Formal Analysis of A Single Sign-on Protocol Implementation for Android

Quanqi Ye, Guangdong Bai, Kailong Wang and Jin Song Dong
National University of Singapore

***Abstract*—As the boom of social networking, Single Sign-On (SSO) services developed by major commercial service providers like Facebook, Google and Twitter, have been widely used by web-based service providers as an alternative authentication scheme. Despite rich research has focused on browser-based web applications, little has been conducted on the implementation of SSO on mobile platforms. However, we reveal that due to the fundamental difference of isolation mechanism in mobile OS and applications from the origin-based isolation in browsers, the SSO encounters a novel attack surface and adversarial models. We perform the first formal analysis on the implementation of the most widely used SSO service–Facebook Login. Our study takes as input the available implementation and dynamic execution traces of Facebook SDK for Android, from which we abstract the implementation-level protocol. The protocol is then modeled in typed Pi-calculus, and automatically checked against the mobile platform specific attack models in a protocol verifier Proverif. Our study has successfully identified a major vulnerability, which allows an attacker to steal authentication credentials from victims and log into their Facebook accounts.**

I. Introduction

Single Sign-On (SSO) [1] is an authentication scheme that allows a user to use his identity registered with an identity provider (IdP) to login to third party applications (simply apps hereafter). As it reduces the cost of managing the website accounts and security risk of password leakage[1] for the users, it has been widely deployed in the modern web environment. Major commercial websites like Facebook, Google and Twitter have developed their SSO services and opened for third party apps to use.

As mobile devices are increasingly becoming a main portal of the web—people use their mobile devices to handle lots of daily activities ranging from online shopping, premium service subscribing to internet banking, traditional web services which were originally designed for web browsers are migrated to Android. There is no exception for SSO service. It is reported that Facebook Login service had been imported by 9 million apps by 2012 [3], and 81 of the top 100 iOS apps and 62 of the top 100 Android apps use Facebook Login [4], [5].

Due to its popularity, security of SSO has become a concern of recent research [6]–[8]. While most prior studies have focused on the desktop or web-based browser environment, few study has inspected the security problems of SSO implementation on mobile platforms. However, we remark that Android has different security assumptions and attack surfaces which are very different from the traditional desktop and browser environment. On one hand, mobile devices are usually under constraints of computational resource and battery power so that it is infeasible for mobile platforms to deploy the real-time malware detection and this will render the system easy to suffer attacks from malicious apps; on the other hand, the unavailability of important security mechanisms like the Same Origin Policy (SOP) [9], [10] which is widely adopted in desktop browsers makes mobile devices easy to have vulnerabilities that were impossible or difficult to exist in the browser environment [11]. Additionally, the concrete implementation of SSO protocol that has additional features not specified by SSO might introduce new vulnerabilities in the protocol. For example, Facebook Login service introduces authorization into its SSO implementation. Thus, it is necessary to take these assumptions and attack surfaces into consideration when evaluating the security of the SSO implementation on mobile platforms.

In this work, we formally analyze the security properties of the implementation of the most widely used SSO service—Facebook Login service. We built a formal model in typed Pi-calculus[2] for the extracted implicit protocol of Facebook Login via examining the network traces and source code of its SDK. Using Proverif to analyze the model against a set of security properties, we successfully identified a major vulnerability which allows a malicious app to obtain the authentication credentials associated with the victim's Facebook account. We have constructed a concrete attack on the real-world implementation after examining the counterexamples.

Over the years, formal analysis has been proved to be a powerful approach to analyze the design of security protocols in a mathematical and rigorous way. Our analysis further demonstrates its effectiveness in analyzing the protocols on the implementation level. Although our analysis targets the implementation on Android, which has been reported to share the largest mobile market [14], this approach is also applicable to other platforms like iOS. To the best of our knowledge, we are the first to perform formal analysis on SSO implementation on a mobile platform. We summarize our contributions in the following:

1) We are the first to perform a formal analysis on an SSO implementation (Facebook Login for Android) on Android.
2) We highlight the mobile platform specific attack models against the authentication implementation. They should be take into consideration for future design and analysis of authentication protocols.
3) We have identified a major vulnerability in the existing Facebook Login for Android implementation,

[1]A recent research reveals that many people tend to use a same password for multiple accounts across websites [2].

[2]A variant of Applied Pi-calculus [12] and modelling language for Proverif [13].

which allows the attacker to steal credentials from victims and access their Facebook accounts.

II. Background

We first introduce the advantages of performing a formal analysis on the implementation of SSO protocol on a mobile platform. We then present a typical SSO process to ease the understanding in later sections. Finally, we introduce Proverif and its input language typed Pi-calculus which is used to model protocols.

A. Advantages of Formal Analysis on SSO

Security vulnerabilities in SSO are multifaceted. They may be introduced due to mistakes in protocol design or misunderstandings of the developers on the documentation [7]. In addition, Android is very different from desktop or web browser environment, which has different attack surfaces for attacker to exploit [15] as aforementioned in Introduction section. To address the above problems, a rigorous approach needs to be adopted to provide a systematic analysis. Formal analysis is such a rigorous approach which is good at proving properties in a formal model.

Formal analysis has been proved to be effective in recent research of security analysis and protocol verification [8], [16], [17]. In this work, we aim to use it to analyze the implementation of SSO protocol—Facebook Login. As a summary, the metrics of formal analysis which make it a powerful analysis tool for security protocol include the following two aspects.

- **A Systematic Approach.** Formal analysis is a systematic approach that help researchers to analyze a protocol from different aspects. Different attack models can be checked against the formal model of the protocol in an incremental manner, and this enables us to check the security properties from different angles.
- **Proving Property.** Property proving is a trait of formal analysis which is very useful in analyzing protocols. Researchers can examine the correctness of a protocol by specifying properties that it should satisfy. Then, verifier can be used to check the satisfiability of these properties in order to prove the correctness of the protocol.

B. Introduction to SSO Login Process

We introduce the general SSO login process and then we use a concrete example to introduce Facebook Login which is a real implementation of SSO login protocol.

SSO is a mechanism which allows a user to use his identity registered with an identity provider to login to a third party app (a.k.a. service provider, SP). There are mainly three principals involved in a typical SSO login flow: a user, a third party app and the Identity Provider (IdP) as is shown in left half of Fig. 1.

