ISO-compliant driving licence protocol [12] or DIDcomm [3] for AnonCreds [2] and W3C Verifiable Credentials [18]. These protocols have the disadvantage that they are designed to exchange a specific credential format. To our knowledge, the OpenID for Verifiable Credentials protocol family is the only transport protocol that is credential format independent. The OID4VC protocol family consists of five standardized protocols. To reduce complexity and make formal security analysis feasible, this thesis focuses on the two most important protocols, OID4VCI and OID4VP. This chapter provides an introduction to these two protocols.

2.1 OpenID for Verifiable Credential Issuance

The OpenID for Verifiable Credential Issuance specification [16] defines a protocol for issuing verifiable credentials regardless of their format. The protocol is based on OAuth 2.0 [9], which means it can use all the parameters of OAuth 2.0 and its extension mechanisms, but the recommendations in the OAuth 2.0 Security Best Current Practice [14] document should be considered.

The specification defines three new endpoints for issuing credentials. The first is a mandatory endpoint to issue a single credential (credential endpoint), the second is an optional endpoint to issue multiple credentials at once (batch credential endpoint), and the last is also an optional endpoint to issue credentials at a later time when they are not ready at issuance (deferred credential endpoint). These endpoints use the access token obtained from the token endpoint to authorize the issuance of verifiable credentials. Any grant type defined for OAuth 2.0 can be used to obtain an access token, but the implementer must consider the OAuth 2.0 Security Best Current Practice [14]. This means, for example, that grant types that return access tokens in the authorization response should not be used. In addition to the existing grant types, the OID4VCI specification defines a new flow called pre-authorized code. In this flow, the issuer, as the IdP, initiates the issuance. The two most common credential issuance flows are the authorization code flow and the pre-authorized code flow. The following subsection describes these flows in more detail. For the sake of brevity, the figures in these subsections show only the most important parameters in the flows.

For high-security use cases, it is important for the issuer to know what type of wallet the credential is being issued to. To ensure that the wallet complies with certain regulations and protects the private keys appropriately, the wallet needs to authenticate itself to the issuer. The specification recommends the use of OAuth 2.0 client authentication as defined in RFC 6749 [9] to establish trust.

In addition, this specification defines new metadata for the issuer that describes what kind of credentials can be issued with what claims and in what format.

2.1.1 Pre-Authorized Code Flow

The pre-authorized code flow is a newly designed grant type in the OID4VCI specification. The flow begins by authenticating the user and sending a credential offer to the wallet. The credential offer always contains an issuer identifier and details about the credentials to be issued. There are two ways to use this flow: First, the credential offer can be used to instruct the wallet to initiate an authorization code flow. In this case, the credential offer can include the *issuer_state* parameter to pass the user's state to the wallet. The second option is for the issuer to create a credential offer with a pre-authorized code and a *user_pin_required* value. The pre-authorized code essentially acts as an authorization code that can be exchanged at the token endpoint. If the user PIN is required, the wallet must send the user PIN along with the pre-authorized code to the token endpoint to obtain the access token. The user PIN is sent to the user out-of-band, such as by email or SMS. The token request typically includes the grant type, pre-authorized code, client ID, and optionally the user PIN. The token response contains the access token and optionally a refresh token and *c_nonce*. The second option with the pre-authorized code can be seen in Figure 2.2.

The access token can be used on the credential endpoint or the batch credential endpoint to obtain the credential. To invoke the credential endpoint, the wallet has to authenticate with the access token, send information about the format of the credential, and optionally include a cryptographic proof of ownership of a private key. The proof of ownership must include the *c_nonce* obtained at the token endpoint or the credential endpoint. The credential response contains the credential format, the credential or a transaction ID, and optionally a new *c_nonce*. The transaction ID is used if the credential could not be issued immediately and the wallet needs to try again at a later time. The batch credential endpoint is used in the same way as the credential endpoint, except that all parameters are sent in an array for each credential requested. Accordingly, the response is also an array with one entry for each credential. The last endpoint is the deferred credential endpoint, which is used to issue a credential at a later time if the credential cannot be issued immediately. To call this endpoint, the wallet needs the access token and transaction ID for the credential that could not be issued. The response will contain the format and the credential or an error, e.g. the credential is not yet ready.

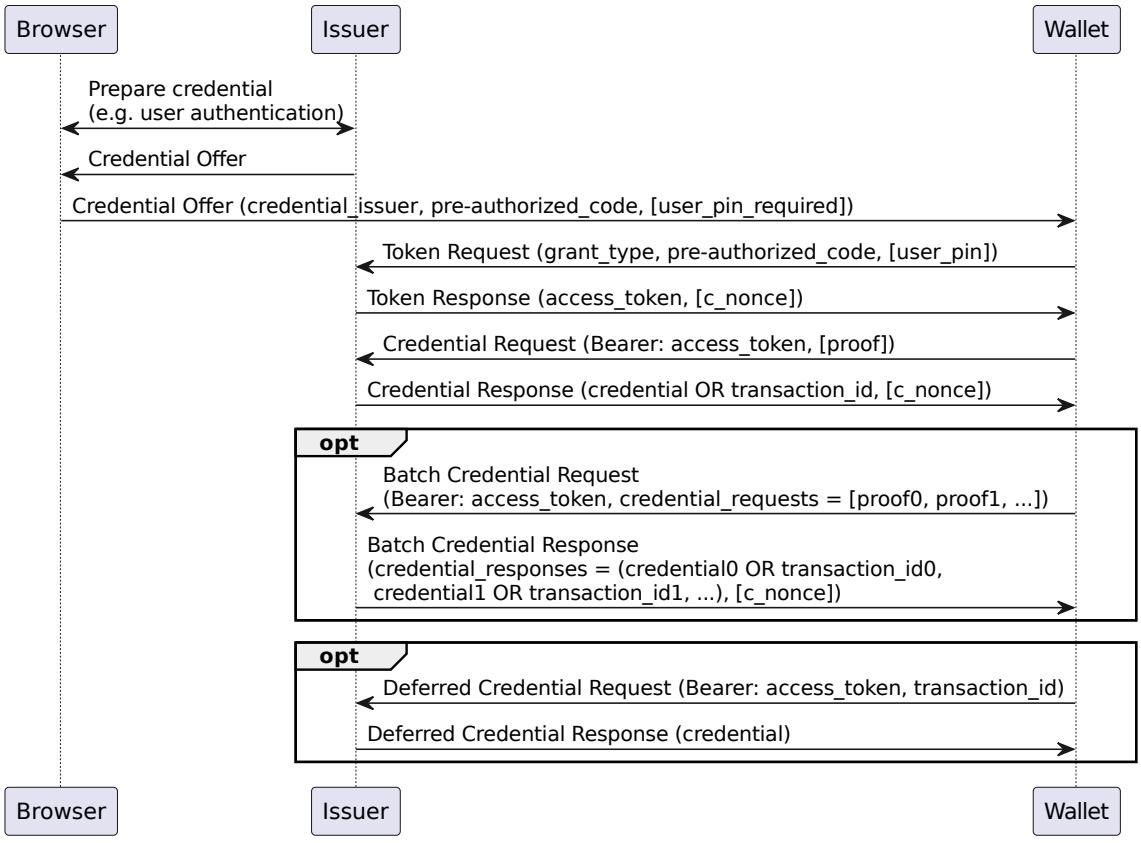

Figure 2.2: Pre-authorized code flow.

2.1.2 Authorization Code Flow

The authorization code flow is another grant type discussed in the OID4VCI specification. In this flow, the wallet initiates the credential issuance instead of the issuer. Figure 2.3 shows the authorization code flow to issue a credential. The flow starts with an authorization request, which

includes parameters from OAuth 2.0 and can also include parameters from its extensions. A typical authorization request would include the response type, client ID, redirect URI, code challenge, code challenge method, and authorization details. The code challenge is included because the use of Proof Key for Code Exchange (PKCE) [17] is recommended, and the authorization details are included to tell the issuer what types of credentials should be issued. In addition to the authorization details, it is also possible to request a predefined credential type with a scope value. The authorization request parameters can be sent directly to the issuer, but for security reasons it is recommended to use OAuth 2.0 Pushed Authorization Requests (PAR) [15] and only send the request URI through the front channel to the issuer. When the user authenticates to the issuer, the issuer may initiate a dynamic credential request to retrieve credentials already present in the wallet. In this case, the issuer becomes the verifier and the wallet is the IdP. It is recommended to use OID4VP to request the credentials. After successfully authenticating the user, the issuer redirects the user back to the wallet with the authorization response containing the authorization code, the iss parameter, and the state parameter if present in the authorization request. The iss parameter is a recommended parameter in the OAuth 2.0 Security Best Current Practice [14] document to prevent mix-up attacks.

The next step for the wallet is to exchange the authorization code for an access token at the token endpoint. This follows the rules of OAuth 2.0, which means that the request typically includes the authorization code, grant type, client ID, redirect URI, and code verifier. The token response contains an access token and optionally a *c_nonce*. With the access token and the *c_nonce*, the wallet can obtain a credential in the same way as described above for the pre-authorized code flow.

2.2 OpenID for Verifiable Presentations

OpenID for Verifiable Presentations [19] is a protocol on top of OAuth 2.0 [9] to present verifiable credentials. The protocol can be used together with OpenID Connect or SIOPv2 and can also send an access token along with the presentation. All parameters from OAuth 2.0 and its extension mechanisms can be used, but the implementer should consider the OAuth 2.0 Security Best Current Practice [14] document. This means that any OAuth 2.0 grant type and response mode can be used to send the presentation to the verifier. In addition, this specification defines a new response mode `direct_post` that allows the wallet to send the authentication response directly to the verifier in a POST request. To transport the presentation, OID4VP defines a new parameter in the authentication response called *vp_token*. A *vp_token* can contain one or more tokens of the same or different formats. Furthermore, the specification extends the OAuth 2.0 metadata with additional parameters to indicate, for example, which cryptographic algorithms are supported in the *vp_token*.

The following subsections describe the three most common OID4VP flows for presenting a presentation. The first is the same device flow with the response mode `fragment`, the second is the same device flow with the response mode `direct_post`, and the third is the cross device flow with the response mode `direct_post`. Same device means that the user started the flow on the device where their wallet is installed. Cross device means that the device where the user started the flow is not the device where their wallet is installed, e.g. a user logs in on a laptop but their wallet is installed on a smartphone. For the sake of brevity, the figures in the following subsections show only the most important parameters in the flows.

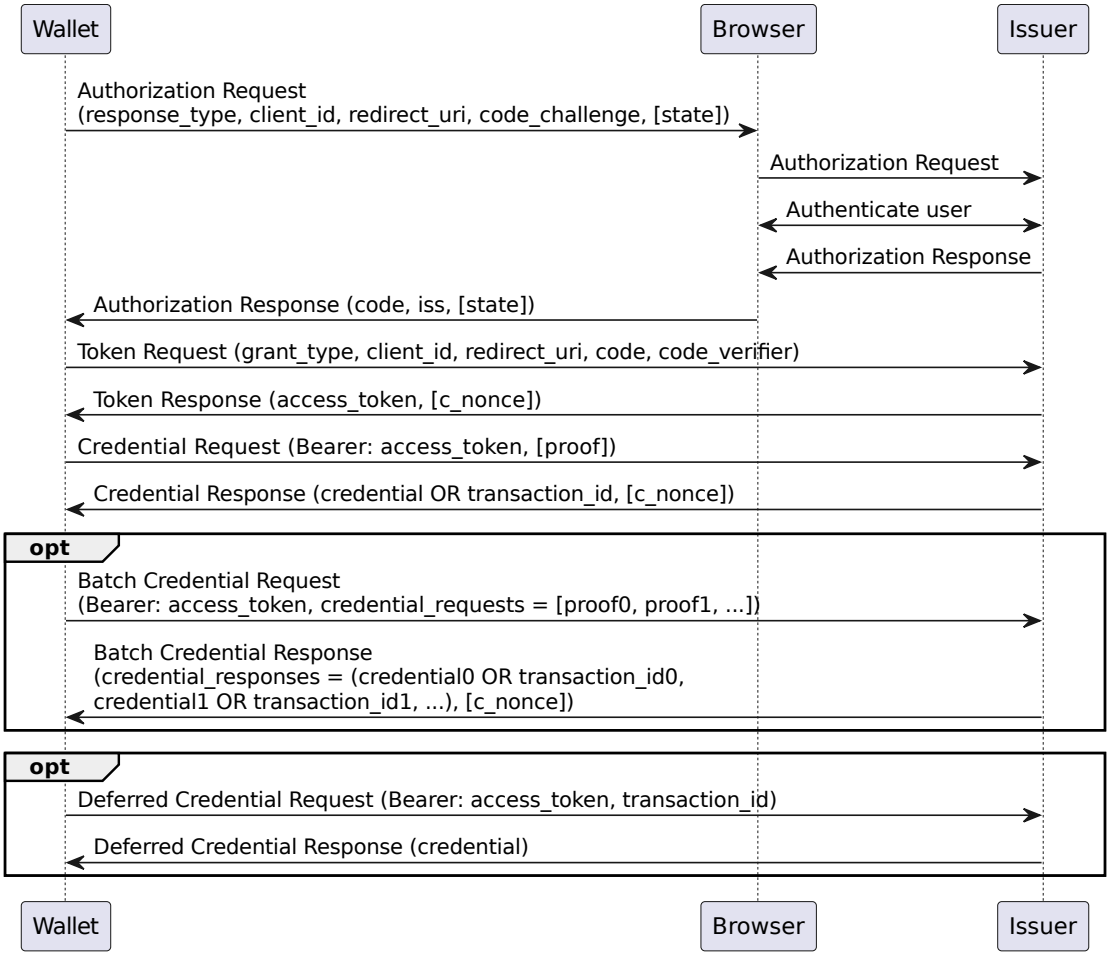

Figure 2.3: Authorization code flow.

2.2.1 Same Device Flow response_mode=fragment

This configuration is the simplest way to get a presentation. The flow starts with an authorization request that typically contains the parameters response type, client ID, redierect URI, presentation definition, nonce, and optionally a state. This is not a complete list of parameters and options that can be included in the authorization request. See the OID4VP specification [19] for more information. The client ID is either the redirect URI if the verifier is not registered with the wallet, or behaves as specified in OAuth 2.0 if the verifier is registered with the wallet. There are other options controlled by the client id scheme parameter that can be found in the specification. The presentation definition expresses what types of credentials the verifier wants to obtain, and the nonce parameter should be embedded in the presentation to bind the presentation to the transaction. The authorization response is sent to the verifier in the redirect URI fragment with the *vp_token*, the presentation submission, the iss parameter, and optionally the state. The state is only included in the response if it is present in the authorization request. The presentation submission contains information about the credentials in the *vp_token*. The iss parameter contains a wallet identifier to prevent mix-up attacks as recommended in the OAuth 2.0 Security Best Current Practice [14]

document. Note that the iss parameter is not required in the OID4VP flows analyzed in this thesis, as shown in the Presentation Authentication security proof (Appendix C.2). This flow can be seen in Figure 2.4.

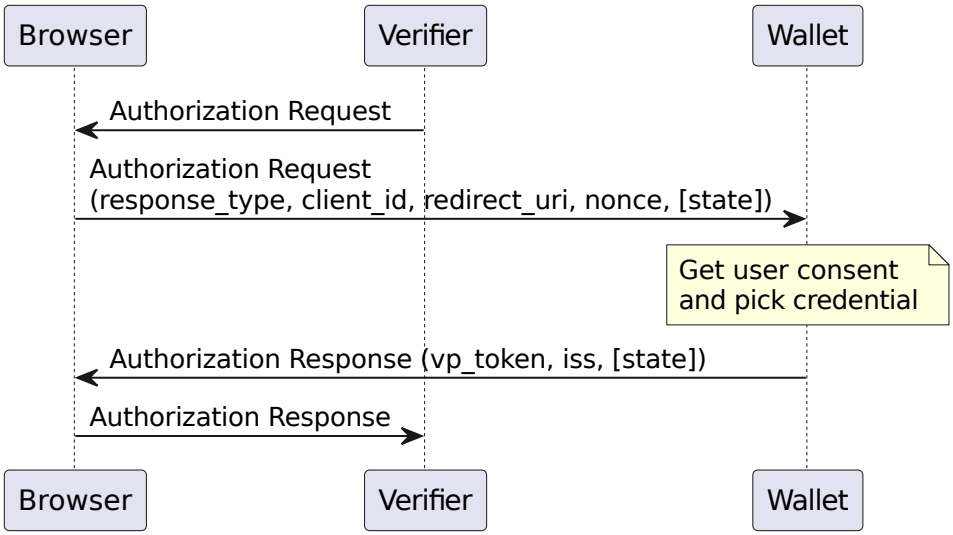

Figure 2.4: Same device flow using the default response_mode=fragment.

2.2.2 Same Device Flow response_mode=direct_post

In this flow, the wallet sends the authorization response to the verifier's direct-post endpoint via an HTTPS POST request. This has the advantage that the wallet and the verifier do not need to be on the same device and that authorization responses that exceed the URL length limit can be transmitted. The authorization request has the same parameters as described above, except that there is the additional parameter response mode and the redirect URI is replaced with the response URI. The response URI contains the verifier's POST endpoint to which the authorization response must be sent. The authorization response is also the same as for the response mode fragment, except that it is sent in the body of the POST request. In the response to the HTTTPS POST request, there is a redirect URI parameter that instructs the wallet to send the user agent to this URI. The redirect URI must contain a secure nonce, called a response code in this work, to ensure that only the owner of the redirect URI can retrieve the authorization response. After redirecting the user, the flow is complete. Figure 2.5 shows this flow with response_mode=direct_post.

2.2.3 Cross Device Flow

This flow assumes that the verifier and the wallet are on different devices. For example, a QR code can be used to transmit the authorization request from one device to another. The authorization request has the same parameter as in the same device flow with response_mode=direct_post, and the authorization response is also sent via an HTTPS POST request. The difference is that the response to the POST request does not include a redirect URI. After the POST request is sent, the flow is complete. Figure 2.6 shows the cross device flow.

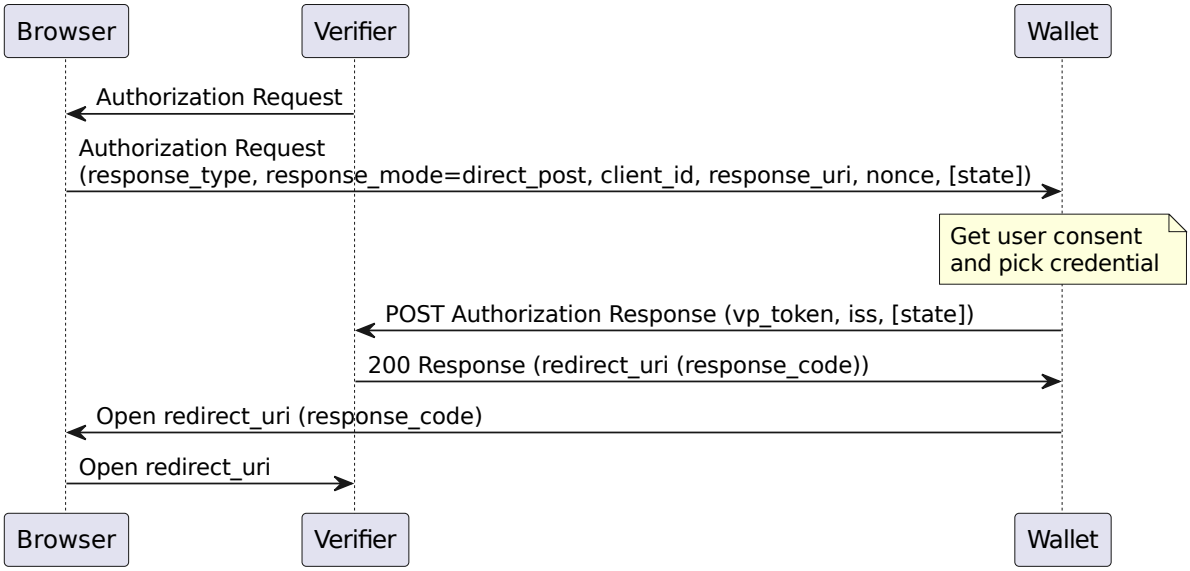

Figure 2.5: Same device flow using response_mode=direct_post.

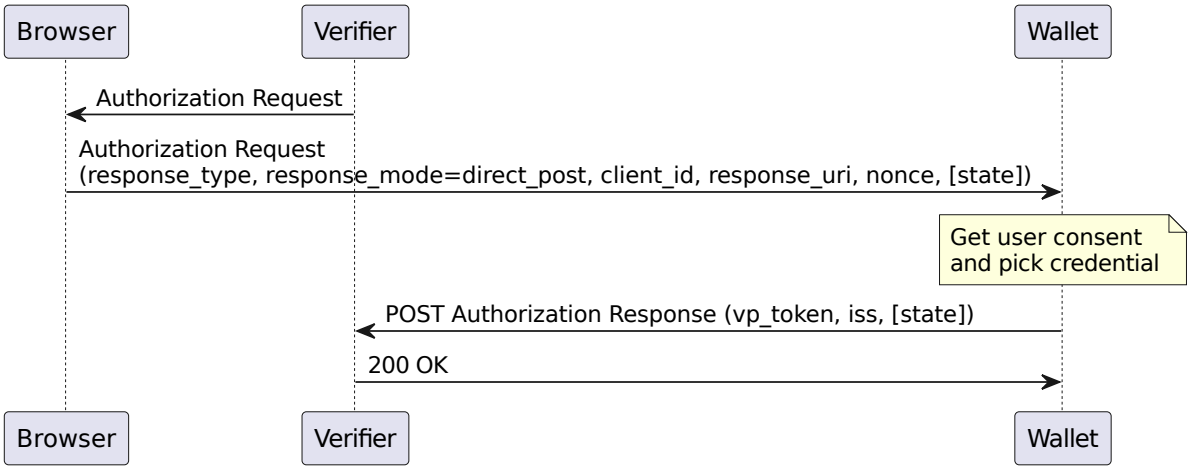

Figure 2.6: Cross device flow.

3 Formal Model

This chapter introduces the formal model of the OpenID for Verifiable Credentials protocols. The model uses the WIM which is first introduced in [6] and briefly described in Section 3.1. This chapter does not give a comprehensive explanation of how the WIM works, but rather focuses on the OpenID for Verifiable Credentials model. This work is based on the current version of the WIM, which can be found in [7] for detailed information. Section 3.2 describes the processes involved in the OID4VC model and the web browser extension needed for the security proof.

3.1 Web Infrastructure Model

This section describes the basics of the Web Infrastructure Model and is based on the description of the WIM in [5]. A detailed formal description of the WIM can be found in [7]. The WIM is a Dolev-Yao (DY) style model that closely models existing web standards such as HTTP and HTML.

The central building block of the WIM is the communication model. Each entity in the model is a process that listens to one or more IP addresses and consumes events. An event has a message as well as a sender and a receiver IP address. At each processing step of the model, an event is non-deterministically selected from the "pool" of events and delivered to the appropriate process. The entity processes the event and outputs one or more events that are added to the "pool" of events.

A message is a formal term over a signature Σ. The signature contains constants (such as strings and nonces) and sequence, projection, and function symbols. Examples of functions are methods for encrypting and decrypting messages or signing terms. The signature Σ can be used to show what an example message would look like for an HTTP request to "http://verifier.ex.com/response?response_code=1234":

$$r := \langle \texttt{HTTPReq}, n_0, \texttt{GET}, \texttt{verifier.ex.com}, /\texttt{response}, \langle\langle \textit{response\_code}, \texttt{1234}\rangle\rangle, \langle\rangle, \langle\rangle\rangle$$

The last two arguments are empty because the request does not contain a body or a list of headers.

The equational theory defines a congruence relation $\equiv$ that expresses the relationship between function symbols in Σ. An example is the encryption and decryption with the key n_1 of the term r with a symmetric cipher:

$$\mathsf{dec_s}(\mathsf{enc_s}(r, n_1), n_1) \equiv r$$

A process has a set of IP addresses, a set of states containing terms, and a relation. The relation uses an input event and the state of the process to non-deterministically compute a set of output events.

The WIM defines a special process for the attacker. The attacker process collects all received events and outputs all events that can be derived from the collected events. Note that there is a strict definition of what is derivable, e.g. the attacker cannot decrypt a symmetric cipher without knowing the key. There are two types of attackers: First, there is the web attacker who can only listen to their IP address and can only send messages from their IP, and second, there is the network attacker who can listen to all IP addresses and spoof messages with an arbitrary source IP address. Furthermore, the attacker can corrupt any honest process and thereby learn the contents of the corrupted party's state.

The web browser is another predefined process of the WIM. A web browser models a user by non-deterministically performing user actions such as following a link or entering credentials into a form. The browser stores user credentials in its initial state per domain and only passes them to scripts running under that domain. This means that credentials are not just sent to a malicious site, as could happen in the real world in a phishing attack. The browser also models the behavior of windows, documents in windows, cookies, and web storage data.

The browser can execute scripts, which are similar to DY processes and model JavaScript code. A script receives a state from the web browser and returns a new state and a command to be executed by the browser. This command can be, for example, sending a POST request to a server or following a link.

The web browser is part of a web system that formalizes the web infrastructure and web applications. In addition to web browsers, it typically includes honest web servers, honest DNS servers, and a web or network attacker. To simplify the modeling of a web server, the WIM defines a generic HTTPS server that already handles, among other things, the receiving and sending of HTTPS requests.

3.2 OpenID for Verifiable Credentials Model

This section provides an informal introduction to the OpenID for Verifiable Credentials web system. The formal definition can be found in Appendix A. This model provides the basis for the reasoning in the proof that follows. The web system contains a set of issuers, wallets, verifiers, web browsers (users), and a network attacker. These entities are modeled as DY processes, which are described in the following subsections.

The WIM is designed to model web-based applications, but in this work we also wanted to model native applications running on mobile operating systems. This model treats native applications as an HTTPS server and a browser running on the same device. Since there is no definition of custom schemes in the WIM, this behavior is approximated by leaking the URL to an arbitrary IP address. This is a stronger assumption than custom schemes, because custom schemes only leak the URL if the user selects a malicious application. When the wallet is a native application, the model uses the approximation of custom schemes for the credential offer and authorization request in the OID4VP flow. The authorization responses in both OID4VCI and OID4VP are defined as app links that are verified by the operating system and cannot leak to a malicious application. We also assume that native applications can safely open URLs in a browser on the device without leaking the URLs to the attacker. Using these assumptions, a native application can be modeled on top of the generic HTTPS server defined in the WIM.

Note that anything related to metadata, error handling, or credential types is not part of such a formal model.

3.2.1 Issuer

An issuer is a Dolev-Yao process based on the generic HTTPS server described in [7] and formally defined in Appendix A.2. It implements the PROCESS_HTTPS_REQUEST handler to process incoming HTTPS requests, and there is also a script for user authentication. The issuer is modeled as a web application that supports the pre-authorization code flow and the authorization code flow. Furthermore, there is no trust relationship between issuers and wallets, which means there is no client authentication. This is omitted because in the model it does not make sense for the issuer to check that the wallet complies with certain regulations and protects the private keys because a process is either honest or corrupt.

The issuer model supports pre-authorized code flow with and without the user PIN option. Starting an authorization code flow with the credential offer is not supported. From a security analysis perspective, this would be the same as simply starting the authorization code flow as long as there is no *issuer_state* parameter in the credential offer. The credential offer uses a custom scheme when the wallet is a native application, which means that the credential offer can be leaked to the attacker in the model as mentioned above. We do not distinguish between a native and a web wallet because leaking the credential offer is a worst-case scenario, meaning that a stronger security assertion can be made by always leaking the URL.

The other supported flow is the authorization code flow, which is always used without PAR. In the authorization code flow, there is an option for the issuer to dynamically request credentials from the wallet during user authentication (Section 5.1.5 of [16]), which is not included in the model.

