Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis

Modelling and Analysis of Web Applications in Tamarin

Student: Sandra Dünki
Student ID: 13-914-320
Supervisors: Dr. Christoph Sprenger, Dr. Ralf Sasse
Professor: Prof. David Basin
Date: 1 October 2019

Abstract

In this thesis, we present a generic framework for modelling key elements of the Web infrastructure. We implement the framework using the security protocol verification tool Tamarin. We describe some of the security challenges the Web faces today and show how some of the most common attacks can be modelled using the framework. The framework can also be used to model concrete Web applications and protocols which we demonstrate using the example of the authentication protocol OpenID Connect. In addition, we explain the general usage of the Tamarin prover and give insight into its inner working.

Contents

1 Introduction

The Web has grown historically with many features having been added on-the-go. Consequently, there is no universal standard on how things are done. Some guidelines like the Representational State Transfer (REST) application programming interface (API) emerged but they are not mandatory. In the end, it is mostly up to the companies which protocols and which configurations they want to use. As the Web grows ever more complex, it gets increasingly more important to have tools for automatic analysis available. Manual inspection of Web application protocols can help discover and fix vulnerabilities but it is by no means enough. Many different components can interact with each other and lead to undesired behaviours. For example, a network protocol might be provably secure with respect to a threat model in itself but is this still true once we factor in browser plug-ins or malicious Web sites? Especially for security-critical Web applications manual inspection does not suffice. Manual inspection often misses edge-cases or unlikely combinations of events that can lead to vulnerabilities. In addition to that, assumptions are often made in development and verification of applications. It is near impossible to consider all possible interactions by manual inspection. This is why a formal model allowing formal analysis is so crucial to stay one step ahead of the attackers. The difficulty lies in the complexity of the Web. A model of the Web infrastructure will inevitably be an abstraction of reality and some simplifications must be made. Which simplifications are reasonable and which lead to missing crucial attacks is no easy question to answer. In general, all models are abstractions and can never guarantee the absence of vulnerabilities, only their presence.

Contributions In this work, we develop a framework for modelling elements of the Web infrastructure that can be used as a toolbox by future work. We implement the framework using a security protocol verification tool, specifically the Tamarin prover. Our work is the first general framework of Web applications and the Web infrastructure implemented in Tamarin. Elements include browser functionality, tabs, cookies, redirects, users, Web pages, server logic, login procedures, HTTP(S) messages, different attacker models, and script execution capabilities. We perform several case studies as proof of concept by recreating common Web attacks like request forgery or cookie stealing. To show applicability of the framework to concrete Web applications and protocols, we model and implement two configurations of the authentication protocol OpenID Connect. OpenID Connect was mod-

elled using Tamarin before by Xenia Hofmeier for her Bachelor's thesis [1]. We recreate her model of the OpenID Connect protocol to show our model does not lack any crucial elements to model common Web applications and protocols. In addition, our framework allows for a much more detailed model of OpenID Connect including cookies and scripts. Especially scripts allowed to model a part of the protocol not modelled by Hofmeier, namely the token retrieval script provided by the Relying Party.

Outline In Section 2, we give an introduction to Web applications and the security challenges they face today. We also introduce OpenID Connect, which will be our major case study later on. Before we present our framework in Section 4, we give an introduction to the Tamarin prover in Section 3. We present several case studies to demonstrate applicability of the framework in Section 5. In Section 6, we give an overview of related works. We summarise and conclude in Section 7.

2 Web Applications

Today's everyday life depends heavily on the World Wide Web and its Web applications. The term *Web application* is not formally defined and it is difficult to draw the line between a dynamic Web page and a Web application. Nonetheless, there are many that are largely understood to be Web applications. Popular examples include social media platforms, online banking, file sharing systems, Web mail clients, and online shops.

Web applications are client-server programs. The client usually is a Web browser on a user's computer providing the graphical user interface and the server usually provides the application logic. Often, both sides run a script. On the client side, JavaScript became popular while on the server side PHP is a common language. The shift from traditional applications, which are installed and run solely on the client machine, to Web applications is natural considering the advantages they provide for companies. First, a Web application is easier to maintain than traditional programs because they often required a different version for every operating system and hardware. This made development, maintenance, and releasing updates a huge effort. Second, services offered through a Web application are generally priced using a subscription model compared to traditional programs which are more often sold at a one-time price. Third, the centralised storage of user data at the server side gives companies a lot of information about their customers. Fourth, the end-users often prefer a solely Web-based service because there is no installation required and synchronisation across devices is easy.

2.1 History

While Web applications simplify some aspects of development, they also introduce more complexity to the Web. As the Web has grown historically, many features (especially security features) were not built into the design but added on top as time went by. Understanding how the Web started out can help understand the challenges faced today.

The World Wide Web (WWW) originally started out as a non-profit software to facilitate the fast exchange of documents and information between universities. Its development started in 1989 at the European Organization for Nuclear Research (CERN - Conseil Européen pour la Recherche Nucléaire) by Tim Berners-Lee. The version of the WWW published in 1991 already included Uniform Resource Identifiers (URIs), the HyperText

Transfer Protocol (HTTP), the HyperText Mark-up Language (HTML), the first Web server (`http://info.cern.ch/`), and the first Web browser with a built-in HTML editor. Development continued and many contributors joined the project. In 1993, the WWW was responsible for 1% of Internet traffic. In 1994, there were already 10 million users but no support for e-commerce or any security features. This growth soon led to scalability problems. To solve the scalability issue, Roy Fielding introduced a set of constraints that would help the servers scale to the increasing demand. These include caches between client and server as well as the stateless constraint which frees the server from memorising any client state [2, 20].

The original functionalities soon became too limiting. JavaScript was the main driver of richer applications. Originally developed by Netscape in 1995, JavaScript has the ability to manipulate the document object model (DOM) and, thus, manipulate Web pages directly within the browser. The success of JavaScript was further amplified by the development of AJAX (asynchronous JavaScript). AJAX is an asynchronous Web development technique that allows to retrieve data from the server without having to reload the page. This opened the door for many operations to be performed without the user noticing and, thus, also introducing new vulnerabilities like the CSRF attack described in Section 2.2.2. The second major addition to the original design is a client-side state. Shopping carts of e-commerce stores, for example, rely on some sort of state to keep track of what the user wants to purchase. As HTTP is by design stateless, cookies were introduced. Cookies encode all necessary information and are stored by the server on the client-side. The browser attaches the cookie to every subsequent request to the server. In addition to cookies, client-side local and session storage appeared. This introduced the need to protect resources and data from potentially malicious scripts [14, 19].

2.2 Security Concerns

Web applications perform a lot of security critical functions including payment, online banking, and storage of potentially sensitive data. Access to these functionalities is usually protected by some sort of login and is only granted with the correct login credentials. Most standard Web applications use the common username and password approach. More security-aware applications might offer two-factor authentication but this is usually optional. In Switzerland, only some functionalities that are widely considered security-

critical by the public like payment and online banking enforce a second factor. As with all other applications, usability often beats security. This leads to many problems. In practice, people choose guessable passwords or reuse passwords for multiple Web applications. This leads to many compromised accounts on various platforms if the password database of one platform is leaked. Even if passwords are assumed to be secure, reliable authentication and authorisation protocols are still needed. One of these protocols is the OpenID Connect protocol. We take a closer look at this protocol in Section 2.3.

Password security and server-side database leaks are not the only weak points of Web applications. Countless attack vectors targeting the client-side, i.e., the browser, exist as well. These are often hard to find by manual inspection due to the complexity of the Web infrastructure and the browser behaviour. In the following sections, we will describe three of the most common ones in more detail. For more information on the described attacks or other attacks, see [14].

2.2.1 Cross-Site Scripting

Cross-site scripting (XSS) is a very powerful attack. If successful, the attacker can choose arbitrary (JavaScript) code to be run on the client. Moreover, this code is executed within the application context which means that all client-side resources and programming interfaces are exposed. The attacker can, for example, steal cookies or trigger legitimate looking requests to the server. In order for the attack to succeed, the Web application needs to be vulnerable to a script injection attack. This means that the attacker can send a legitimate request containing a script snippet to the Web application and the Web application will treat the script snippet like any other content. This attack is traditionally prevented by input and output sanitisation. This means that all input and output is changed such that it cannot contain any unwanted executable code. Input/output sanitisation is still an ongoing research area. While older sanitisation techniques simply removed potentially dangerous characters like <> or only allowed whitelisted characters, modern techniques are context dependent. Despite the progress, script injection is still one of the main threats to the Web and leads the OWASP Top Ten list of security risks concerning Web applications both in 2013 and 2017 [3]. XSS attacks themselves dropped from place three in 2013 to place seven in 2017.

Assuming, thus, that the server-side of the Web application is vulnerable

to a script injection attack, one possible XSS attack proceeds as follows. Note that different types of XSS attack exist, the described one is commonly known as a *stored* XSS attack [4, 14]. In a stored XSS attack, the attacker sends a legitimate request to the vulnerable Web application. This request contains a XSS payload, i.e., malicious, executable code. The vulnerable Web application does not perform input sanitisation and so the XSS payload is stored just like any other content on the server-side storage. Next, the attacker needs the victim user to visit the Web site. This can be done, for example, by tricking the user with an e-mail or just by waiting if the user visits the Web page frequently anyway. Recall that the user trusts the Web application and that the Web application is not in itself malicious. Once the client requests the Web page, the server retrieves the corresponding content including the attacker script. As the Web server also does not perform output sanitisation, the script content gets sent to the client unaltered. The browser treats the attacker code the same as the legitimate code and executes it in the application's context [14].

2.2.2 Cross-Site Request Forgery

The goal of a cross-site request forgery (CSRF) attack is to trick the user's browser into sending a request to a target Web application without the user noticing. The Web application has no means of distinguishing such a request from a legitimate, user initiated request. Often, the request is a state changing one. Critical examples include transactions involving money or updating account information.

Triggering the browser to send a request is an alarmingly easy task. Modern Web pages often contain content from external sources, e.g., images. Whenever the browser loads a Web page containing, e.g., an external image a request is sent to this external source. The attacker can just insert some code triggering a request to the target Web application into their own Web page. A state-changing CSRF attack can only succeed if the user is already authenticated to the target Web application. Unfortunately, a user often remains logged in at a Web application for a long time and session information such as login cookies are attached to any outgoing request without differentiating between same-origin and cross-origin requests. This way, the state changing attack succeeds without the user noticing or having to reauthenticate.

Fortunately, there are effective countermeasures against CSRF attacks. CSRF attacks occupied place eight in the OWASP's Top Ten list in 2013

and were no longer on the list in 2017. It is, however, estimated that 5% of Web applications still contain a CSRF vulnerability [3]. The threat should, therefore, not be ignored. The main idea of the countermeasures is to make the request unforgeable. One possibility is to force the user to reauthenticate when performing critical state-changing operations. The e-commerce marketplace amazon.de, for example, requires a reauthentication when changing login information (tested 31.7.2019). Another approach is to include a fresh nonce in two different locations of the request. For example, include a nonce in a hidden form field and the same nonce in the cookie. The server then compares the two values and rejects the request if they are different. The idea is that the attacker cannot forge both values [14].

2.2.3 Session Hijacking

Stolen user credentials allow the attacker to authenticate themselves as the user to the Web application in the attacker's own browser. This give the attacker full control over the session and the same access to the application as the victim. This attack requires knowledge of the user's credentials and possibly possession of second factor devices. Even without two-factor authentication enabled, this attack can largely be prevented by proper password hygiene such as choosing strong passwords and not reusing passwords across platforms. There are other ways, however, for an attacker to gain full control over an authenticated session between victim and application. They can even transfer the authenticated session to their browser. One such attack is called session hijacking and is described in the following.

Recall that today, session management is usually done using cookies. As the HTTP protocol is inherently stateless, a cookie can be stored on the client-side by the server and is attached by the browser to every subsequent request to the same domain. This allows some sort of state which is vital for, e.g., shopping carts in online shops to work properly. A login cookie that authenticates a user is stored on the browser after a successful login by the user. Usually, the login cookie simply contains a fresh random session identifier. The server stores the session identifier on its side and compares them upon receiving a login cookie. If an attacker can get access to the session identifier or cookie, they can just attach it to their requests to the Web application. The server will authenticate the attacker as the victim.

There are several possible ways to steal a cookie. The attacker can use a XSS attack (see Section 2.2.1) to access the cookie storage from within the

target application's context. The attacker can trick the user into installing a malicious browser extension which compromises the browser and allows the attacker to access the cookie storage this way. The attacker can listen into network traffic and extract the cookie from the sniffed messages as the cookie is attached to every request after authentication. The attacker can also try to guess the session identifier if the Web application does not generate it randomly but bases it on user information like their birthday. The problem gets amplified by the many insecure free Wi-Fis available which make sniffing network traffic very easy. Another amplifying factor is that the user need not be logged in at the time of the attack. The attacker can steal the session identifier and then simply wait until the user authenticates himself because many Web applications do not issue a new session identifier after login [14].

2.3 Single Sign-On Protocols

When considering all the risks and possible threats, it is natural that Web application providers are looking for solutions. Especially smaller Web applications benefit from outsourcing security-critical steps like the login procedure to larger companies having more experience and resources to handle it properly. This includes authentication and authorisation of users and proper storage of user credentials. In 2017, there was still at least one company that did not handle password resets properly. One way to for a user to change their password was to send it via e-mail to the company. This means that anyone sniffing network traffic could learn the password and can possibly also forge a password reset e-mail. In addition, the company employee processing the e-mail learns the new password [5]. This is especially bad if the user reuses the password for other Web applications. Because of these issues, single sign-on options became quite popular in recent years. Single sign-on mechanisms allow a user to sign in to a Web application using their existing account at a different Web application. Users are generally happy to use this solution as they need not remember a new password for every platform. The Web application outsourcing the login procedure benefits from not having to handle the login procedure. The Web application providing the login handling benefits by being able to farm more information about the user. Considering the incentive for every party to use single sign-on mechanisms, they are probably here to stay. Maybe in the future there will be only a few centralised platforms providing login mechanisms.

Single sign-on introduces, however, new security concerns and attack

vectors. In the following, we describe the authentication protocol OpenID Connect. OpenID Connect is designed for the purpose of single sign-on described above.

2.3.1 OpenID Connect

OpenID Connect is a protocol based on the OAuth 2.0 standard. It provides authentication guarantees in addition to the authorisation guarantees of OAuth 2.0. OpenID Connect is the third iteration of the OpenID project and is now widely adapted, with companies like Google and Microsoft offering login handling to other sites. OpenID Connect's FAQ page [6] desribes the purpose of it as follows: "It lets app and site developers authenticate users without taking on the responsibility of storing and managing passwords in the face of an Internet that is well-populated with people trying to compromise your users' accounts for their own gain." It praises itself to be developer-friendly and easy to deploy.

Before we dive into the protocol specification of OpenID Connect, we need to establish some terminology. Note that we refer to the OpenID Connect Core 1.0 as described in [7] whenever talking about OpenID Connect. In the OpenID Connect context, the end-user, the person sitting in front of the computer, is called the *Resource Owner* (RO). The browser the RO uses to access the Web is typically called the *User Agent* (UA). The Web application the RO is signing in to is called the *Relying Party* (RP). Confusingly, sometimes the RP is also called the client but we will stick with RP to avoid misunderstandings. This is the application that outsources the login procedure to a third party. This third party providing the login handling is called the *Identity Provider* (IdP). Sometimes the IdP is also called the *OpenID Provider* (OP) or *Authentication Server* (AS). We will not use these two terms here to avoid confusion.

For the OpenID Connect protocol to work, the RP and the IdP need to have already exchanged some information in advance. This is done using the OpenID Connect Discovery Protocol and OpenID Connect Dynamic Client Registration. During these, the RP learns the IdP's configuration, its issuer identifier, the endpoints of the IdP and a URL where the RP can retrieve the public key of the IdP. The IdP learns the RP's redirect URIs and issues a client identifier and an optional client secret to the RP. We will not go any deeper into this preparation stage of OpenID Connect and assume that the mentioned information has been successfully exchanged. There are three

possible OpenID Connect flows specified. Depending on the circumstances, one might be more suitable than the other two.

Authorisation Code Flow The authorisation code flow is suitable if the RP and the IdP can establish a client secret beforehand and, thus, create an authenticated back-channel. This way, critical tokens are not exposed to the UA, which might contain multiple vulnerabilities, e.g., malicious browser extensions. The drawback is that the IdP learns at which RP the RO is signing in. This might be undesirable. The authorisation code flow contains the following high-level steps. For the corresponding message sequence chart (MSC), see Figure 1[1]. For a detailed description of every message, see [7].

1. The RO navigates the UA to the Web page of the RP and chooses the single sign-in option with the preferred IdP (message 1).
2. The RP directs the UA to the IdP and sends an Authentication Request to the IdP's Authentication Endpoint (messages 2-3).
3. The RO authenticates to the IdP.
4. The IdP displays a consent dialogue to the RO. The RO must explicitly authorize the RP to access their data (messages 4-5).
5. The IdP directs the UA back to the RP together with an Authorisation Code (messages 6-7).
6. The RP sends the received Authorisation Code back to the Token Endpoint of the IdP (message 8).
7. The token endpoint responds with an ID Token and an Access Token (message 9). The RP verifies the tokens' signatures.
8. The RP can use the Access Token to retrieve the authorised user data from the IdP's resource server (omitted in MSC).
9. The RP can retrieve the RO's identity from the ID Token and log them in (message 10).

[1] All MSCs and diagrams in this thesis are drawn with `draw.io`

The Authorisation Request consists of the following main components. The `response_type` parameter indicates which OpenID Connect flow should be used. For the authorisation code flow, it is set to `code`. The `redirect_uri` parameter contains the URI the UA is redirected to in step 5. The `client_id` parameter contains the identifier of the RP (client identifier). Optional parameters include a `nonce` to mitigate replay attacks and a `state` parameter. For a full list see [7].

