Aalto University
School of Science

Master's Programme in Security and Cloud Computing

Formal Analysis and Verification of OAuth 2.0 in SSO

Modelling and Verification using PSPSP in Isabelle/HOL, and OFMC

Anand Vasudevan

Master's Thesis
2023

Aalto University
School of Science

Author Anand Vasudevan

Title Formal Analysis and Verification of OAuth 2.0 in SSO — Modelling and Verification using PSPSP in Isabelle/HOL, and OFMC

Degree programme Master's Programme in Security and Cloud Computing

Major Security and Cloud Computing

Supervisor Prof. Tuomas Aura, Assoc. Prof. Sebastian Mödersheim

Advisors Aleksi Peltonen, Tommi Pernilä

Collaborative partner Modirum

Date 5 July 2023 **Number of pages** 54 **Language** English

Abstract

This thesis examines the OAuth 2.0 protocol within Single Sign-On (SSO) systems through modelling and formal analysis. The versatile Performing Security Proofs of Stateful Protocols (PSPSP), a theory for the Isabelle/HOL proof assistant was used to carry out the verification. Additionally the Open-Source Fixedpoint Model-Checker (OFMC), was used in this verification for its accessibility. PSPSP notably supports the modelling of mutable long-term state, a feature not common in many similar tools.

The challenge lies in crafting a model that accurately mirrors real-world scenarios while integrating the OAuth 2.0 protocol on top of the TLS 1.2 protocol. The goal is to produce a model that is both realistic and doesn't induce false attack vectors in its abstraction.

The complexity of combining SSO, OAuth, and TLS often necessitates simplifications for effective verification. This study explores the modelling of OAuth components without drastic over-simplifications, verifying each in isolation, and then applying compositional reasoning available in PSPSP/Isabelle to introduce the TLS protocol as well. This process necessitates a well-defined interface between components and verification of all components individually and in the composition.

Both tools confirm the lack of detectable vulnerabilities in the OAuth 2.0 protocol, reinforcing its security and prominence in SSO systems. The research explores the process of modelling and formally verifying security protocols, and deepens the understanding of OAuth 2.0's role in SSO systems.

Keywords Formal Verification, OAuth 2.0, SSO, Isabelle, PSPSP, OFMC, single sign on

Acknowledgements

This thesis would not have been successful without the invaluable support and guidance of several people to whom I owe immense gratitude.

I would like to begin by expressing my profound gratitude to Associate Professor Sebastian Mödersheim from DTU. His stimulating classes in the Data Security course not only captivated my interest but also led me to the path of formal verification, and eventually guided my choice of a master's thesis topic. Throughout the journey, he has been a steadfast supervisor, showing patience and kindness at every turn.

Equally, I am deeply indebted to Doctoral Researcher Aleksi Peltonen at Aalto University, whose expert guidance throughout this process was critical. His extensive knowledge, openness, encouragement, and faith in my abilities were instrumental and irreplaceable in this undertaking.

I sincerely appreciate Professor Tuomas Aura of Aalto University, who, amidst his demanding schedule, took on the role of my secondary thesis supervisor.

Special thanks are due to Tommi Pernilä for believing in me and offering me the opportunity to work at Modirum on this thesis. The exposure to and inspiration from incredibly knowledgeable and motivated individuals at Modirum have sparked a deeper interest in the field of security and computer science.

My heartfelt appreciation goes to my SECCLO compatriots. Our shared two-year adventure across two countries was deeply enriched by your friendship. The kindness and support within our group forged a familial bond, making me never feel alone in a new country. Special mention to those who shared long hours with me in the library during the final stages of this work.

Lastly, to my family, who have unwaveringly believed in me and instilled the drive for ambition and constant self-improvement, I owe an immeasurable debt of gratitude. My parents, whose nurturing support, tireless work ethic, and deep kindness to everybody in their life continue to inspire me; Aravind, whose knowledge, curiosity, bravery, and gregariousness are among many attributes that I wish to emulate in my life; Sahana, whose humble brilliance and passionate dedication constantly drives me to not only aspire for greatness but also for goodness.

Each one of you has played an integral part in this academic endeavour. Thank you all.

With the support of the
Erasmus+ Programme
of the European Union

Contents

1 Introduction

In today's digital age, organizations rely on a multitude of web services and applications to carry out their operations. These services can be both internal and external tools, each serving a specific purpose. These include email, instant messaging, project management, source code management, and human resources systems. Each of these services typically stores sensitive information and requires users to authenticate themselves before they can use the service.

In the absence of Single Sign On (SSO) systems, users would typically authenticate themselves with a separate set of credentials, such as username and password, for each web service. Some services allow or enforce the use of multi-factor authentication (MFA) [1], whereby a user has to enroll another device, such as their smartphone or a hardware security device, and approve each login request separately. Some services may even ask the users a security question to verify their identity. Repeating these steps can lead to a significant amount of time spent on authentication processes.

Moreover, with multiple different credentials to maintain, users soon hit the wall of password fatigue [2]. This arises when users feel overwhelmed by the task of recalling numerous passwords for various accounts, each with different requirements for password strength. This often results in the reuse of passwords, using the same one for multiple accounts. Such password reuse presents a notable security concern since if any of the websites where the password is employed is compromised, an attacker can engage in brute force attacks to gain access to the user's accounts across all platforms. Within an organization, this vulnerability poses a significant threat to the entire entity.

To counter this, some organizations use password managers to securely store employee credentials. But these have drawbacks of their own. Firstly, changing credentials once an account, or the entire password vault, has been compromised is cumbersome. Furthermore, it is not possible to offer customizable role-based access control, and the organization cannot enforce MFA and risk-based access control.

SSO systems streamline the authentication process by allowing users to authenticate once and subsequently access multiple applications or systems without the need for repeated credential entry. The fundamental concept behind SSO involves establishing a trusted relationship between an identity provider (IDP) and various relying parties (RPs). The IDP is a server that is the authoritative source for user authentication, either controlled or trusted by the organization, while the RPs represent the different web applications the user needs to access.

OAuth 2.0, also referred to as OAuth in this thesis, is a protocol specifically designed to enable third-party services to obtain restricted access to a user's data from another service. In the traditional authentication model, users would typically need to provide their credentials to the third-party service to grant it the ability to authenticate on their behalf and retrieve their data.

This conventional approach results in the third party gaining complete access to the user's data and being able to perform any actions as if they were the user, regardless of the user's intention to grant only limited access. Additionally, users lack the ability to selectively revoke access for individual third-party services; they would need to change their password entirely. Furthermore, since the third party stores the user's credentials, any compromise

of the third-party service could potentially lead to unauthorized access to the user's data. These risks are magnified by the common practice of reusing passwords across multiple services, further exacerbating security concerns [2].

OAuth enables users to authorize and delegate access to their data to third-party applications without the need to disclose their credentials. With OAuth, users can grant specific and limited access to a particular scope of data. Additionally, OAuth provides the flexibility for users to easily revoke access for specific third-party applications when desired. Moreover, by eliminating the need for multiple sets of credentials, OAuth reduces password fatigue and allows users to remain more vigilant against phishing attempts.

OAuth 2.0 has widespread adoption as a protocol for SSO applications, catering to both organizational environments and individual user applications on the wider internet. Although it was not initially designed for authentication, it is routinely adapted to fulfill this role in SSO systems.

Formal verification is a process that uses a methodical approach that employs mathematical reasoning to confirm the correctness of a system. This thesis aims to utilize formal verification tools and methodologies to perform an evaluation of the security properties of the OAuth 2.0 protocol, particularly when implemented in the context of SSO applications. The tools used for conducting the formal verification in this investigation are the Open-Source Fixedpoint Model-Checker (OFMC) [3] and the Performing Security Proofs of Stateful Protocols (PSPSP) [4] theory in Isabelle/HOL.

The study concentrates on the OAuth 2.0 protocol itself rather than any specific real-world implementations. While there may exist numerous intricacies and potential vulnerabilities in various implementations, due to the fact that the implementation process often lacks the robustness found in the standardization of protocols like OAuth 2.0, examining the protocol itself allows for a more comprehensive coverage. Multiple implementations of SSO systems adhere to the protocol specification of OAuth and integrate it into their code. Consequently, a vulnerability within the underlying protocol would compromise all implementations that rely on it.

This thesis aims to achieve several key objectives that collectively contribute to a comprehensive analysis of the security of the OAuth protocol in SSO systems. These objectives are as follows:

1. *Understanding Single Sign-On systems and their requirements:* It is essential to comprehend the nature and functioning of SSO systems in order to analyse their security. The first objective involves examining the characteristics and prerequisites of SSO systems to establish the context for subsequent formal verification and understand their interaction with the OAuth 2.0 protocol.

2. *Gaining in-depth knowledge of the OAuth 2.0 protocol and its implementation in SSO systems:* The second objective is to acquire a systematic and comprehensive understanding of the OAuth 2.0 protocol. This includes studying its theoretical foundations, operational mechanisms, and design philosophy. The focus will be on how the protocol is implemented within SSO systems, particularly regarding user authentication. The investigation will explore the adaptation, interaction, and fulfillment of SSO requirements by the OAuth 2.0 protocol.

3. *Conducting formal verification of the OAuth 2.0 protocol in the Context of SSO Systems:* The central goal of this thesis is to undertake a formal examination of the OAuth 2.0 protocol as it is applied in SSO systems. The third objective involves

constructing a mathematical model of the protocol's operation and employing techniques to systematically prove or disprove its correctness. In the event of identifying vulnerabilities, the analysis will extend to scrutinizing potential attack vectors, understanding their implications, and proposing appropriate remediation strategies.

The structure of this thesis is as follows. Chapter 2 provides an extensive summary of the background concepts that form the foundation of this study. These concepts are essential for a comprehensive understanding of the subsequent chapters. Chapter 3 delves into the methodology employed for the modelling and formal verification process, utilizing the OFMC and PSPSP tools. It elaborates the steps taken to construct the models and verify their security properties. Chapter 4 conducts an analysis of the results obtained from the formal verification. Chapter 5 discusses these results, providing context for the results. Additionally, it explores potential avenues for further improvement and enhancement of the study. Finally, Chapter 6 encapsulates the key points and offers a concluding overview of the thesis.

2 Background

This chapter covers the different concepts being explored in a bit more detail.

2.1 Single Sign On Systems

Single Sign-On systems within organizational settings are the primary focus of this paper, including this particular section. However, a lot of the same issues apply to individual users as well.

2.1.1 SSO systems functioning

As previously mentioned, SSO systems are typically comprised of an Identity Provider (IDP) that deals with authentication, different Relying Parties (RP) that offer a service, and multiple end users. In this framework, the IDP maintains predefined connections with each user and the supported RPs. When a user intends to log in to an RP, they are prompted to authenticate themselves with the IDP, typically by providing their credentials and using additional authentication methods. The IDP then directly communicates with the RP to assert the legitimacy of the login attempt originating from the user.

Numerous RPs will be arranged to operate with a common IDP. This implies that a user can authenticate themselves with just one IDP and subsequently gain access to services offered by multiple RPs. This concept is the underlying principle of Single Sign-On.

This paper primarily focuses on SSO systems that operate via web browsers, which is a very common mode of interaction. The core of an SSO system is the IDP, which is established by the organization that manages the user accounts. The IDP interacts directly with users through a web interface.

The IDP is typically responsible for running various protocols that each RP supports. These protocols could include Security Assertion Markup Language (SAML), OAuth 2.0, and OpenID Connect (OIDC).

RPs register themselves with the IDP, establishing a trust relationship in the process. Similarly, the end user sets up an account with the IDP. Figure 2.1 shows the trust relationship in an SSO system.

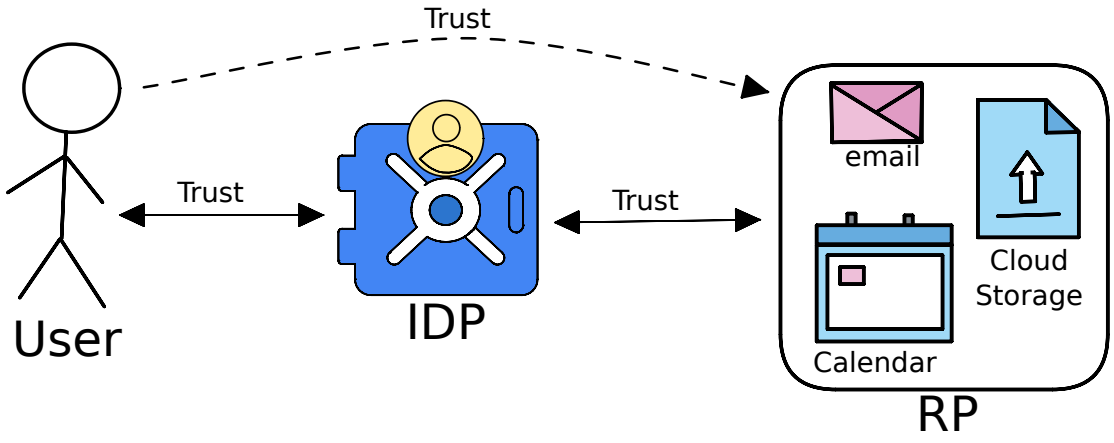

Figure 2.1: Trust relationship between the user, IDP, and RP

2.1.2 Security concerns

A significant security concern associated with SSO systems, as well as other centralized authentication systems like password managers, is their inherent nature as a single point of failure. This means that if the SSO account is compromised, it could lead to the compromise of all connected accounts. Therefore, it's crucial for users to select robust and unique passwords to safeguard their SSO account, and for organizations to enforce multi-factor authentication and other security measures.

Misconfiguration of SSO servers, including both IDPs and RPs, presents another security issue. If these servers are not configured in strict adherence to the specifications of the protocol they are implementing, it could lead to vulnerabilities [5, 6, 7, 8, 9, 10]. This also extends to potential misconfigurations in the communication between the different supported protocols.

Lastly, potential issues could arise from the protocol specifications themselves. These could be inherent flaws or vulnerabilities in the design of the protocols that could be exploited. This paper delves deeper into these potential issues, examining the protocol specifications and their potential vulnerabilities in the context of SSO systems.

In essence, while SSO systems offer numerous benefits in terms of convenience and streamlined access, it's important to be aware of these potential security concerns. Ensuring robust password practices, strict adherence to protocol specifications, and a thorough understanding of potential protocol vulnerabilities are key to maintaining the security of SSO systems.

2.2 OAuth 2.0

OAuth 2.0 is a protocol designed for authorization of data access to third party applications. This section will cover the details about OAuth in more detail.

2.2.1 Roles in OAuth 2.0

Note: the terminology used to describe the roles in the OAuth 2.0 specification [11] differs from the commonly used terminology in SSO systems. In section 2.2, the terminology aligns with the OAuth 2.0 specification. However, in all other sections, the roles will be referred to using the SSO terminology, unless otherwise stated explicitly.

The OAuth 2.0 protocol involves the participation of various distinct parties, each playing a specific role in the execution of the protocol. The roles in OAuth 2.0 are:

- *Resource Owner*: The resource owner is the end-user who owns the data that the third-party application wants to access. It is the user, typically an individual user but could also be an organization, that controls access to a protected resource, such as personal data or services. The resource owner grants permission to the client for accessing their resources.
- *Resource Server*: The resource server hosts the protected resources that the client wants to access on behalf of the resource owner. It can be an API, a database, a file server, or any other system that manages and controls access to the resources. For example, a cloud storage provider where the user's personal files are stored.
- *Client*: The client is the application or service requesting access to the resource on behalf of the resource owner. It can be a web application, mobile app, or other software. The client initiates the authorization process and interacts with the authorization server and resource server to obtain access to the protected resources. For instance, a web application that scans a users cloud storage files for malware.