The pre-authorized code or authorization code can be exchanged at the token endpoint for an access token and a *c_nonce*, but there are no refresh tokens. Note that the *c_nonce* at the token endpoint is not optional in the model and is always returned. Refresh tokens are not included because they are rarely used in real-world deployments, and access tokens never expire in the model. For more details on refresh tokens in the WIM, see [4].

The access token can be used to issue the credential at the credential endpoint or the batch credential endpoint. These endpoints return either credentials or transaction IDs, but not a *c_nonce* because a *c_nonce* is only important if the issuer wants to update it. This model does not include it, because if security can be proven without it, then it should be secure with a fresh nonce. The issued credentials always have a cryptographic holder binding, otherwise the attacker could obtain a credential with a malicious verifier and use the credential to authenticate to an honest verifier. On a more technical level, cryptographic holder binding means that the public key of the private holder key is included in the credential.

The last endpoint implemented by the issuer is the deferred credential endpoint, which is modeled as described in the specification.

The OID4VCI specification has no restrictions on the allowed grant types, but we cannot include all of them in the formal analysis because it would be too complex. Therefore, the model focuses on the most important and commonly used flows in practical applications to make the proof feasible.

3.2.2 Wallet

A wallet is a Dolev-Yao process based on the generic HTTPS server described in [7] and formally defined in Appendix A.3. It implements the PROCESS_HTTPS_REQUEST handler to process incoming HTTPS requests, the PROCESS_HTTPS_RESPONSE handler to process HTTPS responses to requests sent by the wallet, and the PROCESS_TRIGGER handler to send requests to the deferred credential endpoint. The wallet also has two scripts: The first to initiate an OID4VCI authorization code flow with the issuer, and the second to authenticate the user to the wallet. The model includes an OID4VCI client and an OID4VP IdP. To implement cryptographic holder binding, the wallet has one asymmetric holder key pair for all issued credentials. The wallet is modeled as both a web service and a native application running on a mobile device. Each wallet belongs to exactly one browser (user) who knows the credentials to the wallet and can use it. A browser can have multiple wallets. The user PIN in the OID4VCI flow that the user enters into a wallet is only used in honest wallets and never in a malicious wallet. The user of the wallet is a perfect user, which means that they are always aware of the flows they have started. Read more about this in Section 3.2.4 below.

3.2.3 Verifier

A verifier is a Dolev-Yao process based on the generic HTTPS server described in [7] and formally defined in Appendix A.4. It implements the PROCESS_HTTPS_REQUEST handler to process incoming HTTPS requests and two scripts: One to start the OID4VP flow and another to extract the fragment from an authorization response. The verifier can be either a web application running in a browser or a native application running on a mobile operating system. There is no trust relationship between a verifier and a wallet, so there is no client registration.

Three flows are implemented: First, the `response_mode` = `direct_post` with `response_code`, second `response_mode` = `direct_post` without `response_code`, and third `response_mode` = `fragment`. The `response_code` is included as a GET parameter in the redirect URI returned in the direct-post endpoint. This is an important detail because the specification does not specify an exact way to include the `response_code` in the URI. Also, the `response_code` is bound to a browser session, as recommended in Section 11.5. of [19]. The authorization request uses a custom scheme when the wallet is a native application, which means that the authorization request does leak to the attacker in the model.

3.2.4 The Perfect User

As mentioned in the previous Section 3.2.2, the user must always keep track of which flows they have started. This is important in the cross device flow because otherwise an attacker could send an authorization request to an honest user and have them authenticate the request with their wallet. This would allow the attacker to log in as the honest user. Since there are no practical and provable solutions to this problem, as analyzed in previous works [1] and [13], we defined that the user pays attention. To model this behavior, the browser model is extended to store in its state all started flows identified by a nonce and a domain. Subsequently, the method validateRequest($\langle domain, nonce \rangle$)

can be used by the wallet to check if the browser controlling the wallet has a corresponding state or not. To make this work, a new browser script command called `START` has been defined. The formal definition of the browser extension to model a perfect user can be found in Appendix A.5.

This browser extension is also used in the pre-authorized code flow because it is not possible to ensure that the credential offer is created by the honest user. This is important because otherwise, an attacker could inject a credential offer bound to their identity into an honest wallet.

Using this model of the perfect user effectively excludes the attacks described above, but is very close to a user who is careful when using their wallet. This is a minimal mitigation for these attacks, which should not exclude other types of attacks.

In addition, the perfect user checks the redirect URI in the OID4VCI authorization code flow so that no credentials are issued to a malicious wallet, and always uses the user PIN in the pre-authorized code flow in an honest wallet so that the PIN is not leaked to the attacker.

4 Security Properties

This chapter gives a high-level overview of the security properties we have defined for the OpenID for Verifiable Credentials web system model. The formal definition of these security properties can be found in Appendix B. Security properties are needed to define what security means in the context of the model. The intuitive security property we want to achieve is that the attacker cannot log in as an honest user to an honest verifier (Figure 4.1). This is called the authentication security property in the OID4VP protocol.

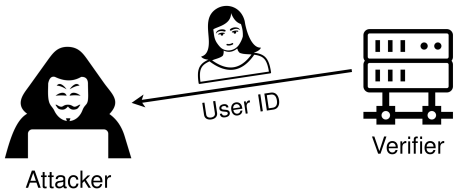

Figure 4.1: Violation of the authentication security property.

It is also desirable to have a security property called session integrity. Informally, session integrity means two things: First, the user has explicitly expressed the wish to log in, and second, after the login flow is complete, the user is logged in under their identity. This means that it should not be possible for an attacker to initiate the login flow and login the honest user under their identity (Figure 4.2). A typical example of such an attack would be a CSRF attack where the attacker logs in the user under their identity. Note that by proving the session integrity security property, we exclude all types of these attacks, not just CSRF attacks. The following sections describe the security properties used in this thesis.

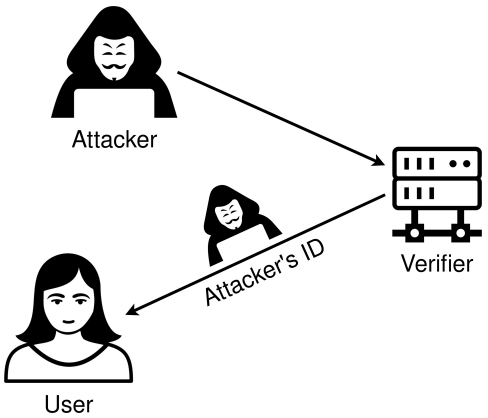

Figure 4.2: Violation of the session integrity security property.

4.1 Presentation Authentication

The informal definition of the Presentation Authentication security property is that an attacker cannot log in under the identity of an honest user to an honest verifier. There are, of course, certain prerequisites for this property to hold: First, the browser, the user's wallets, and the credential issuer must be honest, and second, the user must reject the authorization request in the cross device flow in their wallet that they did not initiate. The attacker has broken the Presentation Authentication security property if they are in possession of a session cookie associated with an honest user ID. The formal definition of this property can be found in Appendix B.1.

4.2 Issuance Authentication

Informally, the Issuance Authentication security property guarantees that an attacker cannot "use" an honest user's credential. Being able to "use" a credential in this context means that the attacker has control of the holder key, because each credential is cryptographically bound to the holder's private key. In order to fulfill this security property, certain preconditions must be met: First, the browser, the user's wallets, and the credential issuer must be honest; second, the pre-authorized code flow is always used with a user PIN; and third, the authorization code is only issued to an honest wallet. The formal definition of this property can be found in Appendix B.2.

4.3 Presentation Session Integrity

At a high level, the Presentation Session Integrity security property means two things: First, the user has explicitly chosen to log in to an honest verifier, and second, the user ends up logging in under the identity they chose in their wallet. On a more technical level, this means that the user's browser does not have a session cookie associated with an attacker's identity. To ensure this, certain prerequisites must be met. One requirement is that the browser, wallet, and credential issuer are honest. Another precondition is that the authorization request does not leak, because then the attacker could inject their presentation into the flow. The formal definition of this property can be found in Appendix B.3.

4.4 Issuance Session Integrity

The informal definition of the Issuance Session Integrity security property is that first, the user has explicitly expressed a desire to issue a credential, and second, the identity in the credential is the identity used to authenticate to the issuer. This means that as long as certain conditions are met, the user cannot load a credential with the attacker's identity into their wallet against their will. The conditions are that the issuer, browser, and wallet involved in the issuance process are honest. The formal definition of this property can be found in Appendix B.4.

5 Security Proof

The formal security proof is the central part of this thesis. It uses the formal model of the OID4VC protocols from Chapter 3 to prove that the security properties from Chapter 4 cannot be violated by the attacker. This proof provides strong security guarantees for the analyzed OID4VC protocols. In this chapter, the high-level ideas of the formal proofs are given and presented in a tree structure. The formal version of the security proofs can be found in Appendix C.

5.1 Proof of Presentation Authentication

To prove the Presentation Authentication security property, it must be shown that the attacker cannot obtain a session cookie associated with the identity of an honest user. The only place such a cookie is created is at the endpoint where the verifier receives the authorization response (redirect endpoint). Since no honest party leaks such a session cookie, the malicious actor must successfully call the redirect endpoint. The redirect endpoint can be called with either a redirect URI and response code, an authorization request with a *vp_token*, or without any parameters.

To call the redirect endpoint with a redirect URI and response code, the attacker needs to know the response code and a login session ID associated with the response code and a *vp_token*. Neither of these are leaked by an honest party, so the malicious actor must create a login session ID for themselves and send a *vp_token* to the direct-post endpoint. As we will see later, the attacker cannot obtain a *vp_token* with the identity of an honest user, so the attacker must send the authorization request in a phishing attack to an honest user. There are two cases, because the attacker can either modify the response URI in the authorization request or not modify it. In the first case, the user sends the *vp_token* to the attacker, but includes the attacker as the audience in the *vp_token*, making it unusable for the attacker to log in to an honest verifier. In the second case, the user sends the *vp_token* to the honest verifier who does not have a session with them.

The attacker also cannot obtain a *vp_token* to send to the redirect endpoint because an honest party does not leak a *vp_token*, the malicious party cannot "use" an honest user's credential (**Issuance Authentication security property**), and the attacker cannot make an honest wallet create a *vp_token*. Forcing an honest wallet to create a *vp_token* does not work as described in the previous paragraph because either the audience value is wrong or the malicious actor cannot obtain the *vp_token*.

The third and final way to use the redirect endpoint is for the attacker to know a login session ID associated with an honest user's *vp_token*. Since a login session ID is not leaked by an honest party, the attacker must trick a user into sending their *vp_token* to the direct-post endpoint, which contains the nonce of the attacker's authorization request. This cannot happen because the perfect user recognizes that the authorization request is not initiated by them and rejects the authentication in their wallet.

In summary, the attacker cannot obtain a session cookie associated with the identity of an honest user. A graphical representation of this proof can be seen in Figure 5.1 and Appendix C.2 contains the formal version of this proof.

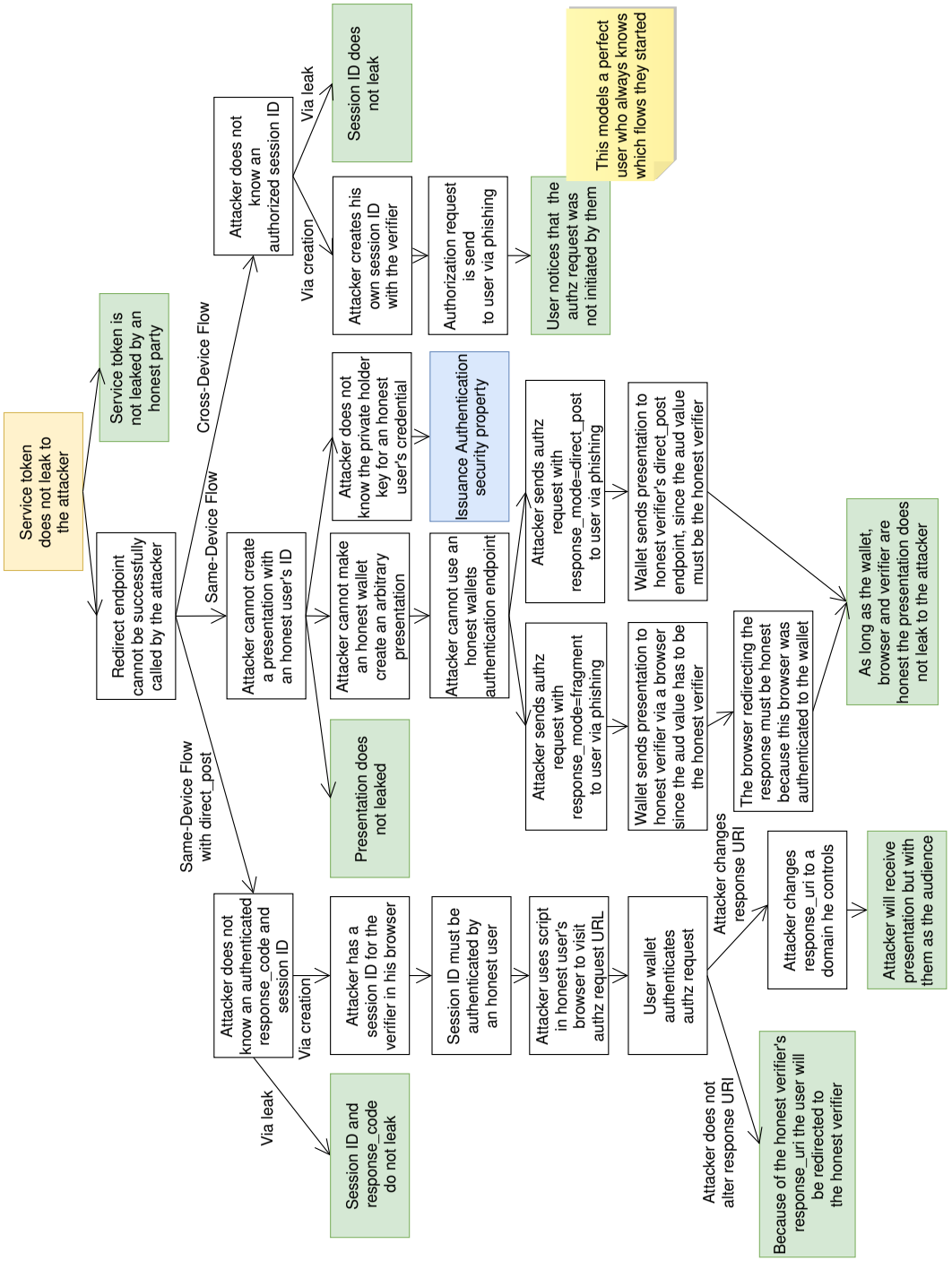

Figure 5.1: Proof structure of the Presentation Authentication security property.

5.2 Proof of Issuance Authentication

Showing that the Issuance Authentication security property holds requires showing that the private key associated with a credential is not known to the attacker. There are two ways that the malicious actor could know the private key: First, an honest wallet leaks the private key, and second, the attacker can inject their own key during credential issuance. Since an honest wallet does not leak the private holder key, the attacker must inject their credential during credential issuance.

The only places where the cryptographic holder binding takes place are in the credential endpoint and the batch credential endpoint. Calling either of these endpoints requires an access token and a *c_nonce* for the holder binding. Since the access token and *c_nonce* are not leaked by an honest party, they must be obtained at the token endpoint. To invoke the token endpoint, either an authorization code and code verifier or a pre-authorized code and user PIN is required.

Since the authorization code and code verifier are not leaked, the attacker must trick a user into authenticating an authorization request. This is not possible because the perfect user rejects authorization requests that they did not initiate, as specified in the security property.

The other way to call the token endpoint is to present a pre-authorized code and a user PIN. The pre-authorized code leaks, but the user PIN does not and is only used in an honest wallet. This means that the attacker cannot obtain an access token from the token endpoint.

In summary, the wallet does not leak its private key, and the attacker cannot inject their key during credential issuance. A graphical representation of this proof can be seen in Figure 5.2 and Appendix C.3 contains the formal version of this proof.

5.3 Proof of Presentation Session Integrity

Proving the Presentation Session Integrity security property requires to show two things: First, the user started the flow. Second, the identity the user chose in their wallet is the one under which they are logged in at the end of the flow. At the end of the login flow the user gets a session cookie that is associated with an identity.

First, we want to prove that this user also started the login flow. The only way to obtain a session ID is by successfully calling the redirect endpoint of the verifier. This is only possible if the request contains a valid login session cookie. This cookie must be stored in the browser under the domain of the verifier. Since the cookie has the `__Host` prefix, it could have been only set by a secure response from the verifier. Such a cookie is only created in the verifier's endpoint that created the authorization request. Considering that this endpoint makes an origin check only a script of the verifier can call this endpoint. This means that the browser (user) must have been explicitly executed a script of the verifier to start the login flow.

To show that the user's wallet selected the credential we track where the *vp_token* came from. It is clear that the verifier does not create *vp_token*s but receives them through different channels. To create a valid *vp_token* one must know the nonce from the authorization request. From the security properties preconditions it is known that the authorization request does not leak. Since the nonce does also not leak through other parties the nonce is only known to the honest verifier, browser, and the user's wallet. Considering that only the wallet creates *vp_token*s, we can conclude that the

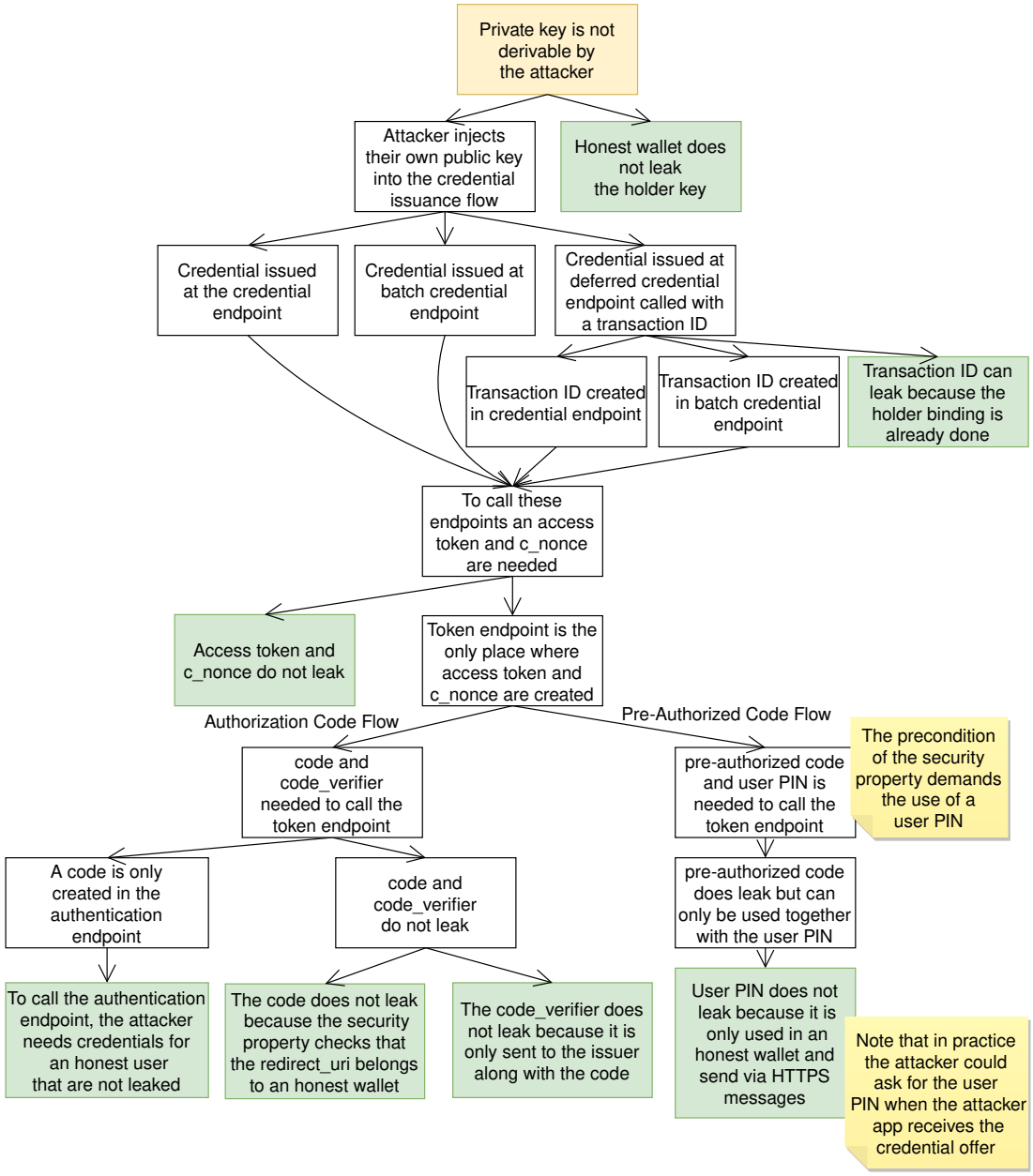

Figure 5.2: Proof structure of the Issuance Authentication security property.

wallet selected one of its credentials and created the *vp_token*. From the **Issuance Session Integrity security property** it is known that the wallet contains only credentials that contain identities of the honest user. This means that the user selected one of their credentials.

In summary, the user started the flow and selected the credential that is used to login. This means that the Presentation Session Integrity security property holds. A graphical representation of this proof can be seen in Figure 5.3 and Appendix C.4 contains the formal version of this proof.

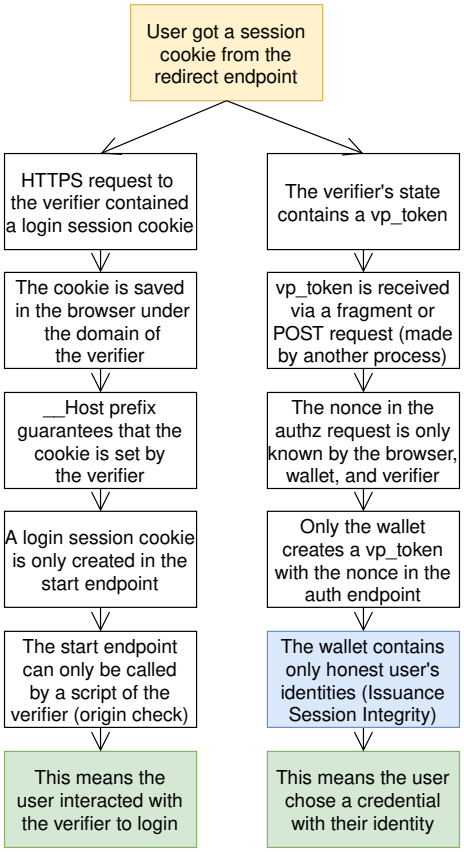

Figure 5.3: Proof structure of the Presentation Session Integrity security property.

5.4 Proof of Issuance Session Integrity

To prove the Issuance Session Integrity security property, we must show that either the user initiated an authorization code flow and selected an identity during user authentication at the issuer, or the user initiated a pre-authorized code flow to issue a credential. The credential is issued at either the deferred credential endpoint, the credential endpoint, or the batch credential endpoint.

If the credential is issued at the deferred credential endpoint, there must have been a previous request to the credential endpoint or the batch credential endpoint, otherwise the wallet would not have a transaction ID. To use either of these endpoints, the wallet must have previously made a token request to obtain an access token. Making a token request requires either an authorization code and code verifier or a pre-authorized code and user PIN.

If a pre-authorized code is used at the token endpoint, that code is received first at the credential offer endpoint. Since the perfect user would reject credential offers not initiated by them, there is a guarantee that the credential offer is created by the user at the issuer. This includes that the user authenticated to the issuer with a username and password, and that this identity is the one contained in the credential. This is the first part of the security property.

If an authorization code is used at the token endpoint, it must have been received with the authorization response. This request must include a valid session cookie. Since the session cookie is stored in the browser under the wallet's domain and has the `__Host` prefix, it must have been set by a secure response from the wallet. The only endpoint that sets such a cookie is the endpoint that creates an authorization request. Considering that this endpoint checks the origin header, only a script of the wallet can call it. This means that the browser (user) explicitly executed a script of the wallet to start the credential issuance flow.

We also need to show that the user authenticated to the issuer. From the token request, it is known that the wallet and the issuer share a code challenge - code verifier pair. Since this pair is not leaked to the attacker, it can be inferred that the browser called the issuer's authentication endpoint with the authorization request. In this step, the browser also provides the user's identity and password. This means that the user provided the identity for the credential.

In summary, in each case the user initiated the issuance flow and provided the identity contained in the credential. This proves that the model fulfills the Issuance Session Integrity security property. A graphical representation of this proof can be seen in Figure 5.4 and Appendix C.5 contains the formal version of this proof.

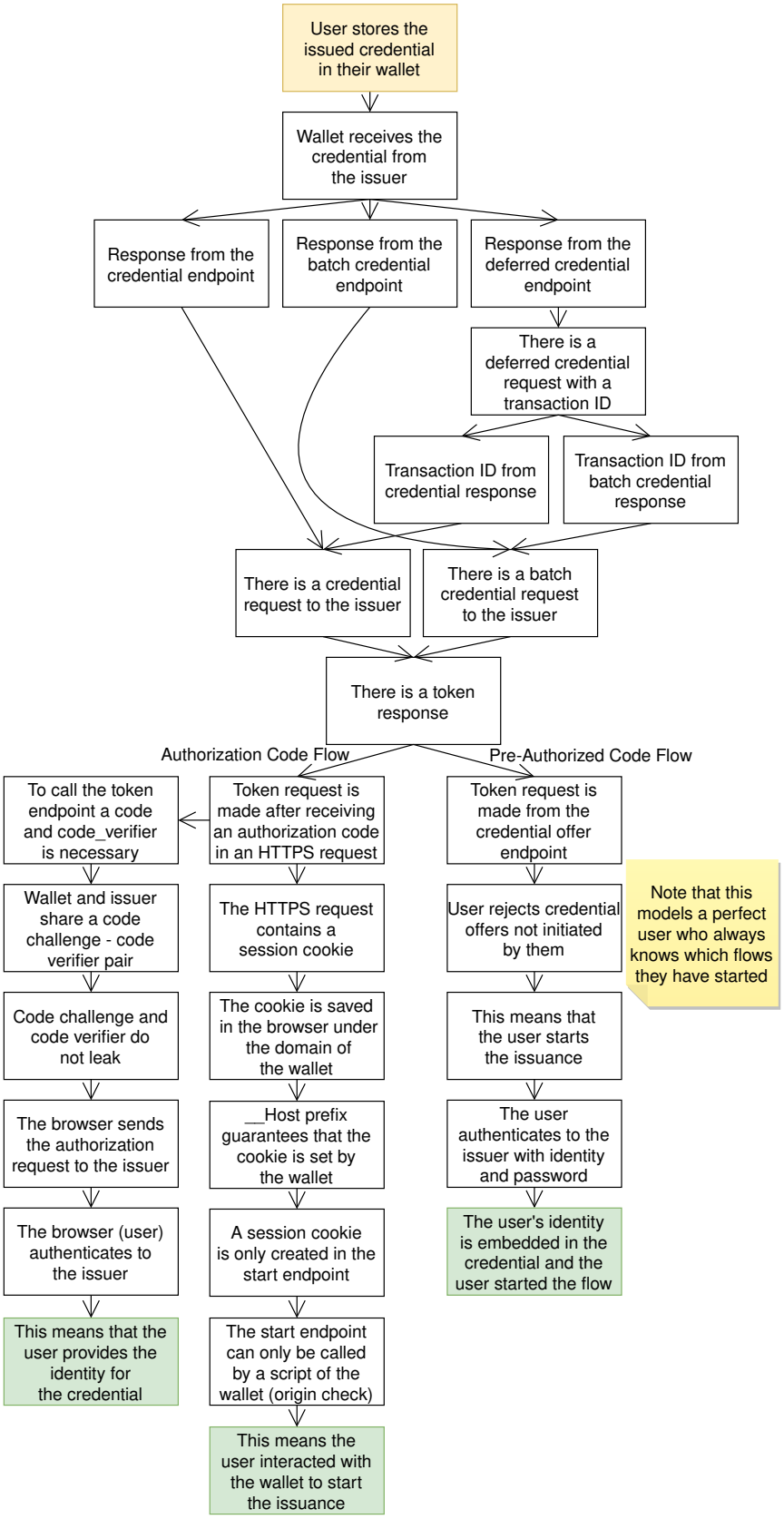

Figure 5.4: Proof structure of the Issuance Session Integrity security property.