The ID Token is the main addition of OpenID Connect to the OAuth 2.0 protocol. It contains information about the RO while the Access Token acts as a bearer token for the RP to access the RO's data on the IdP's resource server [8]. The main components of the ID Token are the following. For a full list, see [7]. The `iss` is the Issuer Identifier identifying the IdP. The `sub` is the Subject Identifier identifying the end-user RO. The `aud` is the Audience Field identifying the RP (client identifier). The token also contains an expiration `exp` time and an optional `nonce`.

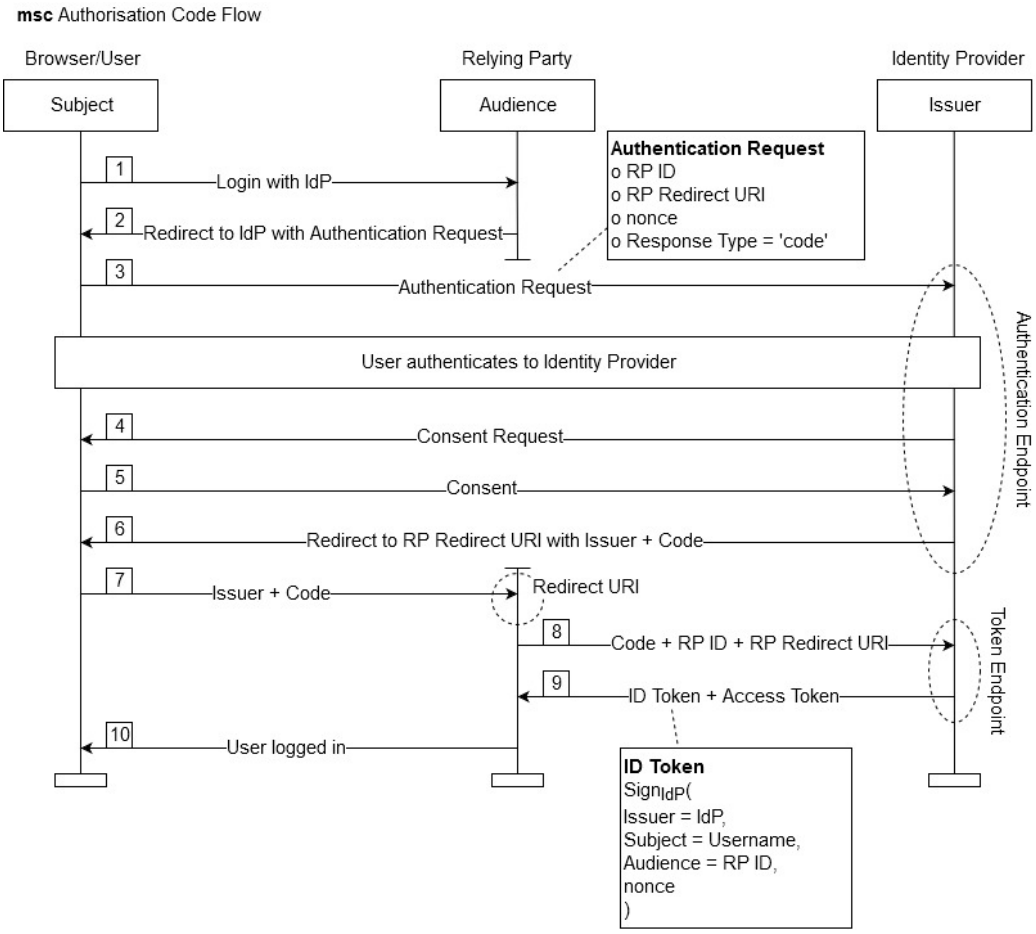

Figure 1: OpenID Connect Authorisation Code Flow

Implicit Flow In the Implicit Flow, no Authorisation Code is used. The ID Token and (if requested) Access Token are returned directly. This has the drawback that the tokens are exposed to the UA. A compromised browser may leak them to an attacker. The other defining feature of the Implicit Flow is that RP and IdP need not share a secret and the IdP does not authenticate the RP. This can be an advantage or a disadvantage depending on the circumstances. On the one hand, it introduces new potential vulnerabilities. On the other hand, it makes OpenID Connect possible whenever a back-channel cannot be established beforehand.

The Implicit Flow contains the following high level steps. For corresponding message sequence chart (MSC), see Figure 2. For a detailed description of every message, see [7].

1. The RO navigates the UA to the Web page of the RP and chooses the single sign-on option with the preferred IdP (message 1).

2. The RP directs the UA to the IdP and sends an Authentication Request to the IdP (messages 2-3).

3. The RO authenticates to the IdP.

4. The IdP displays a consent dialogue to the RO. The RO must explicitly authorize the RP to access their data (messages 4-5).

5. The IdP directs the UA back to the RP (messages 6-7). The IDP includes the ID Token and (if requested) an Access Token in the URL fragment. The RP verifies the signature(s).

6. The RP provides a script that extracts the ID Token (and Access Token if requested) from the URL fragment (messages 8-9)

7. The RP can use (if requested) the Access Token to retrieve the authorised user data from the IdP's resource server (omitted in MSC).

8. The RP can retrieve the RO's identity from the ID Token and log them in (message 10).

Note that the first four steps are analogous to the Authorisation Code Flow. The Authentication Request's `response_type` for the Implicit Flow is `id_token token` to request an ID Token as well as an Access Token and

`id_token` to request only the ID Token. The ID Token has a similar format as for the Authorisation Code Flow except that the `nonce` is required instead of optional [7].

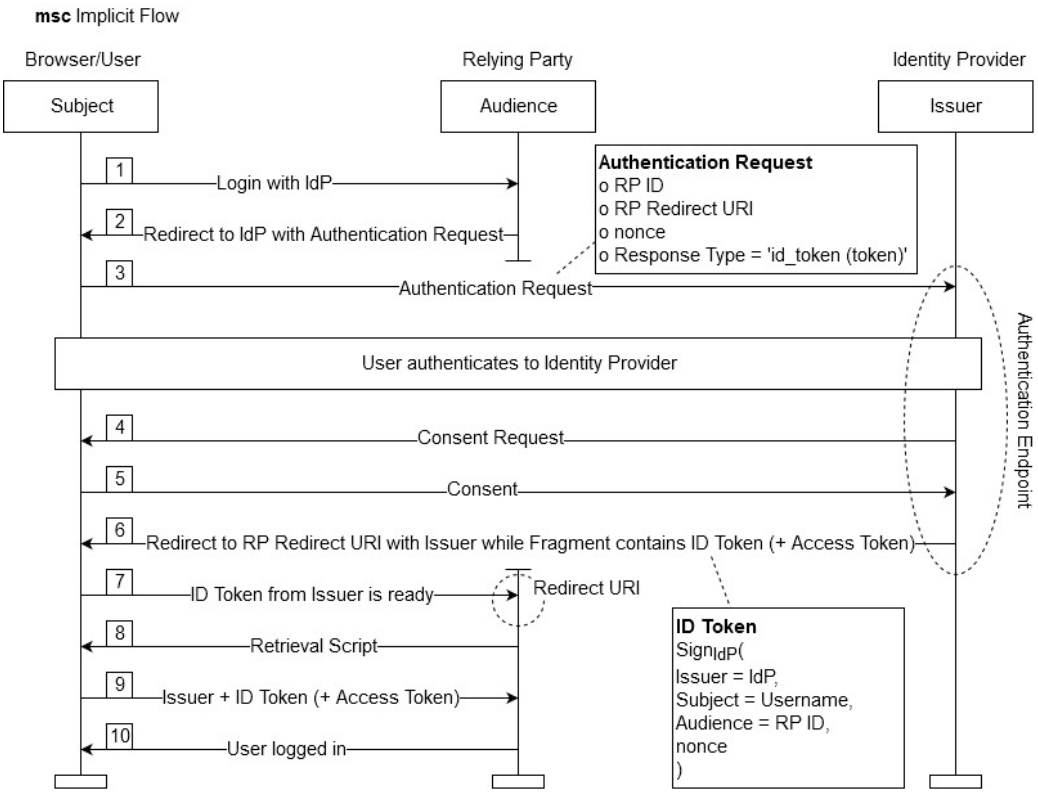

Figure 2: OpenID Connect Implicit Flow

Hybrid Flow The Hybrid Flow is a mixture between the Authorisation Code Flow and the Implicit Flow in the sense that some tokens are returned directly to the RP and thus exposed to the UA and other tokens need to be retrieved separately from the the IdP by the RP using the received Authorisation Code.

3 Tamarin

To formalise and analyse our model of Web applications and the Web infrastructure, we use the Tamarin prover. The Tamarin prover is a tool designed to model and analyse security protocols. This includes verification and falsification of security properties with respect to the model of the protocol. Typical use cases include network protocols like the Needham-Schroeder(-Lowe) key exchange protocols. It turned out, however, to be also useful for modelling and analysing Web applications and the Web infrastructure.

In this section, we give a high-level description of the Tamarin prover and its features. We describe how the Tamarin prover is used and how an input file, a so called theory, to the Tamarin prover is structured. We also give an overview on the inner working of the Tamarin prover, the problems that can occur, and how they can be solved. Note that we focus here solely on the aspects relevant for our model. For an exhaustive description on Tamarin, see the manual [10].

3.1 Theory File

The Tamarin prover expects as input a `.spthy` file. The theory file contains the following elements. It can also contain elements not listed here as we only list and describe the elements used in our model.

- Keywords to indicate start, end, and name of the theory
- Built-in message theories like (a)symmetric encryption and hashing
- Multiset rewriting rules that build the core of every Tamarin theory
- Restrictions to restrict the set of possible traces
- Let bindings to make multiset rewriting rules more readable and modular
- Lemmas to describe the properties of the formalised protocol we want to prove

In the following, we describe each element in more detail.

3.1.1 Keywords

Every theory starts with the keyword `theory`, the name of the theory, and the keyword `begin`. Each theory ends with the keyword `end`.

```
theory name
begin
...
end
```

3.1.2 Built-In Message Theories

Tamarin offers a variety of built-in functions that consist of the most common functions needed to model security protocols, e.g., encryption. On how to define custom functions, see [10]. Built-in functions are usually added right after the `begin` keyword.

```
theory name
begin

builtins: asymmetric-encryption, hashing
...
end
```

The built-in message theories provided by Tamarin are: `hashing`, `asymmetric-encryption`, `symmetric-encryption`, `signing`, `revealing-signing`, `diffie-hellman`, `bilinear-pairing`, `xor`, and `multiset`. For our work, we use the built-ins `hashing`, `asymmetric-encryption`, `symmetric-encryption`, and `revealing-signing`. Thus, we describe those here and refer to [10] for the others.

`hashing` models a hash function. It defines the function symbol `h/1` with one argument and is used like `h(argument)`. A hash function is by design not reversible and the `argument` is not retrievable from just the hash.

`asymmetric-encryption` models asymmetric encryption with a public/private key pair. It defines the function symbols `aenc/2`, `adec/2`, and `pk/1`. `aenc/2` models encryption, `adec/2` the corresponding decryption and `pk/1` the public key corresponding to the given private key. The function symbols relate to each other as would be expected intuitively. Formally, they relate according to the equation `adec(aenc(message, pk(privatekey)), privatekey)= message`.

`symmetric-encryption` models symmetric encryption. It defines the function symbols `senc/2` and `sdec/2`. `senc/2` models encryption, `sdec/2` the corresponding decryption. Both need two arguments, the message to be en-/decrypted, and the symmetric key. The function symbols relate to each other as would be expected intuitively. Formally, they relate according to the equation `sdec(senc(message, symmetrickey), symmetrickey)= message`.

`revealing-signing` models a signature from which the signed message is retrievable. It defines the function symbols `revealSign/2`, `revealVerify/3`, `getMessage/1`, `pk/1`, and `true`. The function symbols relate to each other according to the equations `revealVerify(revealSign(m, sk), m, pk(sk))= true` and `getMessage(revealSign(m, sk))= m`. This is in contrast to the built-in message theory `signing` which includes a function symbol to verify the signature but none to extract the message.

3.1.3 Multiset Rewriting Rules

The theory's multiset rewriting rules (MSR rules) define a labelled transition system. Both the left-hand side and the right-hand side of a MSR rule is a multiset of facts. A fact represents an element in the system, e.g., a public key belonging to an agent. Each MSR rule can be labelled with one or multiple action facts. An action fact is an annotation to be able to reference to the rule and formulate properties. A sequence of action facts generated by an execution is called a trace. Formally, properties are statements about traces. The left-hand side facts of a MSR rule are consumed by the rule and the right-hand side facts are created by the rule. Some facts disappear when consumed (linear facts), but others do not and can be consumed an arbitrary number of times (persistent facts). The multiset of currently consumable facts is the system's state. Below you can find a template of a multiset rewriting rule.

```
1 rule Name_Of_Rule:
2     [ Fact_Consumed(term) ]
3   --[ Action_Fact_Name(term) ]->
4     [ Fact_Generated(term)]
```

Some facts are built into Tamarin and have a special functionality. The three most relevant to us are described in the following. The fresh fact `Fr(~x)` represents a fresh random value `~x`. Fresh terms are marked with a `~`. It can be, for example, used for key generation. A fresh fact can only occur on the left-hand side of a user-defined rule and will generate a fresh value on

every usage. A fresh fact is generated by a built-in rule `[] --> [ Fr(~x)]` that does not need to be implemented in the theory. Below you can see how the fresh fact is used to generate a fresh browser identifier which in turn is used to create persistent Browser fact containing the fresh value as a term. Persistent fact are marked with a `!`.

```
rule Create_Browser:
    [ Fr(~browser_id) ]
  --[  ]->
    [ !Browser(~browser_id) ]
```

The other two special facts are the `In(m)` and `Out(m)` facts. They represent the interface to the untrusted network. `Out(m)` sends the message `m` to the untrusted network. `In(m)` receives the message `m` from the untrusted network. `In(m)` can only occur on the left-hand side of a user-defined rule while `Out(m)` facts can only occur on the right-hand side of a user-defined rule. Both facts have arity one. Examples of `In(m)` and `Out(m)` facts can be found below.

```
rule Server_Send_Over_Http:
    [ Server_Send(request_id, url, response) ]
  --[ Protocol('HTTP')]->
    [ Out(<request_id, url, response>) ]
```

```
rule Server_Receive_Over_Http:
    [ In(<request_id, url, request>) ]
  --[ Protocol('HTTP') ]->
    [ Server_Receive(request_id, url, request) ]
```

A rule can be labelled with action facts that are written in the middle of a rule. The first example above `Create_Browser` does not have any action facts while the second two do. Action facts are used to write restrictions and lemmas but do not have any influence on the application of the rules themselves.

3.1.4 Restrictions

Restrictions are very powerful and enable models that would otherwise be impossible to make. Basically, a restriction is a property that must hold for every trace. Tamarin does not prove the property, it just assumes it. Two of the most useful and widely applicable restrictions are described below.

The restriction `Equality` says that whenever an action fact `Eq(x,y)` occurs in the trace, `x` and `y` must be equal. Similarly, the restriction `Inequality` says that whenever an action fact `Neq(x,y)` occurs in the trace, the two argument must not be equal. These example restrictions can be found in the Tamarin manual [10] in Chapter 6.

Note that `#` denotes a temporal variable and `@i` can be read as "at time point i". Also note that `@i` is short for `@ #i`.

```
restriction Equality:
  " All x y #i. Eq(x,y) @ #i ==> x = y "

restriction Inequality:
  " All x #i. Neq(x,x) @ #i ==> F "
```

The syntax for a restriction uses the keyword `restriction` followed by the restrictions name and a colon. The restriction itself is written as a logical formula in guarded first-order logic inside double-quotes `" ... "`.

3.1.5 Let Bindings

Let bindings allow one to define local macros in the scope of its corresponding rule. They are especially useful if a term occurs multiple times in a rule or we need to access the components of a term.

In the example below, we defined the macro `url = <'HTTPS', $Server, $Path, $File>`. This simply means that all occurrences of `url` are equivalent to the tuple `<'HTTPS', $Server, $Path, $File>`. This has two main advantages. First, we do not have to write `<'HTTPS', $Server, $Path, $File>` twice, which would make it less readable, while still allowing us to access the component `$Server` of `url` which we need for the action fact `Send_To($Server)`. Second, it allows to give a term a name. It is immediately clear that the second term of the `Browser_Send` and `HTTPS_Req_Out` facts represents a URL which would not be so obvious if we wrote `<'HTTPS', $Server, $Path, $File>`. Note that terms marked with `$` are public knowledge and terms enclosed in single-quotes `'...'` are strings.

```
rule Browser_Send_Over_Https:
let url = <'HTTPS', $Server, $Path, $File>
in
    [ Browser_Send(request_id, url, request) ]
  --[ Protocol('HTTPS') ]->
    [ HTTPS_Req_Out(request_id, url, request) ]
```

3.1.6 Lemmas

Lemmas are very much like restrictions with the crucial difference that, unlike restrictions, lemmas are not assumed but have to be proven or disproven. There are two main types of lemmas, `exists-trace` lemmas and `all-traces` lemmas. `exists-trace` lemmas are existential lemmas. It is true if and only if there exists at least one trace that fulfils the specified property. `all-traces` lemmas on the other hand are true if and only if the property holds for all traces. The lemmas are written as logical fomulas.

Examples adapted from the Tamarin manual can be found below. Note that the `all-traces` annotation is usually omitted. The first lemma informally say that there exists a trace where agent `A` sends message `m` and some agent `B`, possibly the same agent `A`, receives it. The second lemma informally says that whenever someone claims to have received an authentic message `m` from `B` then `B` must have actually sent that message beforehand.

```
lemma executable:
  exists-trace
    " Ex A B m #i #j. Send(A,m)@i & Recv(B,m) @j "

lemma message_authentication:
  all-traces
    " All B m #i. Authentic(B,m) @i
     ==> (Ex #j. Send(B,m) @j & j<i) "
```

3.2 Attacker

The standard, built-in adversary in Tamarin is the Dolev-Yao adversary. A Dolev-Yao adversary can read, modify, intercept, and inject any message on the network [15]. Recall that messages are sent and received from the network with `In(m)` and `Out(m)` facts. If we want to model a more powerful adversary, we need to write MSR rules to explicitly give them the additional capabilities. The below rule, for example, gives the built-in adversary the additional capability to reveal the long-term private key of an agent `A` (i.e., send the private key out into the untrusted network) and, thus, impersonate them.

```
rule Reveal_SK:
    [ !Ltk(A, ltk) ]
  --[ Reveal(A) ]->
    [ Out(ltk) ]
```

The above example adds capabilities to the network attacker. It is also possible to define attacker models other than the network attacker. This is also done by implementing the corresponding MSR rules. Action labels and lemmas then allow to show properties and find attacks with respect to this attacker model.