- *Authorization Server*: The server responsible for authenticating the resource owner, validating their authorization, and issuing access tokens to clients. It verifies the identity of the client and the permissions granted by the resource owner. The authorization server plays a crucial role in facilitating secure and authorized access to protected resources. Typically, the authorization server is under the control of the same entity that manages the resource server.

2.2.2 Concepts in OAuth 2.0

This section provides an overview of some key concepts in OAuth 2.0 that are essential for a comprehensive understanding of the protocol and its flow.

OAuth grants

The OAuth specification covers various authentication and authorization flows designed to cater to different use cases. These include the implicit grant, resource owner password credentials, client credentials, and authorization code [11]. The focus of this paper is the authorization code grant, which is widely used in SSO systems.

Authorization Code

An authorization code is a string of characters that does not carry any inherent meaning and serves as a bearer of credentials for the resource owner. It is a single-use code provided by the authorization server and passed through the resource owner to the client. Using this authorization code, the client can communicate with the authorization server and demonstrate that the resource owner has granted them access to data, and receive an access token to access the data.

Due to the authorization code's opaque nature for the client, the specific format of the authorization code is not defined in the OAuth specifications. Implementers of the protocol have the freedom to determine the format that best suits their requirements. The crucial aspect is that the authorization server can consistently issue and authenticate authorization codes. Generally, the authorization code is securely stored within the authorization server and associated with additional transaction details for reference and verification.

The authorization code can be linked to details about the parties involved in the transaction, such as the end user, resource server, authorization server, and client. Additionally, it can be associated with details regarding the allowed scope of data access and the validity period of the code. The OAuth spec recommends that the code validity period is not longer than 10 minutes [11].

Access Token

The access token is a credential issued by the authorization server directly to the client. Just like the authorization code, the client usually perceives this token as a meaningless string, but it holds significance in the authentication process. The client can exchange this token with the resource server to obtain access to the data owned by the resource owner.

The access token can take several forms, but the most common types are bearer tokens [12] and message authentication code (MAC) tokens. Generally, possession of the access token enables data retrieval to whoever possesses it, without further authentication requirements. The presence of an authorization code as an intermediate step in the OAuth flow is designed to address the security concern of exposing the access token to potential attackers by transmitting it through the resource owner's web browser (or other user agent).

Although the OAuth specification offers guidelines for implementing OAuth Bearer Tokens [12], implementers have the flexibility to choose any format they prefer. The primary requirement is that the client, resource server, and authentication server mutually agree on

the token format. However, it is essential for the authorization server and resource server to establish a secure mechanism for validating the legitimacy and integrity of authorization codes submitted by clients to the resource server. This critical aspect is covered by the introspection endpoint [13], which ensures the verification of authorization codes.

Token Introspection
Token introspection [13] is a vital mechanism that allows the resource server to gather detailed information about a token. In OAuth, tokens are often represented as opaque strings, lacking inherent meaning or visibility to the client as well as the resource server. As a result, token introspection plays a crucial role in enabling the resource server to validate the token and ascertain that it is providing the appropriate resources to the client under the correct conditions.

Through token introspection, the resource server can query the authorization server to obtain relevant details about the token. This allows the resource server to verify the authenticity and validity of the token, ensuring that it has not been tampered with or expired. Additionally, token introspection provides insights into the scope and permissions associated with the token, enabling the resource server to make informed decisions regarding the resources it grants access to.

Refresh Token
Typically, access tokens have a short expiration period. In these cases, to address the inconvenience of frequent re-authentication by the resource owner when using the client application, refresh tokens are allowed, but not mandated. These tokens enable the client to obtain new access tokens from the authorization server without requiring any involvement from the resource owner.

When an access token expires, the client can submit the refresh token to the authorization server, along with client authentication using the client ID and secret. In response, the authorization server grants a new access token and a fresh refresh token to the client. The client can then utilize the new access token and retain the new refresh token for subsequent renewals when the access token expires again.

Client ID and secret
Every client that is registered with the authorization server receives a client ID. This client ID is unique for each authorization server and client, ie: the same client id cannot be used by the client with another authorization server. The client ID alone is not enough to authenticate the client with the authorization server.

The client authenticates itself with the authorization server using a a client password, also referred to as a client secret. This secret is typically provided by the authorization server during client registration. The combination of this client secret and the client ID forms the basis for client authentication.

Note: the OAuth specification allows for alternative methods for client authentication, but this paper focuses on client ID and secret-based authentication.

Client state
The *state* is a unique, non-guessable string, generated by the client and sent to the resource owner during the initiation of the authorization request. This *state* value is maintained throughout the process until the client receives the authorization code. It serves multiple purposes:

Primarily, it is used to safeguard against cross-site request forgery attacks (CSRF). It does this by ensuring that the callback received from the resource owner corresponds to the original request initiated by the client.

Moreover, the *state* parameter can be used by the client to maintain uniqueness and distinguish between various OAuth threads for individual end-users. This guarantees that each OAuth process is unique and identifiable. The 'state' parameter can also serve as a container for additional information, such as specifying the precise section of a web page to which the resource owner should be directed upon authentication.

Redirect URI
The redirect URI is the Uniform Resource Identifier (URI) on the client application where the resource owner is redirected after being granted an authorization code. It serves both as a user-friendly way to direct the resource owner to the appropriate web page, especially when used with the state parameter, and as a security measure against phishing attacks.

During the client registration process, the client is required to register the redirect URI with the authorization server. When making an authorization request, the client must include the same redirect URI. The authorization server verifies that the redirect URI in the request matches the registered one, ensuring that the resource owner is redirected to the correct location. This mechanism adds an additional layer of protection, preventing attackers from successfully redirecting the resource owner to a malicious URI and obtaining their authorization code even if the resource owner falls for a phishing attempt.

Scope
The scope is used to specify the precise access permissions that the client is seeking from the resource owner. It outlines the boundaries and limitations of the resources that the client will be granted access to.

Each individual scope option is represented by a distinct string denoting a specific type of resource within the resource server. The exact strings and the permissions associated with these strings are defined by the authorization server and vary across implementations.

During the authorization process, the resource owner is typically presented with the requested scopes on a consent screen. This allows the user to selectively modify the approved scopes, or accept or decline the access request in its entirety.

2.2.3 Data Flow in OAuth

As per the OAuth 2.0 specifications, certain transactions in the OAuth framework mandate the use of the latest widely adopted version of Transport Layer Security (TLS) [14], which is presently version 1.3 and version 1.2. In other transactions, TLS is recommended but not explicitly required. Nevertheless, in practical implementations of OAuth, TLS is employed across the board, just as it is in the majority of web services. This ensures that every message exchanged between the involved parties is encrypted, blocking eavesdropping by attackers.

In this thesis, it is assumed that TLS is employed for all transactions within the OAuth framework.

Authorization Request
As we can see in Figure 2.2, the data flow in OAuth begins with the resource owner (end user) initiating a data access request at the client. The client then redirects the resource owner to the 'authorization endpoint' of the authorization server with all the details needed for the request. This includes the client ID, the redirect URI, and optionally, the scope. The authorization server validates all the details provided and then proceedes to authenticate the resource owner.

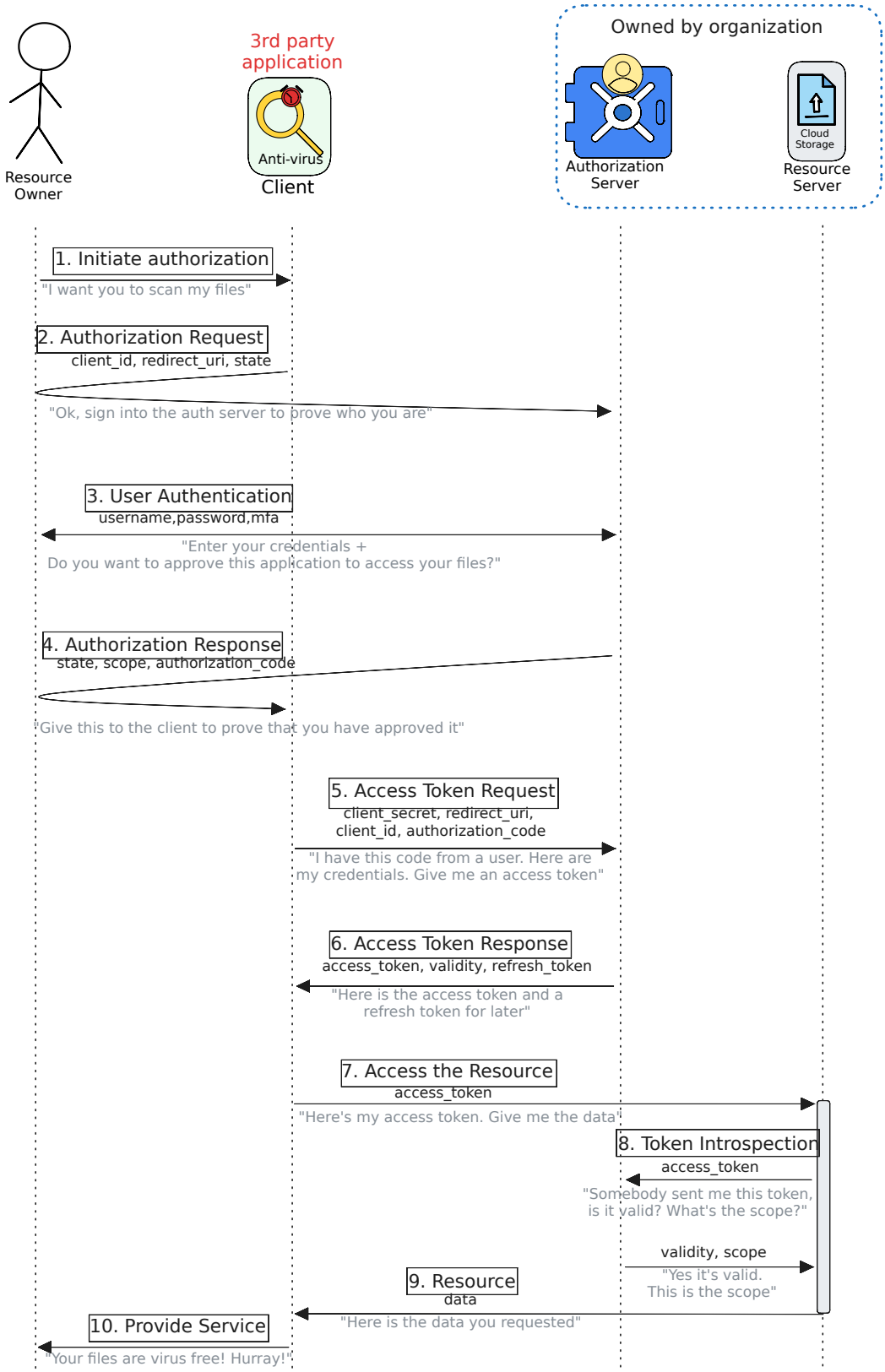

Figure 2.2: The data flow in the OAuth 2.0 authorization code grant [15]

Authenticating the Resource Owner
The authentication of the resource owner is not covered in the OAuth specification and can be done in any way the authorization server sees fit. This could include password authentication, biometric authentication, or multi factor authentication (MFA). If the resource owner has an active valid session cookie with the authorization server, they will not have to re-authenticate themselves. However, to prevent CSRF attacks on the authorization endpoint of the authorization server, the authorization server is required to get some explicit approval from the resource owner for the OAuth request to this specific client. This way an attacker will not be able to get an authorization code without explicit approval from the user.

Authorization Response
After successfully authenticating the resource owner, the authorization server provides the client with an authorization code. This code is transmitted to the client via the resource owner's user agent as part of the authorization response. The authorization response, is a redirection response to the client's designated redirect URI. This response encompasses both the scope of access, the state parameter, and the authorization code, providing the necessary information to proceed with the subsequent steps of the OAuth flow.

Access Token Request
Upon receiving the authorization code from the resource owner, the client performs a validation check to ensure that the state parameter matches the one it generated during the initial authorization request. Once validated, the client proceeds to exchange the authorization code for an access token. This exchange is done through direct communication between the client and the authorization server's token endpoint.

To authenticate itself, the client includes its client secret in the authorization header of a POST request sent to the token endpoint. This request includes the client ID, redirect URI, and authorization code it received.

During the token exchange process, the authorization server performs a series of checks. It verifies that the client secret is correct, ensuring its legitimacy. It checks the authorization code to confirm its validity and expiration status. Additionally, the authorization server confirms that the authorization code was issued to the client making the request and matches the correct client ID and redirect URI.

Access Token Response
Upon successful validation of the access token request, the authorization server provides the client with the actual access token. The response from the authorization server also includes the expiration time of the access token. Additionally, the response may optionally contain a refresh token.

Accessing the Resource
In possession of the access token, the client proceeds to approach the resource server to request access to the desired resource. The access token is included in the client's request to the resource server. Since the client has already been authenticated during the authorization process, there is no need for further client authentication, and therefore the client ID is not required. Instead, the resource server verifies the access token with the authorization server.

During this verification process, the resource server confirms the authenticity of the access token, checks its scope and expiration to ensure it is still valid. The specifics of how the resource server validates the token with the authorization server are not covered by the OAuth specification, as it varies depending on the implementation and configuration.

Overall, this process ensures secure and authorized access to the requested resource

without exposing sensitive information. The limited validity period of access tokens reduces the risk of extended misuse from issuing a bearer token.

2.2.4 The Role of OAuth 2.0 in SSO Systems

As discussed above, the OAuth 2.0 protocol primarily focuses on granting access to protected resources rather than serving as a dedicated user authentication protocol for SSO systems. It does not provide a direct mechanism for explicitly verifying the user's identity.

In contrast, the OpenID Connect (OIDC) protocol was specifically designed as an extension of OAuth 2.0 with the explicit purpose of user authentication [16]. It introduces an ID token, which is an access token type dedicated to asserting the authentication claim of the user.

However, while OAuth 2.0 was not initially designed for user authentication, it is clear that it can still be employed for authentication purposes. For instance, by limiting the scope of access to a specific piece of data that serves as proof of the user's claimed identity. There are several alternative methods available to achieve the common objective of authentication using OAuth.

Token Introspection

To enable OAuth for authentication purposes, a common approach is to expose the introspection endpoint of the authorization server to allow client querying [17]. By doing so, the client can be provided with a token without any specific scope, while still utilizing the authorization server to validate the token's association with the correct user. This validation process can yield essential information, including a unique username and other relevant supporting data.

In such scenarios, clients may solely interact with the authentication server without directly communicating with the resource server. This configuration allows for a streamlined authentication flow where the client primarily communicates with the authentication server for the verification and validation of tokens, ensuring the accurate identification of the associated user.

When the resource server is not present, the roles in the OAuth protocol bear a striking resemblance to the roles found in SSO systems. In this context the resource owner corresponds to the end user, the client takes on the role of a relying party (RP), and the authorization server assumes the role of an identity provider (IDP).

For the remainder of the paper, this perspective of OAuth as an SSO solution through the use of token introspection will be adopted. The terminology used to refer to the roles in OAuth will align with this approach, taking the same naming convention as SSO systems.

2.3 Transport Layer Security

TLS is a protocol used to facilitate secure communications between two parties over a network [14]. Although TLS can be used over any communication protocol, this paper will focus on the use of TLS within the client-server paradigm in the context of the internet services using the TCP/IP protocol stack. Moreover, TLS consists of two layers, the TLS Handshake Protocol, responsible for initiating the secure communications, and the TLS Record Protocol, responsible for ensuring a continued secure and reliable connection [14]. This paper will focus on the TLS Handshake Protocol and abstract away the functioning of the TLS Record Protocol, assuming that it works as expected.