6 Discovered Attacks

This chapter contains detailed descriptions of all attacks discovered during the formal security analysis of the protocols. It is important to note that while several attacks were found during the security proof, no other attacks were discovered in the limits of the model, as the formal security proof shows. Table 6.1 gives an overview of the attacks found, sorted by protocols and the security property they violate. Furthermore, to the best of our knowledge, such a detailed formal security analysis has not yet been performed for other three-party model protocols used in practice. This means that similar attacks may also be found in other protocols.

	Verifiable Credential Issuance	Verifiable Presentations
Authentication Security Property	**Code and user PIN phishing** in pre-authorized code flow (even with wallet attestation) (Section 6.1.1)	Attacker can trick user into authorizing an authorization request in the **cross device flow** (Section 6.2.1)
	Malicious app could start an authorization code flow to **obtain a code** (Section 6.1.2)	
Session Integrity Security Property	**Inject pre-authorized code** bound to attacker identity (Section 6.1.3)	If the **authorization request leaks**, the attacker can answer it with their credential (Section 6.2.2)

Table 6.1: Overview over the discovered attacks.

The attacks discovered in OID4VCI and OID4VP are not entirely new or surprising to people familiar with the specifications. For example, the cross device attack has already been described in the context of SIOPv2 [1]. The underlying issue here is the transfer of a session from one party to another, which is also the reason for the discovered pre-authorized code flow vulnerabilities. In addition, the credential offer with an *issuer_state* parameter also transfers a session between entities, making it vulnerable to this type of attack. This class of vulnerabilities is not specific to OAuth 2.0 based protocols and it is likely to be present in other web flows as well, as described in the Cross-Device Flows: Security Best Current Practice document [13]. To date, no practical and provable fixes have been found for this type of attack. To mitigate this class of vulnerabilities, profound changes must be made to browsers and operating systems. One such technology is the Federated Credential Management API[1], currently implemented in the Chrome browser, which may

[1] https://developer.chrome.com/en/docs/privacy-sandbox/fedcm/

solve these problems in the future. Since no mitigation is currently available, we have introduced the perfect user (Section 3.2.4) into the model. This is not optimal because relying on the user for security is not good practice, as the user is often the weakest link.

6.1 OpenID for Verifiable Credential Issuance

The first section of this chapter contains all discovered attacks against the OID4VCI protocol. Three attacks were found there: First, an attack on the pre-authorized code flow that violates the authentication security property, which works by leaking the credential offer and user PIN to a malicious application. Second, an attack on the authorization code flow that also violates the authentication security property, where a malicious application initiates the issuance flow and obtains an authorization code. Lastly, an attack on the pre-authorized code flow that violates the session integrity security property, which works because the sender of the credential offer cannot be verified.

The OID4VCI specification contains a Security Considerations chapter (Chapter 11 in [16]) in which known attacks and mitigation techniques are discussed, but none of the attacks found in this thesis are already described there.

6.1.1 Authentication Attack on Pre-Authorized Code Flow

The pre-authorized code flow is a new flow introduced in the OID4VCI specification. In this flow, the issuance is initiated by the issuer, not by the wallet, and there is no authorization request. So the issuer immediately sends a pre-authorized code to the wallet in a so-called credential offer. The custom scheme `openid-credential-offer://` is a suggested method in the specification to invoke a wallet. Since there are no restrictions on which application can register for such a custom scheme in typical operating systems, it is possible for a malicious application to register for the custom scheme. As a result, the credential offer, including the pre-authorized code, can leak to the attacker if the user chooses the wrong application when presented with a choice by the operating system. This can happen to less technologically-skilled users, of course, but even people who understand the technology could click on the wrong application by accident or because they are in a hurry. In the absence of other security mechanisms, the attacker could directly exchange the pre-authorized code at the token endpoint for an access token to obtain a credential for an honest user.

The OID4VCI specification includes user PINs as a security feature to prevent attacks where the attacker scans the QR code with the credential offer from the user's computer (see Section 11.3.1 of [16]). The user PIN is a secret that is sent out-of-band (e.g., via e-mail) to the user, and the user must enter this secret into the wallet application. This prevents the attack where the attacker scans the QR code because they do not know the user PIN. Even if the attacker does shoulder surfing to find out the user PIN, the user will most likely send the request first, and since the pre-authorized code is one-time-use only, the attacker will not be able to obtain a credential.

The problem is that the user PIN does not prevent the attack described above, where the user chooses a malicious application to process the credential offer. In this case, the attacking application has UI control and can simply ask for the user PIN. If the user thinks that they are using a legitimate wallet application or does not pay attention, they will enter the user PIN. Note that the user PIN

mechanism is different from, for example, a banking PIN. In the banking use case, the user is supposed to use the PIN in software from the same vendor, whereas in the credential issuance flow, the user is using the PIN in a wallet, which is most likely from a different vendor than the issuer. The attacker then enters the credential offer and the user PIN into their own wallet application and obtains a user credential. This works even if wallet attestation is enabled because the attacker can use an official wallet application to communicate with the credential issuer. This attack can be seen in Figure 6.1.

The underlying problem here is that a session is being transferred from one party to another. This means that the attack works not only on a credential offer with pre-authorized code, but also if the credential offer contains the *issuer_state* parameter. It can be concluded that this is a broad problem that affects other web flows as well, and will require extensive changes to browsers and operating systems in order to be solved.

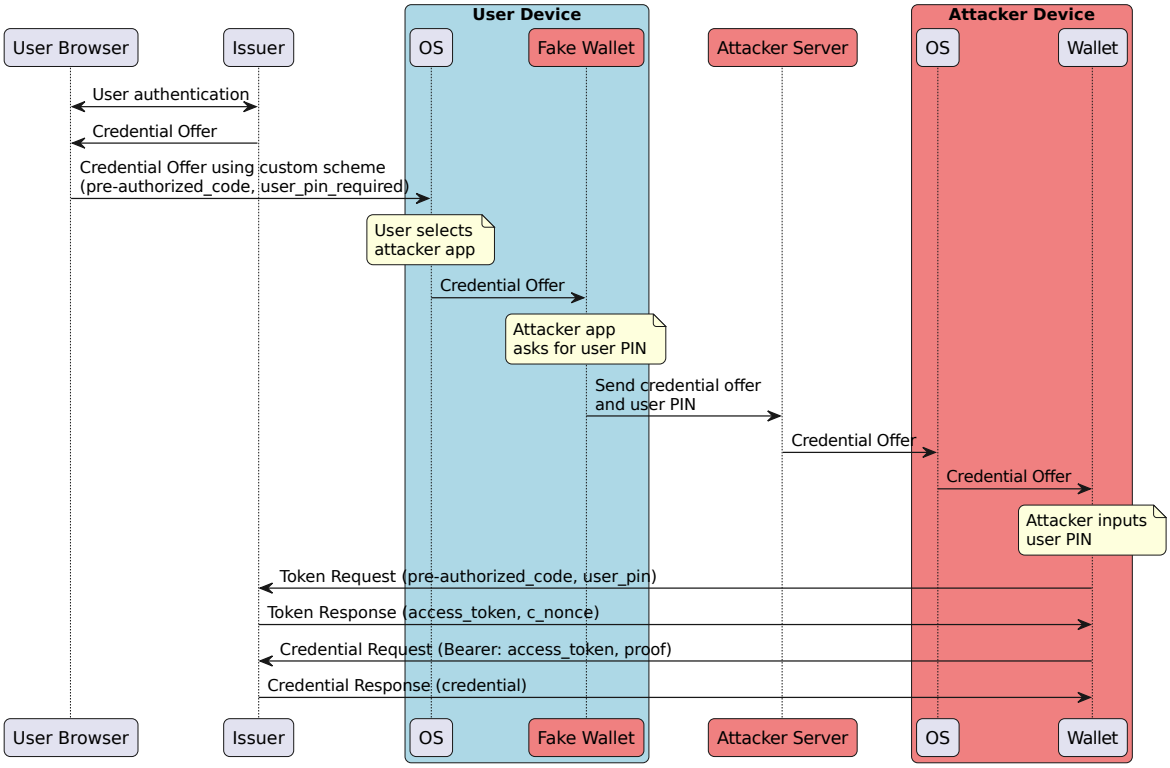

Figure 6.1: Authentication attack against the pre-authorized code flow.

6.1.2 Authentication Attack on Authorization Code Flow

The authentication attack on the authorization code flow assumes that an attacker application is installed on the user's device and that the application can convince the user to log in to the issuer. For example, the malicious application could be a game that requires the user to log in with a credential before playing. Instead of redirecting the user to their wallet application, the app opens the issuer's website to issue a credential. A user paying attention to the process would probably notice that this is an attack, but many users would probably just follow the flow. After authenticating to the issuer, the user is redirected back to the application with the authorization response. Since

the attacking application knows the client ID, redirect URI, code verifier, and code, it can exchange the code for an access token at the token endpoint. With the access token, the attacker can retrieve a user credential from the credential endpoint that is bound to a private key known to the attacker. This means that the attacker can use this credential wherever they want because they have full control over the credential. Using defenses such as nonce, state, or PKCE [17] does not mitigate this attack because the authorization request is initiated by the malicious application. The attack described here can be seen in Figure 6.2.

The attack can also work when wallet attestation is used. In this case, the attacking application uses an authorization request from an honest wallet on the attacker's device and opens that authorization request on the victim's device. To capture the authorization response, the attacker must register for the domain or custom scheme of the authorization response. This may or may not work depending on whether the honest wallet application is already installed on the victim's device, but if it works, the attacker can inject the authorization response into their own unmodified wallet application. Wallet attestation makes the attack more difficult, but not impossible, so ultimately the user has to be careful which wallet the credential is issued to.

This attack also works on a standard OAuth 2.0 or OpenID Connect flow. The malicious application must interact with the IdP, so the IdP must either allow public clients (the malicious application can register the redirect URI of an existing client in the OS), dynamic client registration, or depending on the ecosystem, there could also be a malicious client registered with the IdP. The difference with the OID4VCI attack is that the attacker would only gain access to a user's resources at a particular service or obtain an ID token. Whereas in the attack described here, the malicious actor obtains a credential that can most likely be used across multiple services, which has more severe consequences.

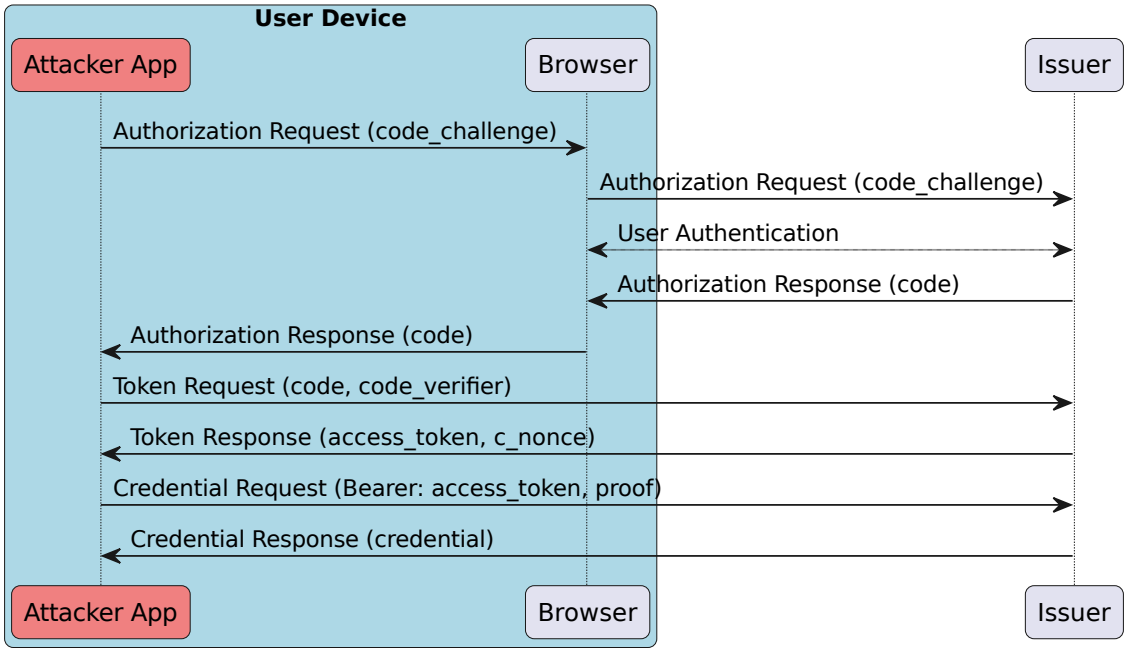

Figure 6.2: Authentication attack against the authorization code flow.

6.1.3 Session Integrity Attack on Pre-Authorized Code Flow

In this attack, the malicious actor uses the credential offer of the pre-authorized code flow to load a credential with the attacker's identity into an honest user's wallet. The underlying problem is the transfer of a session between parties as described above. The attack is carried out as follows. The attacker initiates an issuance flow and authenticates to the issuer. Depending on the issuer's policy, the attacker may also receive a user PIN, e.g. via email. The attacker then embeds the credential offer as a QR code in a web page and sends this web page to a user in a phishing attack. If the issuer requires a user PIN, the PIN is also included on the web page. The user opens the link on their computer and scans the QR code with their wallet. The wallet asks for a user PIN, which is included on the website if required by the issuer. The user then confirms the issuance of the credential and has a credential with the attacker's identity in their wallet.

This can be a problem, for example, if the user uses this credential to authenticate to a cloud storage and uploads sensitive documents, since the attacker also has access to the cloud storage. Note that wallet attestation or a claimed URL for the credential offer would not prevent this attack because the attacker uses a genuine issuer to create the credential offer. This attack can be seen in Figure 6.3.

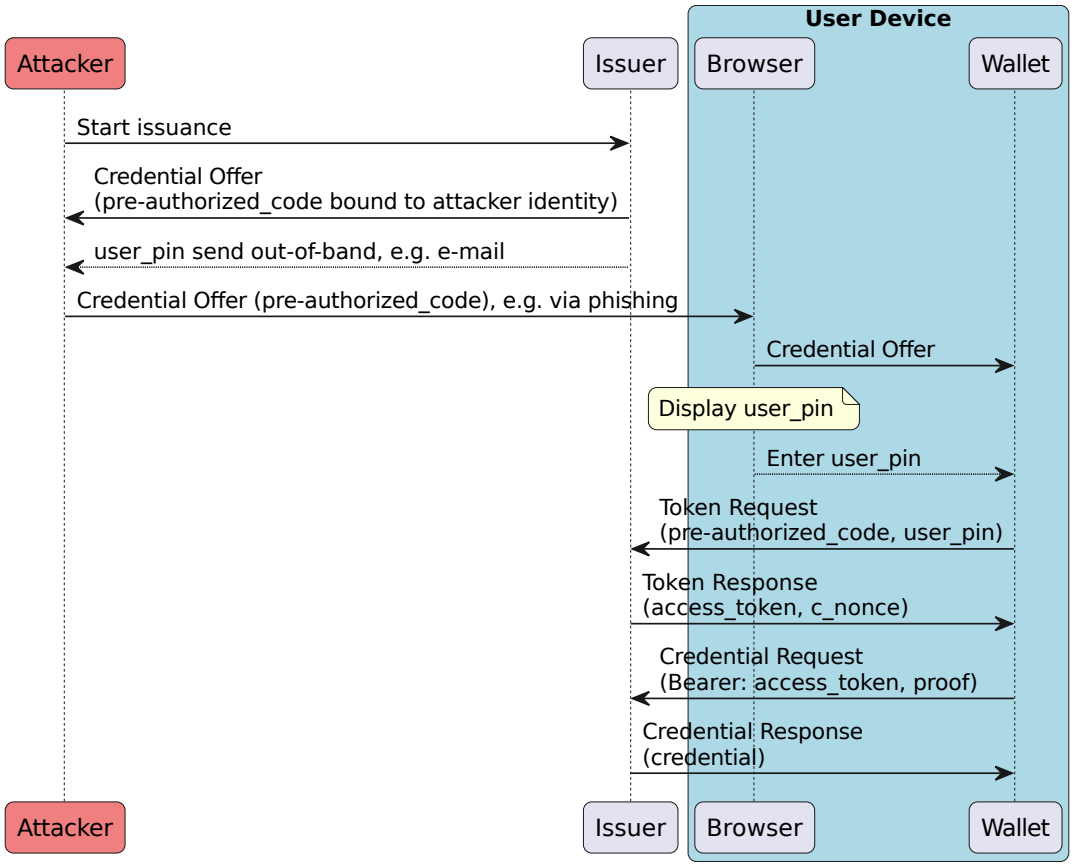

Figure 6.3: Session integrity attack against the pre-authorized code flow.

6.2 OpenID for Verifiable Presentations

This chapter describes all attacks found against the OID4VP protocol. Two attacks have been discovered: First, an attack against the cross device flow that violates the authentication security property, where the attacker sends an authorization request to the victim for authentication. Second, an attack against all analyzed flows that violates the session integrity security property, where an attacker responds to a leaked authorization request. The first attack is described in the Security Considerations chapter of the OID4VP specification (Section 12.2 of [19]), but the second attack is not yet mentioned in the specification.

6.2.1 Authentication Attack on Cross Device Flow

In the cross device flow, the user initiates authentication to a verifier on one device and authenticates the authorization request on another device. This leads to the problem that there is no browser session available at the verifier when the authorization response is received at the direct-post endpoint. Therefore, it is possible for an attacker to initiate authentication at a verifier and send the authorization request in a phishing attack to an honest user. The user authenticates the authorization request in their wallet, which sends a presentation with their credential to the verifier. Since the attacker started the authentication with their browser, it is their browser that is logged in under the honest user's identity. This attack is widely known and already described in Section 12.2 of the OID4VP specification, but there are no practical and provable fixes against it, as described in [13]. The described attack can be seen in Figure 6.4.

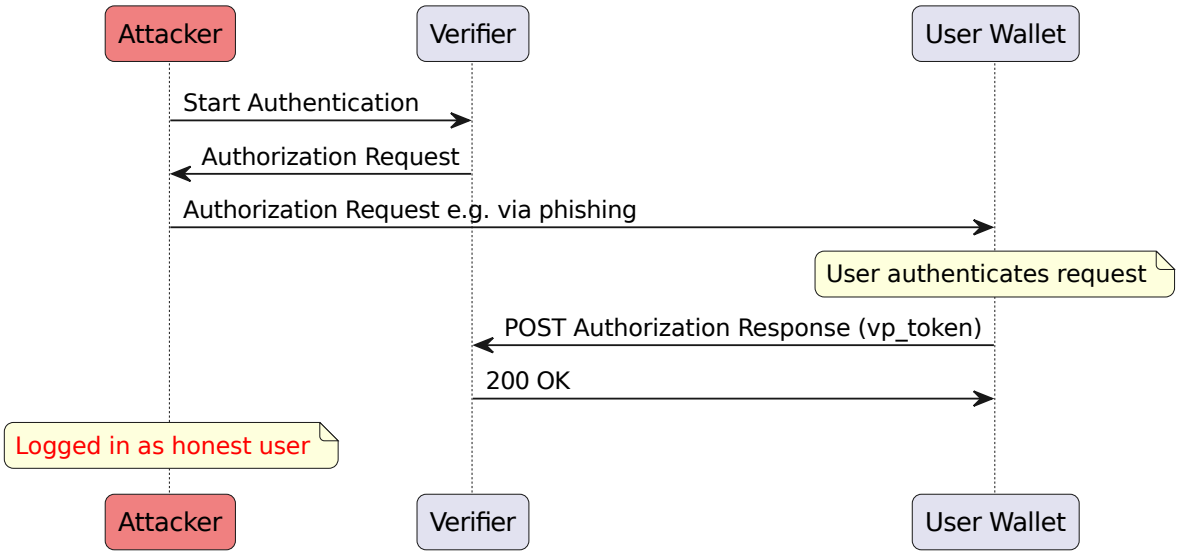

Figure 6.4: Authentication attack against the cross device flow.

6.2.2 Session Integrity Attack

One way to invoke a wallet application in the OID4VP specification is to use the custom scheme `openid4vp://` (Section 7 of [19]). As explained above (Section 6.1.1), there is typically no restriction on which application can register for such a custom scheme, which means that the authorization request can leak to the attacker's application. If this happens, the attacker can use the nonce from the authorization request to generate a presentation with a credential containing the attacker's identity. The malicious application then sends the presentation either to the direct-post endpoint or via the redirect URI fragment to the verifier. If the direct-post endpoint responds with a redirect URI, the application opens the URL in the browser. This means that the attack works with all analyzed response modes of OID4VP. The attack described here can be seen in Figure 6.5.

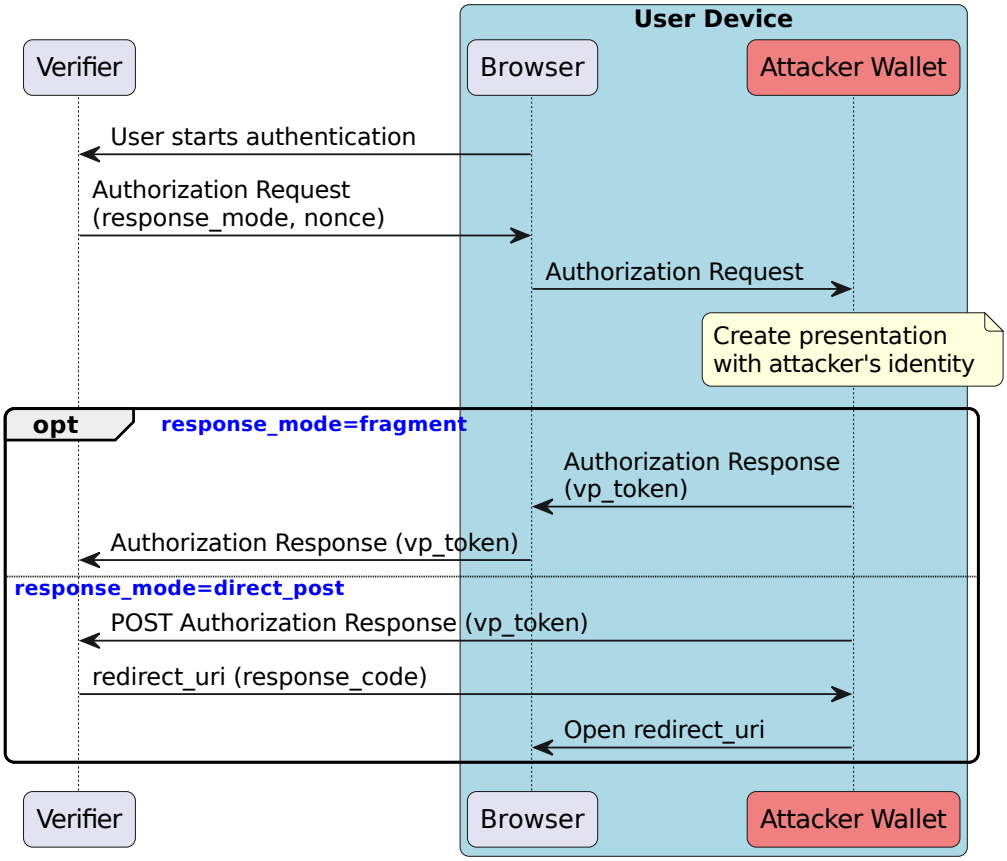

Figure 6.5: Session integrity attack against OID4VP.

7 Contributions to Standards

This chapter provides an overview of this work's contributions to the standards. The two main contributions to the standards were the presentation of results at the OAuth Security Workshop 2023 and the submission of issues. In total, five issues were raised, one of which resulted in a pull request that changed the OID4VCI specification.

The OAuth Security Workshop is one of the most important conferences when it comes to OAuth 2.0 and OpenID Connect security, with attendees from all over the world. The presentation of this work at the OAuth Security Workshop 2023 included a summary of the analyzed protocols, an introduction to the formal analysis technique, and the discovered attacks as described in Chapter 6. The audience included some of the authors of the specifications. The discovered attacks were acknowledged as problems that need to be discussed further in order to find mitigations that can be incorporated into the specifications. In particular, the use of custom schemes was discussed and possible fixes were debated.

7.1 Issues

This section describes the five issues raised during this work on the formal analysis of the OID4VC protocols and indicates the status of each issue at the time of writing. In addition, we have commented on two issues regarding the OID4VP cross device flow and the pre-authorized code flow.

7.1.1 Allowing Arbitrary Grant Types

The OID4VCI specification leaves open which grant types can be used. If the Implicit Grant is used without a claimed redirect URL, the access token could fall into the hands of the attacker because there is no guarantee that the user will be redirected back to the wallet they started with.

The issue was resolved by referring to the OAuth 2.0 Security Best Current Practice [14] document, which states that the Implicit Grant should not be used.

URL: https://bitbucket.org/openid/connect/issues/1939

7.1.2 Is the Acceptance Token One-Time Use?

The OID4VCI specification was unclear as to whether the acceptance token (now transaction ID) was a one-time use or not. The response to this issue is that the acceptance token should be invalidated after the credential is successfully issued, but not if there is an error response at the deferred credential endpoint. Mentioning this is important to avoid security vulnerabilities in implementations. The result of this issue was a pull request to update the specification (https://bitbucket.org/openid/connect/pull-requests/571/).

URL: https://bitbucket.org/openid/connect/issues/1940

7.1.3 Binding Response Code to Session

The OID4VP specification provides the ability to use the direct_post response mode along with a redirect URI containing a cryptographically secure nonce (response code). It is unclear from the specification whether the response code is bound to a browser session or not. It is really important to bind it to a browser session, because otherwise an attacker could violate the Session Integrity security property by starting a login flow and sending the redirect URI with the response code to a victim in a phishing attack. The assumption here is that anyone who opens the redirect URI with the response code is logged in to the verifier because it is not bound to a session. The flow of the attack can be seen in Figure 7.1. This issue was also discussed extensively with some of the authors at the OAuth Security Workshop 2023, and the result was that the response code needs to be bound to a browser session. At the time of writing, this solution is not documented on the issue.

URL: https://github.com/openid/OpenID4VP/issues/27

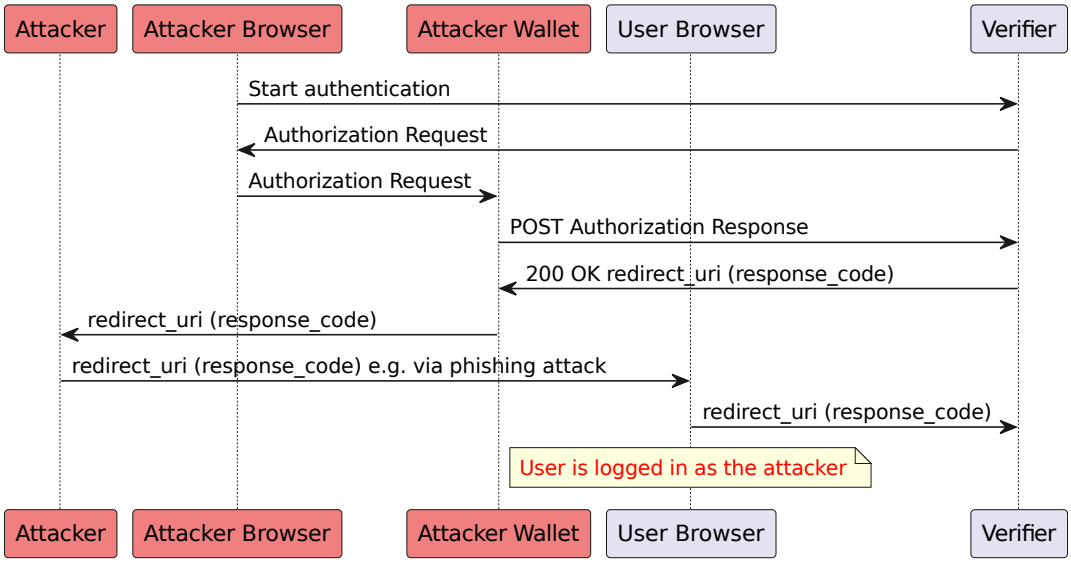

Figure 7.1: Session integrity attack against OID4VP same device flow with response code.