3.3 Running Tamarin

The standard way to run Tamarin after installation is using the command `tamarin-prover interactive filename.spthy`. By navigating to `http://localhost:3001` in your browser you can access the graphical user interface. The theory `filename.spthy` is already loaded. Once you click on the file in the GUI, precomputation starts.

3.3.1 Precomputation

When opening a theory in the GUI, Tamarin first performs a precomputation. During precomputation, Tamarin searches for all possible sources of all left-hand side facts in the rules. Remember that, except for some special built-in facts like `Fr(~x)`, `In(m)`, and `Out(m)`, all facts consumed on the left-hand side of a rule must have been created by some other user-defined rule's right-hand side implemented in the theory. In some cases, the precomputation will not terminate. If one can pinpoint which fact causes the trouble, one can annotate this fact with `[no_precomp]` in order to still look at the theory in the GUI. In general, however, non-termination of precomputation is a sign that the model should be reconsidered.

3.3.2 Raw Sources and Partial Deconstructions

Precomputation yields the so-called raw sources. This means that the sources of each fact have been precomputed. Sometimes, however, Tamarin is unable to figure out where a fact has to come from. This, together with the fact that Tamarin uses an untyped system, can lead to so-called partial deconstruction.

Let us consider a simple, more illustrative than realistic example to explain the concept.

The first rule below simply takes a fresh value `~x` and creates a fact `Some_Fact(~x)` containing the fresh value. This fact is then consumed by the second rule, which sends the term in this fact out into the untrusted network. Note that the second rule has no idea that the term `m` in the consumed fact is a fresh value. Let us assume that Tamarin is unable to resolve (i.e., compute the source of) the fact `Some_Fact(m)`. In this case, the sent term `m` could be anything including private keys, user secrets, session identifiers and many more. In this case, Tamarin cannot exclude the possibility that the network attacker just learned anything they need to know. This usually leads to non-termination of the lemmas.

```
rule Create_Fresh_Value:
    [ Fr(~x) ]
  --[  ]->
    [ Some_Fact(~x) ]

rule Send_Value:
    [ Some_Fact(m) ]
  --[  ]->
    [ Out(m) ]
```

In the example above, there are two possibilities to resolve the issue. The first one is only to be used if you are absolutely sure that you are not altering the model by doing it. You can tell the prover that the term `m` in the second rule is a fresh value by simply marking it with a tilde `~`, see below. This can possibly alter your model because if there indeed was a trace where the term `m` was not fresh, then you just excluded this trace. This can lead to missing attacks.

```
rule Send_Value:
    [ Some_Fact(~m) ]
  --[  ]->
    [ Out(~m) ]
```

The second possibility is to write a sources lemma. A sources lemma looks like a standard lemma but is annotated with `[sources]`. Unlike a standard lemma, however, a sources lemma is not considered using the refined sources but the raw sources. The goal is to write a true and provable sources lemma that gives Tamarin enough information to resolve all unresolved facts

and, thus, partial deconstructions. In our example, we want to tell Tamarin that whenever the second rule is invoked with a term `n`, then an instance of the first rule also with the term `n` must have preceded the second rule. In order to refer to the rules in the lemma, we annotate them with action facts. The second rule is the one that causes the partial deconstruction on the sent term, so we annotate it with `Problem(m)`. The first rule is the source of the term, so we annotate it with `Source(~x)`. The lemma should express that whenever a `Problem(n)` occurs, then a `Source(n)` must have preceded it or the attacker already knew the term `n`. In the first case, Tamarin learns that the term is a fresh value and cannot be arbitrary. In the second case, the attacker already knows the term and, thus, does not learn anything when it is sent out. `KU()` denotes the attacker knowledge. You can find the complete example below. Note that this second method is completely safe. You cannot inadvertently exclude traces as in such a case, Tamarin would find a counterexample to the sources lemma.

```
rule Create_Fresh_Value:
    [ Fr(~x) ]
  --[ Source(~x) ]->
    [ Some_Fact(~x) ]

rule Send_Value:
    [ Some_Fact(m) ]
  --[ Problem(m) ]->
    [ Out(m) ]

lemma typing [sources]:
" All n #i.
( Problem(n) @i
==>
( (Ex #j. Source(n) @j) | (Ex #j. KU(n)  @j & j < i) )
   ) "
```

3.3.3 Refined Sources

For precomputing the refined sources, Tamarin assumes all sources lemmas. Ideally, there are no partial deconstructions left in the refined sources. The refined sources are used for all non-sources lemmas. If there are no sources lemmas, the refined sources are identical to the raw sources. Note that the

refined sources assume the sources lemmas even if they are not proven or even if they are false. Whenever dealing with partial deconstructions, two things must be checked. First, that all partial deconstructions are complete in the refined sources. And second, that the sources lemmas removing the partial deconstruction are actually provable. Otherwise, one works with potentially wrong assumptions.

3.3.4 Proving Lemmas

To prove an `all-traces` lemma, Tamarin first negates the property in the lemma. It then tries to find a trace that satisfies the negated property. If none is found, the lemma is true. For `exists-trace` lemmas, no negation is needed. For `all-traces` lemmas, Tamarin will either give a proof, in which case one can be certain that the property holds, or give a counterexample, in which case one can be certain that the property is violated. For `exists-trace` lemmas, it is reversed. Tamarin will either find a trace satisfying the property, in which case one can be certain that the property holds, or give a proof that the property does not hold, in which case one can also be certain that the property is not satisfied. Tamarin is sound and complete but might, however, not terminate.

In case of non-termination, there are several possible remedies. First, one should always make sure that all partial deconstructions are removed. This is enough for most standard lemmas. For non-terminating sources lemmas and some standard lemmas other methods are available. Before they can be implemented, some detective work is necessary. When using Tamarin's autoprover, Tamarin chooses the next fact to resolve according to a built-in heuristic. If this heuristic does not lead to a terminating proof, there are possibilities to alter the used heuristics. To do this, one first needs to find out how the heuristics should be altered in order for the proof to succeed. Start Tamarin in the interactive mode and open the GUI in the browser. There you have the option to autoprove the lemmas or to prove them manually. Manual proofs can be especially useful for `exists-trace` lemmas, as you can create the trace you are looking for yourself. For now, however, we are more interested in the `all-traces` lemmas that do not terminate. Choose instead of the autoprover to prove the lemma by induction. Now you see in which order Tamarin's built-in heuristic solves the facts. If you click on one of the facts, Tamarin will solve it and a new level of the search tree appears. Try to find a strategy with which the lemma can be proven. This almost certainly

means deviating from the built-in heuristic at some point.

Once you have found a successful strategy you do not want to prove everything manually every time. There are two ways to alter the heuristics. The easiest way is to annotate facts with `[+]` or `[-]`, which will tell Tamarin to move the fact up or down respectively. The annotations can also be combined with the `[no_precomp]` annotation we saw previously `[-, no_precomp]`. Note, however, that regardless of annotations, loop breakers will always be delayed to the very bottom and attacker knowledge will always be solved first. If annotations are not enough, you can also write a so called oracle. An oracle is a python script that directly alters the ordering of facts in any way you want. To use an oracle, start Tamarin with the corresponding flag `tamarin-prover-release interactive --heuristic=O --oraclename=oraclefile filename.spty`. An example oracle can be found in Appendix A. There are, however, always cases where Tamarin does not terminate due to the nature of the problem at hand.

4 Model

In this section, we introduce our framework for modelling the Web infrastructure. The framework includes models for browsers, users, Web applications, communication, and elements like cookies or URLs. All elements in the framework build a toolbox. Depending on the protocol or situation one wants to model, different elements from this toolbox might be suitable and some might not be needed. In addition to this modular approach, each part of the framework is designed to be easily adaptable and extensible to the use case at hand if needed. We present several use cases and examples in Section 5. Note that we assume encryption, public key infrastructure, and certificate authorities to be working reliably and not be vulnerable to any attacks. A diagram giving an overview of the complete model can be found in Appendix B. An implementation of the model in Tamarin showing executability can be found at [9] along with the case studies we will discuss in Section 5.

4.1 Basic Concepts

In this section, we describe how we modelled the basic components of the Web infrastructure including cookies and cookie attributes but also browser state and URLs.

4.1.1 Unified Resources Locators

In our model, every *unified resource locator* (URL) consists of four components: the protocol, the domain, the path, and the file. All of these are public and, thus, every URL is public knowledge. The template below demonstrates how a URL is used in a Tamarin MSR rule.

```
let url = <$Protocol, $Server, $Path, $File>
in
    [...]
  --[ Protocol($Protocol) ]->
    [...]
```

At the moment, the protocol is restricted to either HTTP or HTTPS. This is enforced by the following restriction.

```
restriction protocol_options:
 " All p #i.( Protocol(p) @i ==> ( p = 'HTTP' | p =
    'HTTPS' )) "
```

Due to the modular design of the model, other protocols could be added with little effort. Fragments are not modelled in the URL itself but can be sent along as a parameter with every message, see Section 4.1.4.

4.1.2 Cookies

A *cookie* in our model consists of a name, a value, and the attributes *HTTPonly* and *Secure*. The attributes are either set to `'false'` or `'true'`. Below you can find a template for such a cookie in Tamarin.

```
cookie = <cookiename, cookievalue, <'HTTPonly',
   httponly>, <'Secure', secure>>
```

Browsers, in our model, can store a cookie in a *cookie storage*. Each such cookie storage is associated with a browser and a domain. Thus, in addition to the cookie itself, a cookie storage also contains the corresponding browser identifier and domain. Our framework assumes existence of a cookie storage and, therefore, a cookie storage needs to be created for every domain even if this Web site does not set any cookies. In such a case, the field representing the cookie will always be `'NULL'`. Below you can find how we implemented the creation of a `Cookie_Storage` in Tamarin. Initially, no cookie is stored in the `Cookie_Storage` and the `cookie` is set to `'NULL'`.

```
rule Create_Cookie_Storage:
let cookie = 'NULL'
in
    [ !Browser(browser_id) ]
  --[ Create_Cookie_Storage(browser_id, $Server) ]->
    [ Cookie_Storage(browser_id, $Server, cookie) ]
```

Each cookie storage contains exactly one cookie (which might be `'NULL'`). There can also only be one cookie storage per domain-browser pair in our framework. We enforce this in Tamarin using the following restriction:

```
restriction only_one_cookie_storage:
" All b s #i #j. ((Create_Cookie_Storage(b, s) @i &
   Create_Cookie_Storage(b, s) @j) ==> #i = #j) "
```

This means that, in our framework, the browser can only store at most one cookie per domain. Even though in reality multiple cookies are possible, one cookie already allows to model interesting mechanisms like login cookies or session cookies. We restricted the number of cookies to one because of performance reasons.

The cookie in the cookie storage can be updated by an incoming response from the server. This is done automatically by the response handler or redirect handler as described in Section 4.3.1. The developer of the model does not need to alter the cookie mechanism in any way though they can change the layout of the `cookie` itself without having to change many Tamarin rules. Recall that, depending on the use case, not all functionalities might be needed. If no cookie mechanism is needed at all, one can just ignore them and then the cookie field will always be `'NULL'`. In this case, one can delete or comment out the rules that update the cookie storage in the browser for performance reasons. One still needs the mechanism to create a cookie storage however.

If the *Secure* flag is set to true, then the cookie may only be sent over HTTPS and not over HTTP. We enforce this in Tamarin by including the following restriction `secure` in our theory.

```
1 restriction secure:
2 "All prot ser path file cookiename cookievalue httponly
    #i.
3 Cookie_Is_Sent(<prot, ser, path, file>, <cookiename,
    cookievalue, <'HTTPonly', httponly>, <'Secure',
    'true'>>) @i
4 ==> (prot = 'HTTPS') "
```

Similary, the *HTTPonly* flag is enforced using the below restriction `httponly`. If the HTTPonly flag is set to true, then the cookie is not accessible by any JavaScript [14] code.

```
1 restriction httponly:
2 "All script_domain cookie_domain cookiename cookievalue
    httponly secure #i.
3 Script_Accesses_Cookie(script_domain, cookie_domain,
    <cookiename, cookievalue, <'HTTPonly', httponly>,
    <'Secure', secure>>) @i
4 ==> (httponly = 'false') "
```

Access to cookies is usually restricted in the browser to scripts that are loaded into the same domain as the cookie [14]. We implement this same-domain policy in Tamarin using the restriction `same_domain_policy`.

```
restriction same_domain_policy:
" All script_domain cookie_domain cookie #i.
    Script_Accesses_Cookie(script_domain, cookie_domain,
    cookie) @i
==> (script_domain=cookie_domain)"
```

On the server side, only the value of a cookie is stored if needed at all. For example, the value of login cookies is stored in a linear server storage fact in order to lookup and validate incoming login cookies (see Section 4.4 for more details).

4.1.3 Local Storage

Local storage is modelled similarly to the cookie storage described in Section 4.1.2. Unlike the cookie storage, however, arbitrary information can be stored in a local storage. In Tamarin this can be implemented as a linear `Local_Storage(browser_id, ...)` fact. It can be used similarly to the cookie storage or tab state fact described in Section 4.3, i.e., always present and generated newly whenever consumed. It can also be used similarly to the server storage described in Section 4.4, i.e., only present when needed. We do not recommend using the first option unless needed as it can reduce performance significantly.

4.1.4 HTTP Messages

Every *HTTP(S) message* in our model is accompanied by a mandatory `request_id` and the URL. The message content itself is independent of the HTTP model and the communication mechanism presented in Section 4.2. We distinguish between request messages and response messages. For our purposes, a request message consists of a cookie (which might be `'NULL'`) and a parameter field containing, e.g., user credentials.

```
request = <usersecret, cookie>
```

A response message consists also of a cookie (which might be `'NULL'`) and the requested Web page (which might include scripts or redirects).

```
response = <webpage, sent_cookie>
```

The parameter/Web page field represents all means of passing on information through HTTP, e.g., fragments, headers, and the body. We do not explicitly

model different HTTP methods like GET or POST but they could be added by introducing an additional field.

4.1.5 User Credentials

User credentials can have different forms. The most common one is a username and password (usersecret) combination. Our framework uses fresh values as usersecrets even though this might not always be the case in practice. To model a password guessing vulnerability, a public value could be used as usersecret instead of a fresh one. The framework does not include account creation, i.e., it assumes that all accounts have been set up in advance without attacker interference. The Web application should not store the usersecret unencrypted. We model this by hashing the usersecret. Note, however, that a malicious Web application can learn the unhashed usersecret if the user sends it to the Web application, e.g., when signing in.

We implement the generation of user credentials using the following rule. The fact `User_Credentials` is for the user to use while the Web application receives the hashed version of the usersecret `Hash_Credentials`.

```
/* User Credentials */

rule Gen_Cred:
    [ Fr(~usersecret) ]
  --[ Gen_Cred($Server, $Username) ]->
    [ !User_Credentials($Server, $Username, ~usersecret)
    , !Hash_Credentials($Server, $Username,
       h(~usersecret)) ]
```

For one Web application, no username may be used twice. In Tamarin, we can enforce this with the following restriction.

```
restriction no_duplicate_usernames:
"All s u #i #j. ((Gen_Cred(s, u) @i & Gen_Cred(s, u)
   @j) ==> #i = #j)"
```

4.1.6 Web Pages

A *Web page* is set up as follows. It consists of a URL, the Web page itself, and a security field. In Tamarin, we represented a Web page as a persistant fact with the according fields:

```
1 !Webpage(url, webpage, security)
```

The URL is as described in Section 4.1.1. The security field indicates how access to the Web page is protected. The Web page field consists itself of four fields.

- The first field indicates what kind of Web page it is, i.e., what kind of action it triggers in the browser. This may be, for example, a text only Web page, a redirect, a script, or a Web page containing third-party images. Depending on the use case, different categories are possible. Note that a Web page of a bank that transfers money between accounts is still considered to be of type `'NoAction'` because it does not cause the browser to execute any action.
- The second field is the URL of the Web page. For a user profile, we suggest using the `$File` field for the username.

The third and fourth field are only used if the Web page requires some sort of action of the browser besides loading it. Otherwise these fields are `'NULL'`.

- The third field indicates the target URL. This is, of course, needed for redirects but also for scripts sending requests by themselves.
- The fourth field contains any parameters the Web application needs to send along (except cookies as these are handled separately). For example, OpenID Connect requires sending an authentication request along with the redirect. This can be done using the fourth field.

In the end, there is no constraint on the first and fourth field as long as the corresponding browser and server logic is implemented accordingly. Adding new logic is designed to be as easy as possible, see Sections 4.3 and 4.4 for more details.

The security field of a Web page can be customised as well and also depends on the design of the server logic handling requests to the Web page. We suggest two kinds of Web page protection here. The first one is not protected at all (`security='none'`) and represents a freely accessible Web page The second one is protected by a usersecret (`security = h(usersecret)`) and represents a Web page that requires to know the usersecret to access the user's profile.