1) A user wants to login to the third party app with his identity in IdP. He initiates a login request to the app.
2) The app requires user to provide a token proving he has access to his identity information in IdP.
3) User asks IdP to generate an token for him.
4) IdP generates a token and returns the token to user.
5) User uses the token as a proof to show to the app he has access to his account and logs in to the app with his identity on IdP.

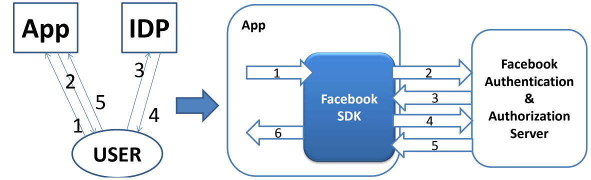


Fig. 1: General SSO Process to A concrete exmaple

In Facebook Login implementation, to facilitate the developers to incorporate Facebook Login service into their apps, Facebook separates the IdP into two parts as shown at the right half of Fig. 1. One is the client side identity provider server (IdP_C) which is normally the Facebook Login SDK. The other part is the server side identity provider (IdP_S) which is the Facebook authentication and authorization server. We demonstrate Facebook Login with an example of a user logging into a news digest app with his Facebook account.

1) User wants to login to the the news digest app with his Facebook account and the app issues a request to Facebook SDK asking for an access token with user's credentials.
2) Facebook SDK redirects the request to Facebook authentication and authorization server.
3) If user has not logged in to Facebook, Facebook server would verify the user's credentials and authenticate the user at this step. Then, Facebook server returns a cookie back to Facebook SDK. If user has already logged in, step 2 and this step are skipped. At the end of this step, user has logged in Facebook with his credentials and has got the cookie.
4) Facebook SDK asks Facebook server to generate access token for user with user's cookie.
5) Facebook server verifies user's cookie and returns the access token to Facebook SDK if user's cookie is correct.
6) Facebook SDK hands over the access token to the app to finish the protocol execution.

In Facebook Login implementation, it enables users to authorize third party apps to access their Facebook accounts. This is not specified by traditional SSO protocol and this makes the protocol complicated. After step 3 and before step 4, Facebook would prompt to user to authorize the app to access to user's Facebook account. The protocol proceeds if the user consents authorizing the app. Therefore, step 2 and 3 are the authentication steps and step 4 and 5 are the authorization steps of Facebook Login.

C. Proverif and Applied Pi-calculus

In this paper, we model Facebook Login protocol in typed Pi-calculus and check it using the tool Proverif. Proverif [18] is an automatic cryptographic protocol verifier which is developed by Blanchet et al. It is able to prove *reachability* properties and *correspondence* assertions. These abilities can be used to check security properties during the execution of a protocol. If some security properties are violated, there might exists vulnerabilities. The reachability property and correspondence assertion are specified in the form of query.

Reachability query is checked by command `query attacker(secret)` in the declaration part of the model to test the secrecy of a secret. Actually the query inside Proverif is proved by deducing the opposite of the original query. For example, if there is a query `query attacker(message)` in the formal model, Proverif tries to prove `query not`

`attacker(message)` and the report also shows the result of `query not attacker(message).`

Correspondence query is in the forms of `query event(e2) ==> event(e1)` which means for each `e2` event that happened, there is a previously happened event `e1`. The one-to-one relationship query can be captured by injective event query, i.e., `query event(e2) ==> inj-event(e1)`, which means that for each `e2` that happened, there is one and only one previous event `e1` happened. There is no difference to write either `event` or `inj-event` before ==> in the correspondence query as they have the same meanings.

III. Method Overview

In this section, we introduce the overview of our method. As shown in Fig. 2, our analysis consists of four stages. Each stage generates an artifact as the input for the next stage. In the following, we intuitively introduce each stage and leave the technique details to the future sections.

- **Protocol Extraction.** Before formally modeling the Facebook Login, it is necessary to understand how the protocol works, including what messages are exchanged among participants, and what are the semantics of the exchanged messages. To learn these, in protocol extraction stage, we first statically analyze the source code of Facebook SDK for Android to understand the communication channel between SDK and the third party app. Then, we dynamically run the protocol and capture the network traffics between the app and Facebook server. After that, by performing protocol semantic inference, we remove the redundant messages in the network traffic and thus get a refined protocol of Facebook Login.
- **Protocol Modelling.** In this step, we aim to formally model the derived protocol in formal language. The challenge is the gap between the implementation and the high-level formal presentation. To solve this, we first specify the derived protocol into an intermediate representation which is close to our modeling language. Then, the intermediate representation is translated into the formal model, combined with the formal specification of the security properties to check the attack models.
- **Protocol Verification** and **Vulnerability Analysis.** The formal models are checked in Proverif, and we study the verification report generated by it. If any property is violated, Proverif generates counterexamples in the form of traces leading to the violation. Based on the counterexamples, we construct concrete attacks and feed them to a dynamic testing which help us to confirm the vulnerabilities in the implementation.

Assumptions. Our analysis makes the following assumptions.

- **IdP_S is trusted.** IdP_S in this paper refers to the Facebook authentication and authorization servers. We assume that the IdP_S can always be trusted and we don't consider the situation when it is compromised by the adversary.

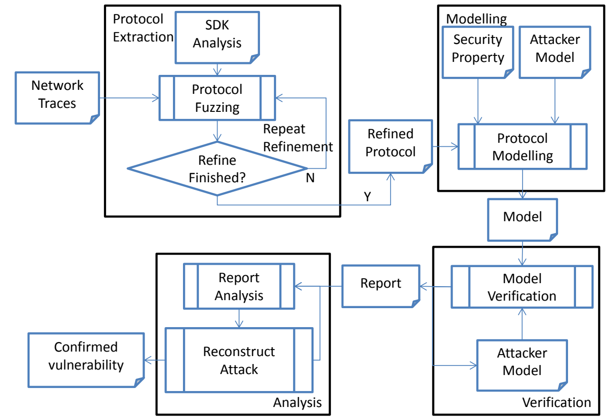


Fig. 2: Overview of Formal Analysis to Facebook Login for Android

- **Cryptographic Algorithms are correct and not exploitable.** We assume that the cryptographic algorithms used in the communication are trusted and secure.
- **Android OS is trusted.** We assume that the Android Operating system itself is trusted and not compromised. Although there might be malicious apps installed in the system, the app can not affect the behavior of the system.