7.1.4 Checks in Section 11.2

Section 11.2 of the OID4VCI specification [16] states that certain checks should be performed upon receipt of a credential offer. There is no further information about the checks in that section, nor is there a reference to another resource. One of the authors responded that checking the credential issuer means obtaining metadata and deciding whether the issuer is trustworthy or not. This is not explicitly stated in the specification and could be improved, but there is no further information on this issue at the time of writing this thesis.

URL: https://github.com/openid/OpenID4VCI/issues/50

7.1.5 Lifetime of an Authorization Code / Pre-Authorized Code

This issue came up during a discussion at the OAuth Security Workshop 2023. Section 6.3 of the OID4VCI specification [16] introduces the "authorization_pending" error response, which means that the wallet must use polling to see when the issuer is ready to issue the credential. This means that the authorization code is long-lived, which is against the recommendation in RFC 6749 [9] section 10.5. The same is true for the pre-authorized code, but the problem is worse there because it can be easily leaked since the credential offer typically uses a custom scheme or the user posts the QR code to their social media thinking the transaction is complete when it is not. To improve the security of the protocol, the lifetime of the authorization code and the pre-authorized code should be as short as possible. At the time of writing, there is an ongoing discussion with no result yet.

URL: https://github.com/openid/OpenID4VCI/issues/60

7.1.6 direct_post response mode, response with a redirect_uri, and protection against session fixation

This issue discusses the use of the redirect URI in the OID4VP cross device flow. We did not create this issue, but commented on it because from a security perspective it does not make sense to have the user enter a response code on their laptop that is displayed in the smartphone's browser. Such a code is easily leaked if the user starts the flow on a malicious website, because the user is supposed to enter the code on a different device. This issue has not been resolved at the time of writing.

URL: https://github.com/openid/OpenID4VP/issues/25

7.1.7 Make pre-authorized code flow optional?

This issue was not created by us, but is based on the results of this work presented at the OAuth Security Workshop 2023. This issue discusses whether the pre-authorized code flow should be used in use cases that require a high level of security, because even with the user PIN there are possible attacks as described in Section 6.1.1. We commented on this issue because there is a comment saying that wallet attestation would prevent this attack, although this is not true. Furthermore, we explained the discussed attack in more detail with a sequence diagram.

URL: https://github.com/vcstuff/oid4vc-haip-sd-jwt-vc/issues/60

8 Summary and Outlook

Within the scope of this thesis, a rigorous formal security analysis was performed on both the OpenID for Verifiable Credential Issuance protocol and the OpenID for Verifiable Presentations protocol. Under the assumption of a vigilant user, we were able to prove the security of both protocols. More formally, we have proven that the authentication security property and the session integrity security property hold in the limits of the formal model. This provides very strong security guarantees for the two most important protocols in the OpenID for Verifiable Credentials protocol family. Since these protocols are part of the European Digital Identity Framework, this work will improve the security of every future wallet in the European Union.

However, the assumption that users are attentive may not always be realistic in practice. Phishing attacks, for example, cause economic damage of several million euro every year in Germany alone.[1] Under the assumption of a non-attentive user, five attacks were discovered during the formal proof in this thesis. Three of them are in the OID4VCI protocol, two of which are violations of the authentication security property and one of which is a violation of the session integrity security property. In the OID4VP protocol we found one violation of the authentication security property and one violation of the session integrity security property. The attacks were disclosed to the working group and possible mitigation strategies were discussed. Discussions took place online in issues and at the OAuth Security Workshop 2023. In the course of this work, five issues were filed and two others were actively discussed to improve the clarity and to strengthen the defenses of the analyzed protocols. Overall, there have been substantial contributions to the specifications that have resulted in lasting improvements in security and quality.

Outlook

The assumption of an attentive user is a strong assumption that often does not hold true in practice. For this reason, protocols or operating system components should be modified to ensure secure operation even when the user is not paying attention. For example, in the future there may be changes in browsers or operating systems to make cross device flows more secure against phishing attacks. One promising technology to address this issue is the Federated Credential Management API[2] in the Chrome browser.

As the OpenID for Verifiable Credentials protocol family grows in popularity, it is important to perform formal security analysis on the other protocols that have not yet been analyzed. One such protocol would be OpenID for Verifiable Presentations over BLE.

[1] https://www.bsi.bund.de/dok/12872520

[2] https://developer.chrome.com/en/docs/privacy-sandbox/fedcm/

Bibliography

[1] C. Bauer. “Formal analysis of self-issued OpenID providers”. masterThesis. 2022. ISBN: 9781817794047. DOI: 10.18419/opus-12398. URL: http://elib.uni-stuttgart.de/handle/11682/12417 (visited on 06/23/2023) (cit. on pp. 32, 45, 80).

[2] S. Curran, A. Philipp, H. Yildiz, S. Curren, V. M. Jurado. *AnonCreds Specification*. URL: https://hyperledger.github.io/anoncreds-spec/ (visited on 09/13/2023) (cit. on pp. 18, 21).

[3] S. Curren, T. Looker, O. Terbu. *DIDComm Messaging v2.0*. URL: https://identity.foundation/didcomm-messaging/spec/v2.0/ (visited on 09/13/2023) (cit. on p. 21).

[4] D. Fett. “An expressive formal model of the web infrastructure”. doctoralThesis. 2018. DOI: 10.18419/opus-10197. URL: http://elib.uni-stuttgart.de/handle/11682/10214 (visited on 05/12/2023) (cit. on pp. 31, 61).

[5] D. Fett, R. Kuesters, G. Schmitz. *The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines*. arXiv:1704.08539. type: article. arXiv, Apr. 27, 2017. arXiv: 1704.08539[cs]. URL: http://arxiv.org/abs/1704.08539 (visited on 08/31/2023) (cit. on p. 29).

[6] D. Fett, R. Küsters, G. 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*. 2014 IEEE Symposium on Security and Privacy. ISSN: 2375-1207. May 2014, pp. 673–688. DOI: 10.1109/SP.2014.49 (cit. on p. 29).

[7] D. Fett, R. Küsters, G. Schmitz. “The Web Infrastructure Model (WIM)”. In: (). URL: https://www.sec.uni-stuttgart.de/research/wim/WIM_V1.0.pdf (visited on 05/15/2023) (cit. on pp. 3, 4, 18, 29, 31, 32, 61, 63, 64, 69, 75, 80, 89, 91–94, 96–103).

[8] D. Fett, K. Yasuda, B. Campbell. *Selective Disclosure for JWTs (SD-JWT)*. Internet Draft draft-ietf-oauth-selective-disclosure-jwt-05. June 30, 2023. 84 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-oauth-selective-disclosure-jwt (visited on 09/15/2023) (cit. on p. 18).

[9] D. Hardt. *The OAuth 2.0 Authorization Framework*. Request for Comments RFC 6749. Oct. 2012. 76 pp. DOI: 10.17487/RFC6749. URL: https://datatracker.ietf.org/doc/rfc6749 (visited on 09/10/2023) (cit. on pp. 22, 24, 55).

[10] F. Helmschmidt. “Security analysis of the Grant Negotiation and Authorization Protocol”. masterThesis. 2022. ISBN: 9781809048349. DOI: 10.18419/opus-12203. URL: http://elib.uni-stuttgart.de/handle/11682/12220 (visited on 06/23/2023) (cit. on p. 80).

[11] P. Hosseyni, R. Küsters, T. Würtele. “Formal Security Analysis of the OpenID Financial-grade API 2.0”. In: (). URL: https://openid.net/wordpress-content/uploads/2022/12/Formal-Security-Analysis-of-FAPI-2.0_FINAL_2022-10.pdf (visited on 09/27/2023) (cit. on pp. 62, 83, 84, 86).

[12] *ISO/IEC 18013-5:2021*. URL: https://www.iso.org/standard/69084.html (visited on 09/13/2023) (cit. on pp. 18, 21).

[13] P. Kasselman, D. Fett, F. Skokan. *Cross-Device Flows: Security Best Current Practice*. Internet Draft draft-ietf-oauth-cross-device-security-02. July 10, 2023. 40 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-oauth-cross-device-security/02 (visited on 09/26/2023) (cit. on pp. 32, 45, 50, 80).

[14] T. Lodderstedt, J. Bradley, A. Labunets, D. Fett. *OAuth 2.0 Security Best Current Practice*. Internet Draft draft-ietf-oauth-security-topics-23. June 5, 2023. 62 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-oauth-security-topics-23 (visited on 09/09/2023) (cit. on pp. 22, 24, 25, 53).

[15] T. Lodderstedt, B. Campbell, N. Sakimura, D. Tonge, F. Skokan. *OAuth 2.0 Pushed Authorization Requests*. Request for Comments RFC 9126. Sept. 2021. 18 pp. DOI: 10.17487/RFC9126. URL: https://datatracker.ietf.org/doc/rfc9126 (visited on 08/30/2023) (cit. on p. 24).

[16] T. Lodderstedt, K. Yasuda, T. Looker. "OpenID for Verifiable Credential Issuance". In: (May 18, 2023). Git commit: e64463b017a31fcbda5a2ebc8571edb1bf07720d. URL: https://openid.bitbucket.io/connect/openid-4-verifiable-credential-issuance-1_0.html (visited on 09/11/2023) (cit. on pp. 3, 4, 18, 22, 31, 46, 55).

[17] N. Sakimura, J. Bradley, N. Agarwal. *Proof Key for Code Exchange by OAuth Public Clients*. Request for Comments RFC 7636. Sept. 2015. 20 pp. DOI: 10.17487/RFC7636. URL: https://datatracker.ietf.org/doc/rfc7636 (visited on 09/08/2023) (cit. on pp. 24, 48).

[18] M. Sporny, G. Noble, D. Longley, D. C. Burnett, B. Zundel, K. D. Hartog. *Verifiable Credentials Data Model v1.1*. Mar. 3, 2022. URL: https://www.w3.org/TR/2022/REC-vc-data-model-20220303/ (visited on 09/13/2023) (cit. on pp. 18, 21).

[19] O. Terbu, T. Lodderstedt, K. Yasuda, T. Looker. "OpenID for Verifiable Presentations". In: (Mar. 25, 2023). Git commit: 11157695d140f4c47f742f5d82e25283b90dd952. URL: https://openid.net/specs/openid-4-verifiable-presentations-1_0-17.html (visited on 09/11/2023) (cit. on pp. 3, 4, 18, 24, 25, 32, 50, 51).

[20] *The European Digital Identity Wallet Architecture and Reference Framework*. Jan. 1, 2023. URL: https://digital-strategy.ec.europa.eu/en/library/european-digital-identity-wallet-architecture-and-reference-framework (visited on 09/15/2023) (cit. on pp. 3, 4, 18).

All links were last followed on September 27, 2023.

The icons in Figure 1.1, Figure 4.1, and Figure 4.2 are taken from uxwing.com.

A Verifiable Credentials Web System

The following algorithms model the OpenID for Verifiable Credential Issuance and the OpenID for Verifiable Presentations protocols. The model is based on the OAuth 2.0 and OpenID Connect model from the dissertation "An expressive formal model of the web infrastructure" by Daniel Fett [4].

These protocols are modeled in a Verifiable Credentials web system $\mathcal{VCWS}^n = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$. The system $\mathcal{W} = \mathsf{Hon} \cup \mathsf{Net}$ consists of a network attacker process in Net. $\mathsf{Hon} = \mathsf{Issuers} \cup \mathsf{Wallets} \cup \mathsf{Verifiers} \cup \mathsf{B}$ contains a finite set of issuers $\mathsf{Issuers}$, a finite set of wallets $\mathsf{Wallets}$, a finite set of verifiers $\mathsf{Verifiers}$, and a finite set of browsers B. The processes are described in more detail in the following sections. There are no DNS servers explicitly modeled because they are subsumed by the network attacker. The set of scripts $\mathcal{S}$ and the mapping script can be seen in Table A.1. The set E^0 is defined as in Definition 42 of [7].

$s \in \mathcal{S}$	script(s)	Defined in
script_issuer_form	`script_issuer_form`	Algorithm 3
script_wallet_index	`script_wallet_index`	Algorithm 5
script_wallet_form	`script_wallet_form`	Algorithm 6
script_verifier_get_fragment	`script_verifier_get_fragment`	Algorithm 11
script_verifier_index	`script_verifier_index`	Algorithm 12

Table A.1: List of scripts in $\mathcal{S}$ with their string representation and definitions.

In the following list there are definitions of functions needed for the model:

- Let $\mathsf{userPinOfId} : \mathsf{ID} \to \mathcal{N}$ associate each user ID with a unique nonce that serves as a user PIN sent out-of-band (e.g., via email).
- Let $\mathsf{browserOfWallet} : \mathsf{Wallets} \to \mathsf{B}$ match every wallet to exactly one browser.
- Let $\mathsf{HASH} : \mathcal{T}_{\mathcal{N}} \to \mathcal{N}$ be a cryptographically secure hash function.
- Let $\mathsf{GETURL}(\mathit{tree}, \mathit{docnonce})$ be defined as in B.1.1 of [4].
- In a run ρ of a Verifiable Credential web system $\mathcal{VCWS}^n$, within a relation of a wallet w we say that $\mathsf{validateRequest}(x) \equiv \top$ iff there exists a configuration (S, E, N) that contains a state with $x \in S(b).\texttt{authStarted}$ and $b = \mathsf{browserOfWallet}(w)$.

Additionally, $\mathsf{WalletDoms} = \{d | d \in \mathsf{dom}(w) \wedge w \in \mathsf{Wallets}\}$ is the set of domains used by wallet processes.

A.1 Identities and Secrets

This section outlines the initialization process for identities, keys, and secrets in the Verifiable Credentials web system $\mathcal{VCWS}^n$. The following subsections are taken from Section 4 in [11] and customized to be suitable for the $\mathcal{VCWS}^n$.

A.1.1 Identities

Identities consist, similar to email addresses, of a user name and a domain part. For our model, this is defined as follows:

Definition A.1.1
An identity i is a term of the form $\langle name, domain \rangle$ with $name \in \mathbb{S}$ and $domain \in \mathsf{Doms}$. Let ID be the finite set of identities. We say that an id is *governed* by the DY process to which the domain of the id belongs. This is formally captured by the mappings $\mathsf{governor} : \mathsf{ID} \to \mathcal{W}$, $\langle name, domain \rangle \longmapsto \mathsf{dom}^{-1}(domain)$ and $\mathsf{ID}^y := \mathsf{governor}^{-1}(y)$.

A.1.2 Keys and Secrets

The set $\mathcal{N}$ of nonces is partitioned into disjoint sets, an infinite set N, and finite sets K_{TLS}, K_{sign}, and $\mathsf{Passwords}$:

$$\mathcal{N} = N \uplus K_{TLS} \uplus K_{sign} \uplus \mathsf{Passwords}$$

These sets are used as follows:

- The set N contains the nonces that are available for the DY processes
- The set K_{TLS} contains the keys that will be used for TLS encryption. Let $\mathsf{tlskey} : \mathsf{Doms} \to K_{TLS}$ be an injective mapping that assigns a (different) private key to every domain. For an atomic DY process p we define $tlskeys^p = \langle \{\langle d, \mathsf{tlskey}(d) \rangle | d \in \mathsf{dom}(p)\} \rangle$ (i.e., a sequence of pairs).
- The set K_{sign} contains the keys that will be used by issuers to sign credentials and by wallets to sign presentations. Let $\mathsf{signkey} : \mathsf{Issuers} \cup \mathsf{Wallets} \to K_{sign}$ be an injective mapping that assigns a (different) signing key to every issuer and wallet.
- The set of $\mathsf{Passwords}$ is the set of passwords (secrets) the browsers share with servers. These are the passwords the users use to log in. Let $\mathsf{secretOfID} : \mathsf{ID} \to \mathsf{Passwords}$ be a bijective mapping that assigns a password to each identity.

A.1.3 Passwords

Definition A.1.2
Let $\mathsf{ownerOfSecret} : \mathsf{Passwords} \to \mathsf{B}$ be a mapping that assigns to each password a browser which *owns* this password. Similarly, we define $\mathsf{ownerOfID} : \mathsf{ID} \to \mathsf{B}$ as $i \longmapsto \mathsf{ownerOfSecret}(\mathsf{secretOfID}(i))$, which assigns to each identity the browser that owns this identity (i.e., this identity belongs to the browser).

A.1.4 Web Browsers

Web browser processes (i.e., processes $b \in \mathsf{B}$) are modeled as described in [7]. Before defining additional constraints on the initial states of web browsers, we introduce the following set (for some process p):

$$\mathsf{Secrets}^{b,p} = \{s | b = \mathsf{ownerOfSecret}(s) \wedge (\exists i : s = \mathsf{secretOfID}(i) \wedge i \in \mathsf{ID}^p)\}$$

Definition A.1.3 (Initial Web Browser State for $\mathcal{VCWS}^{\mathrm{n}}$)
The initial state of a web browser process $b \in \mathsf{B}$ follows the Definition 33 in [7], with the following additional constraints:

- $s_0^b.\texttt{ids} \equiv \langle\{i | b = \mathsf{ownerOfID}(i)\}\rangle$
- $s_0^b.\texttt{secrets}$ contains an entry $\langle\langle d, \mathsf{S}\rangle, \langle\mathsf{Secrets}^{b,p}\rangle\rangle$ for each $p \in \mathsf{Issuer} \cup \{w \in \mathsf{Wallets} | b = \mathsf{browserOfWallet}(w)\}$ and every domain $d \in \mathsf{dom}(p)$ (and nothing else), i.e.,

$$\begin{aligned} s_0^b.\texttt{secrets} \equiv \langle\{\langle\langle d, \mathsf{S}\rangle, \langle\mathsf{Secrets}^{b,p}\rangle\rangle | \exists p, d : p \in \mathsf{Issuer} \\ \cup \{w \in \mathsf{Wallets} | b = \mathsf{browserOfWallet}(w)\} \wedge d \in \mathsf{dom}(p)\}\rangle \end{aligned}$$

- $s_0^b.\texttt{keyMapping} \equiv \langle\{\langle d, \mathsf{pub}(\mathsf{tlskey}(d))\rangle | d \in \mathsf{Doms}\}\rangle$

A.2 Issuers

An issuer $i \in \mathsf{Issuers}$ is a web server modeled as an atomic DY process (I^i, Z^i, R^i, s_0^i) with the address $I^i := \mathsf{addr}(i)$. The following definition defines the states Z^i of i and the initial state s_0^i of i.

Definition A.2.1
A state $s \in Z^i$ of an issuer i is a term of the form

$$\langle pendingDNS, pendingRequests, DNSaddress, keyMapping, tlskeys, corrupt, codes, atokens, signingKey\rangle$$

with *pendingDNS*, *pendingRequests*, *DNSaddress*, *keyMapping*, *tlskeys*, *corrupt* as defined in Definition 43 in [7], $codes \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $atokens \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$ and $signingKey \in \mathcal{N}$.
An initial state s_0^i of i is a state of i with $s_0^i.\mathtt{pendingDNS} = \langle\rangle$, $s_0^i.\mathtt{pendingRequests} = \langle\rangle$, $s_0^i.\mathtt{DNSaddress} \in \mathsf{IPs}$, $s_0^i.\mathtt{keyMapping} = \langle\{\langle d, \mathsf{pub}(\mathsf{tlskey}(d))\rangle | d \in \mathsf{Doms}\}\rangle$, $s_0^i.\mathtt{tlskeys} = tlskeys^i$, $s_0^i.\mathtt{corrupt} = \bot$, $s_0^i.\mathtt{codes} = \langle\rangle$, $s_0^i.\mathtt{atokens} = \langle\rangle$ and $s_0^i.\mathtt{signingKey} = \mathsf{signkey}(i)$.

The relation of an issuer i is based on the generic HTTPS server as defined in [7]. The following algorithms 1 to 3 overwrite and extend the generic HTTPS server. Methods that are not overwritten are defined as in [7]. Table A.2 shows a list of placeholders used by an issuer.

Placeholder	Usage
ν_1	new pre-authorized code
ν_2	new authorization code
ν_3	new access token
ν_4	new c_nonce
ν_5	new transaction ID

Table A.2: List of placeholders used by an issuer.

Algorithm 1 Relation of an issuer R^i: Processing HTTPS requests.

```
1: function PROCESS_HTTPS_REQUEST(m, k, a, f, s')
2:     let preAuthCode := urn:ietf:params:oauth:grant-type:pre-authorized_code
3:     if m.path ≡ / then      → Start pre-authorized code flow
4:         let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, ⟨script_issuer_form,
           ↪ ⟨/startCredOffer⟩⟩⟩, k)
           ↪      → Reply with script_issuer_form in pre-authorized code mode
5:         stop ⟨⟨f, a, m'⟩⟩, s'
6:     else if m.path ≡ /startCredOffer ∧ m.method ≡ POST then
       ↪      → Start credential offer endpoint
7:         if m.body[username] ≡ ⟨⟩ ∨ m.body[password] ≡ ⟨⟩ ∨ m.body[wallet] ≡ ⟨⟩ then
8:             stop
9:         let username := m.body[username]
10:        let wallet := m.body[wallet]
11:        if m.body[password] ≢ secretOfID(username) then
12:            stop
13:        let useUserPin ← {⊤, ⊥}
14:        let preAuthorizedCode := ν_1
15:        let s'.codes := s'.codes +^⟨⟩ ⟨preAuthorizedCode, ⟨useUserPin, username⟩⟩
16:        let credOffer := ⟨m.host, preAuthorizedCode, useUserPin⟩
17:        let parameters := ⟨credential_offer, credOffer⟩
18:        let credentialOfferURL := ⟨URL, S, wallet, /vci/credentialOffer, parameters, ⊥⟩
19:        let m' := enc_s(⟨HTTPResp, m.nonce, 303, ⟨⟨Location, credentialOfferURL⟩⟩, ⟨⟩⟩, k)
           ↪      → Use status code 303 (see other) as recommended by the OAuth 2.0 Security BCP
20:        let leak := ⟨LEAK, credentialOfferURL⟩    → Model a possible leak via custom scheme by
       sending an unencrypted message with the credential offer URL
21:        let a' ← IPs
22:        stop ⟨⟨f, a, m'⟩, ⟨f, a', leak⟩⟩, s'
23:    else if m.path ≡ /authentication then
24:        if m.method ≡ GET ∨ (m.method ≡ POST ∧ (m.body[username] ≡ ⟨⟩
           ↪ ∨ m.body[password] ≡ ⟨⟩)) then
25:            let data := m.parameters
26:            let m' := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, ⟨script_issuer_form,
               ↪ ⟨/authentication, data⟩⟩⟩, k)
               ↪      → Reply with script_issuer_form in authorization_code mode
27:            stop ⟨⟨f, a, m'⟩⟩, s'
28:        else if m.method ≡ POST then
29:            if m.headers[Origin] ≢ ⟨m.host, S⟩ then
30:                stop    → Without the origin check the session integrity property would break because
       an attacker script could use this endpoint
31:            let username := m.body[username]
32:            if m.body[client_id] ≡ ⟨⟩ ∨ m.body[redirect_uri] ≡ ⟨⟩
               ↪ ∨ m.body[code_challenge] ≡ ⟨⟩ then
33:                stop
34:            let clientID := m.body[client_id]
35:            let redirectUri := m.body[redirect_uri]
36:            let codeChallenge := m.body[code_challenge]
37:            if m.body[password] ≢ secretOfID(username) then
38:                stop
```

```
39:             let authorizationCode := ν2
40:             let s'.codes := s'.codes +⟨⟩ ⟨authorizationCode, ⟨clientID, redirectUri,
                  ↪ codeChallenge, username⟩⟩
41:             let redirectUri.parameters := redirectUri.parameters +⟨⟩ ⟨code, authorizationCode⟩
42:             let redirectUri.parameters := redirectUri.parameters +⟨⟩ ⟨iss, m.host⟩
          ↪      → Issuer identifier to prevent mix-up attacks recommended by the OAuth 2.0 Security BCP
43:             let redirectUri.parameters := redirectUri.parameters +⟨⟩ ⟨state, m.body[state]⟩
44:             let m' := encs(⟨HTTPResp, m.nonce, 303, ⟨⟨Location, redirectUri⟩⟩, ⟨⟩⟩, k)
          ↪      → Use status code 303 (see other) as recommended by the OAuth 2.0 Security BCP
          ↪      → This URL does not leak because it uses a claimed URL
45:             stop ⟨⟨f, a, m'⟩⟩, s'
46:     else if m.path ≡ /token ∧ m.method ≡ POST then
47:         if m.body[client_id] ≡ ⟨⟩ then
48:             stop
49:         if m.body[grant_type] ≡ preAuthCode then
50:             let code := m.body[pre-authorized_code]
51:             let codeInfo := s'.codes[code]
52:             if codeInfo ≡ ⟨⟩ then
53:                 stop
54:             let userPin := m.body[user_pin]
55:             if codeInfo.1 ≡ ⊤ ∧ userPin ≠ userPinOfId(codeInfo.2) then    → Check user pin
56:                 stop
57:             let username := codeInfo.2
58:         else if m.body[grant_type] ≡ authorization_code then
59:             if m.body[redirect_uri] ≡ ⟨⟩ ∨ m.body[code_verifier] ≡ ⟨⟩ then
60:                 stop
61:             let clientID := m.body[client_id]
62:             let redirectURI := m.body[redirect_uri]
63:             let codeVerifier := m.body[code_verifier]
64:             let code := m.body[code]
65:             let codeInfo := s'.codes[code]
66:             if codeInfo ≡ ⟨⟩ ∨ codeInfo.1 ≠ clientID ∨ codeInfo.2 ≠ redirectURI
                  ↪ ∨ codeInfo.3 ≠ HASH(codeVerifier) then
67:                 stop
68:             let username := codeInfo.4
69:         let s'.codes := s'.codes − code
70:         let accessToken := ν3
71:         let cNonce := ν4
72:         let s'.atokens := s'.atokens +⟨⟩ ⟨accessToken, ⟨⟨username, username⟩, ⟨c_nonce, cNonce⟩⟩⟩
73:         let body := ⟨⟨access_token, accessToken⟩, ⟨c_nonce, cNonce⟩⟩
74:         let m' := encs(⟨HTTPResp, m.nonce, 200, ⟨⟩, body⟩, k)
75:         stop ⟨⟨f, a, m'⟩⟩, s'
76:     else if m.path ≡ /credential ∧ m.method ≡ POST then
77:         let authHeader := m.header[Authorization]
78:         if authHeader ≡ ⟨⟩ ∨ authHeader.1 ≠ BEARER ∨ s'.atokens[authHeader.2] ≡ ⟨⟩ then
79:             stop
80:         let tokenInfo := s'.atokens[authHeader.2]
81:         let proof := m.body[proof]
```

```
82:       let aud, cNonce, pubKey such that
          ↪ ⟨aud, cNonce, pubKey⟩ ≡ extractmsg(proof)
          ↪ if possible; otherwise stop
83:       if tokenInfo[c_nonce] ≠ cNonce ∨ m.host ≠ aud ∨ checksig(proof, pubKey) ≡ ⊥ then
84:           stop
85:       let username := tokenInfo[username]
86:       let issueCred ← {⊤, ⊥}
87:       if issueCred ≡ ⊤ then
88:           let credential := BUILD_CREDENTIAL(s′, username, m.host, pubKey)
89:           let m′ := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, ⟨⟨credential, credential⟩⟩⟩, k)
90:           stop ⟨⟨f, a, m′⟩⟩, s′
91:       else
92:           let transactionId := v5
93:           let tokenInfo[transaction_ids] := ⟨⟨transactionId, ⟨public_key, pubKey⟩⟩⟩
94:           let m′ := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, ⟨⟨transaction_id, transactionId⟩⟩⟩, k)
95:           stop ⟨⟨f, a, m′⟩⟩, s′
96:   else if m.path ≡ /batchCredential ∧ m.method ≡ POST then
97:       let authHeader := m.header[Authorization]
98:       if authHeader ≡ ⟨⟩ ∨ authHeader.1 ≠ BEARER ∨ s′.atokens[authHeader.2] ≡ ⟨⟩ then
99:           stop
100:      let tokenInfo := s′.atokens[authHeader.2]
101:      let credResp := ⟨⟩
102:      for proof ∈ m.body[credential_requests] do
103:          let aud, cNonce, pubKey such that
              ↪ ⟨aud, cNonce, pubKey⟩ ≡ extractmsg(proof)
              ↪ if possible; otherwise stop
104:          if tokenInfo[c_nonce] ≠ cNonce ∨ m.host ≠ aud ∨ checksig(proof, pubKey) ≡ ⊥ then
105:              stop
106:          let username := tokenInfo[username]
107:          let issueCred ← {⊤, ⊥}
108:          if issueCred ≡ ⊤ then
109:              let credential := BUILD_CREDENTIAL(s′, username, m.host, pubKey)
110:              let credResp[credentials] := credResp[credentials] +^⟨⟩ credential
111:          else
112:              let transactionId := v5
113:              let tokenInfo[transaction_ids] := tokenInfo[transaction_ids] +^⟨⟩
                  ↪ ⟨transactionId, ⟨public_key, pubKey⟩⟩
114:              let credResp[transaction_ids] := credResp[transaction_ids] +^⟨⟩ transactionId
115:      let m′ := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, ⟨⟨credential_responses, credResp⟩⟩⟩, k)
116:      stop ⟨⟨f, a, m′⟩⟩, s′
117:  else if m.path ≡ /deferredCredential ∧ m.method ≡ POST then
118:      let authHeader := m.header[Authorization]
119:      if authHeader ≡ ⟨⟩ ∨ authHeader.1 ≠ BEARER ∨ s′.atokens[authHeader.2] ≡ ⟨⟩ then
120:          stop
121:      let tokenInfo := s′.atokens[authHeader.2]
122:      let transactionID := m.body[transaction_id]
123:      if transactionID ∉ tokenInfo[transaction_ids] then
124:          stop
125:      let pubKey := tokenInfo[transaction_ids][transactionID].public_key
126:      let tokenInfo[transaction_ids] := tokenInfo[transaction_ids] − transactionID
          ↪ → Transaction ID can only be used once to obtain a credential
```

```
127:        let credential := BUILD_CREDENTIAL(s′, tokenInfo[username], m.host, pubKey)
128:        let m′ := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, ⟨⟨credential, credential⟩⟩⟩, k)
129:        stop ⟨⟨f, a, m′⟩⟩, s′
130:    stop
```

Algorithm 2 Relation of an issuer R^i: Function to build and sign a credential.

```
1: function BUILD_CREDENTIAL(s′, username, iss, pubKey)
2:     let credentialBody := ⟨username, iss, pubKey⟩
3:     let credential := sig(credentialBody, s′.signingKey)
4:     return credential
```

Algorithm 3 Relation of *script_issuer_form*.

```
Input: ⟨tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secrets⟩
 1: let url := GETURL(tree, docnonce)
 2: let url′ := ⟨URL, S, url.host, scriptstate.1, ⟨⟩, ⟨⟩⟩
 3: let formData := scriptstate.2
 4: let username ← ids
 5: let password ← secrets
 6: let wallet ← WalletDoms    → Let the user choose their wallet
 7: let formData[username] := username
 8: let formData[password] := password
 9: let formData[wallet] := wallet
10: let command := ⟨START, url′, formData⟩
11: stop ⟨scriptstate, cookies, localStorage, sessionStorage, command⟩
```

A.3 Wallets

An wallet $w \in \mathsf{Wallets}$ is a web server modeled as an atomic DY process (I^w, Z^w, R^w, s_0^w) with the address $I^w := \mathsf{addr}(w)$. The following definition defines the states Z^w of w and the initial state s_0^w of w.