In the following, we give some examples how various kinds of Web pages in out framework can be implemented in Tamarin. The first example `Setup_Webpage` represents any freely accessible Web page that does not trigger any action in the browser.

```
rule Setup_Webpage:
let url = <$Protocol, $Server, $Path, $File>
    webpage = <'NoAction', url, 'NULL', 'NULL'>
    security = 'none'
in
    [  ]
  --[ Protocol($Protocol)
    , Setup_Webpage(url) ]->
    [ !Webpage(url, webpage, security) ]
```

The second example `Setup_Webpage_Requiring_Login` shows how a Web page protected by a usersecret can be set up. For information about how the server can check the login credentials, see Section 4.4.

```
rule Setup_Webpage_Requiring_Login:
let url = <$Protocol, $Server, $Path, $Username>
    webpage = <'NoAction', url, 'NULL', 'NULL'>
    security = h(usersecret)
in
    [ !Hash_Credentials($Server, $Username,
       h(usersecret)) ]
  --[ Protocol($Protocol)
    , Setup_Webpage(url) ]->
    [ !Webpage(url, webpage, security) ]
```

The third example `Setup_Webpage_Redirect` demonstrates a redirect. The Web page contains `'Redirect'` as a action and specifies a target URL where the browser should be redirected to. Note that a Web page triggering a redirect can, of course, also be protected by a usersecret. In this case, the security of the Web page can be set accordingly, see example above.

```
rule Setup_Webpage_Redirect:
let url = <$Protocol, $Server, $Path, $File>
    target_url = <$Target_Protocol, $Target_Server,
       $Target_Path, $Target_File>
    webpage = <'Redirect', url, target_url, 'NULL'>
    security = 'none'
in
    [  ]
  --[ Protocol($Protocol)
    , Protocol($Target_Protocol)
    , Setup_Webpage(url) ]->
    [ !Webpage(url, webpage, security) ]
```

If the use case at hand requires a representation of non-existing Web pages, we can represent them in Tamarin using the following rule.

```
rule Setup_Webpage_Nonexisting:
let url = <$Protocol, $Server, $Path, $File>
    webpage = <'NotFound', 'NULL', 'NULL', 'NULL'>
    security = 'none'
in
    [ ]
  --[ Setup_Webpage(url)
    , Protocol($Protocol) ]->
    [ !Webpage(url, webpage, security) ]
```

4.2 Browser-Server Communication

We model two ways of communication between browser and server, HTTP and HTTPS. The message format is the same with the difference that a HTTPS message is encrypted. We propose three different models of a HTTPS connection. One of them with explicit encryption and one using a bidirectional channel. In addition, we show how a TLS connection with multiple messages per connection can be modelled if needed.

In Figure 3, a detailed diagram of the model is shown containing the third proposed model of a HTTPS connection. On both the browser-side and the server-side there is an intermediate layer (called channel handlers in the diagram) handling the different options to send and receive a message. Neither the browser nor the server logic needs to handle the difference between HTTP and HTTPS. In both cases, the same fact `Browser_Send(...)`

or `Server_Send(...)` respectively can be used to send a message. The intermediate layer will then send the message according to the protocol indicated in the URL. This adds some complexity to the model and an additional rule is needed for every message sent. However, the benefits of modularity and expandability outweigh the added complexity. As the rules only perform a 1:1 mapping of facts, no significant performance loss is expected.

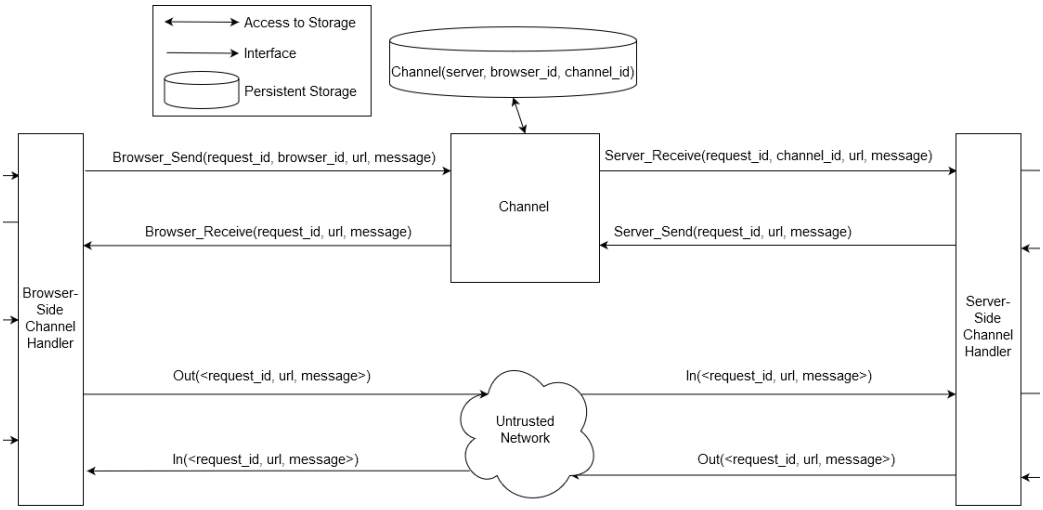

Figure 3: Diagram of Browser-Server Communication

4.2.1 HTTP

Sending a message using HTTP is modelled by simply sending the message in to the untrusted network. Anyone can receive the message and anyone can send a message including the built-in network attacker. In Tamarin, two rules are needed to model this kind of communication: one rule allowing the sender to send the message and one rule allowing the receiver to receive the message. In the following, the rules needed to send a message from the browser to the server are shown. Note that the `browser_id` included by the browser but discarded by the rule `Browser_Send_Over_Http`. This is because the `browser_id` is needed for modelling channels rather than independent messages. The `browser_id` is included to keep interfaces consistent as we need channels for modelling HTTPS (see Section 4.2.2 for more details).

```
/* HTTP */

rule Browser_Send_Over_Http:
let url = <'HTTP', $Server, $Path, $File>
in
    [ Browser_Send(request_id, browser_id, url,
       message) ]
  -->
    [ Out(<request_id, url, message>) ]

rule Server_Receive_Over_Http:
    [ In(<request_id, url, message>) ]
  -->
    [ Server_Receive(request_id, url, message) ]
```

Below one can also find the Tamarin multiset rewriting (MSR) rules for the other direction, i.e., a message sent by the server to the browser. Note that the MSR rules `Browser_Send_Over_Http` and `Browser_Receive_Over_Http` are part of the browser-side channel handler as shown in Figure 3. Respectively, the MSR rules `Server_Send_Over_Http` and `Server_Receive_Over_Http` are part of the server-side channel handler.

```
/* HTTP */

rule Server_Send_Over_Http:
let url = <'HTTP', $Server, $Path, $File>
in
    [ Server_Send(request_id, url, message) ]
  -->
    [ Out(<request_id, url, message>) ]

rule Browser_Receive_Over_Http:
    [ In(<request_id, url, message>) ]
  -->
    [ Browser_Receive(request_id, url, message) ]
```

4.2.2 HTTPS

In the following, we present different models of HTTPS.

HTTPS V1 The first model for HTTPS we developed is based on the proposal by Fett in [16]. It is the same as HTTP but additionally the intermediate communication layers (channel handlers) encrypt/decrypt all outgoing/incoming messages. Requests are encrypted on the client-side using the server's public key. The client-side also generates and sends a fresh symmetric key. This symmetric key is then used by the server-side to encrypt the response. This requires both the client-side and the server-side to remember the key.

For this model of HTTPS, public-key infrastructure (PKI) is needed. PKI is modelled by creating a fresh long-term key for the server. The long-term key is kept secret while the corresponding public key is made public by sending it to the untrusted network. In Tamarin, this can be added as follows. The fact `!Ltk` represents the long-term key of the `$Server`.

```
rule Register_PK:
    [ Fr(~ltk) ]
  -->
    [ !Ltk($Server, ~ltk), !Pk($Server, pk(~ltk)),
      Out(pk(~ltk)) ]
```

Like in the model of HTTP, the messages are sent over the untrusted network where anyone can send and receive messages. This means that, like before, two MSR rules are needed in Tamarin to transmit a message. Below you can find the rules to send an encrypted message from the browser to the server. Note that these rules are part of the browser-side and server-side channel handlers respectively. Both sides store the fresh symmetric key using a linear fact `Client_side_key` and `Server_side_key` respectively. They do not use a shared storage to allow for attacks where an attacker can trick the server into storing a different key from the browser.

```
/* HTTPS V1 */

rule Browser_Send_Over_Https:
let url = <'HTTPS', $Server, $Path, $File>
    aenc_req= aenc(<request_id, url, message, ~symKey>,
       pkServer)
in
    [ Browser_Send(request_id, browser_id, url, message)
    , !Pk($Server, pkServer)
    , Fr(~symKey) ]
  -->
```

```
    [ Out(aenc_req)
    , Client_side_key(request_id, ~symKey) ]

rule Server_Receive_Over_Https:
  let aenc_req=aenc(<request_id, url, message, symKey>,
      pk(skServer))
  in
    [ In(aenc_req)
    , !Ltk(Server, skServer) ]
  -->
    [ Server_Receive(request_id, url, message)
    , Server_side_key(request_id, symKey) ]
```

The rules to send a message from the server to the browser are analogous to the other direction. Instead of the server's public-key, the fresh symmetric key is used for encryption. These rules are also part of the browser-side and server-side channel handlers respectively.

```
/* HTTPS V1 */

rule Server_Send_Over_Https:
  let url = <'HTTPS', $Server, $Path, $File>
      senc_res=senc(<request_id, url, message>, symKey)
  in
    [ Server_Send(request_id, url, message)
    , Server_side_key(request_id, symKey) ]
  -->
    [ Out( senc_res ) ]

rule Browser_Receive_Over_Https:
  let senc_res=senc(<request_id, url,  message>, symKey)
  in
    [ In( senc_res )
    , Client_side_key(request_id, symKey) ]
  -->
    [ Browser_Receive(request_id, url, message) ]
```

HTTPS V2 The above model of HTTPS, however, led to serious performance issues and introduced a lot of partial deconstructions. The sources lemmas removing these partial deconstructions often had to be proven man-

ually or with an oracle. Every slight change in the remaining model required a lot of additional work and sometimes even a new oracle. More often than not, lemmas did not terminate.

We, thus, switched to a model using channels. A channel is modelled by directly passing the message from the sender to the receiver instead of sending it into the untrusted network. This means that only one MSR rule is needed in Tamarin to transmit a message from the browser to the server as shown below.

```
/* HTTPS V2 */

rule Browser_To_Server_Https:
let url = <'HTTPS', $Server, $Path, $File>
in
    [ Browser_Send(request_id, browser_id, url,
       message) ]
  -->
    [ Server_Receive(request_id, url, message) ]
```

The other direction, server to browser, is analogous. Note that the interfaces `Browser_Send`, `Server_Send`, `Browser_Receive`, and `Server_Receive` are the same as before.

```
/* HTTPS V2 */

rule Server_To_Browser_Https:
let url = <'HTTPS', $Server, $Path, $File>
in
    [ Server_Send(request_id, url, message) ]
  -->
    [ Browser_Receive(request_id, url, message) ]
```

The above two rules do not allow for any attacker interference and, thus, model a secure channel in both directions. This significantly weakens the attacker model. Mainly, the network attacker lost their ability to talk to the server. Previously, a network attacker could communicate with the server like any other user, i.e., send requests to the server and receive the corresponding response. Here, we need to give the attacker these capabilities explicitly by implementing the corresponding MSR rules. To ensure that the network attacker can only receive responses to request that they themselves issued, we introduced the `Malicious_Session` fact. Note that the network

attacker can still use HTTP to send messages to both the browser and server if HTTP is included in the model. The corresponding rules can be found below.

```
/* HTTPS V2 */

rule Attacker_Sends_Request:
    [ In(<request_id, url, message>) ]
  --[ Network_Attacker() ]->
    [ Server_Receive(request_id, url, message)
    , Malicious_Session(request_id) ]

rule Attacker_Receives_Response:
    [ Malicious_Session(request_id)
    , Server_Send(request_id, url, message) ]
  --[ Network_Attacker() ]->
    [ Out(<request_id, url, message>) ]
```

HTTPS V3 The two models shown above (HTTPS V1 and HTTPS V2) fail to capture one important property of TLS. Namely, that a TLS channel stays open long enough for more than one message to be sent over it. The models above do not allow to send multiple messages over the same TLS channel. But sending multiple messages is an important feature for many protocols including OpenID Connect. To model OpenID Connect, for example, the IdP needs a mechanism to know when two messages are received over the same channel. Thus, we introduced a new rule to set up a channel between a client and a server as well as a channel identifier. Note that a browser can have multiple channels with the same server without the server knowing. The server just gets the guarantee that messages with the same channel identifier originate from the same client. Note that this slightly changes the `Server_Receive` interface which now includes a `channel_id`.

We modelled the creation of a channel between the browser and the server with the following Tamarin MSR rule.

```
/* HTTPS V3 */

rule Setup_Browser_Server_Channel:
    [ !Browser(browser_id)
    , Fr(~channel_id) ]
  -->
    [ !Channel($Server, browser_id, ~channel_id) ]
```

This `!Channel` fact is then used by the browser-side channel handler to attach the `channel_id` to the outgoing message. Otherwise, the MSC rules to transmit messages between the browser and server are identical to the previous version and can be found below.

```
/* HTTPS V3 */

rule Browser_To_Server_Https:
let url = <'HTTPS', $Server, $Path, $File>
in
    [ Browser_Send(request_id, browser_id, url, message)
    , !Channel($Server, browser_id, channel_id) ]
  -->
    [ Server_Receive(request_id, channel_id, url,
       message) ]

rule Server_To_Browser_Https:
let url = <'HTTPS', $Server, $Path, $File>
in
    [Server_Send(request_id, url, message) ]
  -->
    [Browser_Receive(request_id, url, message) ]
```

Like before, we need to give the attacker the appropriate capabilities to communicate with the server and send messages to the browser. This is done by introducing a `!Malicious_Channel` fact analogous to the honest `!Channel` fact.

```
/* HTTPS V3 */

rule Setup_Attacker_Server_Channel:
    [ Fr(~channel_id) ]
  --[ Network_Attacker() ]->
    [ !Malicious_Channel($Server, ~channel_id) ]
```

```
7
8  rule Attacker_Sends_Request:
9  let url = <'HTTPS', $Server, $Path, $File>
10 in
11     [ In(<request_id, url, message>)
12     , !Malicious_Channel($Server, channel_id) ]
13   --[ Network_Attacker() ]->
14     [ Server_Receive(request_id, channel_id, url,
          message)
15     , Malicious_Session(request_id) ]
16
17 rule Attacker_Receives_Response:
18 let url = <'HTTPS', $Server, $Path, $File>
19 in
20     [ Malicious_Session(request_id)
21     , Server_Send(request_id, url, message) ]
22   --[ Network_Attacker() ]->
23     [ Out(<request_id, url, message>) ]
```

4.3 Browser and User

A *browser* in our model is created by assigning it a fresh browser identifier and is represented in Tamarin by a persistent fact `!Browser`.

```
1 rule Create_Browser:
2     [ Fr(~browser_id) ]
3   --[ Create_Browser(~browser_id) ]->
4     [ !Browser(~browser_id) ]
```

A browser has initially no tabs. Unlike a browser, we model a tab using a linear fact `Tab_State` instead of a persistent fact. This allows for the tab to change, e.g., a user can request a new Web page in a tab that already displays another Web page. The old Web page is removed when the newly requested one is loaded. Each tab is associated with a browser through its browser identifier. A tab consists of a tab identifier, the corresponding browser identifier, the currently loaded Web page, and the request identifier of the currently pending request. The currently pending request is used to model that the browser will only accept responses it is expecting. This also allows to user to abort requests by resetting the pending request field to `'NULL'` (see Section 4.3.3). Initialisation of a browser and tab is shown

below. Initially, no Web page is loaded which is modelled by setting all fields of the Web page to 'NULL'. The same goes for the pending request.

```
rule Create_Tab:
let loaded_webpage = <'NULL', 'NULL', 'NULL', 'NULL'>
    pending_request = 'NULL'
in
    [ Fr(~tab_id)
    , !Browser(browser_id) ]
  --[ Create_Tab(~tab_id, browser_id) ]->
    [ Tab_State(~tab_id, browser_id, loaded_webpage,
       pending_request) ]
```

The browser's logic is split into three parts: the *response and redirect handlers*, the *content handler*, and the *user handler*. The response and redirect handler are responsible for handling all incoming responses. Responses containing a redirect are handled by the redirect handler while regular responses are handled by the response handler (see Section 4.3.1 for more details). The content handler is responsible for reacting to the currently loaded Web page. This includes, e.g., executing scripts or loading third-party images. The content handler is described in more detail in Section 4.3.2. The user handler models the behaviour of the user by, e.g., non-deterministically requesting Web sites. We describe the user handler in Section 4.3.3. We implemented each handler with a set of rules that are described in the respective subsections. For a conceptual diagram of the browser and user model, see Figure 4.

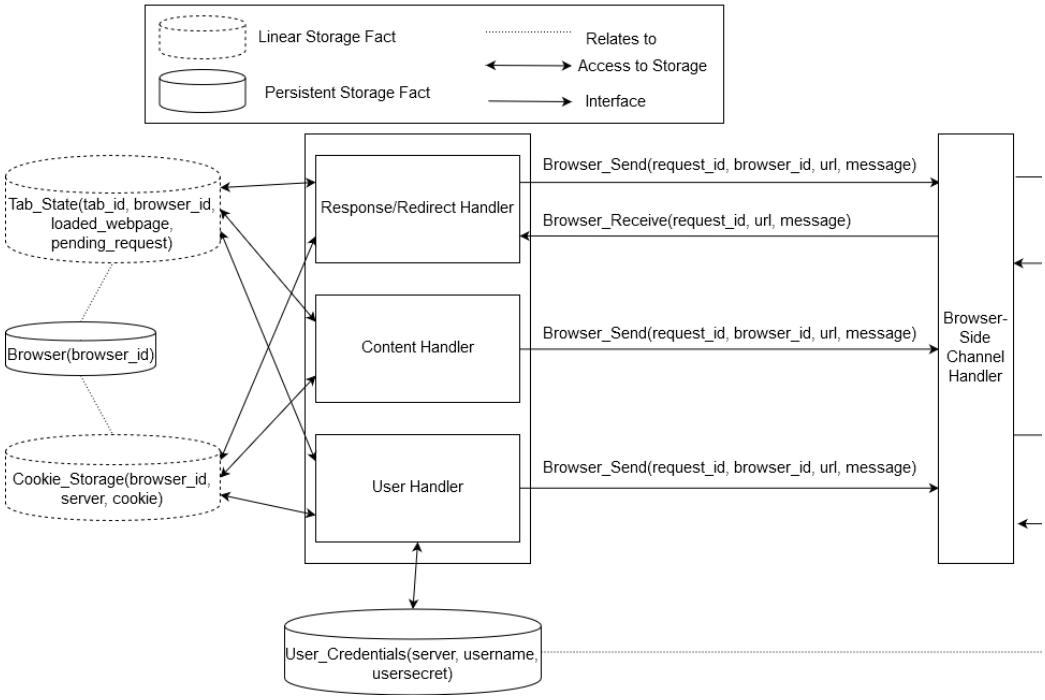

Figure 4: Diagram of the Browser and User Model

4.3.1 Response Handler and Redirect Handler

The response handler and the redirect handler are similar in many ways and, thus, grouped together. Both are responsible for handling incoming responses and both update the currently loaded Web page as well as the cookie if needed. They mainly differ in the type of response they handle. The response handler handles any response that does not contain a redirect. It resets the pending request to `'NULL'` because no request is curretnly pending until a new one is requested. The redirect handler on the other hand handles all responses that contain a redirect. Thus, it does not set the pending request to `'NULL'` but requests the Web page the browser is redirected to and updates the pending request field accordingly with the new request identifier. The redirect handler immediately sends the new request, i.e., the currently loaded page will not be updated and nothing, neither the user nor an attacker, can prevent the redirect request from being sent once the redirect is received.