IV. Protocol Extraction

To build a formal model for Facebook Login protocol, we need to figure out 1) what information is exchanged during the protocol execution and what the semantics of these messages are, and 2) how these messages are transmitted. The key challenges in protocol extraction include the following two aspects.

- **Partially Available Source code.** In most of the time, security analysts are not able to obtain the full source code or they can just obtain part of the source code of a programme. We face this situation during our analysis—we only have the source code of Facebook SDK for Android which is the client side of the SSO implementation, and we do not have any source code of the implementation on the server side. We are not able to know exactly what are the states of the server; how the state transitions take place; and how the server examines the messages obtained from the client-side SDK. We can only infer the server side logic from the responses of the server by fuzzing the server.
- **Undocumented Semantics.** When we are attempting to understand the semantics of the transmitted messages (obtained through capturing the network traces), the challenge is that the meanings of the parameters in a request or response are not easy to understand and infer, because some of them may be a cipher which appears as a random string. To make the problem even worse, there is not any official documentation which explains this.

To address the first question and these two challenges, we take the implementation of the server as a blackbox. In

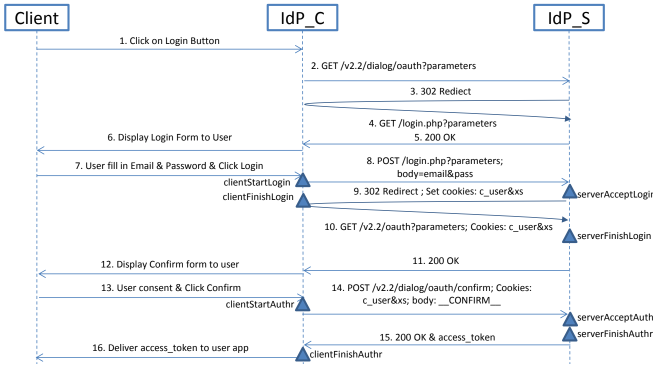


Fig. 3: Refined Protocol. Here the triangle represents the occurrence of certain events. The 4 events from top to down at IdP_C are: clientStartLogin(), clientFinishLogin(), clientStartAuthr() and clientFinishAuthr(); the other 4 event from top to down at IdP_S are: serverAcceptLogin(), severFinishLogin(), serverAcceptAuthr() and serverFinishAuthr().

particular, we study the communication between the IdP_C and IdP_S. We first perform parameter fuzzing to remove redundant parameters from the network traces captured during a successful SSO login process, and through this we can obtain the messages relevant to the protocol. Then we manually examine the remaining parameters to recover their semantics. For the second question, we performed a white box analysis to the source code of Facebook Login SDK to see what are transmitted and how the credentials are delivered in different channels. In the remain of this section, we detail the key techniques we use during the protocol extraction.

A. *Protocol Initialization*

In this section, we introduce the techniques we use to identify the participants in Facebook Login protocol and to identify the participants' communication channels. After that, we summarize a rough protocol steps at the end of this section.

Participants. We use a valid Facebook account to login to a third party app. During the successful login, we record the network traces. By manually inspecting the network traces, we find that there are three participants involved in Facebook Login protocol. They are Facebook authentication and authorization server, Facebook SDK and the third party app.

Communication Channels. In order to identify all the communication channels, we need to understand how Facebook SDK works and figure out the channels it uses to relay credentials. To do this, we manually analyze Facebook SDK source code and the captured network traces. Our manual analysis reveals that Facebook SDK works as follows:

Firstly, the user clicks the login button provided by Facebook SDK and then a new Activity named `com.facebook.LoginActivity` from SDK is started by invoking `startActivityForResult()`. Secondly, at the absence of native Facebook app, a `webView` dialog is launched inside the newly started activity, inside which the user has to fill in his login credentials (email and password) to login to Facebook. Thirdly, Facebook server verfies user's login credentials and if the credentials are correct, it returns a confirmation form indicating what the third party app would like to access to user's account. Fourthly, user consents and clicks confirm button to authorize the app and finally an access token is returned from Facebook server to the `webView` of SDK. By studying the network traces, we find that the last message from Facebook server at the protocol is a piece of Javascript code inside which the access token returned from Facebook server is wrapped as a parameter of the url. The following Javascript is a real example derived from our app.

```
<script type="text/javascript">window.location.href=
"fbconnect://success#access_token=
actual_access_token&expires_in=sometime";</script>
```

From the Javascript code we can know it's a redirection to a url with self-defined `scheme` [19], [20]. Through further inspection of SDK source code, we find that the `webView` dialog has overridden `shouldoverrideurlloading()` method to match the `scheme fbconnect://success`. Inside that overridden method, the access token is extracted from the url's parameter, wrapped in an `intent` and delivered back to the app by `onActivityResult()`.

Therefore, at this point, we have already identified two communication channels through which the credentials flow. First channel is the *https* channel between IdP_S and IdP_C, second channel is the inter-component communication channel used by Intent between IdP_C and the app. We also find that app stores the access token in local storage with Android data storage mechanism. In this case, we also regard this as a communication channel between app and local storage. In a real world case, both app and Facebook SDK can access to the same local storage. Hence, the third communication channel is the one that connects local storage to app and IdP_C.

Protocol Steps. After identifying the involved principles and the communication channels they use, we extract the initial protocol from the captured network traces. The protocol consists of 6 steps in general.

1) IdP_C ->IdP_S: initiate SSO login request
2) IdP_S ->IdP_C: request user login
3) IdP_C ->IdP_S: provide user's credentials to login
4) IdP_S ->IdP_C: ask for user's authorize confirmation
5) IdP_C ->IdP_S: confirm authorization by user
6) IdP_S ->IdP_C: return access token

B. *Protocol Refinement*

To understand the semantics, we need to reduce the protocol by identifying the critical information and removing those redundant or unimportant information from the protocol. Then, we infer the semantics of the reduced protocol by manually inspecting the parameters.

Redundant Reduction. To remove redundant and unimportant information in the protocol, we repeat every request in the Network Traces with less parameters (and less cookies if there is any) to see whether Facebook server replies a same response or not. Same response does not mean that the responses are exactly the same. We regard a same response code and same content in two responses as two same responses. For example, if the response is a redirection with code 302 to the same origin and with a subset of parameters in the redirected url, then we would regard them as two same responses. In this definition, the redirection to http://a.com/page.php?a=xxx&b=xxx&c=xxx would be regarded the same as redirection to http://a.com/page.php?b=xxx. As for those responses with html pages, we check and examine the html files to determine

an oracle for them. If we see a same oracle in the new response as the one in the old response, we regard those two responses as the same responses. We keep removing the redundant information in the network traces in this way until we can no longer get the same responses again so that we get a reduced protocol. We summarize our protocol redundant reduction algorithm in Algorithm 1.