Definition A.3.1
A state $s \in Z^w$ of a wallet w is a term of the form

$$\langle pendingDNS, pendingRequests, DNSaddress, keyMapping, tlskeys, corrupt, \\ credentials, holderKey, userPins, sessions, transactionIds \rangle$$

with *pendingDNS*, *pendingRequests*, *DNSaddress*, *keyMapping*, *tlskeys*, *corrupt* as defined in Definition 43 in [7], $credentials \in \mathcal{T}_{\mathcal{N}}$, $holderKey \in \mathcal{N}$, $userPins \in [\mathsf{Doms} \times \mathcal{N}]$, $sessions \in [\mathcal{N} \times \mathcal{T}_{\mathcal{N}}]$, and $transactionIds \in \mathcal{T}_{\mathcal{N}}$.
An initial state s_0^w of w is a state of w with $s_0^w.\mathtt{pendingDNS} = \langle\rangle$, $s_0^w.\mathtt{pendingRequests} = \langle\rangle$, $s_0^w.\mathtt{DNSaddress} \in \mathsf{IPs}$, $s_0^w.\mathtt{keyMapping} = \langle\{\langle d, \mathsf{pub}(\mathsf{tlskey}(d))\rangle | d \in \mathsf{Doms}\}\rangle$, $s_0^w.\mathtt{tlskeys} = tlskeys^w$, $s_0^w.\mathtt{corrupt} = \bot$, $s_0^w.\mathtt{credentials} = \langle\rangle$, $s_0^w.\mathtt{holderKey} = \mathsf{signkey}(w)$, $s_0^w.\mathtt{userPins} = \langle\{\langle i.\mathsf{domain}, \mathsf{userPinOfId}(i)\rangle | \forall i \in s_0^b.\mathtt{ids} \text{ with } b \in \mathsf{browserOfWallet}(w)\}\rangle$, $s_0^w.\mathtt{sessions} = \langle\rangle$, and $s_0^w.\mathtt{transactionIds} = \langle\rangle$.

The relation of a wallet w is based on the generic HTTPS server as defined in [7]. The following algorithms 4 to 8 overwrite and extend the generic HTTPS server. Methods that are not overwritten are defined as in [7]. Table A.3 shows a list of placeholders used by a wallet.

Placeholder	Usage
ν_1	new code challenge
ν_2	new session ID
ν_3	new HTTP request nonce
ν_4	new state value

Table A.3: List of placeholders used by a wallet.

Algorithm 4 Relation of a wallet R^w: Processing HTTPS requests.

function PROCESS_HTTPS_REQUEST(m, k, a, f, s')
  if m.path ≡ /vci **then**
    ——————————Start issuance via Authorization Code Flow——————————
    let m' := enc$_s$(⟨HTTPResp, m.nonce, 200, ⟨⟩, ⟨script_wallet_index⟩⟩, k)
    stop ⟨⟨f, a, m'⟩⟩, s'
  else if m.path ≡ /vci/credentialOffer **then**
    ——————————Issuance via pre-authorized Code Flow——————————
    let *preAuthCodeFlow* := urn:ietf:params:oauth:grant-type:pre-authorized_code
    let *credentialOffer* := m.parameters[credential_offer]
    let *issHost*, *preAuthorizedCode*, *useUserPin* **such that**
      ↪ ⟨*issHost*, *preAuthorizedCode*, *useUserPin*⟩ ≡ *credentialOffer*
      ↪ **if possible; otherwise stop**
    if ¬validateRequest(⟨*issHost*, *preAuthorizedCode*⟩) **then**
      stop → This check models a perfect user who knows which issuer they started the flow with
    let *sessionID* := ν_2
    let *session* := ⟨⟨host, *issHost*⟩, ⟨pre-authorized_code, *preAuthorizedCode*⟩,
      ↪ ⟨use_user_pin, *useUserPin*⟩⟩
    let s'.sessions := s'.sessions +$^{\langle\rangle}$ ⟨*sessionID*, *session*⟩
    let *clientID* ← {⟨URL, S, d, /vci/redirect, ⟨⟩, ⟨⟩⟩|d ∈ dom(r)} → client_id = redirect_uri
    let *body* := ⟨⟨grant_type, *preAuthCodeFlow*⟩, ⟨client_id, *clientID*⟩,
      ↪ ⟨pre-authorized_code, *preAuthorizedCode*⟩⟩
    if *useUserPin* ≡ ⊤ **then**
      let *body*[user_pin] := s'.userPins[*issHost*]
    let *url* := ⟨URL, S, *issHost*, /token, ⟨⟩, ⊥⟩
    let *message* := ⟨HTTPReq, ν_3, POST, *url*.host, *url*.path, *url*.parameters, ⟨⟩, *body*⟩
    call HTTP_SIMPLE_SEND(⟨⟨responseTo, TOKEN⟩, ⟨session, *sessionID*⟩⟩, *message*, *url*, s')
  else if m.path ≡ /vci/startCodeFlow ∧ m.method ≡ POST **then**
    ——————————Issuance via Authorization Code Flow——————————
    if m.headers[Origin] ≢ ⟨m.host, S⟩ **then**
      stop → Without the origin check the session integrity property would break because an attacker could start a flow in the background
    let *codeVerifier* := ν_1
    let *parameters* := ⟨⟨code_challenge, HASH(*codeVerifier*)⟩⟩
    let *redirectUri* ← {⟨URL, S, d, /vci/redirect, ⟨⟩, ⟨⟩⟩|d ∈ dom(r)}
    let *parameters* := *parameters* +$^{\langle\rangle}$ ⟨redirect_uri, *redirectUri*⟩
    let *parameters* := *parameters* +$^{\langle\rangle}$ ⟨client_id, *redirectUri*⟩
    let *identity* := m.body
    let *authUrl* := ⟨URL, S, *identity*.domain, /authentication, ⟨⟩, ⊥⟩
      ↪ → Use the user selected issuer to start the issuance
    let *useState* ← {⊤, ⊥}
    if *useState* ≡ ⊤ **then**
      let *parameters* := *parameters* +$^{\langle\rangle}$ ⟨state, ν_4⟩
    else
      let *parameters* := *parameters* +$^{\langle\rangle}$ ⟨state, ⟨⟩⟩
    let *sessionID* := ν_2
    let *session* := *parameters* +$^{\langle\rangle}$ ⟨⟨host, *authUrl*.host⟩, ⟨code_verifier, *codeVerifier*⟩⟩
    let s'.sessions := s'.sessions +$^{\langle\rangle}$ ⟨*sessionID*, *session*⟩
    let *authUrl*.parameters := *authUrl*.parameters ∪ *parameters*
    let *headers* := ⟨⟨Location, *authUrl*⟩⟩
    let *headers* := *headers* +$^{\langle\rangle}$ ⟨Set-Cookie, [⟨__Host, sessionID⟩: ⟨*sessionID*, ⊤, ⊤, ⊤⟩]⟩

```
42:         let m′ := enc_s(⟨HTTPResp, m.nonce, 303, headers, ⟨⟩⟩, k)
        ↪       → Use status code 303 (see other) as recommended by the OAuth 2.0 Security BCP
        ↪       → This URL does not leak because a browser can be securely opened
43:         stop ⟨⟨f, a, m′⟩⟩, s′
44:     else if m.path ≡ /vci/redirect then
    ──────────────────────Process Authentication Code Flow issuance response──────────────────────
45:         let sessionID := m.header[Cookie][⟨__Host, sessionID⟩]
46:         if sessionID ∉ s′.sessions then
47:             stop
48:         let session := s′.sessions[sessionID]
49:         let issHost := session[host]
50:         if m.parameters[iss] ≠ issHost then     → Mix-up attack mitigation
51:             stop
52:         if m.parameters[state] ≠ session[state] then
53:             stop
54:         let redirectUri := session[redirect_uri]
55:         let code := m.parameters[code]
56:         let session := session +^⟨⟩ ⟨code, code⟩
            ↪       → Required only for the Issuance Authentication property (this value is never used)
57:         let body := ⟨⟨grant_type, authorization_code⟩, ⟨client_id, redirectUri⟩,
            ↪ ⟨redirect_uri, redirectUri⟩, ⟨code, code⟩, ⟨code_verifier, session[code_verifier]⟩⟩

58:         let url := ⟨URL, S, issHost, /token, ⟨⟩, ⊥⟩
59:         let message := ⟨HTTPReq, v3, POST, url.host, url.path, url.parameters, ⟨⟩, body⟩
60:         call HTTP_SIMPLE_SEND(⟨⟨responseTo, TOKEN⟩, ⟨session, sessionID⟩⟩, message, url, s′)
61:     else if m.path ≡ /vp/authentication then
    ──────────────────Process verifiable presentations authorization request──────────────────
62:         if m.method ≡ GET ∨ (m.method ≡ POST ∧ (m.body[username] ≡ ⟨⟩
            ↪ ∨ m.body[password] ≡ ⟨⟩)) then
63:             let data := m.parameters
64:             let m′ := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, ⟨script_wallet_form, data⟩⟩, k)
                ↪       → Reply with script_issuer_form
65:             stop ⟨⟨f, a, m′⟩⟩, s′
66:         else if m.method ≡ POST then
67:             if m.body[password] ≠ secretOfID(m.body[username]) then
68:                 stop     → Make sure that only the user of the wallet can use the wallet
69:             let responseMode := m.body[response_mode]
70:             if responseMode ≡ fragment then
71:                 let authRespUri := m.body[redirect_uri]
72:             else if responseMode ≡ direct_post then
73:                 let authRespUri := m.body[response_uri]
74:             if authRespUri ≠ m.body[client_id] then
75:                 stop     → Check if the redirect_uri or response_uri parameter equals the client_id
    parameter because it is required by the spec
76:             let credential ← s′.credentials
                ↪       → User chooses a credential and approves the presentation to the verifier
77:             let username, iss, pubKey such that
                ↪ ⟨username, iss, pubKey⟩ ≡ extractmsg(credential)
                ↪ if possible; otherwise stop
```

```
78:             let validateAuthRequest ← {⊤, ⊥}
79:             if validateAuthRequest ≡ ⊤
                  ↪ ∧ ¬validateRequest(⟨authRespUri.host, m.body[nonce]⟩) then
80:               stop    → In the cross-device flow the wallet needs to check whether the user started the
    authentication because otherwise phishing attacks would be possible
81:             let nonce := m.body[nonce]
82:             let aud := authRespUri.host
83:             let presentation := ⟨credential, nonce, aud⟩
84:             let vpToken := sig(presentation, s′.holderKey)
85:             let parameters := ⟨⟨vp_token, vpToken⟩⟩
86:             let parameters := parameters +^⟨⟩ ⟨iss, m.host⟩
87:             let parameters := parameters +^⟨⟩ ⟨state, m.body[state]⟩
88:             if responseMode ≡ fragment then
89:               let authRespUri.fragment := authRespUri.fragment ∪ parameters
90:               let m′ := enc_s(⟨HTTPResp, m.nonce, 303, ⟨⟨Location, authRespUri⟩⟩, ⟨⟩⟩, k)
          ↪   → Use status code 303 (see other) as recommended by the OAuth 2.0 Security BCP
          ↪   → This URL cannot leak as long as a native app verifier uses a verified app link
91:               stop ⟨⟨f, a, m′⟩⟩, s′
92:             else if responseMode ≡ direct_post then
93:               let sessionID := v_2
94:               let session := ⟨⟨message, m⟩⟩
95:               let session := session +^⟨⟩ ⟨receiver, a⟩
96:               let session := session +^⟨⟩ ⟨sender, f⟩
97:               let session := session +^⟨⟩ ⟨key, k⟩
98:               let s′.sessions := s′.sessions +^⟨⟩ ⟨sessionID, session⟩
99:               let message := ⟨HTTPReq, v_3, POST, authRespUri.host, authRespUri.path, ⟨⟩, ⟨⟩, parameters⟩
100:              call HTTP_SIMPLE_SEND(⟨⟨responseTo, DIRECT_POST⟩, ⟨session, sessionID⟩⟩,
                  ↪ message, authRespUri, s′)
```

Algorithm 5 Relation of *script_wallet_index*.

```
Input: ⟨tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secrets⟩
 1: let switch ← {issuance, link}
 2: if switch ≡ issuance then
 3:     let url := GETURL(tree, docnonce)
 4:     let url′ := ⟨URL, S, url.host, /vci/startCodeFlow, ⟨⟩, ⊥⟩
 5:     let id ← ids
 6:     let command := ⟨FORM, url′, POST, id, ⊥⟩
 7:     stop ⟨scriptstate, cookies, localStorage, sessionStorage, command⟩
 8: else
 9:     let protocol ← {P, S}
10:     let host ← Doms
11:     let path ← 𝕊
12:     let parameters ← [𝕊 × 𝕊]
13:     let fragment ← 𝕊
14:     let url := ⟨URL, protocol, host, path, parameters, fragment⟩
15:     let command := ⟨HREF, url, ⊥, ⊥⟩
16:     stop ⟨scriptstate, cookies, localStorage, sessionStorage, command⟩
```

Algorithm 6 Relation of *script_wallet_form*.

Input: ⟨*tree*, *docnonce*, *scriptstate*, *scriptinputs*, *cookies*, *localStorage*, *sessionStorage*, *ids*, *secrets*⟩
let *url* := GETURL(*tree*, *docnonce*)
let *url′* := ⟨URL, S, *url*.host, /vp/authentication, ⟨⟩, ⊥⟩
let *formData* := *scriptstate*
let *username* ← *ids*
let *password* ← *secrets*
let *formData*[username] := *username*
let *formData*[password] := *password*
let *command* := ⟨FORM, *url′*, POST, *formData*, ⊥⟩
stop ⟨*scriptstate*, *cookies*, *localStorage*, *sessionStorage*, *command*⟩

Algorithm 7 Relation of a wallet R^w: Processing HTTPS responses.

function PROCESS_HTTPS_RESPONSE(*m*, *reference*, *request*, *a*, *f*, *s′*)
  if *reference*[responseTo] ≡ TOKEN **then**
    let *sessionID* := *reference*[session]
    let *session* := *s′*.sessions[*sessionID*]
    if *m*.body[access_token] ≡ ⟨⟩ ∨ *m*.body[c_nonce] ≡ ⟨⟩ **then**
      stop
    let *accessToken* := *m*.body[access_token]
    let *session*[access_token] := *accessToken*
    let *cNonce* := *m*.body[c_nonce]
    let *issHost* := *session*[host]
    let *useBatchCredEndpoint* ← {⊤, ⊥}
    if *useBatchCredEndpoint* ≡ ⊥ **then**
      let *proofPossBody* := ⟨*issHost*, *cNonce*, pub(*s′*.holderKey)⟩
      let *proof* := sig(*proofPossBody*, *s′*.holderKey)
      let *url* := ⟨URL, S, *issHost*, /credential, ⟨⟩, ⊥⟩
      let *headers* := ⟨Authorization, ⟨BEARER, *accessToken*⟩⟩
      let *body* := ⟨proof, *proof*⟩
      let *message* := ⟨HTTPReq, v_3, POST, *url*.domain, *url*.path, *url*.parameters, *headers*, *body*⟩
      call HTTP_SIMPLE_SEND(⟨⟨responseTo, CREDENTIAL⟩, ⟨session, *sessionID*⟩⟩, *message*, *url*, *s′*)
    else
      let *credentialRequests* := ⟨⟩
      for *i* ∈ {1, 2, ..., *n*} *n* ∈ ℕ **do**
        let *proofPossBody* := ⟨*issHost*, *cNonce*, pub(*s′*.holderKey)⟩
        let *credentialRequests* := *credentialRequests* +$^{\langle\rangle}$ sig(*proofPossBody*, *s′*.holderKey)
      let *url* := ⟨URL, S, *issHost*, /batchCredential, ⟨⟩, ⊥⟩
      let *headers* := ⟨Authorization, ⟨BEARER, *accessToken*⟩⟩
      let *body* := ⟨credential_requests, *credentialRequests*⟩
      let *message* := ⟨HTTPReq, v_3, POST, *url*.domain, *url*.path, *url*.parameters, *headers*, *body*⟩
      call HTTP_SIMPLE_SEND(⟨⟨responseTo, BATCH_CREDENTIAL⟩, ⟨session, *sessionID*⟩⟩, *message*, *url*, *s′*)
  else if *reference*[responseTo] ≡ CREDENTIAL **then**
    let *sessionID* := *reference*[session]
    let *session* := *s′*.sessions[*sessionID*]
    if *m*.body[credential] ≢ ⟨⟩ **then**
      let *s′*.credentials := *s′*.credentials +$^{\langle\rangle}$ *m*.body[credential]
      let *session*[credentials] := *session*[credentials] +$^{\langle\rangle}$ *m*.body[credential]

```
36:         else if m.body[transaction_id] ≢ ⟨⟩ then
37:             let s'.transactionIds := s'.transactionIds +^⟨⟩
                    ↪ ⟨session[access_token], session[host], m.body[transaction_id], sessionID⟩
38:     else if reference[responseTo] ≡ BATCH_CREDENTIAL then
39:         let sessionID := reference[session]
40:         let session := s'.sessions[sessionID]
41:         for credential ∈ m.body[credential_responses].credentials do
42:             let s'.credentials := s'.credentials +^⟨⟩ credential
43:             let session[credentials] := session[credentials] +^⟨⟩ credential
44:         for transactionId ∈ m.body[credential_responses].transaction_ids do
45:             let s'.transactionIds := s'.transactionIds +^⟨⟩
                    ↪ ⟨session[access_token], session[host], transactionId, sessionID⟩
46:     else if reference[responseTo] ≡ DEFERRED_CREDENTIAL then
47:         let sessionID := reference[session]
48:         let session := s'.sessions[sessionID]
49:         let s'.credentials := s'.credentials +^⟨⟩ m.body[credential]
50:         let session[credentials] := session[credentials] +^⟨⟩ m.body[credential]
51:     else if reference[responseTo] ≡ DIRECT_POST then
52:         if m.body[redirect_uri] ≢ ⟨⟩ then
53:             let sessionID := reference[session]
54:             let session := s'.sessions[sessionID]
55:             let headers := ⟨⟨Location, m.body[redirect_uri]⟩⟩
56:             let m' := enc_s(⟨HTTPResp, session[message].nonce, 303, headers, ⟨⟩⟩, session[key])
        ↪       → Use status code 303 (see other) as recommended by the OAuth 2.0 Security BCP
        ↪       → This URL cannot leak because it is only used if the verifier is a web server
57:             stop ⟨⟨session[sender], session[receiver], m'⟩⟩, s'
58:         else
59:             stop
```

Algorithm 8 Relation of a wallet R^w: Processing trigger messages.

```
1:  function PROCESS_TRIGGER(a, s')
2:      let sendDeferredCredentialRequest ← {⊤, ⊥}
3:      if sendDeferredCredentialRequest ≡ ⊤ then
4:          let tokenInfo ← s'.transactionIds
            ↪       → Choose non-deterministically one of the pending deferred credential requests
5:          if tokenInfo ≡ ⟨⟩ then
6:              stop
7:          let s'.transactionIds := s'.transactionIds −^⟨⟩ tokenInfo
8:          let accessToken := tokenInfo.1
9:          let issHost := tokenInfo.2
10:         let transactionId := tokenInfo.3
11:         let sessionID := tokenInfo.4
12:         let url := ⟨URL, S, issHost, /deferredCredential, ⟨⟩, ⊥⟩
13:         let headers := ⟨Authorization, ⟨BEARER, accessToken⟩⟩
14:         let body := ⟨transaction_id, transactionId⟩
15:         let message := ⟨HTTPReq, ν_3, POST, url.host, url.path, url.parameters, headers, body⟩
16:         call HTTP_SIMPLE_SEND(⟨⟨responseTo, DEFERRED_CREDENTIAL⟩,
            ↪ ⟨session, sessionID⟩⟩, message, url, s')
```

A.4 Verifiers

An verifier $v \in \mathsf{Verifiers}$ is a web server modeled as an atomic DY process (I^v, Z^v, R^v, s_0^v) with the address $I^v := \mathsf{addr}(v)$. The following definition defines the states Z^v of v and the initial state s_0^v of v.

Definition A.4.1
A state $s \in Z^v$ of a verifier v is a term of the form

$$\langle pendingDNS, pendingRequests, DNSaddress, keyMapping, tlskeys, corrupt, issuers, sessions\rangle$$

with *pendingDNS*, *pendingRequests*, *DNSaddress*, *keyMapping*, *tlskeys*, *corrupt* as defined in Definition 43 in [7], $issuers \in \left[\mathsf{Doms} \times \mathsf{pub}(\mathcal{N})\right]$ and $sessions \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$.
An initial state s_0^v of v is a state of v with $s_0^v.\texttt{pendingDNS} = \langle\rangle$, $s_0^v.\texttt{pendingRequests} = \langle\rangle$, $s_0^v.\texttt{DNSaddress} \in \mathsf{IPs}$, $s_0^v.\texttt{keyMapping} = \langle\{\langle d, \mathsf{pub}(\mathsf{tlskey}(d))\rangle | d \in \mathsf{Doms}\}\rangle$, $s_0^v.\texttt{tlskeys} = tlskeys^v$, $s_0^v.\texttt{corrupt} = \bot$, $s_0^v.\texttt{issuers} = \{\langle d, \mathsf{pub}(\mathsf{signkey}(i))\rangle | d \in \mathsf{dom}(i) \wedge i \in \mathsf{Issuers}\}$ and $s_0^v.\texttt{sessions} = \langle\rangle$.

The relation of a verifier v is based on the generic HTTPS server as defined in [7]. The following algorithms 9 to 12 overwrite and extend the generic HTTPS server. Methods that are not overwritten are defined as in [7]. Table A.4 shows a list of placeholders used by a verifier.

Placeholder	Usage
ν_1	new response code
ν_2	new service token
ν_3	new state value
ν_4	new session ID
ν_5	new nonce value

Table A.4: List of placeholders used by a verifier.