Below you can find the response handler implemented as Tamarin MSR rules. Two rules are necessary in order to distinguish between responses

containing a cookie and responses without cookie (`cookie = 'NULL'`). Only if the response contains a cookie, the cookie storage should be updated. If we fail to make this distinction, then the cookie in the cookie storage is deleted every time a response from its domain is received.

```
/* Response Handler*/

rule Response_Handler_NoCookie:
let cookie = 'NULL'
    webpage = <webpage_type, webpage_url,
       webpage_target_url, webpage_action>
    response = <webpage, cookie>
in
    [ Browser_Receive(old_pending_request, url,
       response)
    , Tab_State(tab_id, browser_id, old_webpage,
       old_pending_request) ]
  --[ Not_Redirect(webpage_type)
    , Response_Handler_NoCookie(old_pending_request) ]->
    [ Tab_State(tab_id, browser_id, webpage, 'NULL') ]

rule Response_Handler_Cookie:
let cookie = <cookiename, cookievalue, <'HTTPonly',
   httponly>, <'Secure', secure>>
    webpage_url = <$Webpage_Protocol, $Webpage_Server,
       $Webpage_Path, $Webpage_File>
    webpage = <webpage_type, webpage_url,
       webpage_target_url, webpage_action>
    response = <webpage, cookie>
in
    [ Browser_Receive(old_pending_request, url,
       response)
    , Tab_State(tab_id, browser_id, old_webpage,
       old_pending_request)
    , Cookie_Storage(browser_id, $Webpage_Server,
       old_cookie) ]
  --[ Protocol($Webpage_Protocol)
    , Response_Handler_Cookie(old_pending_request)
    , Not_Redirect(webpage_type) ]->
    [ Tab_State(tab_id, browser_id, webpage, 'NULL')
```

```
    , Cookie_Storage(browser_id, $Webpage_Server,
      cookie) ]
```

To ensure that the response handler does not handle any responses containing a redirect, we include the following restriction into the Tamarin theory.

```
restriction redirect_handling:
"All webpage_type #i. Not_Redirect(webpage_type) @i ==>
   not(webpage_type = 'Redirect')"
```

Below you can find the redirect handler implemented as Tamarin MSR rules. Again, we need to make the distinction between a response with and without a cookie. The redirect handler immediately triggers a request to the Web page indicated in the received response. This also involves attaching the corresponding cookie to the outgoing request.

```
/* Redirect Handler */

rule Redirect_Handler_NoCookie:
let received_cookie = 'NULL'
    webpage_target_url = <$Target_Protocol,
      $Target_Server, $Target_Path, $Target_File>
    webpage = <'Redirect', webpage_url,
      webpage_target_url, webpage_parameters>
    response = <webpage, received_cookie>
    new_pending_request = ~new_request_id
    request = <webpage_parameters, sent_cookie>
in
    [ Browser_Receive(old_pending_request, url,
      response)
    , Tab_State(tab_id, browser_id, old_webpage,
      old_pending_request)
    , Cookie_Storage(browser_id, $Target_Server,
      sent_cookie)
    , Fr(~new_request_id) ]
  --[ Protocol($Target_Protocol)
    , Cookie_Is_Sent(webpage_target_url, sent_cookie)
    , Redirect_Handler_NoCookie(old_pending_request,
      new_pending_request)
    , Request_Origin(new_pending_request,
      webpage_target_url, 'Redirect') ]->
```

```
19      [ Browser_Send(new_pending_request, browser_id,
          webpage_target_url, request)
20      , Tab_State(tab_id, browser_id, old_webpage,
          new_pending_request)
21      , Cookie_Storage(browser_id, $Target_Server,
          sent_cookie) ]
22
23  rule Redirect_Handler_Cookie:
24  let received_cookie = <cookiename, cookievalue,
      <'HTTPonly', httponly>, <'Secure', secure>>
25      webpage_url = <$Webpage_Protocol, $Webpage_Server,
          $Webpage_Path, $Webpage_File>
26      webpage_target_url = <$Target_Protocol,
          $Target_Server, $Target_Path, $Target_File>
27      webpage = <'Redirect', webpage_url,
          webpage_target_url, webpage_parameters>
28      response = <webpage, received_cookie>
29      new_pending_request = ~new_request_id
30      request = <webpage_parameters, sent_cookie>
31  in
32      [ Browser_Receive(old_pending_request, url,
          response)
33      , Tab_State(tab_id, browser_id, old_webpage,
          old_pending_request)
34      , Cookie_Storage(browser_id, $Webpage_Server,
          old_cookie)
35      , Cookie_Storage(browser_id, $Target_Server,
          sent_cookie)
36      , Fr(~new_request_id) ]
37    --[ Protocol($Webpage_Protocol)
38      , Protocol($Target_Protocol)
39      , Cookie_Is_Sent(webpage_target_url, sent_cookie)
40      , Redirect_Handler_Cookie(old_pending_request,
          new_pending_request)
41      , Request_Origin(new_pending_request,
          webpage_target_url, 'Redirect') ]->
42      [ Browser_Send(new_pending_request, browser_id,
          webpage_target_url, request)
43      , Tab_State(tab_id, browser_id, old_webpage,
          new_pending_request)
```

```
44      , Cookie_Storage(browser_id, $Webpage_Server,
            received_cookie)
45      , Cookie_Storage(browser_id, $Target_Server,
            sent_cookie) ]
```

4.3.2 Content Handler

The content handler takes the currently loaded Web page and executes its action. This is the part that is most likely to be custom to the use case at hand while the response and redirect handlers can often be used as it is. Below you can find a content handler that executes scripts triggering a request to a given Web page. It does not update the pending request field in order to ignore the response. This can also be changed if the response should be accepted by the browser. To add additional logic to the content handler, one can simply add a new rule. Through this extensibility, arbitrary scripts could be executed. The content handler is not limited to scripts however. The executed action could also be, for example, loading images or other third party content.

```
1  /* Content Handler */
2
3  rule Content_Handler_TriggerRequest:
4  let target_url = <$Target_Protocol, $Target_Server,
       $Target_Path, $Target_File>
5       loaded_webpage = <'TriggerRequest', url,
            target_url, 'NULL'>
6       new_pending_request = ~request_id
7       request = <'NULL', cookie>
8  in
9       [ Tab_State(tab_id, browser_id, loaded_webpage,
            old_pending_request)
10      , Cookie_Storage(browser_id, $Target_Server, cookie)
11      , Fr(~request_id) ]
12    --[ Protocol($Target_Protocol)
13      , Cookie_Is_Sent(target_url, cookie)
14      , Request_Origin(new_pending_request, url,
            'TriggerRequest') ]->
15      [ Browser_Send(new_pending_request, browser_id,
            target_url, request)
```

```
16      , Tab_State(tab_id, browser_id, loaded_webpage,
           old_pending_request)
17      , Cookie_Storage(browser_id, $Target_Server,
           cookie) ]
```

4.3.3 User Handler

The user handler represents all user actions like requesting a webpage, signing in using their credentials, or pressing a button. Below we give examples of how such user behaviours can be implemented in Tamarin.

We model a user requesting a Web page (e.g., by typing an URL into the URL bar) as the browser sending a request to the server. To distinguish a request made by the user from requests triggered otherwise, we annotate the MSR rule with the action fact `Request_Origin(new_pending_request, url, 'UserRequest')`.

```
1  /* User Handler */
2
3  rule User_Handler_Request:
4  let url = <$Protocol, $Server, $Path, $File>
5      new_pending_request = ~request_id
6      request = <'NULL', cookie>
7  in
8      [ Tab_State(tab_id, browser_id, loaded_webpage,
           old_pending_request)
9      , Cookie_Storage(browser_id, $Server, cookie)
10     , Fr(~request_id) ]
11   --[ Protocol($Protocol)
12     , Cookie_Is_Sent(url, cookie)
13     , User_Handler_Request()
14     , Request_Origin(new_pending_request, url,
           'UserRequest') ]->
15     [ Browser_Send(new_pending_request, browser_id,
           url, request)
16     , Tab_State(tab_id, browser_id, loaded_webpage,
           new_pending_request)
17     , Cookie_Storage(browser_id, $Server, cookie) ]
```

The MSR rule representing a user who sign in to a Web site can be found below. Signing in to a Web site is modelled by the user sending their usersecret as part of the request.

```
/* User Handler */

rule User_Handler_Login:
let url = <$Protocol, $Server, $Path, $Username>
    new_pending_request = ~request_id
    request = <usersecret, cookie>
in
    [ !User_Credentials($Server, $Username, usersecret)
    , Tab_State(tab_id, browser_id, loaded_webpage,
       old_pending_request)
    , Cookie_Storage(browser_id, $Server, cookie)
    , Fr(~request_id) ]
  --[ Protocol($Protocol)
    , User_Logs_In()
    , Cookie_Is_Sent(url, cookie)
    , User_Handler_Login()
    , Request_Origin(new_pending_request, url,
       'UserLogin') ]->
    [ Browser_Send(new_pending_request, browser_id,
       url, request)
    , Tab_State(tab_id, browser_id, loaded_webpage,
       new_pending_request)
    , Cookie_Storage(browser_id, $Server, cookie) ]
```

A user can also abort a request. We model this using the MSR rule below that resets the pending request field to `'NULL'`. This does not stop the server (or attacker) from receiving the request but ensures that the browser ignores the response.

```
/* User Handler */

rule User_Handler_Abort_Request:
    [ Tab_State(tab_id, browser_id, loaded_webpage,
       pending_request) ]
  --[  ]->
    [ Tab_State(tab_id, browser_id, loaded_webpage,
       'NULL') ]
```

Similarly, the user can delete the stored cookie for a domain by setting the cookie in the respective cookie storage to `'NULL'`. We do not allow to cookie storage itself to be deleted as its existence is assumed by the rest of the framework.

```
/* User Handler */

rule User_Handler_Delete_Cookie:
    [ Cookie_Storage(browser_id, $Server, cookie) ]
  --[ ]->
    [ Cookie_Storage(browser_id, $Server, 'NULL')]
```

A user can close a tab in a browser. We model this by introducing the following MSR rule in Tamarin. The rule consumes the `Tab_State` fact without generating a new one. Note that browsers cannot be closed in our framework as they are modelled as persistent Tamarin facts. A browser can, however, have no tabs.

```
/* User Handler */

rule User_Handler_Close_Tab:
    [ Tab_State(tab_id, browser_id, loaded_webpage,
       new_pending_request) ]
  --[ ]->
    [ ]
```

Additional user behaviour can be added by adding the corresponding rules. For example, OpenID Connect requires the user to choose an identity provider. We, thus, added this possible user action as a new rule to the OpenID Connect model in Section 5.4.

4.4 Server and Web Applications

The servers and Web applications are open for customisation depending on the Web application one wants to model. The only constraints are the interfaces for sending and receiving messages. In Figure 5, a diagram of the server-side model can be found. Below, the server logic for the Web pages presented in Section 4.1.6 implemented as Tamarin MSR rules is shown.

In the first rule `Server_Receives_Request`, the server received a request to an unprotected Web page. The server simply responds with the content of the Web page. This rule is applicable to every unprotected Web page including Web pages triggering a redirect.

```
/* Server */

rule Server_Receives_Request:
let security = 'none'
    sent_cookie = 'NULL'
in
    [ Server_Receive(request_id, url, <'NULL',
       received_cookie>)
    , !Webpage(url, webpage, security) ]
  --[  ]->
    [ Server_Send(request_id, url, <webpage,
       sent_cookie>) ]
```

The second rule `Server_Receives_Request_With_Credentials` represents how a server can verify access to a protected Web page and log a user in by creating a login cookie. Verification is done by pattern matching between the stored hash and the received usersecret. If authentication is successful, the server responds with the Web page's content and a login cookie. The cookie contains a fresh identifier and both available security flags are set. The server then stores the fresh value of the cookie together with the associated username in a the linear `Server_Storage` fact.

```
/* Server */

rule Server_Receives_Request_With_Credentials:
let url = <$Protocol, $Server, $Path, $Username>
    security = h(usersecret)
    sent_cookie = <'login_id', ~cookievalue,
       <'HTTPonly', 'true'>, <'Secure', 'true'>>
    response = <webpage, sent_cookie>
in
    [ Server_Receive(request_id, url, <usersecret,
       'NULL'>)
    , !Hash_Credentials($Server, $Username,
       h(usersecret))
    , !Webpage(url, webpage, security)
    , Fr(~cookievalue) ]
  --[ Protocol($Protocol)
    , Protected_Webpage_Accessed(request_id, url) ]->
    [ Server_Send(request_id, url, response)
    , Server_Storage($Username, ~cookievalue) ]
```

The server does also need to be able to verify access to a protected Web page when a previously issued login cookie is used. The third rule `Server_Receives_Request_With_Login_Cookie` provides this functionality. The server matches the received cookie against its storage and, if successful, responds with the requested Web page.

```
/* Server */

rule Server_Receives_Request_With_Login_Cookie:
let url = <$Protocol, $Server, $Path, $Username>
    cookie = <'login_id', cookievalue, <'HTTPOnly',
       httponly>, <'Secure', secure>>
    security = hashed_usersecret
    sent_cookie = 'NULL'
    response = <webpage, 'NULL'>
in
    [ Server_Receive(request_id, url, <'NULL', cookie>)
    , !Hash_Credentials($Server, $Username,
       hashed_usersecret)
    , !Webpage(url, webpage, security)
    , Server_Storage($Username, cookievalue) ]
  --[ Protocol($Protocol)
    , Protected_Webpage_Accessed(request_id, url) ]->
    [ Server_Send(request_id, url, response)
    , Server_Storage($Username, cookievalue) ]
```

To model more specialised server behaviour, the logic of the Web application may be also directly implemented without setting up a Web page first. The relying party and identity provider of the OpenID Connect protocol, for example, we implemented as a set of specialised rules solely for that purpose.

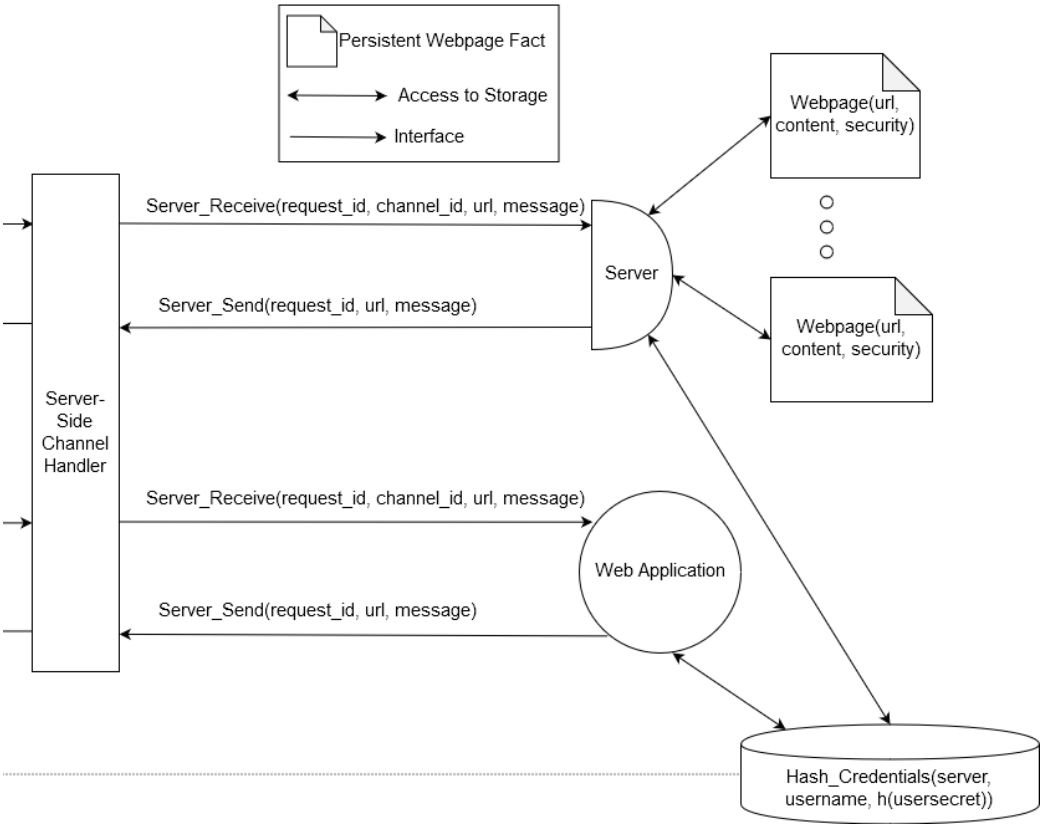

Figure 5: Diagram of the Server-Side Model

4.5 Attacker

Recall that Tamarin has a built-in network attacker. The network attacker can read any message sent to the untrusted network and send any message that they can compose from the current attacker knowledge. This is also true for encrypted messages. The attacker cannot, however, break encryption and, thus, only learns the encrypted message. When using channels as communication medium, the attacker needs to be given the appropriate capabilities, see Section 4.2. For our purposes, this especially means that the attacker can, like any honest user, have a session with the server. The attacker can send requests to the server and receive the corresponding response.