Although the latest TLS version is 1.3, this paper will primarily delve into TLS version 1.2. The choice of TLS 1.2 is driven by the limitations of the tools utilized for implementing this

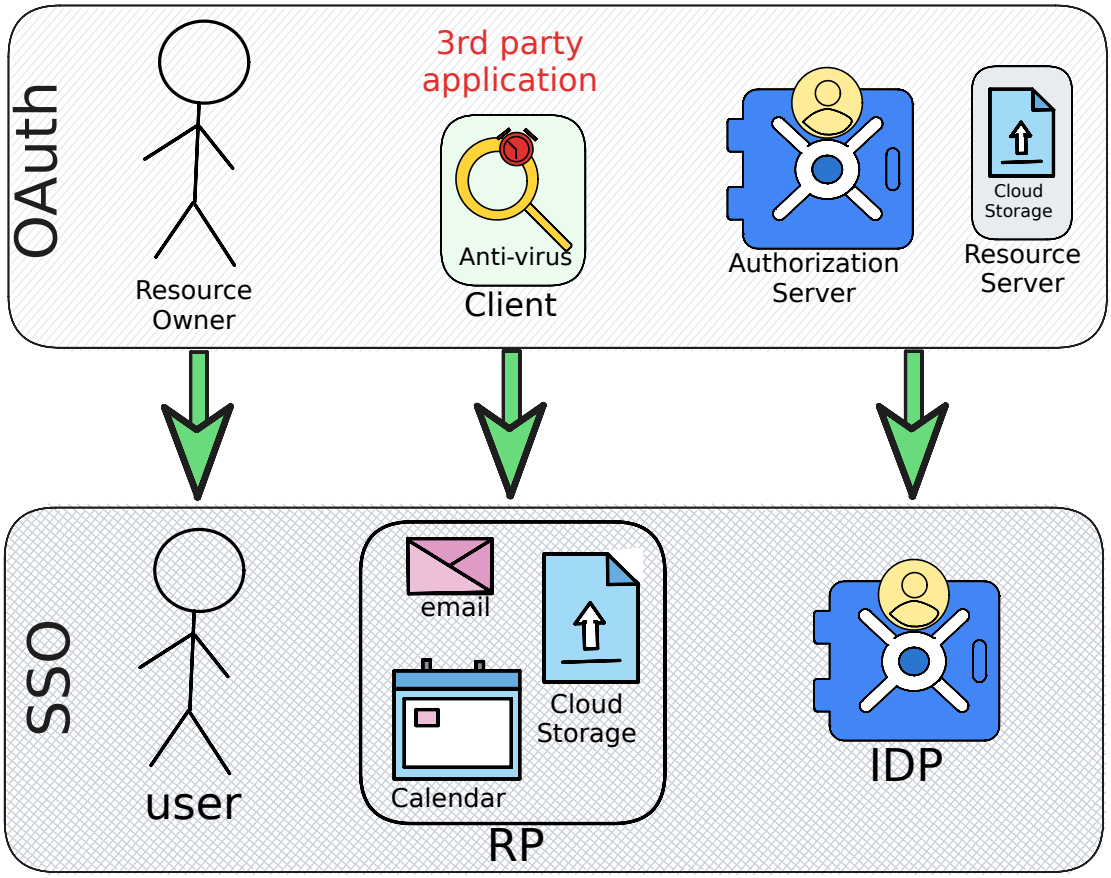

Figure 2.3: The OAuth roles in an SSO context

paper. Therefore, throughout the paper, the term *TLS* will specifically refer to TLS version 1.2, unless explicitly specified otherwise. The TLS 1.2 handshake consists of four round trip messages. Here is a high level overview of the messages in each round trip:

1. Client Hello: The TLS handshake begins with the client initiating the process. The client sends a message to the server, specifying the list of supported ciphers and the TLS version it can handle. Additionally, the client generates a random number called the client random and includes it in the message.

2. Server Hello: In response, the server selects the appropriate TLS version and ciphers for the session. It sends a message back to the client, indicating the chosen TLS version and ciphers. The server also generates a random number called the server random and includes it in the message. The server provides its signed certificate in another message. If the server doesn't support the TLS version requested by the client, it sends a failure message and terminates the session.

3. Client Response: Upon receiving the server's message, the client verifies the server's certificate by checking if it was signed by a trusted certificate authority (CA). The client then selects a randomly generated value, the premaster secret (PMS), and encrypts it using the server's public key obtained from the server's certificate. This encrypted PMS is sent to the server as part of the ClientKeyExchange message. The client also derives the session key using the client random, server random, and PMS. Next, using the ChangeCipherSpec message, the client informs the server that all future messages within this session will be encrypted using the agreed-upon

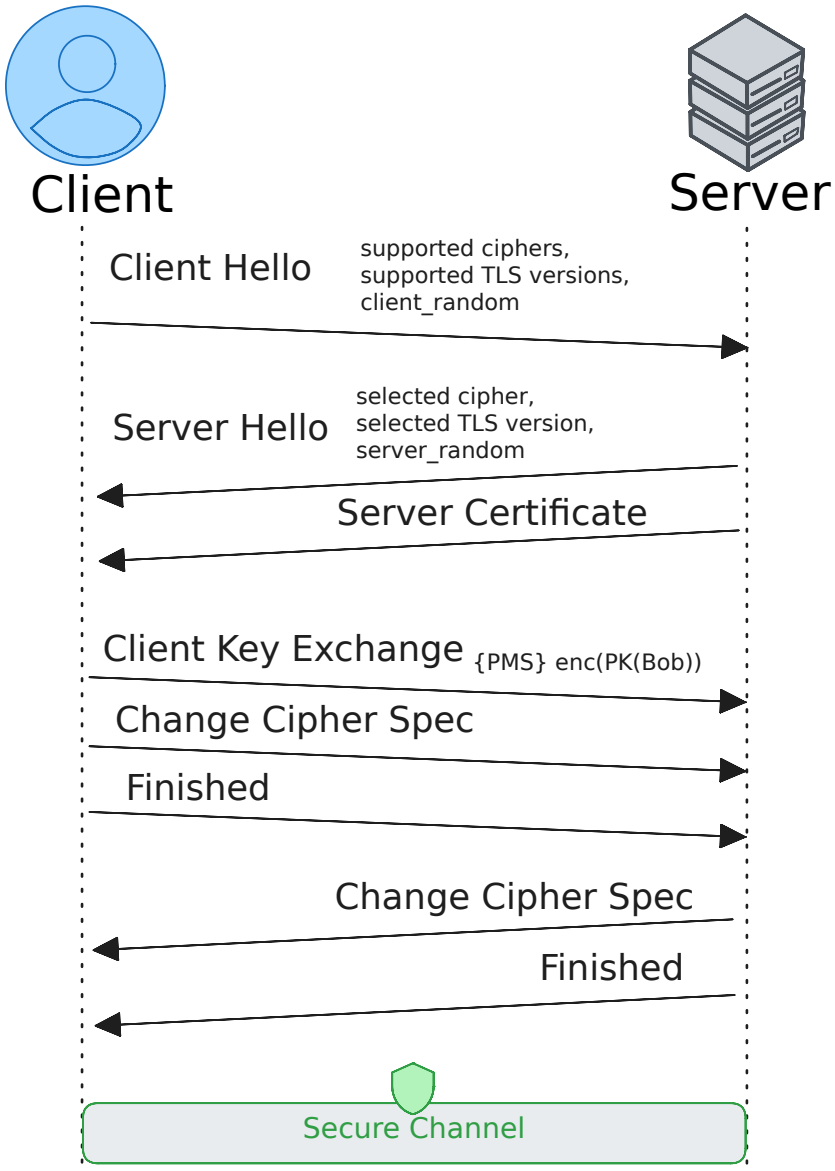

Figure 2.4: Data flow in version TLS 1.2

ciphers and session key (which the server will be able to derive, given the PMS). Finally, the client sends a *Finished* message to indicate that it's part of the handshake is done.

4. Server Response: The server receives the encrypted PMS from the client and decrypts it using its private key. Using the client random, server random, and decrypted PMS, the server computes the session key. It then sends a message to the client, confirming that all subsequent messages will be encrypted using the newly established session key and cipher. It sends the client a *Finished* message and now the two parties have a secure channel to communicate over.

After completing this exchange, both the client and the server can generate a session key using a key derivation function. This session key enables them to employ symmetric cryptography with the agreed-upon ciphers, ensuring the encryption of all exchanged messages. As a result, despite relying on an inherently insecure TCP connection, TLS

facilitates secure communication between two parties within an insecure environment.

2.4 Formal Verification

As software systems grow increasingly complex, ensuring that they operate as intended becomes more challenging. Additionally, guaranteeing their functionality without any security vulnerabilities adds extra complexity.

The testing of such systems can be performed using various methods. One approach involves manual testing, where the software designer and/or their peers meticulously review the design and implementation to identify any errors or issues. Another method is unit testing [18], where the system authors create specific test cases to ensure the expected functionality.

However, users may interact with the system in unanticipated ways, and it becomes challenging for manual testers to anticipate all possible user interactions, hampering their ability to fully validate the system's behavior. Automated testing tools can simulate pseudo-random user inputs and behaviors to analyse system outputs [19], but they also have their limitations. Conventional testing approaches may struggle to adequately capture complex scenarios such as network communications involving multiple messages and potential attackers.

Formal verification is a process that allows one to establish proof of a system's correct behavior. This can be extended to both software and hardware systems, as well as combinations of the two. While formal verification can be applied to software products in general, this paper will specifically explore it in the context of network communication protocols.

2.4.1 Model Checking

Model checking is a powerful technique used to algorithmically verify the correctness of systems that have a finite number of states [20]. This process involves employing a dedicated program or tool called a model checker. These tools use various methods to verify the model, one frequent method being systematically exploring all possible states that the system can reach and examining each state to determine if it complies with the specified correctness criteria.

It is crucial to note that model checking does not directly validate the actual system but rather verifies a model of the system. This model is a simplified representation of the system, that focuses on the logical aspects rather than the specific code, syntax, or implementation details. Therefore, one of the most critical aspects of model checking is ensuring that the model accurately reflects the actual system. If the model is inaccurate or flawed, the results obtained from the model checker cannot be reliably interpreted or trusted.

The other important aspect of model checking is to provide an accurate representation of the expected behavior of the system, the property specification. This includes defining the system requirements and specifying the conditions that constitute errors. It is essential to have well-defined and comprehensive error definitions to ensure that the model checker can effectively identify and flag potential issues. Without clear and precise error definitions, the model checker may raise false positive errors or miss out on true positive errors. Therefore, careful and thorough consideration of error conditions is vital to maximize the effectiveness of model checking and obtain reliable results.

A model checker can systematically explore numerous states in the model to identify errors. The discovery of an error can indicate one of the following:

1. An error in the model, which is linked to an issue in the system design
2. An error in the model that cannot be replicated in the actual system, either due to limitations in the model or its inability to fully represent the real-world system
3. Inaccurate property specifications in the model that raise an error that is not of concern in the actual system.

Similarly, if a model checker fails to detect any errors in the model, it could be due to the following reasons:

1. There are no errors in the system.
2. The model checker is unable to capture certain errors in the system due to inaccuracies in the model's design or limitations in the model's capability to represent the complexities of the real-world system.
3. The property specifications used in the model were not sufficiently precise to identify a potential error.

By using an accurate model and specifications, model checking enables the identification of potential errors, flaws, or unexpected behavior in the system being analysed.

2.4.2 Proof Assistants

A limitation of model checkers, as mentioned in Section 2.4.1, is their constraint to verify models with a limited number of states [21]. As the complexity of a model increases, resulting in a larger number of states, the computational resources required to exhaustively cover all states also significantly escalate. Consequently, to ensure reasonable verification times, it may be necessary to impose restrictions on the number of states when employing a model checker. For example, reducing the number of participants, messages, and functions allowed in a networking protocol. However, this restriction may not be feasible or practical in certain systems, and could compromise the utility of the model checker in other cases.

This is where proof assistants come in. They are built upon the foundation of formal logic [22], which has long been used to prove theorems in academic disciplines such as mathematics and physics. Similar to hand-written proofs, proof assistants define their own language to express theories and algorithms. By defining the system within the rules of the proof assistant's language in a precise logical form, complex mathematical reasoning can be applied to analyse properties and relationships within the system.

Proof assistants usually operate on higher-order logic, allowing for intricate analysis of logical consistency within the system. Rather than iterating over every possible state of a system, like model checkers do, they logically try to disprove the input, looking for improper assertions, leaps in logic, and other logical issues. This offers a robust means of verifying complex systems, algorithms, or theories with a high degree of certainty in it's output.

Note that the specific implementation details of proof assistants extend beyond the scope of this paper.

2.4.3 Composition of Protocols

Communication protocols in the modern age are complex, operating over the Internet Protocol suite, Transmission Control Protocol (TCP)/Internet Protocol (IP), and often interacting with additional protocols like TLS and the Hypertext Transfer Protocol (HTTP).

Individually, these protocols are quite complex, which further intensify when combined, presenting significant challenges for verification [23].

To tackle this complexity, various approaches can be employed. One option is to analyse the entire stack of protocols together, considering the extensive interdependencies and modelling the message flow across each layer. However, this approach carries the risk of errors due to the system's vast size and complexity, along with the performance overhead of verifying such a comprehensive protocol.

Alternatively, simulating the impact of underlying protocols on the final protocol can be employed. This involves making assumptions about how the underlying protocols influence the overall system's behavior and incorporating these assumptions into the model. Nevertheless, these assumptions are not always valid and they may introduce verification issues [24].

Another approach involves conducting independent analyses of each protocol, either for all or a selected subset, before composing the results together. This allows for a focused examination of each protocol's properties before integrating them into the larger system. However, it is crucial to acknowledge that one protocol's behavior can impact another, necessitating consideration of their interplay. This is where compositionality becomes important. By defining each protocol separately and establishing their relationships, it becomes possible to verify them both individually and collectively.

2.5 Dolev Yao Attacker

The Dolev-Yao attacker model, originally proposed by D. Dolev and A. Yao, is a model for a type of adversary encountered in networked communications [25]. This attacker possesses complete control over the network, enabling them to eavesdrop on, intercept, modify, delete, or fabricate any messages transmitted on the network. Their capabilities extend to cryptographic operations and they possess unbounded computational power. However, the attacker is unable to break sound cryptographic algorithms. Essentially, the attacker has unrestricted authority over the network, rendering it untrustworthy for the participants. Hence, it is crucial to employ cryptographic primitives to safeguard messages from being read by the attacker and to verify the integrity of messages against unauthorized modifications.

To achieve message confidentiality, preventing the attacker from reading the messages, encryption can be utilized by the parties. To ensure that they can detect any modifications in the messages, to maintain integrity, the parties can employ methods such as MACs or digital signatures.

Enacting such an attack on the modern internet is not feasible due to the significant requirement of gaining control over numerous internet service providers (ISPs), routers, and domain name servers (DNS). However, control over a few key points in the network gives an attacker a significant amount of power.

This paper assumes a Dolev Yao attacker exists on the network.

2.6 Needham-Schroeder Public Key Protocol

The Needham-Schroeder Public Key (NSPK) protocol was developed by R. Needham and M. Schroeder to ensure that two parties can authenticate each other on a network [26]. This protocol is used to serve as a demonstration of the modelling languages selected for the formal verification in this thesis. The protocol follows the following message flow:

1. Alice initiates the protocol by sending Bob a new random nonce, NA, along with her identity, A. The entire message is encrypted using Bob's public key.
2. Bob decrypts the received message and responds to Alice by sending back the nonce NA and a new nonce NB. The entire response message is encrypted using Alice's public key. This exchange serves as proof to Alice that the sender is indeed Bob, as only Bob can decrypt the previous message and retrieve the value of NA.
3. Alice decrypts the response message from Bob and sends Bob the nonce NB, encrypted with Bob's public key. Following the logic above, this step serves as proof to Bob that the sender is genuinely Alice.