Algorithm 9 Relation of a verifier R^v: Processing HTTPS requests.

function PROCESS_HTTPS_REQUEST(m, k, a, f, s')
  if m.path $\equiv$ / **then**
    let m' := enc$_s$(⟨HTTPResp, m.nonce, 200, ⟨⟩, ⟨script_verifier_index⟩⟩, k)
    stop ⟨⟨f, a, m'⟩⟩, s'
  else if m.path $\equiv$ /start $\wedge$ m.method $\equiv$ POST **then**
    Generate authentication request
    if m.headers[Origin] $\not\equiv$ ⟨m.host, S⟩ **then**
      stop → Without the origin check the session integrity property would break because an attacker could start a flow in the background
    let *nonce* := ν_5
    let *responseMode* ← {fragment, direct_post}
    let *parameters* := *parameters* +$^{\langle\rangle}$ ⟨response_mode, *responseMode*⟩
    let *parameters* := *parameters* +$^{\langle\rangle}$ ⟨nonce, *nonce*⟩
    if *responseMode* $\equiv$ direct_post **then**
      let *responseUri* ← {⟨URL, S, d, /directPost, ⟨⟩, ⊥⟩|$d \in$ dom(v)}
      let *parameters* := *parameters* +$^{\langle\rangle}$ ⟨response_uri, *responseUri*⟩
      let *parameters* := *parameters* +$^{\langle\rangle}$ ⟨client_id, *responseUri*⟩
    else
      let *redirectUri* ← {⟨URL, S, d, /redirect, ⟨⟩, ⟨⟩⟩|$d \in$ dom(v)}
      let *parameters* := *parameters* +$^{\langle\rangle}$ ⟨redirect_uri, *redirectUri*⟩
      let *parameters* := *parameters* +$^{\langle\rangle}$ ⟨client_id, *redirectUri*⟩
    let *session* := ⟨⟨nonce, *nonce*⟩⟩
    let *domain* ← WalletDoms → User chooses their wallet
    let *authUrl* := ⟨URL, S, *domain*, /vp/authentication, ⟨⟩, ⊥⟩
    let *session* := *session* +$^{\langle\rangle}$ ⟨host, *authUrl*.host⟩
    let *useState* ← {⊤, ⊥}
    if *useState* $\equiv$ ⊤ **then**
      let *state* := ν_3
      let *parameters* := *parameters* +$^{\langle\rangle}$ ⟨state, *state*⟩
      let *session* := *session* +$^{\langle\rangle}$ ⟨state, *state*⟩
    else
      let *session* := *session* +$^{\langle\rangle}$ ⟨state, ⟨⟩⟩
    let *sessionID* := ν_4
    let s'.sessions := s'.sessions +$^{\langle\rangle}$ ⟨*sessionID*, *session*⟩
    let *authUrl*.parameters := *authUrl*.parameters $\cup$ *parameters*
    let *headers* := ⟨⟨Location, *authUrl*⟩⟩
    let *headers* := *headers* +$^{\langle\rangle}$ ⟨Set-Cookie, [⟨__Host, sessionID⟩ : ⟨*sessionID*, ⊤, ⊤, ⊤⟩]⟩
    let m' := enc$_s$(⟨HTTPResp, m.nonce, 303, *headers*, ⟨⟩⟩, k)
    ↪ → Use status code 303 (see other) as recommended by the OAuth 2.0 Security BCP
    let *leakUrl* ← {⊤, ⊥}
    if *leakUrl* $\equiv$ ⊤ **then**
      let *leak* := ⟨LEAK, *authUrl*⟩
      ↪ → This URL leaks because it can be a custom scheme request to a native app
      let a' ← IPs
      stop ⟨⟨f, a, m'⟩, ⟨f, a', *leak*⟩⟩, s'
    else
      stop ⟨⟨f, a, m'⟩⟩, s'

```
44:     else if m.path ≡ /directPost ∧ m.method ≡ POST then
                                    Process back-channel response
45:         let vpToken := m.body[vp_token]
46:         let state := m.body[state]
47:         if vpToken ≡ ⟨⟩ then
48:             stop
49:         let credential, nonce, aud such that
              ↪ ⟨credential, nonce, aud⟩ ≡ extractmsg(vpToken)
              ↪ if possible; otherwise stop
50:         call VERIFY_PRESENTATION(m, s′, presentation, nonce)
      ↪     → Verify presentation but do not check nonce because it cannot be connected to a session here
51:         let sessionID, host such that
              ↪ ⟨sessionID, ⟨⟨nonce, nonce⟩, ⟨host, host⟩, ⟨state, state⟩⟩⟩ ∈ s′.sessions
              ↪ if possible; otherwise stop
      ↪     → Search with nonce for an existing session. The state parameter can be empty.
52:         if m.body[iss] ≠ host ∨ m.body[state] ≠ state then
      ↪     → Mix-up attack mitigation and state parameter verification (aud parameter is already checked
    in VERIFY_PRESENTATION method)
53:             stop
54:         let s′.sessions[sessionID] := s′.sessions[sessionID] +^⟨⟩ ⟨vp_token, vpToken⟩
55:         let useRedirectUri ← {⊤, ⊥}
56:         let s′.sessions[sessionID] := s′.sessions[sessionID] +^⟨⟩
              ↪ ⟨useRedirectUri, useRedirectUri⟩    → Important for the security properties
57:         if useRedirectUri ≡ ⊤ then
58:             let respCode := ν1
59:             let parameters := ⟨⟨response_code, respCode⟩⟩
60:             let body := ⟨⟨redirect_uri, ⟨URL, S, m.host, /redirect, parameters, ⊥⟩⟩⟩
                  ↪     → This URL cannot leak because it is implemented as a verified app link
61:             let s′.sessions[sessionID] := s′.sessions[sessionID] +^⟨⟩ ⟨response_code, respCode⟩
62:         else
63:             let body := ⟨⟩
64:         let m′ := enc_s(⟨HTTPResp, m.nonce, 200, ⟨⟩, body⟩, k)
65:         stop ⟨⟨f, a, m′⟩⟩, s′
66:     else if m.path ≡ /redirect then
                                    Receive authentication response
67:         let sessionID := m.header[Cookie][⟨__Host, sessionID⟩]
68:         if sessionID ∉ s′.sessions then
69:             stop
70:         let session := s′.sessions[sessionID]
71:         if session[response_code] ≠ ⟨⟩ then
              ↪     → Same-Device Flow with response_mode direct_post and redirect_uri
72:             if m.parameters[response_code] ≠ session[response_code] then
73:                 stop
74:             let presentation := session[vp_token]
75:         else if m.body[iss] ≠ ⟨⟩ ∧ m.body[vp_token] ≠ ⟨⟩ then
                  ↪     → Same-Device Flow with Authentication Response via fragment
76:             if session[host] ≠ m.body[iss] ∨ session[state] ≠ m.body[state] then
              ↪     → Check the iss parameter because it is recommended by the OAuth 2.0 Security BCP
77:                 stop
78:             let session := session +^⟨⟩ ⟨vp_token, m.body[vp_token]⟩
                  ↪     → The vp_token is only saved here to simplify the session integrity proof
79:             let presentation := m.body[vp_token]
```

```
80:         else if session[vp_token] ≢ ⟨⟩ then
                ↪      → Cross-Device Flow (URL contains no parameters)
81:             if session[response_code] ≢ ⟨⟩ then
82:                 stop
83:             let presentation := session[vp_token]
84:         else if m.method ≡ GET then
85:             let m′ := encs(⟨HTTPResp, m.nonce, 200, ⟨⟩, ⟨script_verifier_get_fragment⟩⟩, k)
86:             stop ⟨⟨f, a, m′⟩⟩, s′
87:         else
88:             stop
89:         let nonce := session[nonce]
90:         call VERIFY_PRESENTATION(m, s′, presentation, nonce)
91:         let credential, pNonce, aud such that
              ↪ ⟨credential, pNonce, aud⟩ ≡ extractmsg(presentation)
              ↪ if possible; otherwise stop
92:         let username, iss, pubKey such that
              ↪ ⟨username, iss, pubKey⟩ ≡ extractmsg(credential)
              ↪ if possible; otherwise stop
93:         let serviceToken := v2
94:         let session[serviceTokenId] := serviceToken
95:         let session[userInfo] := ⟨username, iss, session[host]⟩
96:         let url := ⟨URL, S, m.host, /, ⟨⟩, ⊥⟩
97:         let headers := ⟨⟨Location, url⟩⟩
98:         let headers := headers +⟨⟩ ⟨Set-Cookie, [serviceToken: ⟨serviceToken, ⊤, ⊤, ⊤⟩]⟩
99:         let m′ := encs(⟨HTTPResp, m.nonce, 303, headers, ⟨⟩⟩, k)
100:        stop ⟨⟨f, a, m′⟩⟩, s′
101:    stop
```

Algorithm 10 Relation of a verifier R^v: Function to verify a credential.

```
1: function VERIFY_PRESENTATION(m, s′, presentation, nonce)
2:     let credential, pNonce, aud such that
         ↪ ⟨credential, pNonce, aud⟩ ≡ extractmsg(presentation)
         ↪ if possible; otherwise stop
3:     if aud ≢ m.host then
4:         stop
5:     if pNonce ≢ nonce then
6:         stop
7:     let username, iss, pubKey such that
         ↪ ⟨username, iss, pubKey⟩ ≡ extractmsg(credential)
         ↪ if possible; otherwise stop
8:     if checksig(presentation, pubKey) ≡ ⊥ then
9:         stop
10:    if checksig(credential, s′.issuers[iss]) ≡ ⊥ then
11:        stop
```

Algorithm 11 Relation of *script_verifier_get_fragment*.

Input: ⟨*tree*, *docnonce*, *scriptstate*, *scriptinputs*, *cookies*, *localStorage*, *sessionStorage*, *ids*, *secrets*⟩
1: **let** *url* := GETURL(*tree*, *docnonce*)
2: **let** *url′* := ⟨URL, S, *url*.host, /redirect, ⟨⟩, ⊥⟩
3: **let** *command* := ⟨FORM, *url′*, POST, *url*.fragment, ⊥⟩
4: **stop** ⟨*scriptstate*, *cookies*, *localStorage*, *sessionStorage*, *command*⟩

Algorithm 12 Relation of *script_verifier_index*.

Input: ⟨*tree*, *docnonce*, *scriptstate*, *scriptinputs*, *cookies*, *localStorage*, *sessionStorage*, *ids*, *secrets*⟩
1: **let** *switch* ← {issuance, link}
2: **if** *switch* ≡ issuance **then**
3: **let** *url* := GETURL(*tree*, *docnonce*)
4: **let** *url′* := ⟨URL, S, *url*.host, /start, ⟨⟩, ⊥⟩
5: **let** *command* := ⟨START, *url′*, ⟨⟩⟩
6: **stop** ⟨*scriptstate*, *cookies*, *localStorage*, *sessionStorage*, *command*⟩
7: **else**
8: **let** *protocol* ← {P, S}
9: **let** *host* ← Doms
10: **let** *path* ← $\mathbb{S}$
11: **let** *parameters* ← $[\mathbb{S} \times \mathbb{S}]$
12: **let** *fragment* ← $\mathbb{S}$
13: **let** *url* := ⟨URL, *protocol*, *host*, *path*, *parameters*, *fragment*⟩
14: **let** *command* := ⟨HREF, *url*, ⊥, ⊥⟩
15: **stop** ⟨*scriptstate*, *cookies*, *localStorage*, *sessionStorage*, *command*⟩

A.5 Web Browser Extension

In this work, we needed to extend the web browser model to secure the cross-device flow and the pre-authorized code flow. The problem is that an attacker can start a cross-device flow on their computer and send the authentication request to a victim who authenticates the request. As a result, the attacker is logged in under the identity of an honest user. Previous work [1] and [13] did not find a practical and provable solution to this problem. In addition, the user must be able to tell in the pre-authorized code flow whether they have initiated the flow or not. Otherwise, an attacker could send a credential offer with a pre-authorized code tied to their identity to the honest user's wallet. This violates the Issuance Session Integrity security property because the user would be issued a credential with the attacker's identity.

Therefore, in this work, we decided to model a perfect user who always knows which flows they have started and with which parameters, e.g., domain and nonce or domain and pre-authorized code. We model this by introducing a new script command in the browser model, which is similar to a FORM command with the POST method. The difference is that the POST requests must be sent to the same domain that the script is running on. This is important so that only scripts from the honest party can initiate flows. Additionally, a new reference is used in this request so that the response can be identified and the parameters of the response can be stored in a new browser state called "authStarted". After that, the wallet can use the method "validateRequest($\langle domain, nonce \rangle$)" to check if the user has started the flow or not. The extension is similar to the one described in A.1.2 of [10], but has some major differences to fit the problem in this work.

Definition A.5.1 (Web Browser State Extension)
A state $s \in Z_{\text{webbrowser}}$ of a browser b is extended by the following sub term in this work:

$$started$$

with $started \in \mathcal{T}_{\mathcal{N}}$
In the initial state s_0^b of b the variable is initialized with the following value: $s_0^b.\texttt{started} = \langle\rangle$.
Everything else is defined as in Definition 33 of [7].

Algorithm 13 Web Browser Model: Execute a script.

```
1: function RUNSCRIPT(w̄, d̄, a, s′)
                ⋮
17:     let docorigin := s′.d̄.origin
18:     switch command do
19:         case ⟨START, url, data⟩
20:             if url.host ≠ docorigin.host ∨ url.protocol ≠ S then
21:                 stop
        ———— The following code is like the FORM command but adjusted to this use case ————
22:             let w̄′ := GETNAVIGABLEWINDOW(w̄, ⊥, ⊥, s′)
23:             let reference := ⟨START, s′.w̄′.nonce⟩
24:             let parameters := url.parameters
25:             let origin := docorigin
26:             let req := ⟨HTTPReq, v4, POST, url.host, url.path, parameters, ⟨⟩, data⟩
27:             let s′ := CANCELNAV(reference, s′)
28:             call HTTP_SEND(reference, req, url, origin, referrer, referrerPolicy, a, s′)
29:         case ⟨HREF, url, hrefwindow, noreferrer⟩
                ⋮
```

Algorithm 14 Web Browser Model: Process an HTTP response.

```
1: function PROCESSRESPONSE(response, reference, request, requestUrl, a, f, s′)
                ⋮
7:     if Referer ∈ request.headers then
8:         let referrer := request.headers[Referer]
9:     else
10:        let referrer := ⊥
11:    if π1(reference) ≡ START then
12:        let redirectUrl := response.headers[Location]
13:        let preAuthCodeCase ← {⊤, ⊥}
14:        if preAuthCodeCase ≡ ⊤ then
15:            let s′.started := s′.started +⟨⟩
                 ↪ ⟨request.host, redirectUrl.parameters[credential_offer].2⟩
16:        else
17:            let s′.started := s′.started +⟨⟩ ⟨request.host, redirectUrl.parameters[nonce]⟩
18:    if Location ∈ response.headers ∧ response.status ∈ {303, 307} then
                ⋮
```

B Formal Security Properties

B.1 Presentation Authentication

Informally the Presentation Authentication security property means that an attacker cannot log in as a user at an honest verifier as long as certain parties involved in the login processes are not corrupted. An attacker is successfully logged in if they have obtained a service token for a user. The Definition B.1.2 is close to the definition of the authentication property in [11], but has some subtle differences that are very important in this analysis.

Definition B.1.1 (Authentication Request is Validated)
For a run ρ of a Verifiable Credentials web system $\mathcal{VCWS}^n$ with a network attacker we say that the authentication request is validated by a wallet w if there is a processing step Q in ρ with

$$Q = (S, E, N) \xrightarrow{e_{in} \to w} (S', E', N')$$

$e_{\text{in}} = \langle x, y, m \rangle$, $\top$ is selected for *validateAuthRequest* in Line 78 of Algorithm 4, $m_{\text{dec}}.\texttt{body}[\texttt{response\_uri}].\texttt{host} \equiv d_v$, and $m_{\text{dec}}.\texttt{body}[\texttt{nonce}] \equiv nonce$ with $m_{\text{dec}} = \mathsf{dec_a}(m, k)$ and $k \in \mathcal{N}$. For this event we write $\mathsf{authRequestValidated}^Q_\rho(w, d_v, nonce)$.

Before we can define the Presentation Authentication security property we first need to specify the set of all wallets belonging to a browser b: $W_b = \{w \in \mathsf{Wallets} | b = \mathsf{browserOfWallet}(w)\}$.

Definition B.1.2 (Presentation Authentication Property)
Let $\mathcal{VCWS}^n$ be a Verifiable Credentials web system with a network attacker. The web system is secure w.r.t. presentation authentication iff for every run ρ of $\mathcal{VCWS}^n$, every configuration (S^j, E^j, N^j) in ρ, every $v \in \mathsf{Verifiers}$ that is honest in S^j, every $i \in \mathsf{Issuers}$ that is honest in S^j, every $d_i \in \mathsf{dom}(i)$, every $x \in \mathcal{T}_\mathcal{N}$, every $u \in \mathsf{ID}$, the browser b owning u is not fully corrupted in S^j (i.e., the value of *isCorrupted* is not `FULLCORRUPT`), all wallets in W_b are honest in S^j, for every login session $lsid \in \mathcal{N}$, for every service token $n \in \mathcal{N}$ recorded in $S^j(v).\texttt{sessions}[lsid].\texttt{serviceTokenId} \equiv n$ and $S^j(v).\texttt{sessions}[lsid].\texttt{userInfo} \equiv \langle u, d_i, x \rangle$, and if $S^j(v).\texttt{sessions}[lsid].\texttt{useRedirectUri} \equiv \bot$ there exists a processing step $Q = (S^k, E^k, N^k) \to (S^{k+1}, E^{k+1}, N^{k+1})$ with $k < j$ in which $\mathsf{authRequestValidated}^Q_\rho(w', d_v, nonce)$ with $w' \in W_b$, $d_v \in \mathsf{dom}(v)$, and $nonce = S^j(v).\texttt{sessions}[lsid].\texttt{nonce}$ is true, it holds that n is not derivable from the attacker's knowledge in S^j (i.e., $n \notin d_\emptyset(S^j(\mathsf{attacker}))$).

B.2 Issuance Authentication

At a high level, the Issuance Authentication security property guarantees that an attacker cannot use a credential with an honest user's identity from an issuer as long as certain parties involved in the issuance are not corrupted. An attacker can use such a credential if they know the private key to the public key embedded in the credential. The Definition B.2.1 is close to the definition of the authentication property in [11], but has some important additions.

Definition B.2.1 (Issuance Authentication Property)
Let $\mathcal{VCWS}^n$ be a Verifiable Credentials web system with a network attacker. The web system is secure w.r.t. issuance authentication iff for every run ρ of $\mathcal{VCWS}^n$, every configuration (S^j, E^j, N^j) in ρ, every $i \in \mathsf{Issuers}$ that is honest in S^j, every $d_i \in \mathsf{dom}(i)$, every $u \in \mathsf{ID}$ and the browser b owning u is not fully corrupted in S^j (i.e., the value of *isCorrupted* is not `FULLCORRUPT`), all wallets in W_b are honest in S^j, every private key $p \in \mathcal{N}$, every credential $c \in \mathcal{T}_{\mathcal{N}}$ with $c \equiv \mathsf{sig}(\langle u, d_i, \mathsf{pub}(p)\rangle, S^j(i).\texttt{signingKey})$, every issuance session $issid \in \mathcal{N}$, $code \equiv S^j(w).\texttt{sessions}[issid].\texttt{code}$, $S^j(i).\texttt{codes}[code].2 \equiv redirectUri$ with $redirectUri.\texttt{host} \in \mathsf{dom}(w)$ and $w \in \mathsf{Wallets}$ being an honest wallet in S^j, and if $S^j(w).\texttt{sessions}[issid].\texttt{pre-authorized\_code} \not\equiv \langle\rangle$ we have that $S^j(w).\texttt{sessions}[issid].\texttt{use\_user\_pin} = \top$, it holds that p is not derivable from the attacker's knowledge in S^j (i.e., $p \notin d_\emptyset(S^j(\mathsf{attacker}))$).

B.3 Presentation Session Integrity

Intuitively, session integrity in the OID4VP flow means two things. First, a user must explicitly express a desire to log in to a verifier, and second, the user must be logged in with the identity they choose during credential selection in the wallet. This also means that an honest user cannot be logged in under the attacker's identity after the presentation flow.

To formalize the session integrity property we need to define certain events. The first event is the start of the flow which is the execution of the script *script_verifier_index*. The second event is the selection of a credential by the user and the third event is the issuance of a service token by the verifier. The formalized events can be found in the following definitions. The definitions B.3.1, B.3.3, B.3.4, and B.3.5 are close to the definition of the session integrity property for authentication in [11] but have some subtle differences that are very important in this analysis.

Definition B.3.1 (User Started a Login Flow)
For a run ρ of a Verifiable Credentials web system $\mathcal{VCWS}^n$ with a network attacker we say that the user of the browser b started a login session identified by a nonce *lsid* at the verifier v in a processing step Q in ρ if first, the browser b was triggered to select a document loaded from an origin of v, executed the script *script_verifier_index* in that document, and in that script, executed the Line 6 of Algorithm 12, and second, v sends an HTTPS response corresponding to the HTTPS request sent by b in Q and in that response, there is a header of the form $\langle \texttt{Set-Cookie}, [\langle \texttt{\_\_Host}, \texttt{sessionID}\rangle : \langle lsid, \top, \top, \top\rangle]\rangle$. For this event we write $\mathsf{started}^Q_\rho(b, v, lsid)$.

Definition B.3.2 (User Chooses Credential)
For a run ρ of a Verifiable Credentials web system $\mathcal{VCWS}^n$ with a network attacker we say that the user of a wallet w authenticates to a verifier v in a login session *lsid* using an identity *id* with $\mathsf{ownerOfID}(id) = \mathsf{browserOfWallet}(w)$, the audience $aud \in \mathsf{dom}(v)$, and the nonce $nonce \in \mathcal{N}$ if there is a processing step $Q = (S, E, N) \rightarrow (S', E', N')$ in ρ in which the wallet w selects a credential with the identity *id* (Line 76 of Algorithm 4) and sends the credential in a presentation with the audience *aud* and the nonce $nonce \equiv S(v).\texttt{sessions}[lsid].\texttt{nonce}$ (Line 84 of Algorithm 4) to v. For this event we write $\mathsf{chooseCredential}_\rho^Q(v, w, lsid, id)$.

Definition B.3.3 (User is Logged In)
For a run ρ of a Verifiable Credentials web system $\mathcal{VCWS}^n$ with a network attacker we say that a browser b was authenticated to a verifier v using a credential with the identity *id* issued by an issuer i in a login session identified by a nonce *lsid* in a processing step Q in ρ with

$$Q = (S, E, N) \xrightarrow[v \rightarrow E_{out}]{} (S', E', N')$$

and some event $\langle y, y', m\rangle \in E_{out}$ if m is an HTTPS response to an HTTPS request sent by b to v and we have that in the header of m there is a header of the form $\langle \texttt{Set-Cookie}, [\texttt{serviceToken}\colon \langle serviceToken, \top, \top, \top\rangle]\rangle$ for some nonce *serviceToken* such that $S(v).\texttt{sessions}[lsid].\texttt{serviceTokenId} \equiv serviceToken$ and $S(v).\texttt{sessions}[lsid].\texttt{userInfo} \equiv \langle id, d_i, d_w\rangle$ with $d_i \in \mathsf{dom}(i)$, $d_w \in \mathsf{dom}(w)$. For this event we write $\mathsf{loggedIn}_\rho^Q(b, v, i, id, lsid)$.

Definition B.3.4 (Verifier Leaked Authorization Request)
Let $\mathcal{VCWS}^n$ be a Verifiable Credentials web system with a network attacker. For a run ρ of $\mathcal{VCWS}^n$ with a processing step Q, a browser $b \in \mathsf{B}$, a verifier $v \in \mathsf{Verifiers}$, an issuer $i \in \mathsf{Issuers}$, and identity *id*, a login session *lsid*, and $\mathsf{loggedIn}_\rho^Q(b, v, i, id, lsid)$, we say that v leaked the authorization request for *lsid*, if there is a processing step $Q' = (S, E, N) \xrightarrow[v \rightarrow E_{out}]{} (S', E', N')$ in ρ prior to Q such that in Q', v executes Line 41 of Algorithm 9 and there is a nonce $nonce \in \mathcal{N}$ and an event $\langle x, y, m\rangle \in E_{out}$ with $m.1 \equiv \texttt{LEAK}$ and $m.2.\texttt{parameters}[\texttt{nonce}] \equiv nonce$ such that $S'(v).\texttt{sessions}[lsid][\texttt{nonce}] \equiv nonce$.

Definition B.3.5 (Presentation Session Integrity Property)
Let $\mathcal{VCWS}^n$ be a Verifiable Credentials web system with a network attacker. We say that $\mathcal{VCWS}^n$ is secure w.r.t. presentation session integrity iff for every run ρ of $\mathcal{VCWS}^n$, every processing step $Q = (S, E, N) \rightarrow (S', E', N')$ in ρ, every $v \in \mathsf{Verifiers}$ that is honest in S, every $i \in \mathsf{Issuers}$ that is honest in S, every identity *id*, every browser b that is honest in S, every wallet w with $b = \mathsf{browserOfWallet}(w)$ that is honest in S, for every nonce *lsid*, v did not leak the authorization request for *lsid* (see Definition B.3.4), and $\mathsf{loggedIn}_\rho^Q(b, v, i, id, lsid)$, we have that there exists a processing step Q' in ρ (before Q) such that $\mathsf{chooseCredential}_\rho^{Q'}(v, w, lsid, id)$ and there exists a processing step Q'' in ρ (before Q) such that $\mathsf{started}_\rho^{Q''}(b, v, lsid)$.

B.4 Issuance Session Integrity

At a high level, session integrity in the OID4VCI flow means that a user has explicitly expressed a desire to be issued a credential and that the identity in the credential is the one they used to authenticate to the issuer. This also means that an honest user cannot get a credential with the attacker's identity into their wallet during credential issuance.

To formalize this property, we need to define four events. The first and the second events describe the user's desire to issue a credential. This is done either by executing the script *script_issuer_form* to start the pre-authorized code flow or by executing the script *script_wallet_index* to start an authorization code flow. The third event is the authentication of the user to the issuer during an authorization code flow, where the user chooses the identity of the credential to be issued. The fourth and final event is the issuance and storage of the credential in a wallet.

The following definitions are similar to the definition of the session integrity property for authentication in [11] but have some differences that are very important in this analysis.

Definition B.4.1 (User Started Authentication Code Flow)
For a run ρ of a Verifiable Credentials web system $\mathcal{VCWS}^n$ with a network attacker we say that the user of the browser b started an authentication code flow identified by an issuance session id *issid* at the wallet w in a processing step $Q = (S, E, N) \rightarrow (S', E', N')$ in ρ if first the browser b was triggered to select a document loaded from an origin of w, executed the script *script_wallet_index* in that document, and in that script, executed the Line 7 of Algorithm 5, and second, w sends an HTTPS response corresponding to the HTTPS request sent by b in Q and in that response, there is a header of the form $\langle$`Set-Cookie`, [$\langle$`__Host`, `sessionID`$\rangle$: $\langle$*issid*, $\top$, $\top$, $\top\rangle$]$\rangle$. For this event we write $\mathsf{startedCodeFlow}^{Q}_{\rho}(b, w, issid)$.

Definition B.4.2 (User Started Pre-Authorized Code Flow)
For a run ρ of a Verifiable Credentials web system $\mathcal{VCWS}^n$ with a network attacker we say that the user of the browser b started a pre-authorized code flow identified by a nonce *preAuthorizedCode* at the issuer i in a processing step $Q = (S, E, N) \rightarrow (S', E', N')$ in ρ if first the browser b was triggered to select a document loaded from an origin of the issuer i, executed the script *script_issuer_form* in that document with the script state *scriptstate*.`1` = `/startCredOffer`, in that script selected the identity *id* in Line 4 of Algorithm 3, selected $d_w \in \mathsf{dom}(w)$ in Line 6 of Algorithm 3, and executed the Line 11 of Algorithm 3, and second, i sends an HTTPS response corresponding to the HTTPS request sent by b in Q and in that response, there is a header of the form $\langle$`Location`, $\langle$`URL`, `S`, d_w, `/vci/credentialOffer`, *parameters*, $\bot\rangle\rangle$ with *parameters* = $\langle$`credential_offer`, $\langle d_i$, *preAuthorizedCode*, *useUserPin*$\rangle\rangle$, *useUserPin* $\in \{\top, \bot\}$, $d_i \in \mathsf{dom}(i)$ and we have $S(w)$.`sessions`[*issid*].`pre-authorized_code` = *preAuthorizedCode* with *preAuthorizedCode* $\in \mathcal{N}$.
For this event we write $\mathsf{startedPreAuthCodeFlow}^{Q}_{\rho}(b, i, w, issid, id)$.

Definition B.4.3 (User Authenticated at Issuer)
For a run ρ of a Verifiable Credentials web system $\mathcal{VCWS}^n$ with a network attacker we say that the user of the browser b authenticated to an issuer i using an identity *id* for an issuance session identified by a nonce *issid* at a wallet w if there is a processing step $Q = (S, E, N) \rightarrow (S', E', N')$ in ρ in which the browser b was triggered to select a document loaded from an origin of the issuer i, executed the script *script_issuer_form* in that document with the script state *scriptstate*.`1` = `/authentication`, selected in that script the identity *id* in Line 4 of Algorithm 3 and we have that

- the *scriptstate* of that document contains a nonce *code_challenge* such that *scriptstate*.`2[code_challenge]` = *code_challenge*
- $S(w)$.`sessions`[*issid*].`code_challenge` = *code_challenge*

For this event we write $\mathsf{authenticated}^{Q}_{\rho}(b, i, w, id, issid)$.