To model attackers other than a network attacker or to make the network attacker stronger, explicit rules must be in place. The first rule below, for

example, allows a network attacker to gain access to the private key of a Web server in models that use public-key infrastructure. This allows the attacker, besides other things, to set up a (malicious) Web page themselves.

```
/* Attacker */

rule Reveal_SK:
    [ !Ltk(server, ltk) ]
  --[ Reveal(server) ]->
    [ Out(ltk) ]
```

In our framework, we use explicit Web attackers rather than more powerful network attackers to set up malicious Web pages. In Tamarin, we implement this using the rule below. This models a Web with malicious Web sites present. Other kinds of malicious Web pages with other content can also be set up.

```
/* Setup Webpages */

rule Setup_Webpage_Malicious:
let url = <$Protocol, $Server, $Path, $File>
    target_url = <$Target_Protocol, $Target_Server,
       $Target_Path, $Target_File>
    webpage = <'TriggerRequest', url, target_url,
       'NULL'>
    security = 'none'
in
    [  ]
  --[ Protocol($Protocol)
    , Setup_Webpage(url) ]->
    [ !Webpage(url, webpage, security) ]
```

The MSR rule below allows the attacker to inject code into an honest Web page, modelling a Web page with XSS vulnerabilities. It does that by changing the Web pages action triggered in the browser.

```
/* Attacker */

rule Inject_Code:
let url = <$Protocol, $Server, $Path, $File>
    webpage = <'NoAction', url, 'NULL', 'NULL'>
    security = h(usersecret)
in
```

```
8       [ !Webpage(url, webpage, security) ]
9     --[ Inject_Code(url) ]->
10      [ !Webpage(url, <'TriggerRequest', url, url,
          'NULL'>, security) ]
```

By implementing the corresponding MSR rule, any threat model can be implemented.

5 Case Studies

In the previous section, we described how different elements of the Web infrastructure can be modelled. This gave us a toolbox that can be used to model various Web applications and protocols. In this section, we show the usability of this toolbox. We show that it is suitable to implement various attacker models and find attacks on the Web infrastructure within these attacker models. Note that, at the moment, the models are very specific and ill-suited to find unknown attacks on Web applications. They are a proof of concept that these kind of attacks can be found in models constructed with the developed toolbox. To model and verify Web applications a suitable set of functionalities needs to be included in the model, e.g., other Web sites, redirects, cookie policies, etc.

We also show how a protocol can be implemented with this toolbox by modelling the OpenID Connect protocol described in Section 2.3. All theory and proof files can be found at [9].

5.1 Cross-Site Scripting and Request Forgery Attack

The attack we want to demonstrate here is a request forgery attack exploiting a cross-site scripting vulnerability. This means that the attacker is able to run malicious code on the client-side that sends a request to the server looking like a legitimate user request, i.e., that forges a user request. We model an attacker that is capable of exploiting a script injection vulnerability in the Web page as described in Section 2.2.1. We explicitly give this capability to the attacker by using the below rule `Inject_Code` presented in Section 4.5.

```
1 rule Inject_Code:
2 let url = <$Protocol, $Server, $Path, $File>
3     webpage = <'NoAction', url, 'NULL', 'NULL'>
4     security = h(usersecret)
5 in
6     [ !Webpage(url, webpage, security) ]
7   --[ Inject_Code(url) ]->
8     [ !Webpage(url, <'TriggerRequest', url, url,
        'NULL'>, security) ]
```

This rule takes the vulnerable Web page and changes its action in the browser, i.e., attacker-controlled code will be executed by the browser in the honest Web page's context. In this case, the attacker changes the action from

'NoAction' to 'TriggerRequest'. Once the page is loaded in the browser, the content handler non-deterministically runs the injected code using the below rule presented in Section 4.3.2.

```
rule Content_Handler_TriggerRequest:
let target_url = <$Target_Protocol, $Target_Server,
    $Target_Path, $Target_File>
     loaded_webpage = <'TriggerRequest', url,
        target_url, 'NULL'>
     new_pending_request = ~request_id
     request = <'NULL', cookie>
in
     [ Tab_State(tab_id, browser_id, loaded_webpage,
        old_pending_request)
     , Cookie_Storage(browser_id, $Target_Server, cookie)
     , Fr(~request_id) ]
   --[ Protocol($Target_Protocol)
     , Cookie_Is_Sent(target_url, cookie)
     , Request_Origin(new_pending_request, url,
        'TriggerRequest') ]->
     [ Browser_Send(new_pending_request, browser_id,
        target_url, request)
     , Tab_State(tab_id, browser_id, loaded_webpage,
        old_pending_request)
     , Cookie_Storage(browser_id, $Target_Server,
        cookie) ]
```

This sends one or more requests to the honest Web server without the user knowing. If the user has logged in to the Web page earlier and the login cookie has not been deleted, the forged request will succeed in potentially changing the server state (e.g., changing profile information or transferring money). The user does not notice these outgoing requests because the malicious code will discard any responses (modelled by not updating the `pending_request` in the tab state).

To model a request forgery attack on a Web page with a cross-site scripting vulnerability, we used the following elements from the toolbox:

- The user handler with the capability of requesting a Web page and signing in to a Web page.
- The content handler with the capability of executing the malicious code

- The response handler. Note that the redirect handler is not needed to demonstrate this attack.
- The honest Web page. Requests to this Web page will change the server state but trigger no action in the browser. This is a placeholder for any state changing request like updating profile information, transferring money, or posting in a forum.
- Basic browser infrastructure that is always needed, i.e., creating a browser, creating a tab, and creating cookie storage.
- Generation and distribution of user credentials.
- Attacker capability to inject arbitrary code into an existing, honest Web site. In this case, the code triggers a state changing request to the same honest Web site.
- Server functionality that can handle requests, authenticate users, and issue login cookies.
- A communication medium, HTTPS using confidential channels in this case.

The property we want to verify is that, in the absence of a network attacker, whenever a Web page protected by a login is accessed then the origin of that request is the user. The corresponding lemma `Honest_Request_Origin` can be found below. Note that in the final theory, we included some additional constraints in the lemma for two reasons. The first reason is performance and the second one is to steer Tamarin towards finding a concise attack. Using these constraints, Tamarin finds an attack instantly. In Figure 6, you can find the message sequence chart of the attack found by Tamarin.

```
/* Lemmas */

lemma Honest_Request_Origin:
"
not (Ex #i. Network_Attacker() @i)
==>
(All r u #i. Protected_Webpage_Accessed(r, u) @i
==>
((Ex #j. Request_Origin(r, u, 'UserLogin') @j)
```

```
10  | (Ex #j. Request_Origin(r, u, 'UserRequest') @j) ))
11 "
```

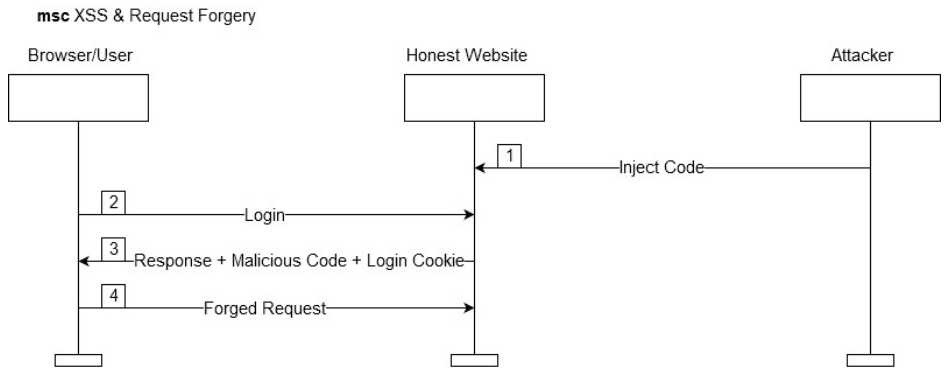

Figure 6: Diagram of the Cross-Site Scripting and Request Forgery Attack

5.2 Cross-Site Request Forgery Attack

In this section, we demonstrate a cross-site request forgery attack. We do not distinguish between different attack vectors that can lead to a CSRF attack (script, images, etc.) here but it would be possible. We also do not implement any mitigation techniques against CSRF attacks here in order to demonstrate the attack. Mitigation techniques like the ones described in Section 2.2.2 can often be implemented with the help of a restriction. For example, one could introduce the server-side mitigation technique of reauthentication before state changing actions by restricting the server to not accept login cookies if the Web page is state changing.

The attacker in this case study does not need any network capabilities or script injection capabilities. The attacker model in this example is the so called Web attacker [14]. These attackers set up their own Web page including registering domains and obtaining a valid certificate for them. Setting up a malicious but legitimate Web page can be modelled using the below rule `Setup_Webpage_Malicious`.

```
1 rule Setup_Webpage_Malicious:
2 let url = <$Protocol, $Server, $Path, $File>
3     webpage = <'TriggerRequest', url, url_bank, 'NULL'>
4     security = 'none'
5 in
```

```
    [ !Webpage(url_bank, webpage_bank, security_bank) ]
  --[ Protocol($Protocol)
    , Setup_Webpage(url) ]->
    [ !Webpage(url, webpage, security) ]
```

The other possibility is to extend a network attacker's capability to registering a domain and obtaining a public-private key pair using the `Reveal_SK` rule below. We went with the first option because we neither need public key infrastructure nor a network attacker for this example.

```
/* Public Key Infrastructure */

rule Register_PK:
    [ Fr(~ltk) ]
  --[ ]->
    [ !Ltk($Server, ~ltk), !Pk($Server, pk(~ltk)),
      Out(pk(~ltk)) ]

rule Reveal_SK:
    [ !Ltk(server, ltk) ]
  --[ Reveal(server) ]->
    [ Out(ltk) ]
```

As the attacker has full control over their own Web page, they can include elements (code, images, etc.) triggering a request to any Web page. Once the user navigates to the attacker's Web page, the harmful elements are loaded and trigger requests to any third party. If the user has previously acquired a login cookie for them, the attack often succeeds in changing the server state (e.g., update profile information). The server, like in the previous demonstrated attack, cannot distinguish between a user initiated request and a malicious request forging a user request. To model a CSRF attack, we need the following building blocks from the toolbox:

- The user handler with the capability of requesting a Web page and signing in to a Web page.
- The content handler with the capability of sending a request triggered by the content of a loaded Web page. This functionality is provided by the rule `Content_Handler_TriggerRequest` presented in Section 4.3.2. The content handler is, thus, identical to the previous case study in Section 5.1.

- The response handler. Note that the redirect handler is not needed to demonstrate this attack.
- The honest Web page the user is logged in to. Requests to this Web page will change the server state but trigger no action in the browser. This is a placeholder for any state changing request like updating profile information, transferring money, or posting in a forum.
- The malicious but legitimate Web page with content triggering a forged cross-site request.
- Basic browser infrastructure that is always needed, i.e., creating a browser, creating a tab, and creating cookie storage.
- Generation and distribution of user credentials.
- Server functionality that can handle requests, authenticate users, and issue login cookies.
- A communication medium, HTTPS using confidential channels in this case.

The property we want to validate is the same as in the previous section and can be found below. Again, the final version of the theory file contains additional constraints for the same reasons as mentioned in the previous section (performance and conciseness of found attack). Using these constraints, Tamarin finds an attack in under 30 seconds. In Figure 7, you can find the message sequence chart of the attack found by Tamarin.

```
/* Lemmas */

lemma Honest_Request_Origin:
"
not (Ex #i. Network_Attacker() @i)
==>
(All r u #i. Protected_Webpage_Accessed(r, u) @i
==>
((Ex #j. Request_Origin(r, u, 'UserLogin') @j)
 | (Ex #j. Request_Origin(r, u, 'UserRequest') @j) ))
"
```

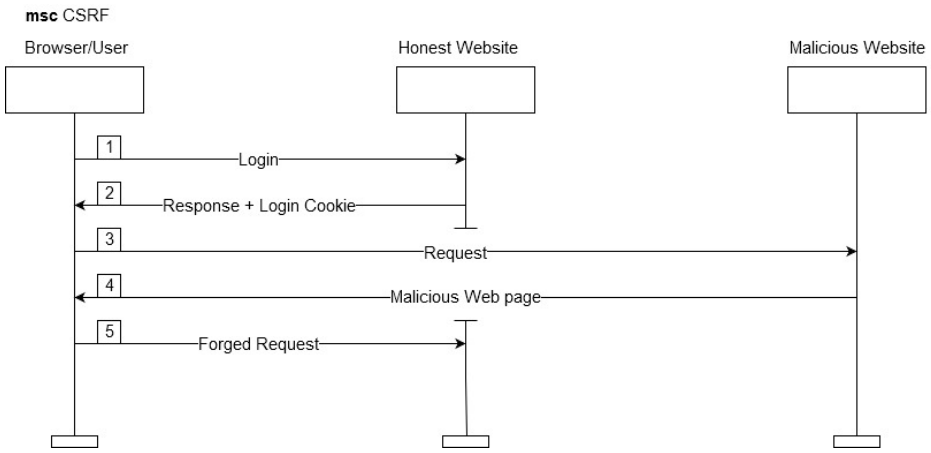

Figure 7: Diagram of the Cross-Site Request Forgery Attack

5.3 Same-Domain Cookie Policy

In this case study, we model a cookie stealing attack on the users browser and a mitigation technique. The user sends a request to the attacker's Web site which responds with a script that extracts a cookie from the browser's cookie storage and sends it back to the attacker without the user realising it. Execution of the malicious script can be modelled in our framework using the below rule as part of the content handler.

```
1  rule Content_Handler_ExtractCookie:
2  let url = <$Protocol, $Server, $Path, $File> // script
       origin domain
3      loaded_webpage = <'ExtractCookie', url, url, 'NULL'>
4      new_pending_request = ~request_id
5      request = <<$Cookie_Domain, stolen_cookie>, cookie>
6  in
7      [ Tab_State(tab_id, browser_id, loaded_webpage,
          old_pending_request)
8      , Cookie_Storage(browser_id, $Cookie_Domain,
          stolen_cookie) // Cookie the malicious code is
          stealing
9      , Cookie_Storage(browser_id, $Server, cookie) //
          Cookie the browser has stored for the malicious
          website
10     , Fr(~request_id) ]
```

```
11   --[ Protocol($Protocol)
12     , Cookie_Is_Sent(url, cookie)
13     , Content_Handler_ExtractCookie()
14     , Script_Accesses_Cookie($Server, $Cookie_Domain,
         stolen_cookie) //script domain, cookie domain
15     , Request_Origin(new_pending_request, url,
         'Script') ]->
16     [ Browser_Send(new_pending_request, browser_id,
         url, request)
17     , Tab_State(tab_id, browser_id, loaded_webpage,
         old_pending_request)
18     , Cookie_Storage(browser_id, $Server, cookie) ]
```

By having access to the login cookie, the attacker can now impersonate the user to the honest Web application. Such an attack is also known as a session hijacking attack. Gaining access to the cookie can also be achieved by other means like sniffing unencrypted network traffic but we focus on the scripting attack here. As cookies are generally stored on a per browser basis and not on a per tab basis, the user can log in to the honest Web site in one tab and navigate to the malicious Web site in another tab. If cookies are not deleted regularly, the attack can also succeed if the user has logged in a long time ago. The interesting thing to show here is that the attack is prevented by including the *same-domain policy* into the model. Unlike other resources in the browser, such as local or session storage, cookies are not protected with a same-origin policy but only with a same-domain policy. This means that a script can access cookies that are associated with the same domain as the context the script is loaded into. Other resources can only be accessed by a script if the script context has the same origin as the resource. An origin is the combination of URI scheme, domain, and port [14]. By including the same-domain policy restriction below into the theory, the attack can be prevented.

```
1 restriction same_domain_policy:
2 " All script_domain cookie_domain cookie #i.
    Script_Accesses_Cookie(script_domain, cookie_domain,
    cookie) @i
3 ==> (script_domain=cookie_domain)"
```

Another possibility to prevent the attack is to set the *HTTPonly* flag of the cookie to `'true'` and enforce the attribute with a restriction like the one shown below. This prevents all scripts from accessing this particular cookie

regardless of the domain. This, however, may be too restrictive for some situations and relies on the server-side to set the cookie attribute. Browsers, thus, enforce the same-domain policy for accessing cookies per default.

```
restriction httponly:
"All script_domain cookie_domain cookiename cookievalue
    httponly secure #i.
Script_Accesses_Cookie(script_domain, cookie_domain,
    <cookiename, cookievalue, <'HTTPonly', httponly>,
    <'Secure', secure>>) @i
==> (httponly = 'false') "
```

We formulate the property that the attacker should not be able to receive a stolen cookie using the below lemma `Cookie_Not_Stolen_By_Script`. In the final theory, there are additional properties added to the lemma in order to increase performance and conciseness of the Tamarin output. Without the same-origin policy enforced, an attack is found by Tamarin in a few seconds, see Figure 8. With the same-origin policy enforced, Tamarin finds no attack and the lemma verifies in under 30 seconds.

```
lemma Cookie_Not_Stolen_By_Script:
" not (Ex cookie_domain cookie #i.
    Attacker_Receives_Cookie(cookie_domain, cookie) @i) "
```

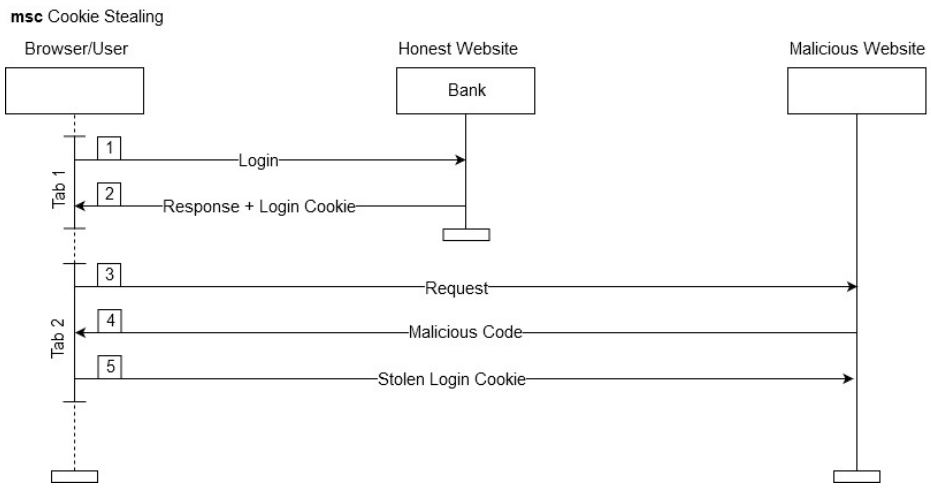

Figure 8: Diagram of the Cookie Stealing Attack

5.4 OpenID Connect

In this section, we apply our framework to the OpenID Connect protocol introduced in Section 2.3. This protocol is already well-studied and serves as a proof of applicability of the framework. While developing the model of OpenID Connect, we oriented ourselves on the work by Fett [16] and Hofmeier [1]. Both abstracted away different steps of the protocol. We combined their abstractions and, thus, work with a protocol abstraction that is more detailed than each of them, i.e., includes more steps than each of them. In comparison with Fett's model, we included user consent meaning that the user has to give explicit consent to the IdP to send their identity to the RP. In comparison with Hofmeier's model, our framework allowed to model the retrieval of the ID Token via a script provided by the RP when using the OpenID Connect Implicit Flow. In addition to the extension of the model of the OpenID Connect protocol, our framework also allowed the modelling of login cookies. Note that the model does not yet allow logging in to the IdP using a login cookie. But as we have shown in the previous case studies, our framework is capable of modelling this functionality by adding just one rule. Incorporating it into the OpenID Connect model is a matter of future work.