At the end of this exchange, Alice and Bob can know that the other party is indeed who they think, ie they have authenticated each other. This holds true as long as the public keys that they posses is the correct public key.

2.7 Formal Verification Tools

This section provides an overview of the tools utilized for the formal verification of OAuth and TLS in this thesis, namely the OFMC and PSPSP tools.

2.7.1 OFMC

The Open-Source Fixedpoint Model-Checker, OFMC, is a model checker designed specifically for analysing security protocols [3]. Its primary objective is to offer a comprehensive, user-friendly, and efficient approach to protocol analysis. OFMC achieves this by utilizing a simplified, almost graphical notation to depict the protocol steps, while incorporating several enhancements to enable comprehensive and swift analysis of protocols, even those with vast or potentially infinite state spaces.

OFMC employs an "Alice and Bob notation", representing commonly used names for two parties involved in a protocol. This notation encourages the author to document the protocol steps in a manner similar to creating a sequence diagram, allowing for easy analysis of the protocol and identification of errors in entry. Additionally, this notation enables bystanders or learners to quickly grasp the protocol's details.

Furthermore, OFMC offers a straightforward means to define the pre-existing knowledge within the system, including secrets, functions, and participants. Furthermore, it allows for the specification of the security requirements that the protocol needs to fulfill.

Example: NSPK algorithm
The following is an example of the NSPK protocol, as seen in section 2.6, in OFMC.

```
Protocol: NSPK

Types:
    Agent A,B;
    Number NA,NB;
    Function pk

Knowledge:
    A: A, pk, inv(pk(A)), B;
    B: B, pk, inv(pk(B))

Actions:
    A -> B: {NA,A}(pk(B))
    B -> A: {NA,NB}(pk(A))
    A -> B: {NB}(pk(B))

Goals:
```

```
    B authenticates A on NA
    A authenticates B on NB
    NA secret between A,B
    NB secret between A,B
```

The *types* section serves to define the various *agents* involved in the protocols, the *numbers* representing variables, and the *functions* utilized.

In the *knowledge* section, the accessible *types* of information for each agent are specified. The `inv()` function denotes that the agent possesses knowledge of the inverse of the specified value. The *numbers* are not explicitly enumerated here, as they are variables that can take on any value.

The *actions* section is responsible for describing the protocol. It is presented in a straightforward, easy to understand, manner. The `{` and `}` braces here indicate that the message is being encrypted using the key specified in the brackets that immediately follow it.

The *goals* section outlines clearly defined objectives for the protocol that are easily comprehensible.

2.7.2 Isabelle HOL

Isabelle is a geneic proof assistant that supports the use of different types of logic to prove mathematical theorems [27][28]. Specifically, Isabelle/HOL (Higher Order Logic) is the specification for working with Isabelle in conjunction with HOL. HOL, is a formal logic system extensively used for mathematical reasoning [29]. It is used to represent and reason about systems or theorems by using objects, properties, relationships, and functions to describe the system.

In Isabelle/HOL, HOL serves as a functional programming language that enables the expression of simple proofs. Additionally, Isabelle introduces Isar, a language that is specifically designed for writing more extensive, complex, and structured proofs. While Isabelle and Isar are powerful tools on their own, a significant advantage of Isabelle is its ability to create and incorporate other logic systems and plugins. This flexibility allows users to harness the underlying strength of Isabelle and extend its capabilities according to their specific needs.

2.7.3 PSPSP

The Performing Security Proofs of Stateful Protocols (PSPSP) theory is specifically designed for performing formal verification of communication protocols in Isabelle/HOL [4]. In contrast to model checkers that exhaustively explore every possible state of a model to determine the precise set of states where the attacker possesses information, PSPSP adopts a few approaches to more efficiently verify the model [30][31]. This results in abstracting all the states into sets and to obtain an ”over-approximation” of the sets with information accessible to the attacker, called the *fixed point*.

Equipped with the sets containing all the information available to the attacker throughout the protocol flow, PSPSP can verify whether the conditions for an attack, as defined in the model specifications, can be satisfied at any given point. By using this approach, PSPSP can be far more efficient compared to exhaustively exploring the entire state space, all the possible states the model can be in, of the model, as some traditional model checkers do. This approach allows for the analysis of protocols in which the state space of the model has the potential to grow quickly, reaching an exceedingly large or even infinite size.

Additionally, PSPSP offers the capability to maintain long-term mutable state and automate the verification of the modeled protocol. With long-term mutable state, sets within the model can be used to store different values among participants, enabling the simulation of long-term cookies, databases, or similar functionalities. The automated verification feature of PSPSP eliminates the need for manual proofs, such as induction, and ensures accuracy in assessing various aspects of the protocol and the composition of multiple protocols. This mitigates the risk of introducing errors during the verification process. These features significantly contribute to the power, robustness, and reliability of PSPSP in protocol analysis.

If PSPSP/Isabelle does not detect any attacks, it can be concluded with a high level of confidence that there is no attack present within the model. However, the absence of an attack in the model does not guarantee the absence of an attack in the protocol itself, as it relies on the accuracy of the model and specifications in representing the actual protocol.

PSPSP offers a high-level language called *trac*, which allows for the description of the protocol using a more abstract representation. While trac differs from the simplicity of the *Alice and Bob notation* style used in OFMC, it still provides an easy to understand way of describing the protocol.

Example: NSPK algorithm
Unlike OFMC, the NSPK proof in PSPSP is much more verbose to define. The protocol setup state looks like this:

```
Enumerations:
    honest = {a,b}
    dishon = {i}
    agent = honest ++ dishon

Sets:
    a_challenge/2
    b_response/2
    nspk_used/2

Functions:
    Public crypt/2 aInit/2 bResp/2 aResp/1 pk/1
    Private inv/1

Analysis:
    aInit(X, Y) -> X, Y
    bResp(X, Y) -> X, Y
    aResp(X) -> X
    crypt(X, Y) ? inv(X) -> Y
```

The protocol definition begins with the *enumerations* part, which includes the different kinds of participants involved in the protocol. Honest and dishonest participants can be defined independently, and participants can be labeled using arbitrary labels. This section also allows for the creation of ”meta” groupings, such as the *agent* group demonstrated here.

Following that, the *sets* part describes the different sets used to store data in the protocol. These sets serve as containers for short-term values like nonces or long-term secrets like private keys. Values can be added or removed from these sets as needed. The `/2` accompanying some set definitions indicates the number of agents associated with each set. Multiple instances of each set can exist, associated with different combinations of agents.

The *functions* part defines the various functions utilized in the protocol and specifies the number of parameters each function takes. Functions are categorized as public or private. Public functions can be invoked by the attacker with any desired parameters, while private functions cannot be invoked by the attacker. In the provided example, the `inv()` function retrieves the inverse public key, the private key, of an agent, and it is a private function that the attacker cannot invoke for any agent.

Although the attacker cannot directly invoke these private functions, they can still use the resulting values of these functions if they gain access to them.

The *analysis* part defines how the attacker can interpret a function whose value they have access to. It explains what the attacker can deduce from intercepting a message containing that function. For instance, `crypt(X, Y) ? inv(X) -> Y` indicates that if the attacker has a message with the crypt function and possesses `inv(X)`, they can obtain the value `Y`.

The actual protocol part of the PSPSP proof looks like this:

```
Transactions:

initialKnowledgeDis(A: dishon)
    send inv(pk(A)).

aInit(A: honest, B: agent)
    new NA
    insert NA a_challenge(A,B)
    send crypt(pk(B),aInit(NA,A)).

bResp(A: agent, B: honest, NA: value)
    receive crypt(pk(B),aInit(NA,A))
    new NB
    insert NB b_response(B,A)
    send crypt(pk(A),bResp(NA,NB)).

aResp(A: honest, B: agent, NA: value, NB: value)
    receive crypt(pk(A),bResp(NA,NB))
    NA in a_challenge(A,B)
    insert NA nspk_used(A,B)
    delete NA a_challenge(A,B)
    send crypt(pk(B),aResp(NB)).

bFinal(A: agent, B: honest, NB: value)
    receive crypt(pk(B),aResp(NB))
    NB in b_response(B,A)
    delete NB b_response(B,A)
    insert NB nspk_used(B,A).

attack_naSecrecy1(A: honest, B: honest, NA: value)
    receive NA
    NA in a_challenge(A,B)
    attack.

attack_naSecrecy2(A: honest, B: honest, NA: value)
    receive NA
    NA in nspk_used(A,B)
    attack.
```

In this representation, the *transactions* are described not in the Alice and Bob notation but

as distinct actions within the system. The specification does not explicitly indicate whether A or B is the sender of the message, as PSPSP does not consider such specifics. However, the implied roles of the participants can be inferred based on how each transaction is defined.

Each transaction defines the relevant agents and values involved in the operation. The transactions encompass various operations such as *receive* to receive messages from the network, *insert*, *in*, *notin*, and *delete* to interact with sets, *new* to create new values, and *send* to transmit messages onto the network.

Messages sent over the network can be analysed by the attacker, and the attacker can invoke transactions in any order. The model should specify the information accessible to the attacker, for example, by using a dishonest participant to reveal their private key. By doing so, the attacker gains the ability to decrypt messages sent to the dishonest participant, hence emulating a Dolev-Yao attacker who is also a participant in the protocol.

Finally, the attack definitions are represented as their own transactions. If the attacker can trigger these transactions and successfully reach the *attack* keyword, the protocol proof is halted, and an attack is considered to be executed.

Note that the above code only consists of the trac language file that focuses on the transaction definition. There are additional components in the Isabelle/HOL proof process, including invoking PSPSP with the appropriate logic, specifying the protocol name, triggering fixed point computations, and other related tasks. It is possible to define this trac segment in a separate file and integrate it into the `.thy` file of Isabelle/HOL for a comprehensive protocol analysis.

2.8 Related Works

The field of formal security verification plays a crucial role in ensuring the reliability of communication protocols that are widely used by individuals and organization. As a result, it remains an active and important area of research. The OAuth 2.0 protocol, being a widely adopted standard for internet authentication and authorization, has garnered significant attention and has been the subject of numerous studies.

One notable study conducted by Fett et al. [15] aimed to comprehensively analyse the OAuth 2.0 protocol. The research identified several security issues within the protocol and proposed remedies for these vulnerabilities. This analysis encompassed a more realistic and comprehenive environment that modelled aspects such as the HTTP and HTTPS standards, web browsers, and other relevant concepts like web storage, iFrames, and JavaScript. By employing this model and simultaneously analysing all OAuth grant types, the study successfully identified four attacks in the protocol. This study served as a valuable foundation for initiating this thesis, primarily due to its employment of a much more comprehensive model that covers a broader range of OAuth components and real-world internet mechanisms. One of the attacks identified in this study, the *307 redirect attack*, was intentionally incorporated into the model created in this thesis as well.

Other studies have utilized various formal verification tools to analyse the OAuth protocol [32, 33], uncovering potential issues within its design. However, it is worth noting that these studies often rely on certain assumptions. One is that attackers have the capability to modify messages on the network, even when protected by TLS. Another is the presence of scenarios where attackers can compromise the security of devices used by honest RPs or users during the protocol execution. While these studies contribute valuable insights, such assumptions are not made in this thesis.

Furthermore, many studies have delved into the analysis of different OAuth 2.0 implementations, highlighting existing flaws and vulnerabilities within them. Some of these studies employ formal analysis methods to scrutinize the security of various OAuth implementations[10], and other use more traditional methods to test the security of implementations [6, 8, 7, 9]. While these investigations often uncover issues within specific implementations, it is important to note that these findings are typically a result of inadequacies in the implementation process itself, rather than inherent shortcomings in the OAuth protocol. These shortcomings can be attributed to implementers who may have overlooked or inadequately addressed important security considerations in their code.

3 Modelling and Verifying OAuth 2.0

This chapter delves into the practical execution of the formal verification process for the OAuth 2.0 protocol using Isabelle/HOL and PSPSP. It provides a detailed description of the modelling process, with the inclusion of OFMC as a means to enhance understanding prior to the utilization of PSPSP.

3.1 The need for both PSPSP and OFMC

The choice of using both tools for formal verification may raise questions regarding the necessity of using PSPSP when OFMC appears to be more approachable. Both tools are indeed powerful and reliable, providing output that can be trusted with a high degree of confidence. Additionally, OFMC offers the advantage of using the intuitive AnB language for protocol modelling and provides an easy-to-understand attack trace, which helps to diagnose the exact mechanism that the attacker uses to exploit the protocol. With these features, it might seem unnecessary to utilize PSPSP for analysis when OFMC appears to fulfill the requirements.

PSPSP offers a significant advantage in this particular use case: the ability to perform stateful composition. This feature proves invaluable when incorporating the creation and deletion of data, such as long-term secrets. This ability to model mutable long-term state proves valuable in modelling various aspects of protocols like OAuth and TLS. In the OAuth protocol, it can be employed to model the client state, authorization codes, and access tokens. Similarly, in the TLS protocol, PSPSP enables the modelling of crucial components such as the client random, server random, and session keys shared between clients and servers. By accommodating these elements, PSPSP enhances the depth and accuracy of the analysis, enabling a more comprehensive assessment of the protocol's security properties.

However, the complementary use of OFMC alongside PSPSP remains valuable. OFMC's simplicity and ease of implementation allow for a focused approach to the modelling process, unburdened by the intricacies of the modelling language and tool features. This minimizes the risk of errors during modelling. Furthermore, the straightforward nature of OFMC serves as a sanity check, reinforcing the results obtained from PSPSP. This additional layer of protection mitigates the potential for undetected attacks in the PSPSP analysis.

3.2 Modelling OAuth 2.0 in OFMC

Since OFMC is arguable easier to set up and begin modelling with, the first step taken in this formal verification process was to model OAuth 2.0 in the context of SSO by using OFMC.

Given that the main objective of the project was to model OAuth, the simplicity of the OFMC tool proved beneficial in mitigating potential challenges associated with the modelling process. The complexity primarily resided in accurately representing OAuth within the formal verification framework, hence, the ease-of-use of OFMC assisted in navigating and addressing these challenges effectively. This step served as a preliminary assessment and a solid foundation from which to develop the PSPSP model and verification.

The steps to execute the modelling process were as follows:

1. *Understanding the context*:
   (a) Identifying the requirements and functioning of a typical SSO system.
   (b) Gaining a comprehensive understanding of the OAuth 2.0 protocol within its intended context: resource sharing authorization.
2. *Adapting OAuth to SSO*: Grasping the necessary modifications needed in the OAuth 2.0 protocol to enable its usage as an authentication protocol in the SSO system context.
3. *Identifying essential protocol messages*: Determining the crucial messages in the protocol that necessitated modelling, while identifying any messages that could be omitted without compromising the analysis.
4. *Identifying protocol types and knowledge*: Identifying the involved participants in the protocol, along with their pre-existing knowledge and the functions necessary for simulating the protocol.
5. *Identifying security verification goals*: Identifying the specific goals required to thoroughly verify the security aspects of the protocol.
6. *Translating messages to the AnB language*: Translating the identified messages into the AnB language, in doing so, simplifying and simulating various components within each message.

Each of these steps will be explained in more detail in the following sections.

3.2.1 Understanding SSO and OAuth

Section 2.1 of this thesis provides an explanation of the functioning of SSO systems. It delves into the mechanisms and principles behind SSO systems, offering an understanding of how they usually operate.

Section 2.2 focuses on the workings of OAuth 2.0. It outlines the core concepts and protocols involved in OAuth 2.0, exploring its functionality. In Section 2.2.4, particular attention is given to the typical modifications required to adapt OAuth 2.0 for use in authentication systems.

3.2.2 Adapting OAuth to SSO

The adapted data flow of OAuth within an SSO system is depicted in Figure 3.1. Notably, the authorization server and resource server are consolidated into a unified IDP, while the client is substituted with a RP. The identified critical messages within this flow are illustrated.