Definition B.4.4 (Credential Stored)
For a run ρ of a Verifiable Credentials web system $\mathcal{VCWS}^n$ with a network attacker we say that the user of the browser b stored a credential from the issuer i containing the identity id in an issuance session $issid$ at a wallet w in a processing step Q in ρ with:

$$Q = (S, E, N) \xrightarrow{e_{in} \to w} (S', E', N')$$

if the event $e_{\text{in}} \equiv \langle x, y, m \rangle$ and m being an HTTPS response to an HTTPS request from w to i containing a body with $c \equiv m_{\text{dec}}.\texttt{body[credential]}$ or $c \in m_{\text{dec}}.\texttt{body[credential\_responses].credentials}$ ($m_{\text{dec}} = \mathsf{dec_a}(m, k)$ and $k \in \mathcal{N}$) and c being a credential of the form $\langle id, iss, pubKey \rangle \equiv \mathsf{extractmsg}(c)$ with $iss \in \mathsf{dom}(i)$ we have that:

- $iss \equiv S(w).\texttt{sessions}[issid].\texttt{host}$
- $c \in S'(w).\texttt{credentials}$
- $c \in S'(w).\texttt{sessions}[issid].\texttt{credentials}$

For this event we write $\mathsf{stored}_\rho^Q(b, w, i, id, issid)$.

Definition B.4.5 (Issuance Session Integrity Property)
Let $\mathcal{VCWS}^n$ be a Verifiable Credentials web system with a network attacker. We say that $\mathcal{VCWS}^n$ is secure w.r.t. issuance session integrity iff for every run ρ of $\mathcal{VCWS}^n$, every processing step $Q = (S, E, N) \to (S', E', N')$ in ρ, every $i \in \mathsf{Issuers}$ that is honest in S, every identity id, every browser b that is honest in S, every wallet w with $b = \mathsf{browserOfWallet}(w)$ that is honest in S, every nonce $issid$, and $\mathsf{stored}_\rho^Q(b, w, i, id, issid)$, we have that:

1. there exists a processing step Q'' in ρ (before Q) such that $\mathsf{startedCodeFlow}_\rho^{Q''}(b, w, issid)$
2. and there exists a processing step Q' in ρ (before Q) such that $\mathsf{authenticated}_\rho^{Q'}(b, i, w, id, issid)$

OR

3. there exists a processing step Q''' in ρ (before Q) such that $\mathsf{startedPreAuthCodeFlow}_\rho^{Q'''}(b, i, w, issid, id)$.

C Proof of Security Properties

This chapter shows that the Verifiable Credentials web system $\mathcal{VCWS}^n$ fulfills the Presentation Authentication (Definition B.1.2) the Issuance Authentication (Definition B.2.1), Presentation Session Integrity (Definition B.3.5), and the Issuance Session Integrity (Definition B.4.5) security properties.

C.1 Lemmas

Lemma 1 (Verifier session id does not leak)
Let $\mathcal{VCWS}^n$ be a Verifiable Credentials web system with a network attacker. For every run ρ of $\mathcal{VCWS}^n$, every configuration (S^j, E^j, N^j) in ρ, every browser b that is honest in S^j, every verifier v that is honest in S^j, every nonce $lsid \in \mathcal{N}$ and a processing step Q in ρ we have that if b sends an HTTPS request to v, v selects a session id $lsid$ in Q (Line 31 of Algorithm 9), and answers with an HTTPS response that contains a cookie of the form $\langle \texttt{Set-Cookie}, [\langle \texttt{\_\_Host}, \texttt{sessionID} \rangle : \langle lsid, \top, \top, \top \rangle] \rangle$ (Line 35 of Algorithm 9) then $lsid$ is not derivable from the attackers knowledge in S^j (i.e., $lsid \notin d_\emptyset(S^j(\mathsf{attacker}))$).

Proof. From the precondition, we know that the honest verifier v sends an encrypted HTTPS response containing a cookie of the form $\langle \texttt{Set-Cookie}, [\langle \texttt{\_\_Host}, \texttt{sessionID} \rangle : \langle lsid, \top, \top, \top \rangle] \rangle$ to an honest browser b. According to Lemma 1 in [7] this response will not leak to any party other than v and b.

The browser uses the host of the corresponding HTTPS request to store the cookie under the domain of v in Line 4 of Algorithm 8. The cookie has two flags: Firstly, the secure flag which means that it will only ever be sent out over an encrypted connection, and since it is stored under the domain of v it will only be sent out to v over an encrypted connection (Line 4 in Algorithm 4 in [7]). Lemma 1 of [7] guarantees that these messages cannot be decrypted by the attacker. Secondly, the cookie uses the HTTP-only flag which means that it cannot be leaked by a script because it cannot be accessed by scripts (Line 3 of Algorithm 7 in [7]). In summary, b will not send the cookie to any other party than v.

Looking at the relation of the verifier it can be seen that there are four endpoints where messages are received. The first endpoint (Line 2 of Algorithm 9) and the second endpoint (Line 5 of Algorithm 9) do not process the cookie header of incoming requests and do not extract data from the state. This means that they cannot leak the $lsid$ session id. The third endpoint (Line 44 of Algorithm 9) does also not process the cookie header but does extract the session from the state via the nonce of the authorization response (**let** $lsid, host$ **such that** $\langle lsid, \langle \langle \texttt{nonce}, nonce \rangle, \langle \texttt{host}, host \rangle, \langle \texttt{state}, state \rangle \rangle \rangle \in s'.\texttt{sessions}$ **if possible**; **otherwise stop**). There are no requests made to other servers and the $lsid$ is also not included in the response in Line 64 of Algorithm 9. The fourth and

last endpoint (Line 66 of Algorithm 9) does process the cookie header and extracts the session id ($lsid = m.\texttt{header}[\texttt{Cookie}][\langle \texttt{\_\_Host}, \texttt{sessionID}\rangle]$). In Line 70 of Algorithm 9 *lsid* is used to get the session info from the state ($session := s'.\texttt{sessions}[lsid]$). After that line, the *lsid* nonce is not used anymore. This means in particular that it is not included in any further requests or the response.

Since the *lsid* nonce is not leaked by the browser and not leaked by the verifier, the attacker cannot derive *lsid* from its knowledge. □

Lemma 2 (Wallet Contains only Honest Identities)
Let $\mathcal{VCWS}^n$ be a Verifiable Credentials web system with a network attacker. If for every run ρ of $\mathcal{VCWS}^n$, every processing step $Q = (S, E, N) \rightarrow (S', E', N')$ in ρ every $i \in \mathsf{Issuers}$ that is honest in S, every identity *id*, every browser b that is honest in S, every wallet w with $b = \mathsf{browserOfWallet}(w)$ that is honest in S, every nonce *issid*, and $\mathsf{stored}^Q_\rho(b, w, i, id, issid)$ (Definition B.4.4), we have that $\mathsf{browserOfWallet}(w) = b = \mathsf{ownerOfID}(id)$.

Proof. The first part $\mathsf{browserOfWallet}(w) = b$ follows immediately from the precondition. For the second part, we have to show that $b = \mathsf{ownerOfID}(id)$. The preconditions says that $\mathsf{stored}^Q_\rho(b, w, i, id, issid)$ is true. From the Issuance Session Integrity property (Definition B.4.5) it is known that either $\mathsf{authenticated}^Q_\rho(b, i, w, id, issid)$ or $\mathsf{startedPreAuthCodeFlow}^Q_\rho(b, i, w, issid, id)$ happened in a previous processing step. Both of these events involve the execution of the script *script_issuer_form* (Algorithm 3) and specifically the Line 4 in which b selects one of its identities. Since the browser chooses the identity, $b = \mathsf{ownerOfID}(id)$ must be true. It follows that $\mathsf{browserOfWallet}(w) = b = \mathsf{ownerOfID}(id)$. □

C.2 Proof of Presentation Authentication

This section contains the formal proof of the Presentation Authentication security property (Definition B.1.2). The property is shown directly by proving that the property holds in a Verifiable Credentials web system $\mathcal{VCWS}^n$.

Lemma 3 (Presentation Authentication Holds)
Definition B.1.2 holds.

Proof. By Definition B.1.2 we have an arbitrary *lsid* with $S^j(v).\texttt{sessions}[lsid][\texttt{serviceTokenId}] \equiv n$ where $n \in \mathcal{N}$ is the service token, which cannot be derived by the attacker. The service token is connected via the login session *lsid* with the user id u ($S^j(v).\texttt{sessions}[lsid].\texttt{userInfo} \equiv \langle u, d_i, x\rangle$ with $d_i \in \mathsf{dom}(i)$ and $x \in \mathcal{T}_{\mathcal{N}}$). The only place a service token is created is at the verifier's redirect endpoint in Line 93 of Algorithm 9. There are two ways an attacker could obtain such a service token: First, the service token is leaked by an honest party, and second, the attacker calls successfully the verifier's redirect endpoint (Line 66 of Algorithm 9). In the following, we will show that both of these cases are not possible.

This paragraph shows that the service token does not leak if it is sent to an honest party. Looking at the relations of the parties it can be seen that an honest wallet, issuer, and verifier will not send a request to the verifier's redirect endpoint (Line 66 of Algorithm 9). Out of these three

parties, only the wallet ever sends HTTPS requests. The wallet sends requests to the token endpoint (Line 20 of Algorithm 4 and Line 60 of Algorithm 4), credential endpoint (Line 19 of Algorithm 4), batch credential endpoint (Line 29 of Algorithm 4) and deferred credential endpoint (Line 16 of Algorithm 4) of the issuer as well as to the direct-post endpoint of the verifier (Line 100 of Algorithm 4) but not to the verifier's redirect endpoint. The service token is created in Line 93 of Algorithm 9 in the redirect endpoint of the verifier. The honest browser is the only party receiving the service token in an encrypted response answering an encrypted request. The browser is storing the cookie with the secure and HTTP-only flag under the origin of the verifier (Line 4 of Algorithm 8 in [7]) so it will only be sent over an encrypted connection to the honest verifier and it cannot be accessed by a script. The service token is transferred via an encrypted message. This makes it impossible for the attacker to derive the token from the message because they cannot break the encryption algorithm, the encryption keys are not leaked, and the domain to public key mapping is correctly preconfigured in the initial state of each process. A more detailed proof of why HTTPS messages between a browser and a generic HTTPS server do not leak can be found in Lemma 1 of [7]. Looking at the relation of the verifier, it can be seen that the service token cookie is never retrieved from a request, so it cannot be leaked if it is sent to v. Since b and v do not leak the service token and no other honest party receives it, the service token will not be leaked to the attacker by an honest party.

The second way for the attacker to obtain the service token is to successfully use the verifier's redirect endpoint (Line 66 of Algorithm 9). To use this endpoint, the attacker has to successfully pass the if-else-if conditions in Line 71 of Algorithm 9:

1. ***session*[response_code] ≢ ⟨⟩ (Line 71 of Algorithm 9):** The first possibility is that the attacker knows the session ID and the response code of a session in which v has already stored the presentation of an honest user. This is the case for the same-device flow that uses the direct-post endpoint (Line 44 of Algorithm 9) to transmit the presentation to the verifier. This means v chooses *useRedirectUri* = $\top$ in Line 55 of Algorithm 9.

   By tracking the response code, it can be seen that it is not being leaked to the attacker by an honest party. The response code is created in Line 58 of Algorithm 9 and stored in the session identified by *lsid* in the subsequent lines. The response code is included in the encrypted response of the direct-post endpoint to $w \in W_b$ (Line 44 of Algorithm 9). Since the presentation is sent to the endpoint only w could have sent it because as argued below (2) the attacker cannot obtain a credential with an honest user's identity. This means the response code is sent to w. The attacker could send a request to the direct-post endpoint and store their presentation in an honest user's session because they know the nonce from the authorization request (*authUrl*.parameters[nonce]). This is due to the fact that *authUrl* leaks (Line 41 of Algorithm 9) but it does not violate the Presentation Authentication security property because the attacker has to be logged in under an honest user's identity. Looking at the other endpoints of a verifier it can be seen that the response code is not leaked. The start-script endpoint in Line 2 of Algorithm 9 and start endpoint in Line 5 of Algorithm 9 do never retrieve the response code from the session and the redirect endpoint (Line 66 of Algorithm 9) does only retrieve the response code to compare the value with parameters in the request (m.parameters[response_code] ≢ *session*[response_code]). Specifically, this endpoint does not include the response code in the response or in a request.

Out of all parties, only the wallet ever makes HTTPS requests to other parties. The only point where a request is sent to the direct-post endpoint of the verifier is in Line 100 of Algorithm 4 in the wallet relation. From Line 13 and Line 17 of Algorithm 9 it is known that the request to the direct-post endpoint is encrypted. According to Lemma 4 of [7], the request and especially the response do not leak to the attacker. The wallet w does receive the response in Line 51 of Algorithm 7 and in the following lines the redirect URL with the response code is processed. As it can be seen in the relation of a wallet the response code is never stored and only sent out in a Location header ($\langle$`Location`, *redirectUri*$\rangle$) in an encrypted response to the browser b. The content of the encrypted message cannot be derived by the attacker according to Lemma 1 of [7]. The browser does store the *redirectUri* in its state but only to send it to *redirectUri*.`host` as it can be seen in Line 11ff. of Algorithm 8 in [7]. Since the *redirectUri* is generated by v (Line 60 of Algorithm 9) and cannot be manipulated by the attacker, only v will receive the response code (*redirectUri*.`host` $\in$ `dom`(v)). From this, it follows that the response code does not leak.

According to Lemma 1 we know that a verifier session id *lsid* send in a header of the form $\langle$`Set-Cookie`, [$\langle$`__Host`, `sessionID`$\rangle$: $\langle$*lsid*, $\top$, $\top$, $\top\rangle$]$\rangle$ from v to b does not leak. This means that the attacker cannot use an honest user's session ID and response code at the verifier's redirect endpoint to obtain a service token.

Besides leaking a session ID, the attacker can also obtain a valid session ID by calling the start endpoint (Line 5 of Algorithm 9) of the verifier. To get a service token, the attacker needs to pass the condition *session*[`response_code`] $\not\equiv \langle\rangle$ in Line 71 of Algorithm 9 and must add a presentation with an honest user's identity to *session*[`vp_token`]. The only endpoint in the verifier relation that allows these values to be added to the session is the direct-post endpoint (Line 44 of Algorithm 9). To call the endpoint and log in as an honest user, the attacker must send a vp_token with the identity of an honest user. From the next case (2) below it follows that the attacker cannot create or obtain such a vp_token. This means that the attacker must load a malicious script into b and redirect the user to their wallet with the authorization request from the start endpoint. The authorization request has the parameter `response_mode` = `direct_post` and the response URI of the honest verifier. If the attacker changes the response URI to their own domain they would get the presentation but the aud parameter (*aud* = *authRespUri*.`host`) would have the wrong value and is rejected by the honest verifier (*aud* $\not\equiv$ m.`host`). The attacker could also try to change the path in the response URI to forward the response code to their domain. This is not possible because the only endpoint on the verifier that has a redirect URI in its response body (Line 60 of Algorithm 9) is the direct-post endpoint, and that redirect URI has the verifier's host (m.`host`). Not changing the response URI means that the user after authenticating the authorization request is redirected to the honest verifier with the response code in the redirect URI (assuming *useRedirectUri* = $\top$ in Line 55 of Algorithm 9). Since b is honest there is no way for the attacker to obtain the response code because it does not leak as shown above. In summary, the attacker knows a session ID that has a presentation with an honest user's identity in *session*[`vp_token`] but they do not know the response code, which makes it impossible for them to get the service token.

2. ***m*.body[vp_token] $\not\equiv \langle\rangle$ (Line 75 of Algorithm 9):** The second way an attacker can use the redirect endpoint of a verifier is to create or obtain a leaked presentation containing a credential with the identity of an honest user. With the presentation, they can answer the

authorization request and use the redirect endpoint to obtain a service token. The presentation has to contain the domain of the honest verifier as the audience because it would not be accepted otherwise. There are several ways to obtain a presentation:

a) **Leaked presentation:** One possibility is that the presentation is leaked to the attacker by an honest party. To find out if a presentation addressed to v can leak we trace the lifecycle of a presentation. The only point where an honest party creates a presentation of the form $\mathsf{extractmsg}(\mathit{presentation}) \equiv \langle\langle u, d_i, \mathit{pubKey}\rangle, \mathit{c\_nonce}, \mathit{aud}\rangle$ ($\mathit{pubKey}, \mathit{c\_nonce} \in \mathcal{N}$ and $\mathit{aud} \in \mathsf{Doms}$) is in Line 84 of Algorithm 4. No honest party will send a request to w to create a presentation except for b which is instructed by v to redirect to w. This is because, as explained above, only the wallet ever sends requests, and never to itself. This means that we have $\mathit{response\_uri} \in \mathsf{dom}(v)$ and $\mathit{redirect\_uri} \in \mathsf{dom}(v)$.

   In the case of `response_mode` = `direct_post` w sends the presentation over an encrypted connection directly to v (Line 100 of Algorithm 4). If the presentation is addressed to v ($\mathit{aud} \in \mathsf{dom}(v)$) and valid ($\mathsf{checksig}(\mathit{presentation}, \mathit{pubKey}) \equiv \top$) it will be saved in the state (Line 54 of Algorithm 9). In following processing steps the presentation is retrieved from the state (Line 74 and Line 83 of Algorithm 9) but never send out over the network again.

   In the case of `response_mode` = `fragment` the presentation is sent over an encrypted connection to a browser that is authenticated via username and password to w. Since passwords are only sent to the domain they are stored for in b (follows from the definition of *secrets* in [7]) and the receiving parties do only use passwords to check them with the method $\mathsf{secretOfID}(u)$, the passwords do not leak. This means that only the honest browser b receives the presentation and does not leak it. In the next step, a script from v sends the presentation via an encrypted POST request to the domain under which the script is running (Line 3 of Algorithm 11), and v stores the presentation in its state. After that the presentation is validated by v (Line 90 of Algorithm 9) and the user id is extracted (Line 92 of Algorithm 9).

   The verifier does not include the presentation in the response of the redirect endpoint (Line 66 of Algorithm 9) or the direct-post endpoint (Line 44 of Algorithm 9). Looking at the other endpoints of the verifier in Line 5 and Line 2 of Algorithm 9 it can be seen that they do not process the presentation and therefore cannot leak it.

   In summary, we can say that no matter which path the presentation takes it will not leak to the attacker.

b) **Use honest wallet:** The only point where an honest party creates a presentation is Line 84 of Algorithm 4. To create a presentation, the attacker has to load a malicious script inside b and redirect the browser with the authorization request to the wallet w. We know that $b = \mathsf{browserOfWallet}(w)$ because only then can b authenticate to the wallet and create a presentation as argued above. The audience of the presentation has to have the value $\mathit{aud} \in \mathsf{dom}(v)$ because otherwise the check $\mathit{aud} \not\equiv m.\mathtt{host}$ in Line 3 of Algorithm 10 is true and the presentation will be rejected. This means that

the attacker has to use *redirect_uri*.host ∈ dom(v) or *response_uri*.host ∈ dom(v) depending on the response_mode parameter in the authorization request because the host of these URIs becomes the value of the *aud* parameter (Line 82 of Algorithm 4).

If the response_mode = direct_post w will send the presentation over an encrypted connection (Line 100 of Algorithm 4) to v because of *response_uri*.host ∈ dom(v). From Lemma 4 of [7] it is known that such an encrypted message cannot be decrypted by the attacker. In addition, there is only one endpoint at the verifier that includes a redirect_uri parameter in the response body (Line 60 of Algorithm 9), and that body does not include any of the parameters sent to the endpoint. Therefore, the attacker cannot use an endpoint of v to forward the presentation to a domain under their control.

If the response_mode = fragment w will redirect b to v with a Location header in an encrypted response (Line 90 of Algorithm 4). During the redirection in the browser, it is clear from Line 11ff. of Algorithm 8 in [7] that the authorization response is not leaked by b to the attacker. The URL is saved in b's state but never sent out to another party than v and is not accessible to scripts not loaded from v's origin. From Lemma 1 in [7] it is known that the encrypted requests and responses between b and w or v cannot be decrypted by the attacker. In addition, v has no endpoint that includes parameters from the request in a redirect to another domain. The verifier has only two places where a redirect is sent to the browser (Line 34 and Line 97 of Algorithm 9). The first one in Line 34 of Algorithm 9 contains no parameters from the request and the second one in Line 97 of Algorithm 9 contains no parameters at all. Therefore, the attacker cannot use a customized redirect URI to forward the presentation to a domain under their control.

In summary, it is known that first, the presentation will not be leaked by an honest party, and second, the attacker cannot force an honest party to create a presentation and send it to him.

c) **Own user credential:** The final way to create a presentation is for the attacker to control the private key of a credential that contains the identity of an honest user. According to the Issuance Authentication security property (Definition B.2.1) it is known that the attacker does not know the private key for a public key contained in an honest user's credential. This means that the holder key of an honest wallet $w \in W_b$ does not leak and that the attacker cannot inject their public key in an issuance flow with i, b, and w.

3. ***session*[vp_token] ≢ ⟨⟩ (Line 80 of Algorithm 9):** The third possibility to use the verifier's redirect endpoint is to execute a cross-device flow (*useRedirectUri* = ⊥ in Line 55 of Algorithm 9) in which the attacker has to call the redirect endpoint (Line 66 of Algorithm 9) with an authenticated login session id *lsid*. This means that the presentation is sent to the verifier via the direct-post endpoint (Line 44 of Algorithm 9) and stored in the session associated with *lsid*. To obtain an authenticated login session ID, the attacker can either derive the *lsid* from their knowledge or create the *lsid* and try to trick an honest user into authenticating.

   Since the Lemma 1 already proves that the login session ID does not leak, there is no way an attacker can derive an authenticated session ID from its knowledge.

The other option for the attacker is to create the *lsid* and trick an honest user into authenticating. Note that this is similar to what is discussed above in (1) but we have *useRedirectUri* = $\perp$ in Line 55 of Algorithm 9. First, the attacker calls the start endpoint (Line 5 of Algorithm 9) of the verifier, and in the response, they receive the *lsid* ($\langle$`Set-Cookie`, [$\langle$`__Host`, `sessionID`$\rangle$: $\langle$*lsid*, $\top$, $\top$, $\top\rangle$]$\rangle$) and an authorization request ($\langle$`Location`, *authUrl*$\rangle$) with `response_mode` = `direct_post`. Second, the *lsid* has to be authenticated by somebody via a POST request to the direct-post endpoint (Line 44 of Algorithm 9) with a vp_token that contains an honest user's identity. From the case (2) above it is known that the attacker cannot obtain such a vp_token. This means the attacker has to load a malicious script into b and let the user authenticate the authorization request with their wallet w (w chooses a credential with the identity u). We know that b = browserOfWallet(w) because only then can b authenticate itself to w. Since $S^j(v)$.`sessions`[*lsid*].`useRedirectUri` $\equiv \perp$ it is clear that the method validateRequest($\langle$*authRespUri*.`host`, *m*.`body`[`nonce`]$\rangle$) (Line 79 of Algorithm 4) is called. This method checks whether b has $\langle$*authRespUri*.`host`, *m*.`body`[`nonce`]$\rangle \in S(b)$.`started`. The only way a domain-nonce combination is added to $S(b)$.`started` is by executing Line 17 of Algorithm 14. To execute this line a response to a request with the reference `START` is needed (π_1(*reference*) $\equiv$ `START` Line 11 of Algorithm 14). This request can only be sent by a script executing a new script command defined in Line 19 of Algorithm 13. This script command sends a POST request to a URL and checks that the URL has the same domain as the script (*url*.`host` $\not\equiv$ *docorigin*.`host` $\vee$ *url*.`protocol` $\not\equiv$ `S` Line 20 of Algorithm 13). That makes it impossible for a malicious script to add the domain-nonce combination to $S(b)$.`started` for a domain not controlled by the attacker, e.g. a domain of v. This means that the attacker cannot force a user into authenticating an arbitrary authorization request. Consequently, the attacker cannot use the verifier's redirect endpoint in the cross-device flow mode to obtain a service token.

Looking at all the possible uses of the verifier's redirect endpoint, there is no way an attacker could use it to obtain a service token. Since the service token is also not leaked by an honest party, the attacker cannot derive a service token for an honest user from its knowledge. This proves Lemma 3. □

C.3 Proof of Issuance Authentication

This section contains the proof of the Issuance Authentication security property (Definition B.2.1). The property is shown directly by proving that the property holds in a Verifiable Credentials web system $\mathcal{VCWS}^n$.

Lemma 4 (Issuance Authentication Holds)
Definition B.2.1 holds.

Proof. By Definition B.2.1 there is an arbitrary credential $c \equiv$ sig($\langle u, d_i$, pub(p)$\rangle$, $S^j(i)$.`signingKey`). To show that p is not derivable by the attacker ($p \notin d_\emptyset(S^j$(attacker))), we trace all the ways a malicious actor could control the private key in the credential. There are

two ways that p can be derived by the attacker: First, the private holder key ($S^j(w).\texttt{holderKey}$) of $w \in W_b$ is leaked, and second, the attacker is able to provide their public key during credential issuance.

This paragraph shows that $S^j(w).\texttt{holderKey}$ does not leak. $S^j(w).\texttt{holderKey}$ is used in the authentication endpoint (Line 61 of Algorithm 4) to create a signature in Line 84 of Algorithm 4. Since the cryptography is secure the private key will not be exposed from the signature or the signed data. Signatures with this key are also created in the token response (Line 2 of Algorithm 7) in Line 13 and Line 23 of Algorithm 7. In the token response the public key ($\mathsf{pub}(S^j(w).\texttt{holderKey})$) of the holder key is embedded in the proof of possession (Line 13 and Line 23 of Algorithm 7) and included in the HTTPS requests to the credential endpoint (Line 19 of Algorithm 4) and the batch credential endpoint (Line 29 of Algorithm 4). But again: Since cryptography is secure, the private key cannot be derived from the public key. The start endpoint (Line 2 of Algorithm 4), the credential offer endpoint (Line 5 of Algorithm 4), the start code flow endpoint (Line 21 of Algorithm 4), and the redirect endpoint (Line 44 of Algorithm 4) never use the holder key and the requests to the token endpoint (Line 20 and Line 60 of Algorithm 4), the request to the direct-post endpoint (Line 100 of Algorithm 4), and the deferred credential endpoint (Line 16 of Algorithm 4) do not include the holder key $S^j(w).\texttt{holderKey}$. In addition, the holder key is never written in the wallet relation because it is set in the initial state, so there is no way to overwrite the key.

The other way an attacker knows p is by injecting $\mathsf{pub}(p)$ with $p \in d_\emptyset(S^j(\mathsf{attacker}))$ into c during an issuance flow between $w \in W_b$, b and i. To show that this is not possible, we trace the creation of c at the issuer. A credential can be created in the credential endpoint (Line 76 of Algorithm 1), batch credential endpoint (Line 96 of Algorithm 1), or deferred credential endpoint (Line 117 of Algorithm 1).

The public key cannot be injected into c at the deferred credential endpoint because this endpoint only needs an *access_token* and a *transaction_id* to issue the credential. At this point, the public key for the credential is already contained in the state of i and the attacker would only obtain the credential. This means that it is not important whether the attacker knows the *transaction_id* or not because they can always obtain such a credential by tricking the user into logging in at a corrupted verifier.

From the relation of an issuer, it is clear that the public key can only be injected into the credential in the credential endpoint or the batch credential endpoint. To call one of these endpoints an *access_token* and a *c_nonce* are needed. The POST requests to these endpoints include a proof of possession of p which is a signature of the form $proof \equiv \mathsf{sig}(\langle d_i, c\_nonce, \mathsf{pub}(p)\rangle, p)$. In the credential endpoint the public key is included in c in Line 88 of Algorithm 1 or written to the state in Line 93 of Algorithm 1 to later be included in the credential created in the deferred credential endpoint (Line 127 of Algorithm 1). In the batch credential endpoint the public key is included in c in Line 109 of Algorithm 1 or added to the state in Line 113 of Algorithm 1 to be included at the deferred credential endpoint.

To derive the *access_token* from the attacker's knowledge, the attacker has to either call the token endpoint (Line 46 of Algorithm 1) or obtain it via a leakage. To show that the *access_token* does not leak we trace the lifecycle of the token. The *access_token* is created in Line 70 of Algorithm 1, stored in the state of i, and sent in an encrypted response (secure after Lemma 4 of [7]) to w. To

prove that it is indeed sent to w there must be two cases considered: First, the token endpoint is called with a *code* and a *code_verifier* and second, the token endpoint is called with a *pre-authorized_code* and a *user_pin* because of $S^j(w).\texttt{sessions}[issid].\texttt{use\_user\_pin} = \top$.