The algorithm needs input of the network traces as list. Firstly, the algorithm initials RT_list to an empty list. Then it extracts the requests and responses from NT_list. After that, as long as the number in RT_list is not the same as that in NT_list, it will go into the loop. Inside the loop, it first gets the first request from Req_list and response from Res_list; then removes redundant parameters from req and sends it to server with function `make_request()`; if the response is the same, then it keeps removing and sending the request until it gets a different response; when the response is different, it stops and appends the last request that has the same response as that in NT_list to RT_list and appends the response to RT_list. The algorithm finally returns the RT_list which is the reduced protocol containing requests and responses of refined parameters.

Algorithm 1: Redundant Reduction

```
Input  : NT_list: network traces list
Output: RT_list: reduced network traces list
1  RT_list ← ∅;
2  Req_list ← extractRequest(NT_list);
3  Res_list ← extractResponse(NT_list);
4  while length(RT_list) ≠ length(NT_list) do
5      req ← popFirst(Req_list);
6      response ← popFirst(Res_list);
7      rm_req ← remove_param(req);
8      res ← make_request(rm_req);
9      if res = response then
10         req ← rm_req ;
11         goto 7 ;
12     else
13         append(RT_list, req);
14         append(RT_list, response);
15     end
16 end
17 return RT_list
```

By applying Algorithm 1, we get the reduced protocol out of the original network traces. The reduced protocol is the communication flow shown at right half of Fig. 3 between IdP_C and IdP_S.

Semantics Inference. After the redundant reduction step, we identify three critical parameters from the refined protocol. They are *c_user*, *xs* and *access_token* respectively.

- ***c_user* is a user id**: we use two valid Facebook accounts to perform several successful SSO logins and record those network traces. Through analyzing the network traces, we find that *c_user* is always identical in the same account's network traces across different sessions. Therefore, we conclude that *c_user* is a user id.
- ***xs* is a session id**: we use one valid Facebook account to perform multiple successful SSO logins and also record network traces. We find that *xs* changes every time we successfully log in and it is returned right after the user's login step. In the authorization process, Facebook server needs to verify this information. Therefore, we conclude that xs is a session id.
- ***access_token* is an access token**: by white box inspecting Facebook SDK's source code, we find that this is used as an access token in SDK and delivered as an access token to the app. Hence, we conclude this is an access token.

After this stage, we have figured out the semantics in the network traffics and have extracted the refined protocol for Facebook Login.

V. Protocol Modeling

During the protocol modeling, we model the protocol derived from the extraction step. We first specify the derived protocol into an intermediate representation which is close to the input language, such that we can bridge the gap between the implementation and the formal language. Then, we translate the intermediate representation into a protocol formal model. Third, we model the attacks that may compromise the protocol implementation on Android in three attacker models. Fourth, we specify the security properties for the protocol formal model.

A. Intermediate Representation

We present the refined protocol for Facebook Login including the communication between Facebook SDK and third party app in Fig. 3. The refined protocol indicates that Facebook Login process is divided into two phases. The first phase is authentication process started by user initiating login and ended by SDK getting the cookie credentials. The second phase is authorization process started from the request after the redirect response and ended by SDK receiving the access token.

In order to ease the protocol modelling, we translate the protocol in Fig. 3 into an intermediate representation shown in Table I. We intuitively introduce the translation method as follows. For every communication message in Fig. 3, the sender of that message initiates `Send()` to send out the message which consists of the parameters and cookies used to complete the authentication and authorization process, while the receiver uses `Rec()` to receive this message. The first parameter of `Send()` is the receiver and the rest is the message to be sent. Similarly, the first parameter of `Rec()` specifies from whom to receive this message and the rest is the actual message received. Other keywords in Table I represent the state of the a principal. For example, `Verify_Cookies` represents the server state in which it is verifying the received cookies and `Require_User_Identity` means the server needs user identity information in order to proceed.

B. Modelling Facebook Login Protocol

To ease readers' understanding of our model, we present the overall structure of Facebook Login in Fig. 4 In the figure, the app refers to a third party application app on Android system; IdP_C refers to Facebook Login SDK and IdP_S refers to Facebook authentication and authorization server. We use

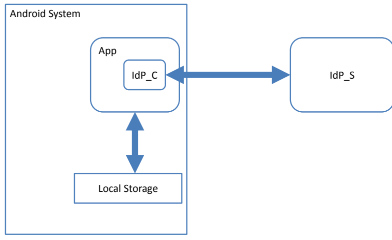


Fig. 4: Abstract Structure of Facebook Login Protocol

typed Pi-calculus as our modeling language. Generally, a typed Pi-calculus model consists of three main parts: declarations, queries and processes. Declaration part is used to declare channels, variables and functions; query part is used to specify the properties which a secure protocol should hold and the process part is used to model the main protocol logic. In the following, we introduce our modeling process.

Modeling Cryptographic Primitives. We model the cryptographic primitives that we use in modelling the communication channels used by participants as follows.

```
1  type public_key.
2  type private_key.
3
4  (*used to model the private channel between app and idpc*)
5  free app_pvKey: private_key [private].
6  free idpc_pvKey: private_key [private].
7  free idps_pvKey: private_key [private].
8  free localStorage_pvKey: private_key [private].
9
10 fun enc(bitstring, bitstring): bitstring. (*used to encrypt
       *)
11 reduc forall a: bitstring, b: bitstring;
12   dec(enc(a, b), b) = a. (*used to decrypt*)
13
14 fun get_public_key(private_key): public_key.
15 fun aenc(bitstring, public_key): bitstring. (*asymmetric
       encrypt*)
16 reduc forall message: bitstring, pri: private_key;
17   adec(aenc(message, get_public_key(pri)), pri) = message.
         (*asymmetric decrypt*)
```

First, we define the data types for public_key and private_key to model the asymmetric key (line 1 and 2). Then, we define the function that is used to generate a public key from private key in line 14. This function here is just used to associate a public key with a private key. It doesn't mean how the public key is generated in the real world scenario. In line 15 to 17, we define the functions for asymmetric encryption and decryption. We also define the symmetric cryptographic algorithm in line 10 and 11 used to encrypt the local variable to be tested its secrecy. In line 4 to 8, we define the private keys for each of the principal involved in the protocol. With these data types and functions, we can model the asymmetric cryptography in our model.

Modeling Protocol Participants. In our models, each principle is represented by a separated process. We also model the Local Storage into a process because from the formal model perspective, it also acts independently and can be access by other apps in the system. In this case, there are totally the following four processes defined in the protocol model.

```
1  (*model modified for display here*)
2  (*App Process*)
3  let app =
4    out(appAndIdpc, clickOnLogin);
5    out(appAndIdpc, email, password);
6    out(appAndIdpc, confirm_OK);
7    in(appAndIdpc, m_accessToken: bitstring);
8    out(localStorageChannel, m_accessToken); 0.
```

This app process takes in login, email and password as parameters which stand for user clicking the login button, typing in user's email and password event. In the process's body, there are five steps and we omit the steps that IdP_C displays the login form and authorization form to user for simplicity of model. `appAndIdpc` stands for the channel between app and IdP_C. First, app requests on behalf of user to login. Then, user types in email and password. After that, user clicks to confirm authorization and app receives the access token sent back by the IdP_C. 0 stands for the end of the process.

```
1  (*IdP_C Process*)
2  let idp_c =
3    in(appAndIdpc, m_clickOnLogin: bitstring);
4    out(idpcAndIdps, initial_request);
5    in(idpcAndIdps, m_login_form: bitstring);
6    in(appAndIdpc, m_2: bitstring);
7    out(idpcAndIdps, em, ps);
8    in(idpcAndIdps, m_3: bitstring);
9    out(localStorageChannel, cookie_xs);
10   out(localStorageChannel, cookie_c_user);
11   out(idpcAndIdps,auth_request, cookie_c_user, cookie_xs);
12   in(idpcAndIdps, m_4: bitstring);
13   in(appAndIdpc, m_5: bitstring);
14   out(idpcAndIdps, cookie_c_user, cookie_xs, anti_CSRF_token
         );
15   in(idpcAndIdps, m_token: bitstring);
16   out(appAndIdpc, token); 0.
```

IdP_C process performs as a relay between the app and IdP_S. It receives user's SSO login request and forwards it to IdP_S (line 1 and 2); after that, IdP_C receives login form from IdP_S and user's input credentials and then IdP_C sends user's credentials to IdP_S for identifying user's identity (line 4, 5 and 5); after that, IdP_C gets the cookies representing user's logged in status and stores them in local storage (line 8 and 9); line 10 to 13 stand for the process in which IdP_C first receives the user's authorization confirmation and delivers the confirmation to IdP_S, and then receives the access token back from Idp_S; finally, line 14 and 15 represent that IdP_C delivers the access token to the app.

```
1  (*IdP_S Process*)
2  let idp_s =
3    in(idpcAndIdps, ms_1: bitstring);
4    out(idpcAndIdps, generated_login_form);
5    in(idpcAndIdps, ms_3: bitstring);
6    let (=user_email, =user_password, dumb:bitstring) = adec(
         ms_3, idps_pvKey) in
7    let xs = make_xs(c_user) in
8    out(idpcAndIdps, xs);
9    in(idpcAndIdps, ms_4: bitstring);
10   let (=auth_request, =c_user, =xs) = adec(ms_4, idps_pvKey)
          in
11   out(idpcAndIdps, confirm_form);
12   in(idpcAndIdps, ms_5: bitstring);
13   let (=c_user, =xs, =antiCSRFToken) = adec(ms_5, idps_pvKey
         ) in
14   let token = make_token(c_user) in
15   out(idpcAndIdps, token); 0.
```

IdP_S process first receives user's login request and replies the login form asking for user's credentials (line 2 and 3); in line 4 to 7, after receiving and verifying user's identity, IdP_S generates the cookie representing user's logged in status and sends it back to IdP_C; later in line 8 to 12, IdP_S verifies the user's identity by the cookie xs and asks user to confirm authorization; finally in line 13 and 14, IdP_S generates the access token and sends it back to IdP_C.

```
1  (*Local Storage Process*)
2  let localStorage =
```

```
    in(localStorageChannel, sd_cookie_xs:bitstring);
    in(localStorageChannel, sd_cookie_c_user:bitstring);
    in(localStorageChannel, sd_accessToken:bitstring);0.
```

localStorage process mainly receives and saves the credentials from app process and IdP_C process in local storage.

Modeling Communication Channels. We use the in/out channels in typed Pi-calculus to model the communication channels. However, during our modelling, we meet a major challenge. Proverif doesn't stop verifying a model for a very long time if we use private channel to model the communication channels. Later we find a solution to it which is modelling the channels as public channels instead of private channels. However, this introduces another problem that breaks our second assumption in section III. Because the channels are public, the attackers can get the plain text in this case. Therefore, we need to encrypt the content which is sent through the public channel just like in real world case of https channel.

We create for each principle a public key and a private key. Before one principle wants to communicate with another, it first encrypts the message with public key of the other principle it wants to communicate with. Then when the other principle receives the message, it decrypts the message using its own private key.

We model all the three communication channels in our model using public encrypted channel. The following code shows the channels and the functions used to encrypt and decrypt the messages.

```
free appAndIdpc: channel. (*channel between app and
    idp_client*)
free idpcAndIdps: channel. (*channel between idp_c and idp_s
    *)
free localStorageChannel: channel.
```

These are the public channels defined for the communications among the principals. By using the asymmetric cryptographic primitives defined earlier, we model the public encrypted channels.

C. Modeling Attackers

After we get the basic model, we add the attackers into the basic model and create three attacker models, under the assumptions and scope aforementioned in section III. We consider the attackers to exist in different parts of the path through which credentials are transmitted. They are summarized below.

Network Attacker. The circle Z represents the adversary in real world. In this model, the attacker eavesdrops on the channel between IdP_C and IdP_S. And this model is similar to Man-in-the-Middle attacker model. The adversary can intercept, drop and replay the network messages on the channel. This attacker model is shown in Fig. 5a. To model this attacker, we send out a private global variable encrypted with the corresponding credential to a public channel where the adversary can get the messages.