In the following, we describe how we modelled different aspects of the OpenID Connect protocol in our framework that are not flow-specific. The user behaviour is independent from the chosen flow but specific to OpenID Connect. We, thus, added the below custom MSR rules to the Tamarin implementation of the user handler. We model the user initiating the OpenID Connect protocol by the rule `User_Starts_OIDC`. This rule sends a message to the RP requesting to sign in with the chosen IdP.

```
1  rule User_Starts_OIDC:
2  let rp_url = <'HTTPS', $RP, $RP_Path, $RP_File>
3      idp_url = <'HTTPS', $IdP, $IdP_Path, $IdP_File>
4      new_pending_request = ~request_id
5  in
6      [ Tab_State(tab_id, browser_id, loaded_webpage,
          old_pending_request)
7      , Cookie_Storage(browser_id, $RP, cookie)
8      , !IdP(idp_url)
9      , Fr(~request_id) ]
10   --[ User_Starts_OIDC()
```

```
11      , User_Starts_OIDC2($RP, $IdP, browser_id)
12      , Cookie_Is_Sent(rp_url, cookie) ]->
13      [ Browser_Send(new_pending_request, browser_id,
           rp_url, <<'SingleSignOn', idp_url>, cookie>)
14      , Tab_State(tab_id, browser_id, loaded_webpage,
           new_pending_request)
15      , Cookie_Storage(browser_id, $RP, cookie) ]
```

We model the user authenticating to the IdP using the rule `User_Authenticates`. The OpenID Connect documentation does not specify how the IdP authenticates the user. We decided to model authentication using the classic username and usersecret approach from our framework.

```
1  rule User_Authenticates:
2  let idp_url = <'HTTPS', $IdP, $Target_Path,
       $Target_File>
3       loaded_webpage = <'LoginForm', idp_url, 'NULL',
           'NULL'>
4       new_pending_request = ~request_id
5  in
6       [ Tab_State(tab_id, browser_id, loaded_webpage,
           old_pending_request)
7       , Cookie_Storage(browser_id, $IdP, cookie)
8       , !User_Credentials($IdP, $Username, usersecret)
9       , Fr(~request_id) ]
10    --[ User_Authenticates()
11      , User_Authenticates2($IdP, $Username, browser_id)
12      , Cookie_Is_Sent(idp_url, cookie) ]->
13      [ Browser_Send(new_pending_request, browser_id,
           idp_url, <<'Authenticate', $IdP, $Username,
           usersecret>, cookie>)
14      , Tab_State(tab_id, browser_id, loaded_webpage,
           new_pending_request)
15      , Cookie_Storage(browser_id, $IdP, cookie) ]
```

We model the user sending their consent to the IdP using the below rule `User_Gives_Consent`. The consent message specifies to which IdP the user consents, for which RP the user consents, and for which username the user consents.

```
rule User_Gives_Consent:
let idp_url = <'HTTPS', $IdP, $Target_Path,
    $Target_File>
    loaded_webpage = <'ConsentForm', idp_url, 'NULL',
       <$RP, $Username>>
    new_pending_request = ~request_id
in
    [ Tab_State(tab_id, browser_id, loaded_webpage,
       old_pending_request)
    , Cookie_Storage(browser_id, $IdP, cookie)
    , Fr(~request_id) ]
  --[ User_Gives_Consent()
    , User_Gives_Consent2($IdP, $RP, $Username)
    , Cookie_Is_Sent(idp_url, cookie) ]->
    [ Browser_Send(new_pending_request, browser_id,
       idp_url, <<'Consent', $IdP, $RP, $Username>,
       cookie>)
    , Tab_State(tab_id, browser_id, loaded_webpage,
       new_pending_request)
    , Cookie_Storage(browser_id, $IdP, cookie) ]
```

The logic of the RP server and the IdP server to specific to the protocol and cannot be covered by the generic server functionality presented in Section 4.4. We, thus, implement custom Tamarin MSR rules to represent the IdP and RP logic. The interfaces to the communication medium, however, stay the same. This logic is largely flow-specific. One common component are the rules to set up the two Web applications.

```
rule Setup_RP:
let rp_url = <'HTTPS', $RP, $Path, $File>
    rp_redirect_url = <'HTTPS', $RP, $Redirect_Path,
       $Redirect_File>
in
    [ ]
  --[ ]->
    [ !RP(rp_url, rp_redirect_url) ]

rule Setup_IdP:
let idp_url = <'HTTPS', $IdP, $Path, $File>
in
    [  ]
```

```
13   --[ ]->
14     [ !IdP(idp_url) ]
```

We do not model the discovery protocol of OpenID Connect separately, but represent it by the following discovery rule `Discovery`.

```
1 rule Discovery:
2     [ !IdP(idp_url), !RP(rp_url, rp_redirect_url),
        Fr(~rp_id) ]
3   --[ Source_rpid(~rp_id) ]->
4     [ !Discovered(idp_url, rp_url, rp_redirect_url,
        ~rp_id) ]
```

We model the the authentication request sent by the RP as follows:

```
1 <'AuthReq', rp_id, rp_redirect_url, ~nonce,
    responsetype>
```

The value of the response type field depends on the flow. Once the user initiated the OpenID Connect protocol, the RP redirects the user to the IdP together with the authentication request. We represent this by the following rule that is the same for both flows. Note that `responsetype` is a placeholder for the string indicating the flow type.

```
1 rule RP_Receives_UserStart:
2 let idp_url = <'HTTPS', $IdP, $Path, $File>
3     rp_url = <'HTTPS', $RP, $RP_Path, $RP_File>
4     webpage = <'Redirect', rp_url, idp_url, <'AuthReq',
        rp_id, rp_redirect_url, ~nonce, responsetype>>
5     sent_cookie = 'NULL'
6 in
7     [ Server_Receive(request_id, channel_id, rp_url,
        <<'SingleSignOn', idp_url>, received_cookie>)
8     , !Discovered(idp_url, rp_url, rp_redirect_url,
        rp_id)
9     , Fr(~nonce) ]
10  --[ RP_Receives_UserStart()
11    , RP_Sends_AuthReq($RP, $IdP, ~nonce)
12    , Problem_rpid(rp_id)
13    , Source_nonce(~nonce) ]->
14    [ Server_Send(request_id, rp_url, <webpage,
        sent_cookie>)
15    , RP_State('1', <$RP, $IdP, channel_id, rp_id,
        rp_redirect_url, ~nonce>) ]
```

Independent of the flow type is also the behaviour of the Authentication Endpoint of the IdP responsible for authenticating the user and obtaining consent. We model this behaviour with the following two rules:

```
rule IdP_Receives_AuthReq:
let idp_url = <'HTTPS', $IdP, $Path, $File>
    webpage = <'LoginForm', idp_url, 'NULL', 'NULL'>
    sent_cookie = 'NULL'
in
    [ Server_Receive(request_id, channel_id, idp_url,
       <<'AuthReq', rp_id, rp_redirect_url, nonce,
       responsetype>, received_cookie>)
    , !Discovered(idp_url, rp_url, rp_redirect_url,
       rp_id) ]
  --[ IdP_Receives_AuthReq() ]->
    [ Server_Send(request_id, idp_url, <webpage,
       sent_cookie>)
    , IdP_State('1', <$IdP, channel_id, rp_id,
       rp_redirect_url, nonce>) ]
```

```
rule IdP_Receives_Credentials:
let idtoken = revealSign(<'idtoken', $Username, $IdP,
    rp_id, nonce>, ltk)
    rp_url = <'HTTPS', $RP, $RP_Path, $RP_File>
    idp_url = <'HTTPS', $IdP, $Path, $File>
    webpage = <'ConsentForm', idp_url, 'NULL', <$RP,
       $Username>>
    sent_cookie = <'login_id', ~cookievalue,
       <'HTTPonly', 'true'>, <'Secure', 'true'>>
in
    [ Server_Receive(request_id, channel_id, idp_url,
       <<'Authenticate', $IdP, $Username, usersecret>,
       received_cookie>)
    , !Hash_Credentials($IdP, $Username, h(usersecret))
    , !Discovered(idp_url, rp_url, rp_redirect_url,
       rp_id)
    , IdP_State('1', <$IdP, channel_id, rp_id,
       rp_redirect_url, nonce>)
    , Fr(~cookievalue )
    , !Ltk($IdP, ltk) ]
  --[ IdP_Receives_Credentials()
```

```
15      , Problem_rpid(rp_id)
16      , Problem_nonce(nonce) ]->
17      [ Server_Send(request_id, idp_url, <webpage,
           sent_cookie>)
18      , Server_Storage($IdP, $Username, ~cookievalue)
19      , IdP_State('2', <$IdP, channel_id, rp_id,
           rp_redirect_url, nonce>) ]
```

Note that both the IdP and the RP use server-side state (`IdP_State` and `RP_State` respectively) to keep track of the protocols progress.

We model the signed ID Token as follows:

```
1 revealSign(<'IdToken', $Username, $IdP, rp_id, nonce>,
    ltk)
```

Other elements from the framework used in both flows include:

- The response handler and also the redirect handler as communication between RP and IdP is often redirected via the user's browser.
- Basic browser infrastructure that is always needed, i.e., creating a browser, creating a tab, and creating cookie storage.
- Generation and distribution of user credentials.
- A communication medium including a network attacker. HTTPS using confidential channels allowing multiple messages in this case.
- Public-key infrastructure used for signing and verifying the ID Token.
- Server-side storage for storing the values of login cookies as described in Section 4.4.

5.4.1 Authorisation Code Flow

The response type of the Authorisation Code Flow is `code`. Therefore, the Authentication Request sent by the RP has the following form:

```
1 <'AuthReq', rp_id, rp_redirect_url, ~nonce,
    'ResponseType_is_Code'>
```

The Authorisation Code Flow of OpenID Connect can be used if a back-channel between IdP and RP is available. We modelled this back-channel by bypassing the standard communication medium and directly passing the

message using a fact. We thereby also modelled the difference between the Authentication End-Point and the Token End-Point of the IdP. Communication to the Authentication End-Point is over the usual communication medium while communication to the Token End-Point is exclusively over the back-channel. In our model, we assume no attacker interference on the back-channel.

Below we describe the IdP and RP functionalities specific to the Authorisation Code Flow as rules. Once the IdP obtained the user's consent, it generates a fresh Authentication Code. The IdP redirects the user's browser back to the RP together with the Authentication Code. At the same time, the IdP stores the Authentication Code in its server-side state.

```
1  rule IdP_Receives_Consent:
2
3  let idp_url = <'HTTPS', $IdP, $Path, $File>
4      webpage = <'Redirect', idp_url, rp_redirect_url,
           ~authcode>
5      sent_cookie = 'NULL'
6  in
7      [ Server_Receive(request_id, channel_id, idp_url,
           <<'Consent', $IdP, $RP, $Username>,
           received_cookie>)
8      , !Discovered(idp_url, rp_url, rp_redirect_url,
           rp_id)
9      , IdP_State('2', <$IdP, channel_id, rp_id,
           rp_redirect_url, nonce>)
10     , !Ltk($IdP, ltk)
11     , Fr(~authcode) ]
12   --[ IdP_Receives_Consent()
13     , IdP_Receives_Consent2($IdP, $RP, $Username)
14     , Problem_rpid(rp_id)
15     , Problem_nonce(nonce) ]->
16     [ Server_Send(request_id, idp_url, <webpage,
           sent_cookie>)
17     , Server_Storage($IdP, $Username, sent_cookie)
18     , IdP_State('3', <$IdP, rp_id, rp_redirect_url,
           nonce, ~authcode, $Username>) ]
```

When the RP receives the Authorisation Code, it sends it to the IdP via the back-channel `RP_To_IdP`.

```
rule RP_Receives_AuthCode:
let rp_redirect_url = <'HTTPS', $RP, $Redirect_Path,
    $Redirect_File>


    idp_url = <'HTTPS', $IdP, $Path, $File>
in
    [ Server_Receive(request_id, channel_id,
       rp_redirect_url, <authcode, received_cookie>)
    , RP_State('1', <$RP, $IdP, channel_id, rp_id,
       rp_redirect_url, nonce>)
    , !Discovered(idp_url, rp_url, rp_redirect_url,
       rp_id)
    , !Pk($IdP, pk(ltk))
    , Fr(~cookievalue) ]
  --[ RP_Receives_AuthCode()
    , RP_Receives_AuthCode2($RP, $IdP, authcode) ]->
    [ RP_To_IdP($RP, rp_id, $IdP, authcode, nonce)
    , RP_State('2', <$RP, $IdP, channel_id, rp_id,
       rp_redirect_url, nonce, request_id>) ]
```

When the IdP receives the Authorisation Code, it matches it against its state. If successful, the IdP signs the ID Token and sends it to the RP via the back-channel `IdP_To_RP`.

```
rule IdP_Receives_AuthCode:
let idtoken = revealSign(<'IdToken', $Username, $IdP,
    rp_id, nonce>, ltk)
in
    [ IdP_State('3', <$IdP, rp_id, rp_redirect_url,
       nonce, authcode, $Username>)
    , RP_To_IdP($RP, rp_id, $IdP, authcode, nonce)
    , !Ltk($IdP, ltk)
    , !Discovered(idp_url, rp_url, rp_redirect_url,
       rp_id) ]
  --[ IdP_Receives_AuthCode()
    , IdP_Receives_AuthCode2($IdP, $RP, authcode,
       $Username) ]->
    [ IdP_To_RP($IdP, $RP, rp_id, idtoken) ]
```

The RP then receives the ID Token and generates a login cookie for the user. This concludes the protocol.

```
rule RP_Receives_IdToken:
let rp_redirect_url = <'HTTPS', $RP, $Redirect_Path,
    $Redirect_File>
     webpage = <'NoAction', rp_redirect_url, 'NULL',
        'NULL'>
     sent_cookie = <'login_id', ~cookievalue,
        <'HTTPonly', 'true'>, <'Secure', 'true'>>
     idtoken = revealSign(<'IdToken', $Username, $IdP,
        rp_id, nonce>, ltk)
     idp_url = <'HTTPS', $IdP, $Path, $File>
in
     [ IdP_To_RP($IdP, $RP, rp_id, idtoken)
     , RP_State('2', <$RP, $IdP, channel_id, rp_id,
        rp_redirect_url, nonce, request_id>)
     , !Discovered(idp_url, rp_url, rp_redirect_url,
        rp_id)
     , !Pk($IdP, pk(ltk))
     , Fr(~cookievalue) ]
   --[ RP_Receives_IdToken()
     , RP_Receives_IdToken2($RP, $IdP, $Username, nonce)
        ]->
     [ Server_Send(request_id, rp_redirect_url,
        <webpage, sent_cookie>)
     , Server_Storage($RP, $Username, sent_cookie) ]
```

Included in the theory, one can find an executability lemma and the below property lemma `RP_Sent_AuthReq`. The property lemma shows that whenever the RP accepts an ID Token then the same RP has sent the corresponding Authentication Request. In other words, the RP will not accept an ID Token unless it sent the corresponding Authentication Request. Performance of the model is good with about one minute needed to automatically prove the executability lemma and only a few seconds for the property lemma.

```
lemma RP_Sent_AuthReq:
"
All idp rp nonce username #i.
((RP_Receives_IdToken2(rp, idp, username, nonce) @i  )
==>
 Ex #j. RP_Sends_AuthReq(rp, idp, nonce) @j)
"
```

5.4.2 Implicit Flow

For the OpenID Connect Implicit Flow protocol, no back-channel is needed. Instead, the IdP sends the ID Token to the browser and the RP has to provide a script that retrieves the token from the URL fragment. This means that the model of the Implicit Flow, unlike the one modelling the Authorisation Code Flow, includes a content handler able to execute this script, i.e., send the ID Token to the RP. We implemented this functionality by adding the following rule to the content handler:

```
rule Content_Handler_Retrieval_Script:
let idp_url = <'HTTPS', $IdP, $Target_Path,
    $Target_File>
     loaded_webpage = <'RetrievalScript',
        rp_redirect_url, rp_redirect_url, $IdP>
     new_pending_request = ~request_id
in
     [ Tab_State(tab_id, browser_id, loaded_webpage,
        old_pending_request)
     , Cookie_Storage(browser_id, $IdP, cookie)
     , URL_Fragment(browser_id, 'IdToken', $IdP, idtoken)
     , Fr(~request_id) ]
   --[ Content_Handler_Retrieval_Script()
     , Content_Handler_Retrieval_Script2($IdP,
        browser_id, idtoken)
     , Cookie_Is_Sent(idp_url, cookie) ]->
     [ Browser_Send(new_pending_request, browser_id,
        rp_redirect_url, <<'IdToken', $IdP, idtoken>,
        cookie>)
     , Tab_State(tab_id, browser_id, loaded_webpage,
        new_pending_request)
     , Cookie_Storage(browser_id, $IdP, cookie) ]
```

The content handler also needs to inform the RP that the ID Token is ready to be fetched. The following rule provides this functionality:

```
rule Content_Handler_IdToken_Ready:
let idp_url = <'HTTPS', $IdP, $Target_Path,
    $Target_File>
     idtoken = revealSign(<'IdToken', $Username, $IdP,
        rp_id, nonce>, ltk)
```

```
4      loaded_webpage = <'IdToken', idp_url,
          rp_redirect_url, idtoken>
5      new_pending_request = ~request_id
6  in
7      [ Tab_State(tab_id, browser_id, loaded_webpage,
          old_pending_request)
8      , Cookie_Storage(browser_id, $RP, cookie)
9      , Fr(~request_id) ]
10   --[ Content_Handler_IdToken_Ready()
11     , Content_Handler_IdToken_Ready2($IdP, browser_id,
          $Username, idtoken, nonce)
12     , Cookie_Is_Sent(idp_url, cookie) ]->
13     [ Browser_Send(new_pending_request, browser_id,
          rp_redirect_url, <<'IdTokenReady', $IdP>,
          cookie>)
14     , Tab_State(tab_id, browser_id, loaded_webpage,
          new_pending_request)
15     , Cookie_Storage(browser_id, $IdP, cookie)
16     , URL_Fragment(browser_id, 'IdToken', $IdP,
          idtoken) ]
```

The IdP does not need any additional functionality to the ones already described. The only difference to the Authorisation Code flow is that the IdP sends the ID Token to the browser instead of sending an Authorisation Code to the RP after obtaining the user's consent. The Token Endpoint of the IdP (`IdP_Receives_AuthCode`) is not needed here.