3.2.3 Identifying Essential Protocol Messages

In Figure 3.1, one message is intentionally greyed out, and omitted from the modelling: the authentication of the user by the IDP, *message 4*. This is because the authentication techniques may vary across different implementations, and the authentication step itself is not essential to the security of the protocol. A flawed authentication process would signify an untrustworthy IDP, which undermines one of the fundamental premises of the entire protocol: the IDP as a trusted participant in the protocol. The authentication step can be simulated within the authorization request initiated by the user to the IDP, capturing its purpose without explicitly depicting it as a separate message. The other messages shown in Figure 3.1 were identified as crucial to the modelling of the protocol.

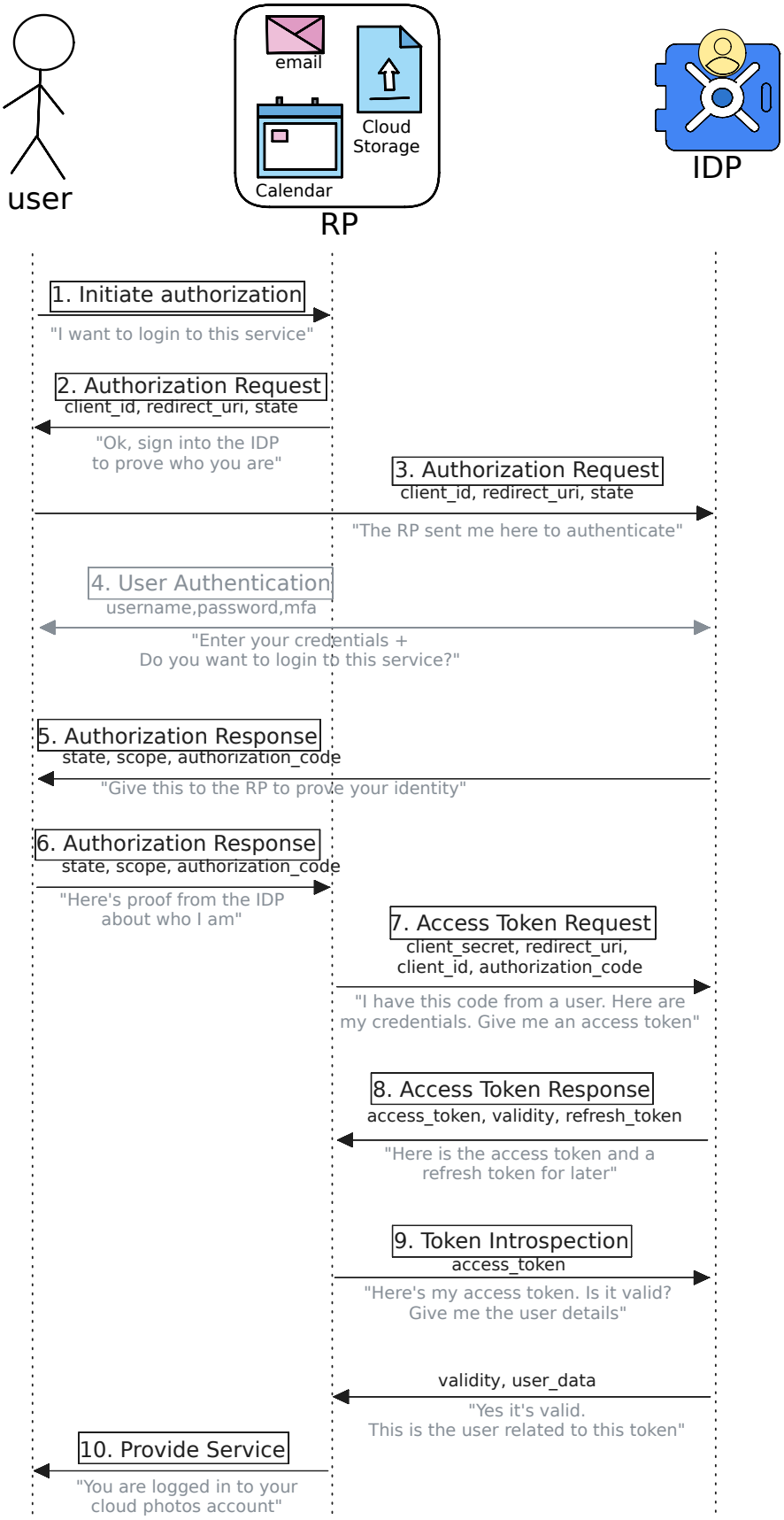

Figure 3.1: The OAuth in SSO flow being modelled

3.2.4 Identifying Protocol Types and Knowledge

This stage of the modelling process involves establishing the essential background information in the protocol. The subsequent sections will address various types of background information covered in detail.

Agent

The agents in the protocol are the active participants who engage in the protocol. They can be classified as either honest or dishonest, depending on their behavior within the protocol. In this model, there are three distinct participants: the User, the RP, and the IDP. OFMC automatically incorporates the presence of attackers, eliminating the need for explicitly mentioning them. Agents with names beginning with lowercase characters are considered trusted third parties. All other agents can be instantiated as required by OFMC, and the placeholder `i`, denoting the intruder/attacker, can be used with any of these agents. However, the attacker cannot assume the role of a trusted third party.

Number

This encompasses the different values used within the protocol, such as nonces, shared secrets, and other values passed among the participants. Numbers can be either *constants* or *variables*. *Variables* are denoted with names beginning with an *uppercase character*, serving as placeholders and set by OFMC during protocol execution. *Constant* numbers, on the other hand, begin with a *lowercase character* and retain a fixed value.

Function

This part of the model encompasses the different functions required throughout the protocol. Functions can be used to represent long-term secrets, such as passwords or shared secrets. For example, `pw(A)` can denote the password utilized by agent `A` within the protocol.

The protocol model incorporates the following functions:

1. `h`: A hash function to provide the Identity Provider (IDP) with the user's password or client secret.
2. `pw`: The password function that end user needs to authenticate itself at the IDP.
3. `cpw`: The client password function represents the client secret in the context of OAuth. It is worth noting that the normal password function can be used interchangeably without impacting the overall outcome.
4. `info`: This function serves as an assertion from the IDP, confirming the authenticity of the user's identity.

It is important to note that the descriptions provided for these functions do not necessarily reflect their actual operations. They are only represented as distinct functions within the model. Behind the scenes, OFMC treats them as opaque values, generating results based on the parameters passed to them, without any understanding of their underlying functionality.

Knowledge

The knowledge part of the model specifies the type of knowledge possessed by each agent. This includes constant numbers and functions. Variable numbers do not require explicit inclusion in this part. Agents may also possess awareness of other agents and trusted third parties within their knowledge scope.

This part can also be home to constraints regarding the roles played by agents. For example, that the Server and the Client roles cannot be assigned to the same agent simultaneously throughout the protocol execution.

The finished types and knowledge part looks like this:

```
Types: Agent U, RP, idp;
       Number STATE;
       Function h, pw, cpw, info;

Knowledge:   U : U, idp, RP, pw(U);
           idp : U, idp, RP, info(U), h(pw(U)), h(cpw(RP));
            RP : idp, RP, cpw(RP);

        where RP!=U, RP!=idp, U!=idp
```

3.2.5 Identifying Security Verification Goals

The primary objective of any SSO system is to ensure that only the authorized user can access their data stored within a RP. It is crucial to prevent any unauthorized individuals, posing as the user, from exploiting the SSO system to gain access to sensitive information. Several key factors contribute to achieving this goal, such as:

1. Weak authentication method: If a user utilizes an extremely vulnerable authentication method, such as a password that is identical to their username, it creates an opportunity for exploitation.

2. Compromised IDP: The IDP could be compromised through collusion with an attacker or by being infected with malware, compromising the security of the SSO system.

3. Stolen credentials: If the attacker gains access to the user's password, either through intentional sharing by the user or by using a keylogger installed on the user computer, they can impersonate the user and access their account.

It is important to note that these types of attacks are outside the scope of the formal analysis conducted in this study. The analysis assumes a Dolev-Yao attacker, who lacks control over the participants' computers in the network. While the attacker can manipulate network packets and masquerade as other computers, they are unable to exert direct control over another computer. The analysis focuses on threats within this framework, considering the attacker's capabilities and constraints.

In this particular context, the security goals of the protocol revolve around safeguarding against the various means through which an attacker can gain unauthorized access to user data from a RP. Specifically, these goals aim to prevent scenarios where the attacker successfully convinces the RP of their legitimacy as the IDP or manipulates the RP's perception regarding the origin of critical elements such as the authorization code or access token. This can also include deceiving the RP into believing that the token introspection and assertions regarding user authentication were provided by the legitimate IDP. Any of these situations could result in the RP mistakenly placing trust in the attacker rather than relying on the genuine IDP, thereby compromising the security of the system.

Furthermore, it is essential for the RP to be capable of verifying that the user assertion received from the IDP corresponds to the same user who initiated the authentication process. One potential attack scenario arises when an attacker, Eve, observes that a user, Alice, intends to log in to her social media account (the RP) and simultaneously proceeds to log in to her own account. With control over the network, Eve manipulates this situation by convincing the social media site (RP) that she is Alice. This is achieved by substituting

Alice's authorization code, access token, or token introspection with her own, thus granting Eve access to Alice's social media account. Conversely, a similar attack can occur where Eve deceives the RP into making Alice log in to Eve's account instead of her own.

Given the capabilities of modelling offered by OFMC, the following security goals were established:

- *Authenticity of messages:* The RP can verify the origin of crucial messages to ensure they indeed originate from the correct IDP. This includes verifying the authenticity of the authorization code, the access token, and the User introspection
- *Protection of user's secret data:* The RP ensures that the user's sensitive data remains confidential and is not shared with any unauthorized entities. This emulates a secure system where only the intended user can login to the RP, preventing unauthorized access by other parties.
- *Session integrity:* To maintain session integrity, it is ensured that one session cannot be confused with another. In real-world scenarios, this aspect is handled by the Record Protocol component of TLS, which ensures reliable, secure communication and prevents session confusion [14].
- *Confidentiality of messages:* Certain messages are shared confidentially between the involved parties. The OAuth specification only urges that the access token be kept secret, the secrecy of the other messages was added as an extra factor to ensure added security. This includes the authorization code being known only to the RP, user, and IDP, and the access token and user introspection being known only to the RP and the IDP.

By incorporating these security goals into the modelling process, OFMC enables a comprehensive analysis of the protocol's ability to achieve the desired security objectives.

This is what the security requirements look like in the AnB language:

```
RP authenticates idp on U, info(U), Code, Token
NU2 secret between U, RP
Token, info(U) secret between RP, idp
Code secret between U, RP, idp
```

3.2.6 Translating messages to the AnB language

Once all the preceding steps have been completed, the translation of messages into the `AnB` language becomes a straightforward task. It is necessary to have a comprehensive understanding of the various notations used by `AnB` to represent different actions.

`[A] *->* B`: This notation signifies the establishment of an encrypted communication channel between A and B. The presence of `[ ]` around A indicates that A remains unauthenticated within the channel, while B has provided its certificate and successfully authenticated itself.

This is what the messages look like when translated to the AnB language:

```
# the user initiates a login with the RP
[U] *->* RP  : U, RP, idp
# the RP gives the user a the session state
RP  *->* [U] : STATE, RP

# the user is redirected to the IDP and authenticates
```

```
    [U] *->* idp : U, RP, idp, STATE, pw(U)
    # the IDP gives the user the auth code
    idp *->* [U] : STATE, Code

    # the user gives the RP the auth code from the IDP
    [U] *->* RP  : STATE, Code

    # RP exchanges the code for an access token by authenticating itself
    [RP]*->* idp : cpw(RP), Code
    idp *->* [RP]: Token

    # RP exchanges the access token code and gets the user information
    [RP]*->* idp : Token
    idp *->* [RP]: U, info(U), RP, idp
```

3.3 Formal verification with OFMC

The completed `.AnB` file can be provided to OFMC to check. The `numSess` parameter in OFMC specifies the desired number of concurrent sessions that OFMC will initiate. The `numSess` parameter was set to `2` for all checks. This fixed number of sessions guarantees the termination of the model checking process, but is robust enough to check the protocol.

Several iterations of the model were analysed utilizing OFMC, each with different variations. The most significant versions are outlined below. The base model is the one described in Section 3.2.

3.3.1 The base model with nonces for each TLS section

This amendment was prompted by a limitation in OFMC channels where the TLS Record Protocol is not fully reproduced. Consequently, a party could potentially mix up messages from various TLS sessions in which it was involved. By introducing nonces, this issue was circumvented.

The modifications required adding distinct nonce variables within the `Number` section and incorporating these nonces within the `Actions` section. The updated `Actions` section is presented below. Notice that a separate nonce is assigned for each TLS session that would be established between the parties in a real-world scenario.

```
    [RP]*->* idp : cpw(RP), Code, NRP2
    idp *->* [RP]: Token, NRP2

    [RP]*->* idp : Token, NRP3
    idp *->* [RP]: U, info(U), RP, idp, NRP3
```

3.3.2 The base model with nonces, coupled with SecretData

In the final step of the protocol, the `SecretData` variable was introduced to emulate a genuine *logging in* scenario where the RP transmits the user's personal data. Various approaches were explored, some of which revealed vulnerabilities within the system.

`SecretData` was appended to the final steps of the model, following the token exchange. OFMC has a limitation that prevents the resumption of an ongoing TLS session. This limitation hinders the direct utilization of SecretData as a response to the message where the user provides the authorization code. As a result, alternative techniques were explored to overcome this limitation:

1. Initiating a new connection with the user unauthenticated: This approach involves establishing a connection with the user while the RP authenticates itself.
2. Initiating a new connection with the user unauthenticated, and using a nonce for verification: In this method, the RP sends a nonce to the user and expects the user to respond with the nonce while appending the previous nonce utilized during the transmission of the authorization code code.
3. Initiating a new connection with both parties being authenticated: This approach involves initiating a fresh connection where both the RP and the user authenticate themselves with their public keys.

An analysis of the results obtained from this modelling and verification process using OFMC is presented in Chapter 4.

3.4 Modelling OAuth 2.0 in PSPSP

The process of modelling OAuth for PSPSP follows a similar approach to modelling OAuth for OFMC, with some notable differences. The first three modelling steps outlined in Section 3.2 are identical to the initial three steps of the PSPSP modelling process.

The main distinction between the two tools lies in the translation of modelling ideas into the respective languages. In the case of PSPSP, the language being used is trac, which differs significantly from the AnB language used in OFMC. These variations necessitate a different approach to address the following steps in the modelling process:

4. Identifying protocol types and knowledge: Identifying the enumeration, sets, functions, and analysis necessary for simulating the protocol.
5. Translating messages to the trac language: Converting the identified transactions into the trac language, specifying the logic of each transaction in a simplified manner.
6. Identifying security verification goals: Identifying the specific scenarios that lead to attacks in the protocol.

The specific steps that differed between the two modelling processes are discussed in greater detail in the following subsections.

3.4.1 Identifying protocol types and knowledge

The setup of the model in PSPSP introduces different parts of the model compared to OFMC. Instead of the `types` and `knowledge` parts, PSPSP includes `Enumerations`, `Sets`, `Functions`, and `Analysis`. An explanation of the contents and purpose of each of these parts is provided in Section 2.7.3.

The `Enumeration` part, displayed below, outlines the different types of participants in the model. There are three primary types: `hon` for honest participants, `dis` for dishonest participants, and `idp` representing the IDP. Additionally, there is an `agent` type that encompasses both honest and dishonest participants, while the IDP remains a distinct entity. The `user` and `rp` types serve as placeholders, allowing for easy modification of their meaning throughout the model. Similar to variables in a program, their values can be changed globally by adjusting the values specified here to `hon`, `dis`, or `agent`.

Adding multiple types of participants significantly impacts the verification time required by PSPSP. While various versions of these participant types were used throughout different modelling iterations, the ones shown here were the most frequently used.

```
1    Enumerations:
```

```
        hon = {a}           # honest users
        dis = {i}           # dishonest users
        idp = {p}           # the independent IDP
        agent = hon ++ dis  # honest and dishonest users combined
        user = agent        # refers to agents
        rp = agent          # honest and dishonest users
```

Below are the sets used in the model, along with a brief explanation of their purpose. These sets are straightforward to determine and implement, and easily comprehensible.

```
    Sets:
        secretData/2        # secret data between the RP and user
        rp_state/2          # the RP state that corresponds to a user
        auth_codes/3        # authorization codes corresponding to a
                            #   user, RP, and IDP
        access_tokens/3     # access tokens corresponding to a user,
                            #   RP, and IDP
```

The functions included in the model along with their descriptions are shown below. As mentioned in the previous 2.7.3, the public functions are accessible for the attacker to invoke, while the private functions cannot be invoked by the attacker.

```
Functions:
Public
    scrypt/2        # Symmetric encryption. Key and value
    tuple/2         # A generic set of 2 items
    cid/1           # Derive client id of an RP

    initiateSSO/2   # Format for the user initiating the sso
                    #   authentication
    authReq/4       # Format for the authentication request.
                    #   User, client_id of RP, state, user password
    authResp/3      # Format for the authentication response.
                    #   User, code, state
    tokenReq/3      # Format for the access token request.
                    #   Code, client_id of RP, RP password
    tok/1           # Format for the access token response. Token
    cred/1          # Format for the user credential/assertion
                    #   from IDP
    sec/1           # Format to send the secret data of the user

    pk/1            # used for TLS
    serverK/1       # Derive the server TLS key of a server
    clientK/1       # Derive the client TLS key of a client

Private
    inv/1           # Derive the inverse public key
    pw/1            # Password of user or RP
    session/4       # Used to store the relation between the user,
                    #   RP, IDP, and auth code
```

The concluding phase of this step is to define the analysis corresponding to each public function.

```
    Analysis:
```

```
    # the attacker can read these functions with no extra information
        tuple(X,Y) -> X,Y
        initiateSSO(X,Y) -> X,Y
        authReq(W,X,Y,Z) -> W,X,Y,Z
        authResp(X,Y,Z) -> X,Y,Z
        tokenReq(X,Y,Z) -> X,Y,Z
        tok(X) -> X
        cred(X) -> X
        sec(X) -> X
    # the attacker needs to know "X" in order to find "Y" in this function
        scrypt(X,Y) ? X -> Y
```

3.4.2 Translating messages to the trac language

The process of translating messages into the trac language presents certain complexities compared to the corresponding step involving AnB. Below is an excerpt showcasing a sample of the exchanged messages. Notably, lines beginning with $*$ denote their association with the TLS protocol, which has been composed into this verification process. Section 3.5 delves into the modelling of the TLS protocol, elucidating its composition in greater detail. However, for this stage, it is sufficient to know that the $*$ symbol signifies the sharing of a specific component within that particular line with the TLS protocol. In this given example, the shared component is the session key, which is generated as an outcome of the TLS protocol.

```
# the user initiates the sso process
userRquest(A:user,B:rp,K:value)
*   K in clientSessionKeys(A,B)
    send scrypt(clientK(K),initiateSSO(A,B)).

# the RP recieves the user request and
#   responds with its state and client id
rpRedirectUser(A:user,B:rp,K:value)
    receive scrypt(clientK(K),initiateSSO(A,B))
*   K in serverSessionKeysUnauth(B)
    new RP_STATE
    insert RP_STATE rp_state(A,B)
    send scrypt(serverK(K),tuple(RP_STATE,cid(B))).
```

These are the initial stages of the OAuth protocol in PSPSP. It is worth noting that each transaction in `trac` concludes with a period, denoting its completion. Within the transaction, multiple steps can be added from its definition until its culmination. Certain transactions can encompass greater complexity compared to others.

The transaction definition specifies the permissible types of participants in each transaction. In this scenario, the participant types of `user` and `rp` dynamically change, yet predominantly assume the roles of `agent` i.e., honest or dishonest entities. Occasionally, different variations of the same transaction may exist, designed separately for honest and dishonest participants.

The distinction between honest and dishonest participants is determined by the author. This distinction is established in the early stages of the `transactions` section. The approach adopted in this model involves intentionally disclosing additional data from the dishonest participant to the attacker. The following code depicts this aspect of the model.

```
initialKnowledge(A:hon)
```

```
    send A.

initialKnowledge2(B:dis)
*   send inv(pk(B))
    send cid(B)
    send pw(B).
```

Equipped with the private key, client_id, and password corresponding to all dishonest participants, the attacker possesses the capability to decrypt all transmitted messages and assume the identity of each dishonest user. Furthermore, within the TLS component of the model, the dishonest participants reveal their PMS to the attacker, enabling the attacker to obtain the TLS session key used for encrypting all exchanged messages.

3.4.3 Identifying security verification goals

The key security goals addressed in the modelling section using OFMC remain unchanged. The aim is to provide utmost protection for the authorization code, access token, the user and client passwords, and the secret data of the user. The impact of our security goals aligns precisely with those defined in the `goals` section of OFMC.

However, there are notable differences in the definition of security goals between PSPSP and OFMC. In PSPSP, we define various transactions that the attacker can initiate to fulfill the attack condition. These transactions include the keyword `attack` during their execution. If the attacker successfully triggers these transactions, it compromises our security goal. Thus, instead of specifying the security goals as in OFMC, we focus on defining the attack conditions in PSPSP.

The security goals employed across various iterations of the model followed the same vein as this example.

```
# ensure that the attacker doesn't have the auth code
codeSecrecy(A:hon, B:hon, C:idp, CODE:value)
    receive CODE
    CODE in auth_codes(A,B,C)
    attack.

# ensure that the attacker doesn't have the user password or client secret
passwordSecrecy(A:hon)
    receive pw(A)
    attack.

# ensure that the attacker doesn't have the access token
tokenSecrecy(A:hon, B:hon, C:idp, TOKEN:value)
    receive TOKEN
    TOKEN in access_tokens(A,C)
    attack.

# ensure that the attacker doesn't have the secret data of the user
resourceSecrecy(A:hon, B:hon, SECRET: value)
    receive SECRET
    SECRET in secretData(A,B)
    attack.
```

3.5 Modelling and composing TLS with OAuth

As stated previously, it is assumed within the scope of this study that TLS is employed throughout every stage of the OAuth protocol. Therefore, it is imperative to incorporate TLS in the model used for the formal verification of the OAuth protocol. Although it is technically feasible to compose multiple protocols within OFMC, the primary objective of this study was to conduct a comprehensive analysis utilizing the PSPSP tool. Therefore, the inclusion of TLS composed with OAuth was exclusively pursued using PSPSP.

3.5.1 Modelling TLS

This project utilized a TLS model based on the example model provided in the additional materials to the journal publication by Hess, Mödersheim, and Brucker [34]. However, the stages of modelling were carefully adhered to in order to meet the specific requirements of TLS for OAuth within the given context. Some modifications were made to accommodate these requirements.

The messages essential to the protocol were considered identical to the messages outlined in Section 2.3. In other words, all the messages were essential, albeit in a simplified manner, and each set of messages exchanged between the client and server were consolidated into a single PSPSP transaction.

Presented below is an excerpt of transactions from the PSPSP TLS model, showcasing some concepts previously discussed. Specifically, note the inclusion of both honest and dishonest versions of the client key exchange transaction. The dishonest version includes the transmission of the PMS, allowing the attacker to decrypt all TLS messages from dishonest users.

The $*$ label reappears in this context. As previously mentioned, this label serves to indicate all transaction steps associated with the other protocol incorporated in the model. Specifically, it signifies an operation performed on a shared set or value within the composition of the two protocols. This includes the actions of declassifying data to make it visible to and usable by the attacker. For example, in this case, the PMS is employed in the OAuth protocol, while the nonces are declassified to enhance transparency in the TLS protocol.

```
1  serverHello(B:TLShon,NA:value)
2      receive NA
3      new NB
4      insert NB tls_step2(B)
5  *   send NB
6      send sign(inv(pk(s)),tuple(B,pk(B))).
7
8  client_key_ex(A:TLShon,B:TLShon,NA:value,NB:value)
9      receive NB
10     NA in tls_step1(A,B)
11     delete NA tls_step1(A,B)
12     new PMS
13 *   insert PMS tls_step3(A,B)
14 *   send ox(PMS)
15     send crypt(pk(B),PMS)
16     send scrypt(clientK(PMS),hash(PMS,NA,NB)).
17
18 client_key_exD(A:TLShon,B:dis,NA:value,NB:value)
19     receive NB
20     NA in tls_step1(A,B)
21     delete NA tls_step1(A,B)
22     new PMS
23 *   insert PMS tls_step3(A,B)
```

```
24 *    send ox(PMS), PMS
25      send crypt(pk(B),PMS)
26      send scrypt(clientK(PMS),hash(PMS,NA,NB)).
```

Although the sets and functions utilized in this TLS model are relatively straightforward, which is why they are not elaborated upon here, the `ox` function appears to serve no apparent purpose. However, its presence in PSPSP is necessary due to the behavior that if a value is not transmitted over the network, it is considered lost. The `ox` function remains opaque to the attacker (it is not included in the `analysis` section), allowing it to be used for transmitting any value, whether confidential or not, that should not be lost. The PMS value is transmitted using the `ox` function to ensure its inclusion in the $*$ step, as required for the composition with the OAuth protocol.

3.5.2 Modifications to TLS

In the initial stages of verification, the TLS example provided with the PSPSP installation was utilized with minimal modifications. However, it was subsequently observed that an oversight was made in this adaptation.

The issue was that only only `honest` and `dishonest` participants were permitted to engage in the TLS exchange. With the IDP modelled as a trusted third party, this effectively excluded the IDP from participating in the protocol. As a result, the analysis conducted before this was fixed became irrelevant since the IDP was unable to participate in the OAuth verification process, given that establishing a TLS connection is a prerequisite for involvement in the OAuth protocol.

This issue was not a result of any inherent errors within the TLS model itself, but rather due to the TLS model being initially designed for a different scenario than its use in composition with the OAuth protocol. This oversight underscores the possibility of introducing bugs when composing multiple protocols, highlighting the crucial importance of conducting meticulous analyses for each individual protocol, both independently and in conjunction with other protocols, to ensure seamless compatibility in composition.

3.5.3 Composing TLS and OAuth in PSPSP

Once the two protocol models have been developed independently with all the sets and functions that are required, composing them together in PSPSP is straightforward.

To begin with, the enumerations, sets, functions, and analysis sections should be combined. It is essential to ensure that the enumerations of the two protocols exhibit some degree commonality; otherwise, the process of composing the protocol becomes meaningless as the participants from each protocol would never interact. Additionally, it is advisable to maximize the reuse of sets and functions to reduce the verification time required by Isabelle in confirming the validity of the protocol. However, there will still be certain sets and functions that remain independent to each protocol, while others are shared between both protocols.

The transactions section will now be divided into two separate sections: *Transactions of tls* and *Transactions of oauth*. The name for each of the protocols is arbitrary but constant throughout the protocol. Moreover, the attack definitions for each protocol will be included within their respective transactions, independent of each other.

After completing these adjustments within the `trac` language, the remaining PSPSP commands in Isabelle must also be modified to accommodate the composition of both pro-

tocols. These steps can be flexibly adapted to accommodate any number of protocols being composed.

3.6 Formal verification with PSPSP

This section focuses on the execution process of formal verification using PSPSP, including the necessary commands within the Isabelle file required for any model. It also provides a more detailed explanation of iterations on the model with OAuth composed with TLS.

3.6.1 Setting up Isabelle

Once the protocol has been modeled in the trac language and saved in a separate file, the next step is to prepare the Isabelle proof around that protocol. This section will outline some of the commands offered by PSPSP that are used in the proof.

Firstly, the name of the Isabelle theory being proven is specified. Next, the `Automated_Stateful_Protocol_Verification.PSPSP` logic is imported to enable the use of PSPSP. Then, the `trac_import` command is employed to import the `.trac` file containing the model.

Moving on to the proof itself, the sub-message patterns (SMP)[34] are computed for each individual protocol using the symbolic SMP. Additionally, the ground sub-message patterns (GSMP) are computed for the composition of the two protocols, incorporating the $*$ messages by utilizing the `with_star_projs` command along with the protocol name.

Subsequently, the fixpoint of the protocol is calculated for each composed protocol. This fixpoint serves as an over-approximation of all the information accessible to the attacker at any given point. It is worth noting that this step is typically the most computationally heavy in the Isabelle verification process.

Following that, the `protocol_security_proof` command from PSPSP is utilized to verify multiple properties within each composed protocol. This includes checking for the absence of attack signals in the fixpoint, ensuring the entire protocol is covered by the fixpoint, verifying that the fixpoint has been fully analysed, and validating the well-formedness of both the protocol and the fixpoint. PSPSP automates these steps.

At this stage, the protocols have been computed and proven independently. The next step involves computing the shared secrets between the composed protocols using PSPSP. This computation specifically focuses on the messages within the GSMP that are secret between both protocols, such as the `ox` messages with $*$ in them. Subsequently, a `protocol_composition_proof` and a lemma are employed to establish the security of the composed protocols.

Throughout the development process, several versions of the trac file were employed, each incorporating different properties. The following sections describe the most notable versions used.

3.6.2 The base model

This is the unmodified OAuth model, as described in Section 3.4. The initial versions of this model had shortcomings, as outlined in Section 3.5.2. Although these early versions did not reveal any possible attacks, it is important to note that the results are unreliable due to the aforementioned flaws. After adapting the TLS model to align with the requirements of OAuth, the analysis of this model was rerun and verified by Isabelle to be free of flaws.

3.6.3 Terminating TLS sessions

One limitation of the base model is the reuse of TLS session keys between any two participants. This scenario does not accurately reflect real-world practices, and it is important to consider the termination of TLS sessions as they occur in actual environments.

Two approaches can be utilized to ensure the termination of TLS sessions. The first approach involves directly deleting the session key once the session is completed. This is a straightforward solution that would necessitate the participants to establish a new session key if they intend to communicate again. The second approach involves using another set to store the used keys of each participant. Prior to using a key and communicating, participants would check if it has been previously used, proceeding only if it hasn't. Upon concluding the TLS session, the key is added to the `used_keys` set specific to the participant. For the purpose of optimizing performance, the second approach was adopted.

Below is an example illustrating the termination of TLS in the trac language. The scenario shows the RP receiving the access token from the IDP and subsequently proceeding to request token introspection from the IDP. Both these steps are performed using different keys, `K1` and `K2` respectively.

```
# rp gets access token from idp
# rp asks idp for user details
rpTokenExchange(A:user, B:rp, C:idp, K1:value, K2:value, ACC_TOKEN:value)
    receive scrypt(serverK(K1), tok(ACC_TOKEN))
*   K1 in clientSessionKeys(B,C)
    K1 notin used_keys(B)
*   K2 in clientSessionKeys(B,C)
    K2 notin used_keys(B)
#terminate tls rp-idp
    insert K1 used_keys(B)
    send scrypt(clientK(K2), tok(ACC_TOKEN)).
```

3.6.4 Modelling an attack

It is important to ensure that the model can detect attacks in cases where they exist. So to test that, attacks were introduced into the model. Various methods were used for this, ranging from disclosing the TLS session key of honest users, to revealing the user's secret data in the final step. However, in an effort to simulate real-world issues known in OAuth 2.0, the HTTP 307 redirect attack as described in Fett et al. [15] was specifically replicated.

This attack arises due to flawed implementations of the IDP, which employ the HTTP 307 redirect code [35] to redirect the user to the RP's designated redirect URI. The vulnerability arises from the forwarding of request parameters from the initial request to the new HTTP server, whereby the HTTP client sends new data to the new server in addition to the previous request sent to the old server. In the case of OAuth, this flaw becomes critical when the IDP redirects the user back to the RP with the authorization code, while also including parameters from the user's previous request to the IDP. Frequently, the user's previous request to the IDP involves entering their username and password.

In such a scenario, the RP, alongside the authorization code, gains access to the user's username and password for the IDP. If the user is authenticating themselves at a malicious RP, it could harvest these credentials and exploit them to impersonate the user at the IDP.

This attack was induced into the model by including the user's password in the authorization response function sent from the IDP. The user directly forwards this function to the

RP without any modifications.

```
# idp gets user credentials
# idp gives user auth code to give to rp along with user pw
idpAuthResponse(A:user,B:rp,C:idp,K:value,RP_STATE:value)
    receive scrypt(clientK(K),authReq(A,cid(B),RP_STATE,pw(A)))
*   K in serverSessionKeysUnauth(C)
    new AUTH_CODE
    insert AUTH_CODE auth_codes(A,B,C)
    send session(A,B,C,AUTH_CODE)
# send the password of A in the auth response. To simulate a 307 redir
    send scrypt(serverK(K),authRespAttack(B,AUTH_CODE,RP_STATE,pw(A))).
```

An analysis of the results obtained from this modelling and verification process using PSPSP is presented in Chapter 4.

4 Analysis

This chapter presents and analyses the outcome of the formal verification process.

4.1 Detailed Analysis from OFMC

The overall observation reveals certain constraints in OFMC's ability to effectively verify protocols of this nature. The main limitation is in OFMC's ability to faithfully carry out the TLS handshake and setup the TLS connection between parties. The other limitation lies in the RP's inability to resume the TLS session and transmit the user's secret data. However, disregarding this specific scenario and assuming that, in practical a implementation, the RP can successfully deliver the secret data to the intended user, the verification process proves to be fruitful. A summary of the analysis conducted on the various attempted models is presented.

4.1.1 Iterations with no nonce

All the iterations of models that lacked the use of a nonce were found to be vulnerable to attacks in OFMC with the `--numSess` parameter set to 2. These vulnerabilities arose due to the limitation of OFMC's *channels* mechanism in adequately replicating all the properties of a TLS session.

The OFMC channel, represented by $*{-}{>}*$ in bullet notation, ensures that the sender can trust that the message will only be readable by the intended receiver, while the receiver can verify its origin. However, this channel does not maintain any session context, allowing messages from the same set of participants at different times to be interchanged.

The vulnerabilities in models without a nonce stemmed from Alice initiating two sessions with the same RP. The attacker, Eve, intercepted all the messages interchanged, transmitting messages from one session of Alice to the other session of the RP. Consequently, the RP authenticated Alice's Session 1 with the authorization code and access token intended for Alice's Session 2.

While this may not appear to be a significant attack, as Alice still needs to authenticate herself with the RP and no data is being stolen by Eve, it undermines the principles established in the verification process. Any form of bypassing the strong authentication between Alice and the RP constitutes an attack. Depending on the configuration of the IDP and the RP, this kind of compromise could potentially lead to more severe issues.

However, it is important to recognize that this type of attack would not be feasible in a real-world scenario where an actual TLS channel is employed. This is due to the inherent security measures provided by TLS sessions, where a unique session key is generated for each session. In the context of the same set of participants, such as Alice and Bob, every session they initiate will have a distinct session key. Consequently, if a message from one session is erroneously or intentionally sent to another session, the intended receiver will reject the message as it cannot be decrypted using the incorrect session key. The robust security guarantees offered by the TLS protocol prevent the attacks observed in the model without a nonce from occurring in real-world implementations. Hence, it is crucial to modify the model to eliminate the detection of such infeasible attacks, allowing for a more accurate assessment of the actual security risks present in the model.

4.1.2 Adding a nonce

To mitigate the impossible attack described in Section 4.1.1, one effective approach is to introduce a nonce into each message within a session. This nonce is generated randomly at the initiation of the session, and it must be included at the end of every message exchanged within the session. The receiving participant verifies that the nonce matches the expected session nonce, thereby ensuring the integrity and authenticity of the message within the correct session.

By introducing a nonce, we introduce a mechanism that emulates the functionality of a TLS session key to a certain extent. While it does not provide protection against an attacker who already possesses the long-term encryption key, it effectively safeguards against attacks involving the reuse of messages across different sessions.

By configuring OFMC to allow for two simultaneous sessions, the revised model incorporating nonces in each session successfully avoided any detected attacks. The inclusion of nonces served as a key modification, differentiating this model from the previous one.

Furthermore, the security requirements, in the ”Goals” part of the model, were strengthened to specify the condition that the nonce, `NU2`, must remain confidential between the user (U) and the RP. Additionally, it mandated that the RP authenticate the validity of `NU2`, verifying that it was indeed sent by the user. These stringent requirements were implemented to fortify the secrecy and authenticity of the authorization code transmitted within the same message.

With the incorporation of these modifications, OFMC detected no attacks, confirming the effectiveness of the nonce-based approach and even with the strengthened security requirements. Additional iterations and variations on this method consistently demonstrated the desired security properties, further solidifying the robustness of the protocol.

4.1.3 Adding SecretData

Up to this point, there has been no simulation of an actual user login in the protocol. The protocol concludes once the RP receives the assertion from the IDP, either directly after providing the authorization code or through access token introspection. While it is acceptable to end the protocol at this stage in theory, it is still valuable to assess whether this additional step can be successfully modeled using OFMC. It is also valuable to check if there are any vulnerabilities associated with the modelling of the user login step.

However, accurately modelling real-world behavior in this case is not possible due to the limitation in OFMC channels, which prevents session resumption. Different approaches were explored to address this issue, each yielding varying results.

One approach that proved to be successful and straightforward involved using the secure OFMC channel with mutual authentication between the client and server. However, it should be acknowledged that this approach does not align with typical real-world scenarios. In practice, clients do not usually authenticate themselves with certificates during the TLS handshake, and servers primarily respond to client-initiated messages rather than initiating their own.

Nevertheless, this approach can be seen as representative of the ultimate goal of establishing a secure channel after the IDP provides the user assertion. The user assertion indirectly authenticates the user to the RP, serving the purpose of user authentication in the SSO system.

While the formal verification using OFMC with ”numSess” set to 2 did not uncover any attacks, it can be argued that employing this two-sided client-server authentication does

not provide insights into the security of the preceding steps. In fact, by utilizing this mutual authentication, the RP could potentially skip all the previous steps altogether and directly send the SecretData to the user without compromising security.

A more reasonable approach is to introduce a nonce to verify that the user who receives the SecretData is the same as the one who sent the RP the authorization code. Another nonce is added to ensure that the RP sending the SecretData is indeed the same one that received the authorization code from the user. Although this does not fully replicate real-world scenarios, it guarantees that the user sending the authorization code is the same as the user receiving the SecretData, providing some level of security while still leaving room for potential attacker exploitation without introducing new vulnerabilities to the protocol.

The primary objective of this type of modelling is to accurately represent real-world systems within the limitations of the selected tool. While this may be straightforward in some cases, it often necessitates various approaches and adjustments to achieve the desired simulation. Furthermore, it is relatively easy to identify false positives, where the model erroneously introduces an infeasible attack. Whereas, detecting false negatives, where the model obscures an existing attack in the system, is considerably more challenging. Additionally, there may be cases where the specific tool is unable to faithfully replicate the system, regardless of modelling efforts.

4.2 Detailed Analysis from PSPSP

In the initial stages of the verification process, the TLS model used in the PSPSP model exhibited some shortcomings. However, through subsequent modifications to the TLS model these were overcome. The next shortcoming is in the inability of the model to faithfully carry out the handing over of the secret data from the RP to the user, by resuming a TLS connection based on a session cookie. However, with some workarounds implemented and rigorous testing with different iterations of the OAuth model, the verification process yields fruitful results. This section provides a summary of the analysis conducted on the different iterations of the OAuth and TLS model composed.

4.2.1 Base model

The term ”base model” in this context is defined as the model without the terminating TLS sessions, induced attacks, or modifications made to the TLS model to align it with the requirements of this model.

One variation often implemented in this model involved changing the participant labeled as ”idp” to have the value ”hon”. Thereby making the IDP the same as any honest user in the system. This modification aimed to expedite the verification process in Isabelle by reducing the time required for assessing the impact of smaller changes. The rationale behind this adjustment was that if allowing the IDP to act as a normal participant did not lead to any attacks within the system, then assuming the IDP as a trusted third party would only enhance the overall security. Consequently, if any attacks were detected following this alteration –which was not the case–, the change could be reverted and retested accordingly.

In the scenario where the IDP is assumed to be a regular honest user, the formal verification process revealed no potential attack vectors, and these results can be deemed reliable. Conversely, when considering the IDP as an independent third party, the outcomes cannot be relied upon due to the TLS model’s inadequacy in accommodating the IDP as an independent entity in OAuth, as explicated in Section 3.5.2. The appropriate changes were made to the TLS protocol and, similar to the base model, no potential attack vectors were triggered in this adjusted model by PSPSP.

4.2.2 Terminating TLS

In real-world scenarios, when the user is redirected by the RP to the IDP for authentication, the TLS connection between the user's web browser and the RP is terminated. Consequently, when the user is redirected back from the IDP to the RP, a new TLS connection is established between the user and RP. Similarly, the RP and the IDP establish two distinct connections: one for the authorization code - access token exchange and another for the access token - user assertion exchange.

However, this behavior is not accounted for in the base model. Here, the TLS connection between the user and RP remains the same from the initialization of the authentication until the user receives its secret data. The same is true for the RP performing the authorization code exchange and the token introspection with the IDP. Furthermore, the same connection can be reused in multiple runs of the protocol in PSPSP.

This shortcoming hinders the effectiveness of an attacker acting as a PITM attacker. If the attacker compromises the single session key being used, it can compromise the entire protocol. Conversely, if the attacker cannot compromise that specific session key, its capabilities are greatly limited.

In order to address this issue, a model was developed that incorporates the termination of TLS sessions. Figure 4.1 illustrates the stages within the OAuth SSO data flow where the TLS connection between two entities is reset. In this new model, each termination point triggers the initiation of a fresh TLS session, resulting in a more accurate representation of real-world scenarios.

Similar to the base model, this adjusted version of the model also did not uncover any potential attack vectors within the protocol.

4.2.3 Inducing an attack

While it is indeed encouraging to witness positive results from Isabelle, it is crucial to ensure that the model does not inadvertently suppress attacks through any means. To verify the integrity of this model, deliberate attempts were made to introduce vulnerabilities into the protocol, with the intention of checking if PSPSP could detect the attacks. Multiple methods were used to achieve this objective.

The most straightforward attack that could be executed involved the revelation of the user's secret data without encryption, thereby directly exposing it to the attacker. Another attack possibility is to expose the TLS session key of the user with the RP and the user with the IDP, allowing the attacker to observe the user's password transmitted to the IDP or the secret data from the RP. As expected, these actions triggered an attack in PSPSP.

The implementation of the HTTP 307 attack, as identified by Fett et al. and outlined in Section 3.6.4, was also executed. This attack triggered the password secrecy attack in PSPSP. However, depending on the iteration of model that was used, it was unable to trigger the resource secrecy attack, which tests for leaked data. This limitation arises from the inability of the model to properly simulate the final step of the user logging into the RP.

The base model ensures that the user to whom the RP is sending the secretData is definitely the user whose secret data it is. This means that unless the attacker possesses an honest user's TLS session key, it cannot masquerade as an honest user. In the real-world, a session cookie would be used in the user browser and the RP to remember which client the particular authorization code belongs to.

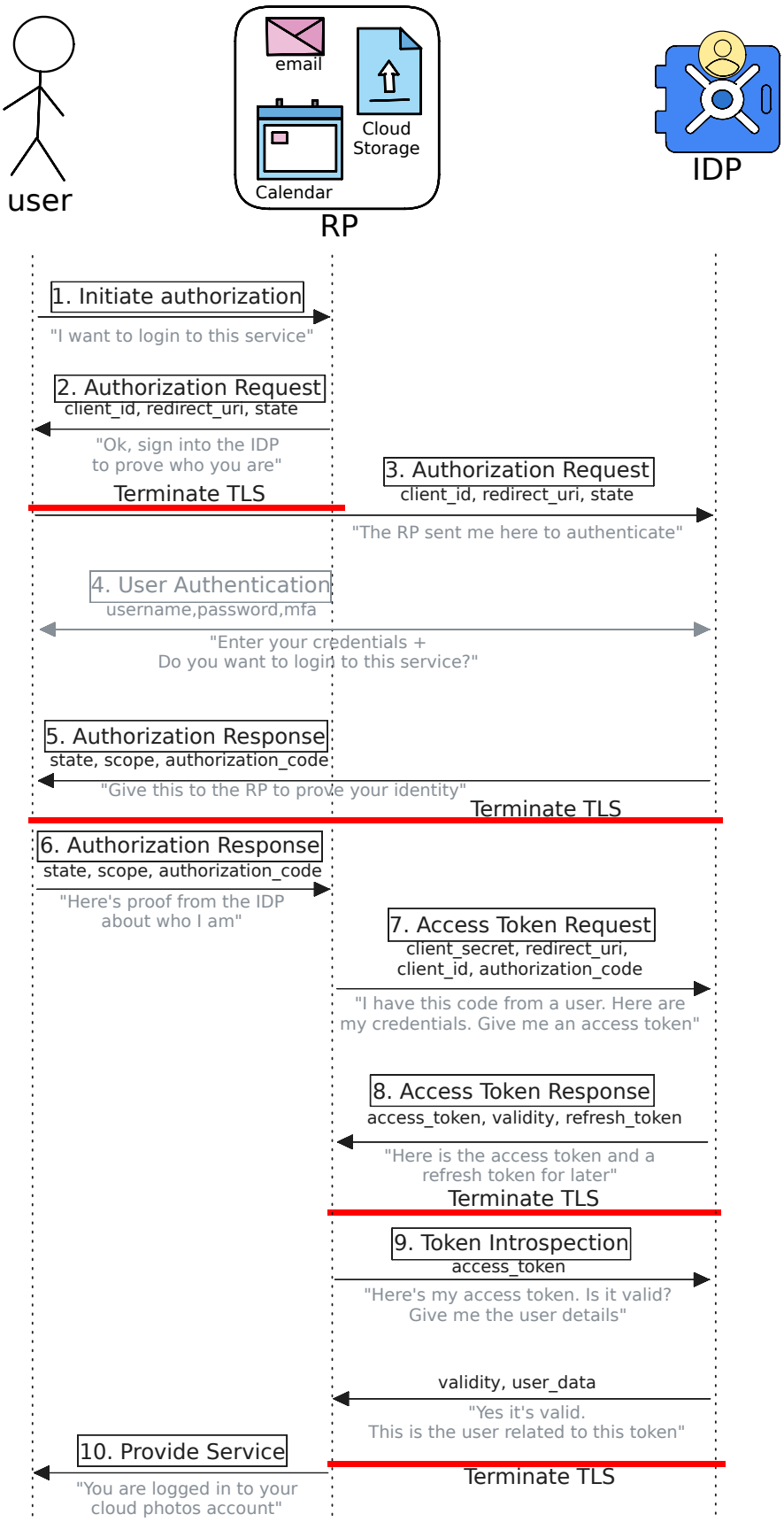

Figure 4.1: The areas where the TLS sessions are terminated in the OAuth data flow

There were attempts to simulate this in the model by sending out a separate session function that records these values, and then receiving this session function but it is not the same as the session cookie being stored.