Malicious SP_C Attacker. The circle represents the Malicious App in this attacker model. In this model, the attacker can communicate with IdP_C as normal app. It can access the local storage and access whatever data that belongs to it. But it can not intercept on the communication channel between IdP_C and IdP_S. This attacker model is shown in Fig. 5b. To model this attacker, we send out whatever credentials the SP_C app receives to the public channel. And to the end of the model, we also send the credentials in local storage to that public channel in order to model the malicious SP_C communicates with and sends back the credentials to the attacker.

TABLE I: Abstracted Protocol for App, IdP_C and IdP_S

App Protocol	IdP_C Protocol	IdP_S Protocol
Send(Idpc, login)	Rec(App, command)	
	Send(Idps, auth req)	Rec(Idpc, req)
		Require_User_Identity
	Rec(Idps, redirect, intention)	Send(Idpc, redirect, login)
	Send(Idps, login)	Rec(Idpc, command)
	Rec(Idps, loginForm)	Send(Idpc, loginForm)
Rec(Idpc, loginForm)	Send(App, loginForm)	
Send(Idpc, userCredential)	Rec(App, userCredential)	
	Send(Idps, userCredential)	Rec(Idpc, userCredential)
		Verify_User_Credential
	Rec(Idps, redirect, intention)	Send(Idpc, redirect, authr)
	Send(Idps, authr)	Rec(Idpc, authr)
		Verify_Cookies
	Rec(Idps, authrForm)	Send(Idpc, authrForm)
Rec(Idpc, authrForm)	Send(App, authrForm)	
Send(Idpc, userConsent)	Rec(App, userConsent)	
	Send(Idps, userConsent)	Rec(Idpc, userConsent)
		Verify_Cookies
	Rec(Idps, access token)	Send(Idpc, access token)
Rec(Idpc, access token)	Send(App, access token)	

Malicious App in System. This attacker model is shown in Fig. 5c. Z represents a malicious app that user installed in the system. It can communicate with the benign app. This attacker model can be further divided into two sub models:

- **App with root privilege.** In this model, Z can access to the local storage and access to all the data including app's data. But since the system is still benign, it can not change the system's behavior.
- **App without root privilege.** In this model, Z can also access to Local Storage, but it can only access to its own data in it.

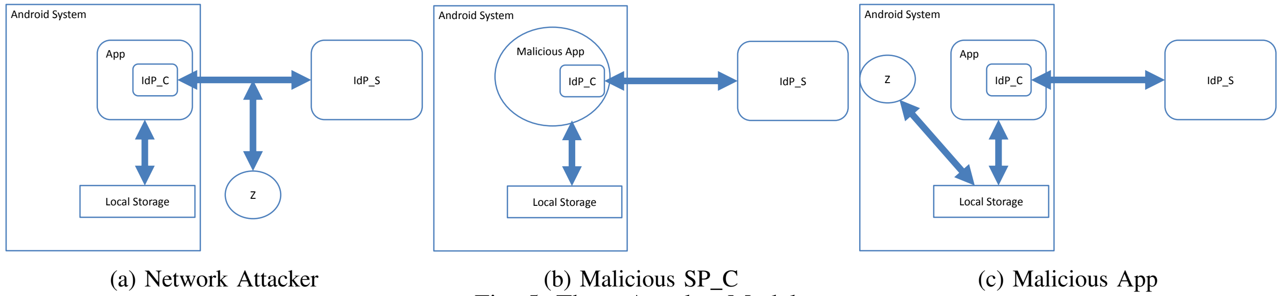
In the actual modelling, we model these two scenarios explicitly into two separated models. For convenient illustration, Fig. 5c represents these two scenarios. Due to the length limits, we are not going to show the source code of our models here but release our source code online [21].

D. Security Properties

The next step is to identify the security properties for the protocol. Security properties include three kinds of properties. The first kind is the authentication properties; the second is the authorization properties and the third is secrecy properties.

Authentication Property. Authentication means that the two interlocutors communicating to each other are sure about the identity of one another that they are talking to [22]. It can be achieved if principal A finished a protocol with principal B and A believes that it has finished the protocol with B which indeed it is (vice versa) [13].

Authorization Property. Since Facebook Login adds authorization feature to third party app to access user's account in its implementation which is not specified by traditional SSO protocol, we have to consider the authorization properties



(a) Network Attacker (b) Malicious SP_C (c) Malicious App

Fig. 5: Three Attacker Models

in this protocol. Particularly, authorization refers to if the user thinks he has authorized an app to access his Facebook account, he has really done so (vice versa).

Secrecy Property. Secrecy should also be achieved in Facebook Login protocol and the credentials should not be available to adversary. Otherwise, adversary can abuse the credentials and destroy the authentication properties. In Proverif, secrecy is checked with `query attacker(secret)`. As shown in Table II that the secrecies of `email`, `password`, `xs` and `access_token` are checked in our model.

Authentication and authorization in Facebook Login protocol are similar as they are both initiated by user (user provide credentials to login and user provide consents to authorize, server return credentials), they can be checked in similar method. Because there are two phases in Facebook Login protocol, one is the authentication phase, the other is the authorization phase, in order to check mutual authentication and authorization, we need 8 events for it (4 each for a mutual authentication and an authorization). The 8 events are shown in Fig. 3 with triangles indicating the occurrences of them. They are used to indicate the beginning and end of the authentication and authorization processes as their names indicate. The first 4 events are for checking authentication properties: clientStartLogin event indicates client starts the login procedure just before he sends message 8 out; serverAcceptLogin indicates the server accepts the login request after receiving message 8 and before sending out message 9; clientFinishLogin indicates that client finishes the login process after receiving message 9 but before sending out message 10 and serverFinishLogin event indicates server finishes user logging in procedure after receiving message 10. While the later 4 events are for checking authorization properties and their meaning are very similar to the authentication events and it should be easy to understand.

With these eight events, we define 4 queries for mutual authentication in login phase and authorization phase. They are summarized in Table II. Queries 6 and 7 are not injective queries because a server can accept login or authorization requests from multiple clients at the same time. Therefore their relationships are not one-to-one mapping, while the rest two queries should be in one-to-one manner.

VI. Checking and Result Analysis

In this step, we use Proverif to check the protocol model against the specified attacker models to analyze the satisfiability of the properties. Proverif generates a verification report after each checking. If any violation is reported, we manually examine the report and apply our domain knowledge to construct a concrete attack. Then we confirm the attack by dynamically testing the server and check whether it is a true positive or not. In this way, we have successfully identified a major vulnerability in the model built and we have successfully reproduced the vulnerability in real world device. In this section, we illustrate the details of our checking.