The RP needs to be able to send the retrieval script to the browser. We implement this using the following rule:

```
1  rule RP_Receives_IdToken_Ready:
2  let rp_redirect_url = <'HTTPS', $RP, $Redirect_Path,
      $Redirect_File>
3      webpage = <'RetrievalScript', rp_redirect_url,
          rp_redirect_url, $IdP>
4      sent_cookie = 'NULL'
5      idp_url = <'HTTPS', $IdP, $Path, $File>
6  in
7      [ Server_Receive(request_id, channel_id,
          rp_redirect_url, <<'IdTokenReady', $IdP>,
          received_cookie>)
8      , RP_State('1', <$RP, $IdP, channel_id, rp_id,
```

```
        rp_redirect_url, nonce>)
9     , !Discovered(idp_url, rp_url, rp_redirect_url,
        rp_id)
10    , !Pk($IdP, pk(ltk))
11    , Fr(~cookievalue) ]
12  --[ RP_Receives_IdToken_Ready()
13    , RP_Receives_IdToken_Ready2($RP, $IdP) ]->
14    [ Server_Send(request_id, rp_redirect_url,
        <webpage, sent_cookie>)
15    , RP_State('2', <$RP, $IdP, channel_id, rp_id,
        rp_redirect_url, nonce>) ]
```

Included in the theory, one can again find an executability lemma and the property lemma describing the same property `RP_Sent_AuthReq` as also shown for the Authentication Code Flow. Performance of this model is not as good as for the Authentication Code Flow but still manageable with nine minutes for the executability lemma and a few seconds for the property lemma. The Implicit Flow model also includes a sources lemma to remove eight partial deconstructions. Tamarin proves this lemma automatically in a few seconds.

6 Related Work

Akhaew et al. [11] made the first step in the direction of a formal analysis of the Web infrastructure in 2010 using the SAT-based model-checking tool Alloy. They proposed a first formal model of the Web platform and Web security threats. They developed three threat models depending on the attacker's capabilities. A *Web attacker* can register a Web site and use a browser but has no further insight into the network. An *active network attacker* has all the capabilities of a Web attacker plus the ability to block, forge, and eavesdrop on messages in the network. In addition to these standard Dolev-Yao abilities, an active network attacker can also use the victim's browser API to, e.g., launch a CSRF attack. Finally, the *gadget attacker* has the ability to inject content into honest Web sites not controlled by the attacker. This can happen, for example, through blog entries or third-party ads. For more information on how a gadget attacker can inject content into honest Web sites, see [14].

The authors identified two security goals that are almost universally applicable. The first one is that no new feature should violate any assumptions or invariants that Web sites rely upon for security. The second one is session integrity, meaning that the attacker should not be able to trick the server into executing any sensitive actions like transferring money. To formally validate these goals they developed a model of the Web infrastructure including browsers, servers, scripts, script context/origin, cookies, HTTP, and DNS and implemented them in Alloy. Alloy is a declarative object-modelling language. The Alloy Analyser is a modelling tool that translates a model written in Alloy into the propositional input expected by a SAT-solver. The SAT-solver then tries to find a counterexample to the assertions within a given bound. If a counterexample is found it is guaranteed to be a valid counterexample. If none is found within the bound, no guarantees can be made about the validity of the assertions, as a counterexample might exist within a larger bound. On the flip side, the solver always terminates although performance might decrease exponentially with the increase of the bound. They also developed a model for frames, location bar, and UI security features like the lock icon but have not yet included them into their implementation.

Bansal et al. proposed a library to model Web applications called WebSpi based on the ProVerif verifier in 2012. They showed how the library can be used to model, e.g., the OAuth 2.0 authorisation protocol. A extended version was published in 2014 [13]. In 2013, the same authors used

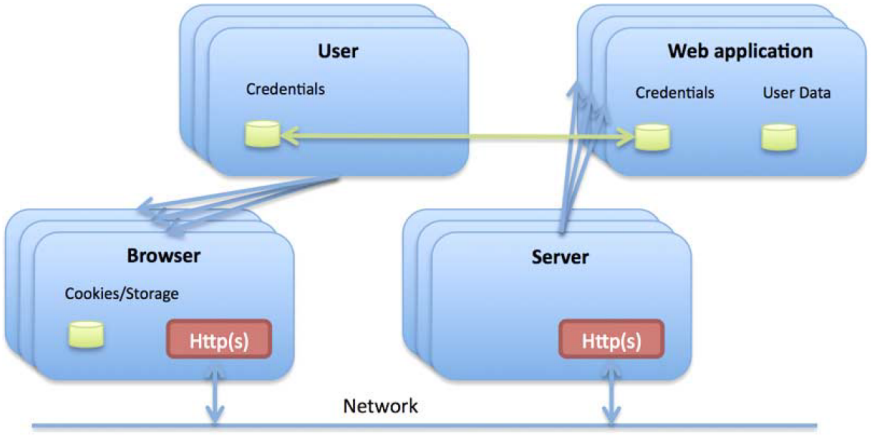

Figure 9: WebSpi Architectural Diagram from [13]

the WebSpi library to analyse encrypted Web storage [12]. In Figure 9 the high-level architecture of the WebSpi model is shown. It is possible that a so-called principal is a user and a Web application simultaneously. A principal could, for example, register and host two Web applications and also be a user of other Web applications through a browser. Users and Web applications share pre-existing credentials and Web applications have a public/private key pair. Key distribution is, thus, not modelled separately but assumed to be pre-existing. This allows communication over HTTP as well as HTTPS.

The WebSpi library is a generic framework to model Web applications in ProVerif. To make the modelling process even easier for a Web developer, the authors present an automated model-extraction tool that generates a ProVerif model directly from the server PHP and client JavaScript code. The tool supports only a subset of PHP and JavaScript instructions but seems nevertheless useful. The authors suggest isolating the security critical parts of the Web application and write these only in the supported subset. The remainder of the application can use the full instruction set of PHP and JavaScript. This is more difficult for already existing Web applications but a good option for Web applications still in development.

ProVerif, like Tamarin, is complete and sound but may not terminate. ProVerif is also unbound in size, unlike Alloy. This means that one can be sure that all security goals of the model are met whenever the tool succeeds in verifying them. If the tool fails to validate the security goals, it gives a

proof derivation thereof. It might, however, fail to provide a concrete attack trace. It also might not terminate due to the nature of the problem at hand.

As the above mentioned models were implemented in an actual verifier, the models are somewhat limited by performance constraints and the capabilities of the respective verifier. Daniel Fett, on the other hand, developed an extensive model of the Web on paper for his dissertation in 2018 [16]. His model includes many aspects of the Web from HTTP(S) messages to WebRTC. It is intended as a reference for implementing models in a verifier for formal analysis. Fett et al. also published further works related to this in [17, 18]. As a case study, Fett applied his model of the Web to OAuth 2.0 and OpenID Connect and manually analysed the protocols' properties within the model. We started with the work of Fett as a basis to develop our own model in Tamarin. Some aspects of Fett's model we included in our own, e.g., modelling the user as a non-deterministic part of the browser. Other aspects of Fett's model did not work in our model, e.g., the model of HTTPS. It lead to performance issues and lacked the possibility to send multiple messages over the same TLS channel. As the OpenID Connect protocol relies heavily on this functionality, we improved this shortcoming of his work by introducing channels that allow to send multiple messages. To model OpenID connect, we again started with Fett's abstraction of the protocol and added some details, mainly user consent.

7 Conclusion

In this work, we have developed a generic framework to model elements of the Web infrastructure and implemented it in the security protocol verification tool Tamarin. The framework includes a detailed browser model including tabs, cookies, local storage, and script execution capabilities. It also contains models of the unencrypted and encrypted hypertext transfer protocol HTTP(S). The framework provides models for servers, Web pages, server storage, URLs, redirects, and also provides an interface to add custom models of Web applications. It further includes a set of policies: The *same-domain policy* for cookies, the *HTTPonly* policy for cookies with the according attribute as well as and the *Secure* policy for cookies with the according attribute set. The framework functions as a toolbox that developers of a model of a Web application can use. Most elements of the framework can be used independently of the others and developers can choose the ones that are suitable for their use case. New elements can be added to the framework easily as long as any new rules comply with the existing interfaces, for example, new user behaviour or script execution capabilities.

We performed several case studies to show applicability of the framework. This showed that different threat models, e.g., a network attacker, a Web attacker, or XSS vulnerabilities, can be implemented and common attacks like CSRF can be found automatically by Tamarin. In addition, we modelled the steps of two different configurations of the OpenID Connect authentication protocol in greater detail than previous work [16, 1].

The framework does have its inherent limitations. Not only can a model not include every aspect of the Web but the Web is also constantly changing. In addition, large portions of the Web infrastructure are also unknown and non-uniform. It is hard to know from the outside which policies are enforced by a certain browser or how (or if) a Web application implements input/output sanitisation for example.

Future work could combine the case studies we presented. At the moment, we have individual models for OpenID Connect Implicit Flow, Code Flow, CSRF, session hijacking etc. One could merge them to see, for example, how OpenID Connect behaves in presence of a CSRF vulnerability. The framework could also be further extended by adding new elements to it. These could include, for example, iframes or a browser history. The biggest challenge when combining or adding more elements to the model will be performance. With every addition, complexity and the number of possible

traces increases, highlighting the need for automated analysis.

We conclude that our framework is a suitable model of the Web infrastructure. Its implementation in an automated analysis tool allows to efficiently model real world attacks. Even though models are inevitably an abstraction of reality, they provide important results. In practice, the key question seldom is whether a certain protocol or application is secure or not. The more realistic question is, how strong an attacker needs to be in order for the security properties to be broken and what the consequences would be. Models able to be analysed automatically, such as the presented work, can help to answer this question.

References

[1] Xenia Hofmeier. Supervised by Ralf Sasse, Sven Hammann, David Basin. *Formal Analysis of Web Single-Sign On Protocols using Tamarin*. Bachelor's thesis 2019. https://www.ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/information-security-group-dam/research/software/ba-19-hofmeier-oidc.pdf.

[2] A short history of the Web. https://home.cern/science/computing/birth-web/short-history-web. Accessed 19.8.2019.

[3] OWASP Top Ten Project. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project. Accessed 31.7.2019.

[4] Types Of Cross-Site Scripting. https://www.owasp.org/index.php/Types_of_Cross-Site_Scripting. Accessed 17.9.2019.

[5] Sandra Dünki. Supervised by Ralf Sasse, David Basin. *Testing Password Recovery Protocols*. Bachelor's thesis 2017.

[6] OpenID Connect FAQ. https://openid.net/connect/faq/. Accessed 1.8.2019.

[7] OpenID Connect Core 1.0 incorporating errata set 1. https://openid.net/specs/openid-connect-core-1_0.html. Accessed 1.8.2019.

[8] OAuth Access Tokens. https://www.oauth.com/oauth2-servers/access-tokens/. Accessed 2.8.2019.

[9] https://www.ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/information-security-group-dam/research/software/duenkis-webmodel.zip.

[10] Tamarin-prover manual. January 2019. https://tamarin-prover.github.io/manual/tex/tamarin-manual.pdf.

[11] Devdatta Akhawe, Adam Barth, Peifung E Lam, John Mitchell, and Dawn Song. Towards a formal foundation of web security. In *2010 23rd IEEE Computer Security Foundations Symposium*, pages 290–304. IEEE, 2010.

[12] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Keys to the cloud: formal analysis and concrete attacks on encrypted web storage. In *International Conference on Principles of Security and Trust*, pages 126–146. Springer, 2013.

[13] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Discovering concrete attacks on website authorization by formal analysis 1. *Journal of Computer Security*, 22(4):601–657, 2014.

[14] Philippe De Ryck, Lieven Desmet, Frank Piessens, and Martin Johns. *Primer on client-side web security*. Springer, 2014.

[15] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.

[16] Daniel Fett. An expressive formal model of the web infrastructure. 2018.

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

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

[19] Stefan Koch. *JavaScript: Einführung, Programmierung und Referenz*. dpunkt-Verlag, 2011.

[20] Mark Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. " O'Reilly Media, Inc.", 2011.

A Oracle Template

This oracle was needed in a previous version of the framework where HTTPS was modelled using encryption instead of confidential channels. We include the oracle here as a version of it might be needed when working with this earlier HTTPS model. Thanks to Ralf Sasse for providing me with a template I could use to write this oracle.

```
#!/usr/bin/python

import re
import os
import sys
debug = True

lines = sys.stdin.readlines()
lemma = sys.argv[1]

# INPUT:
# - lines contain a list of "%i:goal" where "%i" is the
    index of the goal
# - lemma contain the name of the lemma under scrutiny
# OUTPUT:
# - (on stdout) a list of ordered index separated by EOL


rank = []              # list of list of goals, main
    list is ordered by priority
maxPrio = 110
for i in range(0,maxPrio):
  rank.append([])

# SOURCES LEMMA
if lemma == "typing":
  for line in lines:
    num = line.split(':')[0]

    if re.match('.*!KU\( senc\(.*', line):
       rank[109].append(num)

```

```
30      # Branches where Sqn_HSS makes no sense.
31      # elif re.match('.*Sqn_HSS\(.*~sqn_root,
           ~sqn_root.*', line): rank[99].append(num)
32
33
34
35  else:
36    exit(0)
37
38  # Ordering all goals by ranking (higher first)
39  for listGoals in reversed(rank):
40    for goal in listGoals:
41      sys.stderr.write(goal)
42      print goal
```

B Model

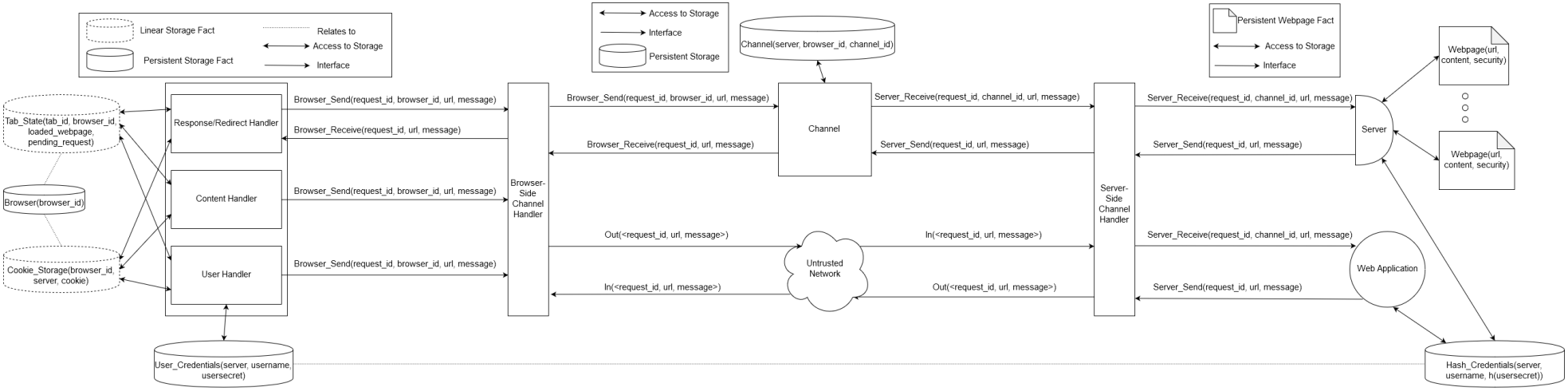

Figure 10: Complete Diagram of Model

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Modelling and Analysis of Web Applications in Tamarin

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Dünki	Sandra

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Dietikon, 30.9.2019	S. Dünki

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.