Nevertheless, the attack scenarios within the model effectively capture all sensitive data in the OAuth protocol, including the user password, client secret, authorization code, and access token. Furthermore, the attack scenarios in the TLS protocol ensure the successful establishment of the TLS connection without any breaches of the session key.

4.3 Results

In summary, the adaptation of the OAuth protocol for SSO, employing token introspection, revealed no identifiable attacks.

4.3.1 Results from OFMC

The multiple iterations of the OFMC formal verification yielded no identifiable attacks. The model encountered some attacks that were subsequently identified as being protocol model inaccuracies. These attacks resulted from other shortcomings in the modelling process and the OFMC tool itself.

4.3.2 Results from PSPSP

During multiple iterations of the model, the PSPSP tool did not identify any attacks. These iterations included varying numbers of honest users, the use of the IDP as an independent third party, and the incorporation of varying security goals.

Furthermore, when deliberate attacks were introduced to the model to assess its effectiveness, the attacks were successfully detected. This further affirms the robustness of the system.

5 Discussion

This chapter provides an analysis of the study results within a broader context. It includes the interpretation of the results in relation to existing research, a discussion of the limitations of the study, and potential areas for future work.

5.1 Interpretation of results

The findings from this analysis align with prior research conducted in this domain, affirming the security of the OAuth protocol across a multitude of contexts. Assuming that the protocol is accurately implemented, avoiding the mistakes highlighted in studies such as those by Fett et al. [15] and Li and Mitchell [9], it can be concluded that the OAuth 2.0 protocol is a secure choice for SSO applications.

Comparable studies have been undertaken, utilizing formal verification tools to analyse the OAuth protocol and specific implementation details, as discussed in Section 2.8. However, the distinctive feature of this study lies in the use of the OFMC and the PSPSP tools, which have not been used in this specific circumstance before. The incorporation of these tools presents unique advantages and benefits of their own.

The incorporation of the OFMC tool in this investigation offers an advantage in terms of simplicity and ease of understanding, benefiting both the researcher and any individuals aiming to review or learn from this study. In contrast, the PSPSP tool presents significant technical advantages, including:

1. The capability to maintain a mutable long-term state, implying that long-term states can be modified over time. For instance, a previously used TLS session key can be invalidated and discarded.
2. The provision for the verifier to employ ”non-monotonic” reasoning, meaning that alterations in the state of the system may lead to changes in assertions about the system. As a result, a statement that was initially accurate may subsequently become false, and the converse is also possible.

5.2 Qualification of Results

While the formal verification results obtained from both tools demonstrate the absence of attacks, it is crucial to acknowledge certain qualifications regarding these findings. As discussed in Section 2.4, the outputs of formal verification can occasionally be unreliable due to various factors, including:

1. *Model errors*: Errors within the model can either mask existing attacks or introduce new ones. While it is relatively straightforward to identify infeasible attacks, uncovering hidden attacks can be more challenging.
2. *Tool limitations*: The tools used might not accurately capture all aspects of the real-world system or could have bugs or shortcomings that hide or introduce attacks.
3. *Inaccurate security properties*: The specified security properties in the model may not encompass every possible scenario that could lead to an attack in real-world circumstances, allowing certain attacks to go undetected. Conversely, the security properties may overly restrict normal scenarios and flag them as attacks.

Despite the diligent efforts made to ensure the integrity of the model and property specifications, as well as the reliability of the tools selected for the analysis, it is important to acknowledge that the results obtained in this study are still susceptible to these potential errors.

5.2.1 Shortcomings in the model

This study definitely posses limitations, particularly regarding the TLS issues present in OFMC as discussed in Section 4.1.1. Moreover, the inability of both OFMC and PSPSP to adequately simulate the final login step of the user into the RP, as described in Sections 4.1.3 and 4.2.3, hinders the model's performance as an accurate representation of the real-world.

5.3 Future Work

While this study serves as a promising foundation for utilizing the PSPSP tool in Isabelle to conduct an in-depth analysis of the OAuth protocol, there are still opportunities for enhancing the model. These improvements, although not essential to address the initial research goals set out in this thesis, augment the study by addressing additional research questions regarding the security of the OAuth protocol.

One avenue for future research involves incorporating multiple IDPs into the authentication process. This would introduce complexities concerning potential misuse of authorization codes or access tokens from one IDP to gain unauthorized access at another IDP. Furthermore, within this scope, one IDP could be designated as a dishonest entity, enabling the identification of potential vulnerabilities when a service provider trusts a compromised IDP. Both of these scenarios closely resemble real-world use cases of SSO systems.

Another area for exploration is the integration of two-factor authentication (2FA) or multi-factor authentication (MFA) capabilities into the system. One possible approach for implementing 2FA is by establishing a separate trusted channel, distinct from TLS, between the IDP and the user. In this scenario, the IDP would transmit a one-time challenge to the user, who would then enter this value into the primary TLS channel between the user and IDP after entering their password. This setup could be combined with intentional leakage of the user's password to evaluate whether the 2FA method effectively prevents the attacker from exploiting the compromised password. Jacomme and Kremer [36] extensively investigates and details the formal verification of a couple of other MFA mechanisms that could be incorporated into future SSO studies.

Additionally, an opportunity for further investigation lies in upgrading the TLS version used in this study from version 1.2 to the latest available version, namely version 1.3. Version 1.3 introduces added complexity to the handshake process. While the PSPSP tool currently lacks the ability to model this upgrade, it can be pursued using alternative tools or future iterations of PSPSP.

Although there are numerous other features within SSO systems that could be incorporated into this analysis, this thesis proposes one additional recommendation: the inclusion of a user onboarding and password reset mechanism. This would involve implementing a distinct endpoint at the IDP where users can register and set their passwords. In this case, the password would be represented as a mutable long-term value in PSPSP, diverging from its current function-based representation in PSPSP and OFMC. The IDP would also provide an endpoint for users to change their passwords, requiring the entry of their old password and potentially even incorporating MFA.

The advantage of using an automated tool like PSPSP, which supports composition, is that all these additions can be seamlessly composed into the existing model. This significantly reduces the effort and complexity involved compared to directly modifying the existing OAuth section of the model, thereby mitigating the likelihood of introducing errors into the system.

6 Conclusion

The motivation behind this study was to investigate potential vulnerabilities in the structure and usage of the OAuth 2.0 protocol, particularly when applied in the realm of Single Sign-On (SSO) systems. SSO systems are widely implemented in organizations for employees and on the wider internet for end-users, and OAuth is a popular choice to be used in these protocols. Although OAuth was not originally designed for authentication, it has been adapted for use in this context.

This study successfully addressed all the research questions established at the outset:

1. Understanding SSO systems and their requirements.
2. Understanding the OAuth 2.0 protocol and its application in SSO systems.
3. Undertake a formal examination of the OAuth protocol, as tailored for SSO systems, either to validate its correctness or to scrutinize and suggest remedies for any discovered attack vectors.

The context of SSO systems was thoroughly explored and delineated, while the OAuth protocol was subjected to meticulous analysis. One mechanism for adapting the OAuth protocol to suit the SSO use case, namely token introspection, was presented and explained.

A comprehensive analysis was performed using two distinct formal verification tools: OFMC and PSPSP. Neither of these analyses unveiled new attack vectors in the OAuth protocol, thus further affirming the protocol's security. This reinforces the continued adoption of OAuth, not only for its original resource sharing purpose but also as an authentication protocol within SSO systems. This thesis contributes to the existing body of formal analysis conducted in this field, that have arrived at similar conclusions regarding the absence of errors in the protocol itself. However, some of them identify attacks that arise due to intricacies in implementations and different capabilities of the modelling tools used.

By focusing on the protocol itself rather than specific implementations and their intricacies, this study offers broader applicability. Although numerous implementations exist, a flaw in the protocol would imply a flaw in every single implementation that adheres to the protocol.

While this study did not unveil any groundbreaking discoveries, it is essential to validate the protocol's security using new tools and techniques that enhance the analysis capabilities. Notably, the utilization of mutable long-term state introduces capabilities that are not commonly used in formal protocol verification. Furthermore, this study serves as a solid foundation for future research endeavors, allowing for the incorporation of additional features specific to SSO systems.

In conclusion, the rigorous formal verification undertaken in this thesis underscores the robustness of the OAuth 2.0 protocol, validating its versatility, and security. This finding strengthens the protocol's prominent standing as a widely adopted solution for authentication in Single Sign-On systems.

Bibliography

[1] Aleksandr Ometov et al. “Multi-factor authentication: A survey”. In: *Cryptography* 2.1 (2018).

[2] Anupam Das et al. “The tangled web of password reuse.” In: *NDSS*. Vol. 14. 2014.

[3] David Basin, Sebastian Mödersheim, and Luca Vigano. “OFMC: A symbolic model checker for security protocols”. In: *International Journal of Information Security* 4 (2005), pp. 181–208.

[4] Andreas V Hess et al. “Performing security proofs of stateful protocols”. In: *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. IEEE. 2021, pp. 1–16.

[5] Alessandro Armando et al. “Formal analysis of SAML 2.0 web browser single sign-on: breaking the SAML-based single sign-on for google apps”. In: *Proceedings of the 6th ACM workshop on Formal methods in security engineering*. 2008, pp. 1–10.

[6] Wanpeng Li and Chris J Mitchell. “Security issues in OAuth 2.0 SSO implementations”. In: *Information Security: 17th International Conference, ISC 2014, Hong Kong, China, October 12-14, 2014. Proceedings 17*. Springer. 2014, pp. 529–541.

[7] Hui Wang et al. “The achilles heel of OAuth: a multi-platform study of OAuth-based authentication”. In: *Proceedings of the 32nd Annual Conference on Computer Security Applications*. 2016, pp. 167–176.

[8] San-Tsai Sun and Konstantin Beznosov. “The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems”. In: *Proceedings of the 2012 ACM conference on Computer and communications security*. 2012, pp. 378–390.

[9] Wanpeng Li and Chris J Mitchell. “Security issues in OAuth 2.0 SSO implementations”. In: *Information Security: 17th International Conference, ISC 2014, Hong Kong, China, October 12-14, 2014. Proceedings 17*. Springer. 2014, pp. 529–541.

[10] Chetan Bansal et al. “Discovering concrete attacks on website authorization by formal analysis”. In: *Journal of Computer Security* 22.4 (2014), pp. 601–657.

[11] Dick Hardt. *The OAuth 2.0 Authorization Framework*. RFC RFC 6749. RFC Editor, Oct. 2012. DOI: 10.17487/RFC6749. URL: https://datatracker.ietf.org/doc/rfc6749.

[12] Michael B. Jones and Dick Hardt. *The OAuth 2.0 Authorization Framework: Bearer Token Usage*. RFC RFC 6750. RFC Editor, Oct. 2012. DOI: 10.17487/RFC6750. URL: https://datatracker.ietf.org/doc/rfc6750.

[13] Justin Richer. *OAuth 2.0 Token Introspection*. RFC RFC 7662. RFC Editor, Oct. 2015. DOI: 10.17487/RFC7662. URL: https://datatracker.ietf.org/doc/rfc7662.

[14] Eric Rescorla and Tim Dierks. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC RFC 5246. RFC Editor, Aug. 2008. DOI: 10.17487/RFC5246. URL: https://datatracker.ietf.org/doc/rfc5246 (visited on 05/27/2023).

[15] Daniel Fett, Ralf Küsters, and Guido Schmitz. “A comprehensive formal security analysis of OAuth 2.0”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, pp. 1204–1215.

[16] Yvonne Wilson and Abhishek Hingnikar. *6. OpenID Connect*. Springer, 2019. ISBN: 978-1-4842-8261-8. URL: https://learning.oreilly.com/library/view/solving-identity-management/9781484282618/html/475485_2_En_6_Chapter.xhtml.

[17] Yvonne Wilson and Abhishek Hingnikar. *5. OAuth 2 and API Authorization*. Springer, 2019. ISBN: 978-1-4842-8261-8. URL: https://learning.oreilly.com/library/view/solving-identity-management/9781484282618/html/475485_2_En_6_Chapter.xhtml.

[18] Per Runeson. “A survey of unit testing practices”. In: *IEEE software* 23.4 (2006).

[19] George Klees et al. “Evaluating fuzz testing”. In: *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2018, pp. 2123–2138.
[20] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. “Automatic verification of finite-state concurrent systems using temporal logic specifications”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8.2 (1986).
[21] Edmund Clarke et al. “Progress on the state explosion problem in model checking”. In: *Informatics*. Vol. 10. 2001, pp. 176–194.
[22] Herman Geuvers. “Proof assistants: History, ideas and future”. In: *Sadhana* 34 (2009), pp. 3–25.
[23] Cas Cremers. “Compositionality of security protocols: A research agenda”. In: *Electronic Notes in Theoretical Computer Science* 142 (2006).
[24] F.J. Thayer Fabrega, J.C. Herzog, and J.D. Guttman. “Mixed strand spaces”. In: *Proceedings of the 12th IEEE Computer Security Foundations Workshop*. Proceedings of the 12th IEEE Computer Security Foundations Workshop. June 1999. DOI: 10.1109/CSFW.1999.779763.
[25] Danny Dolev and Andrew Yao. “On the security of public key protocols”. In: *IEEE Transactions on information theory* 29.2 (1983).
[26] Roger M Needham and Michael D Schroeder. “Using encryption for authentication in large networks of computers”. In: *Communications of the ACM* 21.12 (1978), pp. 993–999.
[27] Makarius Wenzel et al. *The isabelle/isar reference manual*. 2004.
[28] Tobias Nipkow. “Programming and proving in Isabelle/HOL”. In: *Technical report, University of Cambridge*. 2013.
[29] “Chapter 5 Higher order predicate logic”. In: *Categorical logic and type theory*. Ed. by Bart Jacobs. Vol. 141. Studies in Logic and the Foundations of Mathematics. ISSN: 0049-237X. Elsevier, 1998, pp. 311–372. DOI: https://doi.org/10.1016/S0049-237X(98)80035-9. URL: https://www.sciencedirect.com/science/article/pii/S0049237X98800359.
[30] Sebastian Mödersheim and Alessandro Bruni. “AIF-ω ω: Set-Based Protocol Abstraction with Countable Families”. In: *Principles of Security and Trust: 5th International Conference, POST 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings 5*. Springer. 2016, pp. 233–253.
[31] Alessandro Bruni et al. “Set-pi: Set membership p-calculus”. In: *2015 IEEE 28th Computer Security Foundations Symposium*. IEEE. 2015, pp. 185–198.
[32] Suhas Pai et al. “Formal Verification of OAuth 2.0 Using Alloy Framework”. In: *2011 International Conference on Communication Systems and Network Technologies*. 2011, pp. 655–659. DOI: 10.1109/CSNT.2011.141.
[33] Haixing Yan et al. “Verification for OAuth Using ASLan++”. In: *2015 IEEE 16th International Symposium on High Assurance Systems Engineering*. 2015, pp. 76–84. DOI: 10.1109/HASE.2015.20.
[34] Andreas V Hess, Sebastian A Mödersheim, and Achim D Brucker. “Stateful Protocol Composition in Isabelle/HOL”. In: *ACM Transactions on Privacy and Security* 26.3 (2023).
[35] Roy T. Fielding, Mark Nottingham, and Julian Reschke. *HTTP Semantics*. RFC RFC 9110. Num Pages: 194. Internet Engineering Task Force, June 2022. DOI: 10.17487/RFC9110. URL: https://datatracker.ietf.org/doc/rfc9110.

[36] Charlie Jacomme and Steve Kremer. “An extensive formal analysis of multi-factor authentication protocols”. In: *ACM Transactions on Privacy and Security (TOPS)* 24.2 (2021).

A Appendix

Files: https://github.com/anandv96/thesis-sso