A. Checking against Network Attack

The verification results are shown in Table II on the fourth column. All of the secrecy properties for network attacker are proved to be true which means although the attacker can obtain and manipulate the network traffic, he can't decrypt the message of the traffic to get the credentials. User's credentials are secure using Facebook Login under the network attacker.

As for the authentication and authorization properties, we can see from the table that in all the scenarios, the verification results are the same. The falsities of the sixth and seventh properties mean that for every clientFinishLogin or clientFinishAuthr event that happens, there are might be no serverAcceptLogin or serverAcceptAuthr event that happened previously. This implies there exists a replay attack vulnerability in this protocol.

However, after carefully analyzing the verification result, we regard it as a false positive because of the channel we use in our model. In the model, we use an encrypted public channel and thus the communication can be intercepted by the attacker. Moreover, the attacker in this model can pretend to be the Facebook server which responses the requests from IdP_C. However, the https communication protocol in real world guarantees that the adversary can not eavesdrop on the channel or steal the messages in it and the attacker can't pretend to be somebody else. Nevertheless, this is the best choice for modelling the channel and if not using public encrypted channel, the model verification process does not terminate within a reasonable amount of time.

For the rest of two queries, although the first one can not be proved either true or false, but Proverif dose give us report that `inj-event(serverFinishLogin) ==> event(clientStartLogin)` which is not the one-to-one relationship is instead can be proved to be true indicating that Facebook server would not be cheated to login a fake user as it does need a real user to start login process in advance for it to finish the login process with a user.

The one-to-one relationship that for every server finishes authorization event, there is a client started authorization event is proved to be true as in the last correspondence query. It is reasonable because the final authorization is protected by an anti CSRF token in the previous responded user confirmation

TABLE II: **Queries and results summary**. N.A. Res. represents for network attacker results; M_SPC Res. represents for malicious SP_C results; M.A.N. Res. represents for malicious app without root privilege results; M.A.R. Res. represents for malicious app with root privilege results;and Cn.P. represents for can't be proved.

No.	Query		N.A. Res.	M_SPC Res.	M.A.N. Res.	M.A.R. Res.
1	Secrecy	`not attacker(email)`	True	False	True	True
2		`not attacker(password)`	True	False	True	True
3		`not attacker(xs)`	True	False	True	False
4		`not attacker(access_token)`	True	False	True	False
5	Authentication	`inj-event(serverFinishLogin) ==> inj-event(clientStartLogin)`	Cn.P.	Cn.P.	Cn.P.	Cn.P.
6		`inj-event(clientFinishLogin) ==> event(serverAcceptLogin)`	False	False	False	False
7	Authorization	`inj-event(clientFinishAuthr) ==> event(serverAcceptAuthr)`	False	False	False	False
8		`inj-event(serverFinishAuthr) ==> inj-event(clientStartAuthr)`	True	True	True	True

form. Moreover, the server also requires user's cookie for the authorization step. We also notice that if the email and password of user are known by the adversary, then he can pretend to be the client and finish authorization process with server which will violate the third query as the login process is not protected like in the authorization process. However, it is shown in the first two reachability queries in Table II that if the user uses his credentials correctly, it is not possible for the adversary to know the user's credentials.

B. Checking against a Malicious SP_C App

The fifth column of Table II shows the results of malicious SP_C attacker model. Different from the network attacker model, the secrecy queries of this model are all proved to be false, which indicates that if an app is malicious, all the data a user enters and saves in that app is not safe. This is actually true in real world. If a user enters his credentials in the malicious app, his credentials entered can be recorded and be sent to the attacker. After stealing the credentials, the app functions as a normal app and uses user's credentials to login to Facebook. If the credentials are correct, the malicious app can then record the cookie returned from Facebook.

To one's surprise, the Authentication properties of this attacker model are proved exactly the same as in the network attacker model. One might think that if the user's credentials are compromised, then the attacker can pretend to be the user and he can log in to Facebook. Therefore all the authentication properties should proved to be false. However, the falsity of the sixth and the seventh queries are because of the problem of modelling channel and even if the attacker have the user's credentials, the event of for every serverFinishAuthr, there is a previous event clientStartAuthr that happens is also true. Because the authorization process are protected by an anti-CSRF token and even the attacker abuses the user's credentials to login and authorize, these two events are also in one-to-one manner and Facebook server thinks the attacker is the user.

Again, Proverif can't prove the query for every serverFinishLogin, there is a previous event clientStartLogin happened to be either true or false but instead can prove for every serverFinishLogin, there is at least one clientStartLogin event happened previously to be true. Even the attacker has user's credentials, it won't change the outcome of this query because if the attacker wants a serverFinishLogin event to happen, he must pretend to be the user and initiates a clientStartLogin event unless he also compromises Facebook server.

C. Checking against a Malicious App

This model consists of two sub-models as whether the malicious app has root privilege or not.

Malicious App without Root Privilege. The verification results for the sub-model are the same as in the results of network attacker model. This is not a surprise as is compared to the network attacker model, the information attacker can obtain is even less than in network attacker model. Because attacker in this model can not access the local storage which contains user's credentials and he can not intercept and decrypt the message sent from IdP_C to IdP_S.

Malicious App with Root Privilege. Different from the results in app without root privilege sub-model, the email and password of the user are safe while the credentials `xs` and `access_token` can be obtained by the attacker which are indicated from results of the latter two secrecy queries. After careful inspection on the counterexample, we think this indicates a serious flaw in model. In this attacker model, the IdP_C would store the credentials in the local storage including the cookies and access token. Although the access token is reasonable to be stored locally, it is not correct for the cookies to be stored locally. Because the cookies belong to Facebook SDK which should not be accessed by a third party app and this violates SOP. With root privilege, an third party app can have access to other app's private storage and when the cookies belong to Facebook are stored in local storage without being encrypted, it is possible for an attacker to get this credential and have access to user's Facebook account.

Identified Vulnerability. The verification result of cookie `xs` in malicious app attacker model shows that the cookies can be known by the adversary. Based on our domain knowledge, this is a vulnerability as the browser uses cookie `xs` as a proof to show to server an authenticated session of a user. If this cookie is stolen by the adversary, he can use it to login to user's Facebook account. The `access_token` is also shown to be known by the adversary when a malicious app has root privilege which enables the adversary to use the `access_token` to access to whatever the user has authorized the third party app to access.

Attack Confirmation. We reconstruct the attack on a real device and we confirm that it is possible for a malicious app to steal these two sensitive credentials from the storage in the mobile phone. Firstly, we build a dummy app using the Facebook Login and we authorize the

app with public profile permission. Then we used adb tool kit with root privilege to access to storage of the mobile phone. We successfully locate the cookies `c_user` and `xs` as well as the credential `access_token`. The cookies are stored in an sqlite database in mobile phone's storage at path `/data/data/<Apps package name>/databases/webviewCookiesChromium.db` and the `access_token` is stored in an `xml` file at path `/data/data/<App's package name>/com.facebook.AuthorizationClient.WebViewAuthHandler.TOKEN_STORE_KEY.xml`.

Using the cookies, we successfully login to user's Facebook account on a browser and browse through the victim's posts. With `access_token` obtained, we manually craft a request using our proxy to Facebook server asking for the resource that user has granted. Facebook server returned with user's public profile the user granted which shows the `access_token` stolen by adversary would render user's data in danger.

VII. Related Work

Most of the prior research focused on SSO on desktop or in web-based browser environment, few did research to the SSO implementation on mobile environment. To our best knowledge, we are the first to perform formal analysis to SSO implementation on mobile platform.

SSO Analysis. In Wang et al.'s paper [6], they studied about different SSO implementations mainly on desktop browser environment. But different from our work, they didn't perform formal analysis to the implementation and they were not on mobile platform. The most related work was done by Chen et al. in this paper [23]. They analyzed OAuth from a mobile developer's perspective and found flaws in OAuth protocol. However, the most different between our work and theirs is we performed formal analysis while they mainly study the applications manually.

Formal Analysis. In Wang et al.'s another paper [7], through formally analyzing the implementations of different SDKs, they summarized the assumptions of using these SDKs. When overlook these assumptions, there might be security issues for adversaries to exploit. However, they mainly performed the formal analysis to the desktop or web SDKs but not the SDKs for mobile environment. In Bai et al.'s work [17], they took one step further and try to build a platform tool to perform an automatically protocol extraction from raw network traffic. Nevertheless, different from this work, their work focuses more on the platform tool building and protocol extracting process but didn't formally analyze protocols on mobile platform.

VIII. Conclusion

We perform a formal analysis to Facebook's implementation of its SSO service, Facebook Login SSO service, for Android. Through this formal analysis, we identify a major vulnerability in it and later successfully reconstruct an attack in a real device. We also point out explicitly that it is suitable to carry out a formal analysis to existing SSO implementations for mobile. Although our research is derived from Android, we believe that the method used in this paper can also be easily applied to other mobile platforms.

Acknowledgement

This research is partially supported by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate.

References

[1] "Single sign-on," Apr 2015. [Online]. Available: http://goo.gl/i362Qt

[2] "Security risk as people use same password on all websites," Sep 2009. [Online]. Available: http://goo.gl/OJO7eK

[3] B. Darwell, "Facebook platform supports more than 42 million pages and 9 million apps," Apr 2012. [Online]. Available: http://goo.gl/pqSKPZ

[4] D. Cohen, "Login with facebook update: Apps must now separately request permission to post on behalf of users," Aug 2013. [Online]. Available: http://goo.gl/3AVzJ9

[5] "Facebook login overview," Apr 2015. [Online]. Available: https://goo.gl/WeLrBV

[6] R. Wang, S. Chen, and X. Wang, "Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services," in *S&P, 2012*.

[7] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich, "Explicating sdks: Uncovering assumptions underlying secure authentication and authorization." in *USENIX Security, 2013*.

[8] C. Bansal, K. Bhargavan, and S. Maffeis, "Discovering concrete attacks on website authorization by formal analysis," in *CSF, 2012*.

[9] A. Barth, "The web origin concept," RFC 6454, Dec 2011. [Online]. Available: http://goo.gl/bldZ9Q

[10] M. Zelwski, *Browser Security Handbook, part 2*, 2011. [Online]. Available: https://goo.gl/Vi3dGO

[11] G. Bai, J. Sun, J. Wu, Q. Ye, L. Li, J. S. Dong, and S. Guo, "All your sessions are belong to us: Investigating authenticator leakage through backup channels on android," in *ICECCS, 2015*.

[12] M. D. Ryan and B. Smyth, "Applied pi calculus," in *Formal Models and Techniques for Analyzing Security Protocols*, V. Cortier and S. Kremer, Eds. IOS Press, 2011. [Online]. Available: http://goo.gl/acRXtl

[13] B. Blanchet, B. Smyth, and V. Cheval, "Proverif 1.86: Automatic cryptographic protocol verifier, user manual and tutorial," *INRIA Paris-Rocquencourt, LSV, ENS Cachan & CNRS & INRIA Saclay IIle-de-France, Paris, France*, 2013.

[14] "Android and ios squeeze the competition, swelling to 96.3% of the smartphone operating system market for both 4q14 and cy14, according to idc," Feb 2015. [Online]. Available: http://goo.gl/HsfU8z

[15] J. Oberheide and F. Jahanian, "When mobile is harder than fixed (and vice versa): demystifying security challenges in mobile environments," in *ACM HotMobile, 2010*.

[16] H. Bagheri, E. Kang, S. Malek, and D. Jackson, "Detection of design flaws in the android permission protocol through bounded verification," in *FM, 2015*.

[17] G. Bai, J. Lei, G. Meng, S. S. Venkatraman, P. Saxena, J. Sun, Y. Liu, and J. S. Dong, "Authscan: Automatic extraction of web authentication protocols from implementations." in *NDSS, 2013*.

[18] "Proverif: Cryptographic protocol verifier in the formal model," Jul 2015. [Online]. Available: http://goo.gl/eqjSPX

[19] "Intents and intent filters," Apr 2015. [Online]. Available: http://goo.gl/uuyY5z

[20] "<data>," Oct 2015. [Online]. Available: http://goo.gl/dY8o3w

[21] "Android_sso_model," May 2015. [Online]. Available: https://goo.gl/3Lb5xm

[22] "User authentication with oauth 2.0," Apr 2015. [Online]. Available: http://goo.gl/1CljVW

[23] E. Y. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague, "Oauth demystified for mobile application developers," in *CCS, 2014*.