1. **$\langle\langle$`code`, *code*$\rangle$, $\langle$`code_verifier`, *code_verifier*$\rangle\rangle$ (Line 58 of Algorithm 1)**

   To show that the token endpoint is called by w we have to prove that the *code* and the *code_verifier* do not leak to the attacker. The *code* is created in the authentication endpoint (Line 23 of Algorithm 1) in Line 39 of Algorithm 1. In the following lines, it is stored in the state of i and sent via a browser redirect to the *redirect_uri*. From the precondition, it is known that the *redirect_uri* belongs to an honest wallet. This means that the authorization request came from w and that the *code* is being sent to w. It makes no sense for an attacker to send an authorization request with a domain of w in the redirect URI because there is no way to get the authorization code. Looking at the relation of a wallet, there are only two places where a Location header is included in the response (Line 40 and Line 90 of Algorithm 4) and in both places there is no *code* parameter from the request in the redirect URI. The browser that authenticated towards the authentication endpoint must be b because otherwise there would be no honest identity u in c as shown below. It is also clear that the *redirect_uri* is always an HTTPS URL because of Line 26 of Algorithm 4 and because the authorization request is an encrypted request (Line 30 of Algorithm 4). The *code* is not leaked during redirect in b because the *redirect_uri* is written to the state of b but only sent to *redirect_uri*.`host` (Line 11ff. Algorithm 8 in [7]). The wallet w receives the *code* in the redirect endpoint (Line 44 of Algorithm 4) and sends it back to $m.\texttt{parameters}[\texttt{iss}] \equiv S^j(w).\texttt{sessions}[issid].\texttt{host}$. From this, it is clear that the *code* is sent back to i via an encrypted message to the token endpoint (Line 58 of Algorithm 4). The wallet also stores the *code* in the state, but only for formal reasons of the model. The *code* in the state is never retrieved or written to again. At the token endpoint i checks that the *code* is issued by itself (Line 66 of Algorithm 1) and in the following lines it is deleted from the state. The issuer does not include the *code* in the response or a request. Looking at the other endpoints it can be seen that only the start credential offer endpoint writes to $S^j(i).\texttt{codes}$ as well but does not read from it. The start endpoint (Line 3 of Algorithm 1), credential endpoint (Line 76 of Algorithm 1), batch credential endpoint (Line 96 of Algorithm 1), and deferred credential endpoint (Line 117 of Algorithm 1) do not read or write to $S^j(i).\texttt{codes}$.

   The attacker can also try to authenticate as an honest user at the authentication endpoint. For this, they need an honest password that is leaked somewhere. The only two locations where passwords are sent from b to a server are in Line 5 of Algorithm 3 and Line 5 of Algorithm 6. Both scripts select a password from the list of passwords for their origin (follows from the definition of *secrets* in [7]) and send it via an HTTPS request to the instance. From Lemma 1 in [7] it is clear that the attacker cannot decrypt such a message. At the endpoints the passwords are compared with the result of the method $\mathsf{secretOfID}(id)$ in Line 67 of Algorithm 4 and Line 37 of Algorithm 1 but are not stored or send out again. That means that passwords do not leak and the attacker cannot authenticate to the authentication endpoint as an honest user. From this, we can conclude that the *code* does not leak.

   To show that the *code_verifier* does not leak, we look at its lifecycle. The *code_verifier* is created in Line 24 of Algorithm 4 in the start code flow endpoint of the wallet (Line 21 of Algorithm 4) and stored in the state. A hashed version of it is sent out over the network but

since the cryptography is secure it is impossible to recover the plain text. The only place where the *code_verifier* is read from the state is Line 57 of Algorithm 4 and in the following it is sent to $m.\texttt{parameters[iss]} \equiv S^j(w).\texttt{sessions}[issid].\texttt{host}$ which is i as shown above. The issuer compares the *code_verifier* in Line 66 of Algorithm 1 with the *code_challenge* from the authorization request and does not do any further processing of the *code_verifier*. In particular, it is not stored, included in the response, or sent out in a request. It can be seen that the *code_verifier* does not leak as well.

Since the *code* and the *code_verifier* do not leak, w is the only party that can send the token request. This means that the *access_token* will be sent to w in the token response.

2. $\langle\langle \texttt{pre-authorized\_code}, \textit{pre-authorized\_code}\rangle, \langle \texttt{user\_pin}, \textit{user\_pin}\rangle\rangle$
(Line 49 of Algorithm 1)

Sending a *pre-authorized_code* and a *user_pin* to the token endpoint is the second option to use the endpoint. To guarantee that only w can send the token request either the *pre-authorized_code* or the *user_pin* are not allowed to leak. Since the *pre-authorized_code* leaks in the credential offer (Line 22 of Algorithm 1), we must show that the *user_pin* remains a secret. The only location where a user PIN is added to a request or a response in a wallet is in Line 17 of Algorithm 4 ($body[\texttt{user\_pin}] = S^j(w).\texttt{userPins}[issHost]$). The user PIN is sent to the domain for which it is selected ($url = \langle \texttt{URL}, \texttt{S}, issHost, \texttt{/token}, \langle\rangle, \bot\rangle$) via an HTTPS request in Line 20 of Algorithm 4. It is known from Lemma 4 in [7] that this message cannot be decrypted by the attacker. The issuer compares the user PIN from the token request with the result of the method $\mathsf{userPinOfId}(u)$ in Line 55 of Algorithm 1. The issuer does not process the user PIN any further and has no user pins in its state. Note that the verifier also does not have any user pins in its state and does not receive any user pins. Since the comparison of the user PIN is successful it can be concluded, that the token request is sent from w to i because only these two know the user pin. It follows that the *access_token* is sent to w.

The wallet w stores the *access_token* in the state in Line 8 of Algorithm 7 and sends it in an encrypted message either to the credential endpoint (Line 19 of Algorithm 4), batch credential endpoint (Line 29 of Algorithm 4) or deferred credential endpoint (Line 16 of Algorithm 8) of i. It is clear that the messages go to i because all these requests use the same domain as the token request ($S^j(w).\texttt{sessions}[issid].\texttt{host}$). The wallet does not include the *access_token* in any other response or request. The issuer receives the *access_token* in the credential endpoint (Line 76 of Algorithm 1), batch credential endpoint (Line 96 of Algorithm 1), or deferred credential endpoint (Line 117 of Algorithm 1) and verifies that it is issued by itself. After that, the *access_token* is not processed further in those endpoints and is not included in the response or any request. The start endpoint (Line 3 of Algorithm 1), the start credential offer endpoint (Line 6 of Algorithm 1), and the authentication endpoint (Line 23 of Algorithm 1) do not read or write to $S^j(i).\texttt{atokens}$ which contains access tokens. All of the above shows that the *access_token* does not leak.

The attacker also cannot use the token endpoint to obtain an *access_token* because, as shown above, the *code*, *code_verifier*, and *user_pin* do not leak.

From this, it can be concluded that the attacker cannot derive the *c_nonce* from the token response (Line 74 of Algorithm 1). On the wallet side the *c_nonce* is never stored in the state and only included in the *proof* (Line 13 and Line 23 of Algorithm 7) that is sent via an encrypted message

to the credential or batch credential endpoint of i. These requests are sent to i because otherwise the *access_token* leaks to the attacker which contradicts the proof above. The issuer compares the *c_nonce* with the value from its state (Line 83 and Line 104 of Algorithm 1) and does not do any further processing. The *c_nonce* is not contained in any other response or request of i, so there is no way the attacker can derive the value.

Since the attacker does not know the *access_token* and the *c_nonce*, they cannot inject $\mathsf{pub}(p)$ into c. This, and the fact that $S^j(w).\texttt{holderKey}$ does not leak, proves Lemma 4. □

C.4 Proof of Presentation Session Integrity

This section contains the formal proof of the Presentation Session Integrity security property (Definition B.3.5). The property is shown directly by proving that the property holds in a Verifiable Credentials web system $\mathcal{VCWS}^n$.

Lemma 5 (Presentation Session Integrity Holds)
Definition B.3.5 holds.

Proof. Definition B.3.5 says that $\mathsf{loggedIn}^Q_\rho(b, v, i, id, lsid)$ is true and the state of v contains $S(v).\texttt{sessions}[lsid].\texttt{serviceTokenId} \equiv serviceToken$ and $S(v).\texttt{sessions}[lsid].\texttt{userInfo} \equiv \langle id, d_i, d_w\rangle$. This means in particular that v contains a valid login session with the nonce *lsid* (i.e. $S(v).\texttt{sessions}[lsid] \not\equiv \langle\rangle$). To show that the Presentation Session Integrity security property holds, the two events $\mathsf{started}^{Q''}_\rho(b, v, lsid)$ and $\mathsf{chooseCredential}^{Q'}_\rho(v, lsid, id)$ must be shown.

1. **$\mathsf{started}^{Q''}_\rho(b, v, lsid)$**

   A service token for the session id *lsid* is only created in Line 93 of Algorithm 9. To execute this line v has to receive an encrypted HTTPS request containing the header $\langle\texttt{Cookie}, \langle\langle\texttt{\_\_Host}, \texttt{sessionID}\rangle, lsid\rangle\rangle$. Definition B.3.3 says that this request is sent by the browser b. From the definition of the browser in the WIM it is easy to see that Algorithm 4 in [7] is always used to send HTTP requests. From Line 4 of Algorithm 4 in [7] ($cookies = \langle\{\langle c.\texttt{name}, c.\texttt{content.value}\rangle | c \in^{\langle\rangle} s'.\texttt{cookies}[message.\texttt{host}] \wedge (c.\texttt{content.secure} \equiv \top \Rightarrow (url.\texttt{protocol} \equiv \texttt{S}))\}\rangle$) it is clear that the `sessionID` cookie is only included in a request to v if it is stored under a domain of v. In Line 4 of Algorithm 8 in [7] it can be seen that a cookie is always stored under the host of the request ($s'.\texttt{cookies}[request.\texttt{host}] = \mathsf{AddCookie}(s'.\texttt{cookies}[request.\texttt{host}], c, requestUrl.\texttt{protocol})$) and Definition 37 of [7] assures that cookies with the `__Host` prefix can only be set via an encrypted response. This means that b received an encrypted response from v with the header $\langle\texttt{Set-Cookie}, [\langle\texttt{\_\_Host}, \texttt{sessionID}\rangle : \langle lsid, \top, \top, \top\rangle]\rangle$. The only point where v creates a response with such a header is Line 32 and Line 35 of Algorithm 9 and in the same processing step Q'' the login session *lsid* is created. In Line 6 of Algorithm 9, the origin header is checked in the processing step Q'', which means that only a script loaded from v can send this request. A verifier has only two scripts: the script *script_verifier_get_fragment* is to get the authorization response from a URL fragment and the script *script_verifier_index* is to start the authentication flow. If the latter is executed, specifically Line 6 of Algorithm 12, the script command `START` is executed, which sends a POST request to v containing the origin

header $\langle d_v, \mathsf{S}\rangle$ with $d_v \in \mathsf{dom}(v)$ (Line 25 of Algorithm 13). In a subsequent processing step, Q'' will be executed in the run ρ. This means that all conditions for $\mathsf{started}_{\rho}^{Q''}(b, v, lsid)$ are fulfilled.

2. **$\mathsf{chooseCredential}_{\rho}^{Q'}(v, lsid, id)$**

   From Definition B.3.3 we know that v contains the state $S(v).\texttt{sessions}[lsid].\texttt{userInfo} \equiv \langle id, d_i, d_w\rangle$, which means that Line 95 of Algorithm 9 is executed. From this, it can be concluded that the if-else-if condition in Line 70ff. of Algorithm 9 is executed without stopping the processing step which means there exists the state $S(v).\texttt{sessions}[lsid].\texttt{vp\_token} \equiv vp\_token$ and $\mathsf{extractmsg}(vp\_token) \equiv \langle\langle id, d_i, pubKey\rangle, n, aud\rangle$ with $n, pubKey \in \mathcal{N}$ and $aud \in \mathsf{dom}(v)$. The *aud* parameter must be a domain of v because it is checked in Line 90 of Algorithm 9. A term of the form of the *vp_token* is never created by a verifier because a verifier does never sign terms. By tracing the *vp_token* back it can be seen that it is either received via a URL fragment (Line 75 of Algorithm 9) and send to v by the script *script_verifier_get_fragment* (Line 3 of Algorithm 11) or it is received via a POST request in Line 45 of Algorithm 9. This means that another process must have created the *vp_token*. The only honest parties creating signatures are wallets and issuers. To create a valid *vp_token* the party must know the nonce n from the authorization request.

   The nonce n is created in Line 8 of Algorithm 9, stored in the session state ($S(v).\texttt{sessions}[lsid].\texttt{nonce}$) and subsequently send in the authorization request via a Location header to the browser b in an encrypted response. The authorization request is not leaked in Line 41 of Algorithm 9 because of Definition B.3.5. The browser redirects to the Location header in Line 11ff. of Algorithm 8 in [7] and stores the authorization request in its state, but never sends it to anyone other than w. The authorization request is then transferred via an encrypted message to w. The wallet uses n only to create a valid presentation, does not store it anywhere, and does only send the nonce via a HTTPS POST message (Line 100 of Algorithm 4) directly to v or uses the Location header (Line 90 of Algorithm 4) again to redirect the user back to the verifier. The redirect also uses only encrypted HTTPS messages. The verifier compares the nonce in the presentation with the nonce from the session in Line 5 of Algorithm 10 and does not do any further operations with n. This especially includes that n is not send out in a response again in Line 64 of Algorithm 9 or Line 99 of Algorithm 9 and a verifier does also not do any requests to other servers. As a side note the endpoint in Line 2 of Algorithm 9 does also not process or send n. This means that the nonce n does not leak since it is never sent over an unencrypted connection. Note that as argued earlier the transfer of secrets via an encrypted connection is secure. In conclusion, only v, b and w get to know the nonce n.

   It follows that only w knows the nonce n and signs a term of the form of a *vp_token* in the processing step Q' in Line 84 of Algorithm 4. From this, it can be concluded that only w can create the *vp_token*. From $\mathsf{started}_{\rho}^{Q''}(b, v, lsid)$ it is clear that the authorization request is sent by v to w, that means the *aud* parameter of the *vp_token* is set to $authRespUri.\texttt{host} \in \mathsf{dom}(v)$ in Line 82 of Algorithm 4. Before this in Line 76 of Algorithm 4 a credential with the identity *id* is chosen from the state of w. Since i, b and w are honest Lemma 2 can be applied which guarantees for the identity *id* of the credential stored in w that $\mathsf{ownerOfID}(id) = \mathsf{browserOfWallet}(w)$. With all the above the requirements for $\mathsf{chooseCredential}_{\rho}^{Q'}(v, lsid, id)$ are fulfilled.

This proves Lemma 5 because if we have $\mathsf{loggedIn}^{Q}_{\rho}(b, v, i, id, lsid)$, there must have been the events $\mathsf{started}^{Q''}_{\rho}(b, v, lsid)$ and $\mathsf{chooseCredential}^{Q'}_{\rho}(v, w, lsid, id)$ before. □

C.5 Proof of Issuance Session Integrity

The following is the proof of the Issuance Session Integrity security property (Definition B.4.5). The property is shown directly by proving that the property holds in a Verifiable Credentials web system $\mathcal{VCWS}^{n}$.

Lemma 6 (Issuance Session Integrity Holds)
Definition B.4.5 holds.

Proof. From Definition B.4.5 we know that the $\mathsf{stored}^{Q}_{\rho}(b, w, i, id, issid)$ event happened. This event says that the state of w contains a credential c with $c \in S'(w).\texttt{sessions}[issid].\texttt{credentials}$ and that c is received in an encrypted response body ($c \equiv m_{\text{dec}}.\texttt{body}[\texttt{credential}]$ or $c \in m_{\text{dec}}.\texttt{body}[\texttt{credential\_responses}].\texttt{credentials}$). There are three points in the wallet relation where such a state is created: First, in the credential response (Line 35 of Algorithm 4), second, in the batch credential response (Line 43 of Algorithm 4), and third, in the deferred credential response (Line 50 of Algorithm 4). The attacker cannot send such a response because they would have to know the symmetric key (Line 20 of Algorithm 18 in [7]) of a previously sent HTTPS request with the correct reference. This is not possible because the attacker cannot learn symmetric keys in an HTTPS response or request (Lemma 4 of [7]). That means there must have been a corresponding request made by w first.

Starting with the deferred credential response there must have been a deferred credential request first. The only point in the wallet relation where such a request is sent is Line 16 of Algorithm 4. To execute this line there must have been a trigger, $\top$ must have been assigned to *sendDeferredCredentialRequest* in Line 2 of Algorithm 8, and $S(w).\texttt{transactionIds} \not\equiv \langle\rangle$. From this, it is known that something must have been added to $S(w).\texttt{transactionIds}$ in a previous processing step. There are two points where something is written to $S(w).\texttt{transactionIds}$: First, in the credential response (Line 37 of Algorithm 4), and second, in the batch credential response (Line 45 of Algorithm 4). This means one of these two responses must have been received previously and for them to be received there must have been a corresponding request first.

As it can be seen in the relation of the wallet the batch credential request is only made in the token response in Line 29 of Algorithm 4 if *useBatchCredEndpoint* is true in Line 11 of Algorithm 7. The credential request is also only sent in the token response if *useBatchCredEndpoint* is false in Line 11 of Algorithm 7. This in turn means that there must have been a token request. The token request must have been made regardless of whether the deferred credential endpoint was used or not because if the deferred credential endpoint was not used, the credential would have been written to $S'(w).\texttt{sessions}[issid].\texttt{credentials}$ by the credential response or the batch credential response, as discussed above.

There are two locations in the relation of a wallet where a token request is made: The first one is in Line 60 of Algorithm 4 where an authorization request is received and the second one is in Line 20 of Algorithm 4 where a credential offer is received.

1. **startedCodeFlow$^{Q''}_{\rho}$ ($b, w, issid$)**

   To execute Line 60 of Algorithm 4 the redirect endpoint (Line 44 of Algorithm 4) has to be called successfully. This requires an HTTPS request from b containing a cookie of the form $\langle$`Cookie`, $\langle\langle$`__Host`, `sessionID`$\rangle$, $issid\rangle\rangle$. From Line 4 of Algorithm 4 in [7] it is clear that the `sessionID` cookie is only included in a request to w if it is stored under the domain of w. In Line 4 of Algorithm 8 in [7] it can be seen that a cookie is always stored under the host of the request and Definition 37 of [7] assures that cookies with the `__Host` prefix can only be set via an encrypted response. This means that b received an encrypted response from w with the header $\langle$`Set-Cookie`, $[\langle$`__Host`, `sessionID`$\rangle : \langle issid, \top, \top, \top\rangle]\rangle$. The only point where w creates a response with such a header is in the start-code-flow endpoint (Line 21 of Algorithm 4) in Line 41 of Algorithm 4 and in Line 38 of Algorithm 4 in the same processing step Q'' the issuance session *issid* is created. The *issid* session contains the parameters from the authorization request, e.g. $S(w)$.`sessions`$[issid]$.`code_challenge` $\equiv$ *code_challenge* (Line 25 of Algorithm 4). In Line 22 of Algorithm 4, the origin header is checked in the processing step Q''. Since the browser b is honest only a script from w can execute this endpoint to start the issuance. A wallet has only two scripts: the script *script_wallet_form* is to authenticate a user towards the wallet and the script *script_wallet_index* is to start the issuance flow. If the latter one is executed particularly Line 7 of Algorithm 5, the `FORM` script command is run which sends a POST request to w containing the origin header $\langle d_w, \mathtt{S}\rangle$ with $d_w \in \mathsf{dom}(w)$ (Line 48 of Algorithm 7 in [7]). In a subsequent processing step, Q'' will be executed in the run ρ. This means that all conditions for startedCodeFlow$^{Q''}_{\rho}$ $(b, w, issid)$ are fulfilled.

2. **authenticated$^{Q'}_{\rho}$ ($b, i, w, id, issid$)**

   We know that the token endpoint of the issuer (Line 46 of Algorithm 1) was successfully called and that two parameters are required: First, a valid *code* and second a valid *code_verifier*. The *code* is sent by b to w in the authorization response to the redirect endpoint in Line 44 of Algorithm 4 and the *code_verifier* is taken from the state ($S(w)$.`sessions`$[issid]$.`code_verifier` = *code_verifier*). This means that the start-code-flow endpoint (Line 21 of Algorithm 4) was executed in a previous processing step because this is the only location where the *code_verifier* is stored in the state. From Line 25 of Algorithm 4 we can conclude that there must also be the state $S(w)$.`sessions`$[issid]$.`code_challenge` = *code_challenge* = HASH(*code_verifier*). The token endpoint of the issuer compares the *code_verifier* with the *code_challenge* (*codeInfo*.`3` $\not\equiv$ HASH(*codeVerifier*) Line 66 of Algorithm 1). The only place where *codeInfo*.`3` is stored in the state is Line 40 of Algorithm 1 in the authentication endpoint of the issuer (Line 23 of Algorithm 1). The parameter *code_challenge* is received via the body of a POST request. Since the endpoint checks the origin header in Line 29 of Algorithm 1, only an issuer script can send this POST request. The issuer only has the script *script_issuer_form* that adds parameters from the script state to the POST request (Line 3 of Algorithm 3) and chooses the identity *id* in Line 4 of Algorithm 3. The script can be loaded from two locations: First, in Line 4 of Algorithm 1 for the pre-authorized code flow and second in Line 26 of Algorithm 1 for authentication at the authentication endpoint of the issuer. The latter one must have been executed because otherwise the *code_challenge* would not have been stored in *codeInfo*.`3`. This means we have *scriptstate*.`1` = /`authentication`. The *code_challenge* parameter must have been received by the authentication endpoint in a GET parameter (*data* = *m*.`parameters`)

because these parameters are added to the script state *scriptstate*.2 (Line 26 of Algorithm 1). This means we have *scriptstate*.2[code_challenge] = *code_challenge*. From the relation of the parties it is known that an issuer and a verifier never make HTTP requests and the wallet only calls the token endpoint (Line 20 of Algorithm 4 and Line 60 of Algorithm 4), credential endpoint (Line 19 of Algorithm 4), batch credential endpoint (Line 29 of Algorithm 4) and deferred credential endpoint (Line 16 of Algorithm 4) of the issuer as well as to the direct-post endpoint of the verifier (Line 100 of Algorithm 4) but never the authentication endpoint of the issuer.

Looking at the *code_challenge* it can be seen that it will not leak. To show that the *code_challenge* does not leak it is also necessary to show that *code_verifier* does not leak because of *code_challenge* = HASH(*code_verifier*). The *code_challenge* is created in Line 24 of Algorithm 4 and subsequently send out in the Location header (Line 40 of Algorithm 4) of an encrypted response to b (secure after Lemma 1 of [7]) and stored in the state of w (Line 38 of Algorithm 4). During the redirection in the browser, it is clear from Line 11ff. of Algorithm 8 in [7] that the authorization request is not leaked by b to the attacker. The URL is stored in the state of b but never sent out to another party than i and is not accessible to scripts not loaded from the origin of i.

The issuer receives the *code_challenge* at the authentication endpoint (Line 23 of Algorithm 1) and stores it in its state (Line 40 of Algorithm 1). The only location where i retrieves this state is in Line 51 and Line 65 of Algorithm 1 but only in Line 65ff. the *code_challenge* is used to compare it with the incoming *code_verifier* (*codeInfo*.3 ≠ HASH(*code_verifier*)). In all other endpoints in Line 3, Line 6, Line 76, Line 96, and Line 117 of Algorithm 1 and in the function BUILD_CREDENTIAL (Algorithm 2) the state $S(i)$.codes is never read or written to. The *code_verifier* is received by i in the token endpoint (Line 46 of Algorithm 1) and compared to the *code_challenge* in Line 66 of Algorithm 1 but never stored in the state or send out again to another party.

Taking a closer look at the wallet relation it can be seen that the *code_challenge* and *code_verifier* are never retrieved from the state in the endpoints of Line 2, Line 5, Line 61 of Algorithm 4, in Algorithm 7, and in Algorithm 8. The *code_challenge* is also not retrieved in the redirect endpoint in Line 44 of Algorithm 4 but the *code_verifier* is retrieved there and send out to the domain $S(w)$.sessions[*issid*].host via an encrypted message. The attacker cannot derive the *code_verifier* from the encrypted message after Lemma 4 of [7]. The domain in $S(w)$.sessions[*issid*].host is the domain of the identity that is chosen by b in Line 5 of Algorithm 5 and thus a domain of i ($S(w)$.sessions[*issid*].host ∈ dom(i)).

Since only w, i, and b know the *code_challenge* the browser b has to make the request to the authentication endpoint and subsequently picks the identity *id* in Line 4 of Algorithm 3. This means that all conditions for $\mathsf{authenticated}^{Q'}_{\rho}(b, i, w, id, issid)$ are fulfilled.

From the above it can be seen that if there is the $\mathsf{stored}^{Q}_{\rho}(b, w, i, id, issid)$ event and the token endpoint is called from Line 60 of Algorithm 4 there has to be before the $\mathsf{authenticated}^{Q'}_{\rho}(b, i, w, id, issid)$ and the $\mathsf{startedCodeFlow}^{Q''}_{\rho}(b, w, issid)$ events.

3. **startedPreAuthCodeFlow**$_{\rho}^{Q'''}(b, i, w, issid, id)$

   The other place where a wallet makes a token request is in Line 20 of Algorithm 4. This means the state of w contains the *preAuthorizedCode* ($S(w)$.`sessions`[*issid*] .`pre-authorized_code` = *preAuthorizedCode* in Line 12 of Algorithm 4) and the validateRequest($\langle d_i, preAuthorizedCode\rangle$) method validated the credential offer received in the credential offer endpoint (Line 5 of Algorithm 4) successfully. From this method it is clear that b must contain $\langle d_i, preAuthorizedCode\rangle$ in $S(b)$.`started`. The only place where such parameters are written into the state of b is in Line 15 of Algorithm 14. To execute this line there must be a response with the reference `START` and a corresponding request. Such a request is only sent in Line 28 of Algorithm 13 which means that the script command `START` must have been executed. This script command can only make an encrypted POST request to the domain the script is running on, due to the check in Line 20 of Algorithm 13. From the validateRequest($\langle d_i, preAuthorizedCode\rangle$) method and Line 15 of Algorithm 14 it can be concluded that d_i = *request*.`host`. This means that d_i is the domain under which the script that executes the `START` command is running. The only script an issuer has is *script_issuer_form* and the only location where a credential offer is created is Line 16 of Algorithm 1 in the start credential offer endpoint (Line 6 of Algorithm 1). To call this endpoint, the script must have the script state *scriptstate*.`1` = /`startCredOffer`. It can be concluded that b executes that script and selects the identity *id* in Line 4 of Algorithm 3. Furthermore, b selects the domain $d_w \in \mathsf{dom}(w)$ of the wallet in Line 6 of Algorithm 3 and sends a POST request (Line 11 of Algorithm 3) to the start credential offer endpoint of i. From the relation of the issuer it can be seen that the response to this request contains a header of the form $\langle$`Location`, $\langle$`URL`, `S`, d_w, /`vci`/`credentialOffer`, *parameters*, $\bot\rangle\rangle$ with *parameters* = $\langle$`credential_offer`, $\langle d_i, preAuthorizedCode, useUserPin\rangle\rangle$. This means all the conditions for startedPreAuthCodeFlow$_{\rho}^{Q'''}(b, i, w, issid, id)$ are fulfilled.

This proves Lemma 6 because if there is a stored$_{\rho}^{Q}(b, w, i, id, issid)$ event in a run ρ of $\mathcal{VCWS}^n$ either the events startedCodeFlow$_{\rho}^{Q''}(b, w, issid)$ and authenticated$_{\rho}^{Q'}(b, i, w, id, issid)$ or the event startedPreAuthCodeFlow$_{\rho}^{Q'''}(b, i, w, issid, id)$ happend before. □

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature