UCLA
UCLA Electronic Theses and Dissertations

Title

Automated Detection and Mitigation of Vulnerabilities in Single Sign-on Implementations

Permalink

https://escholarship.org/uc/item/2dq2v5vk

Author

Rahat, Tamjid Al

Publication Date

2024

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA

Los Angeles

Automated Detection and Mitigation of Vulnerabilities in Single Sign-on Implementations

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Electrical and Computer Engineering

by

Tamjid Al Rahat

2024

© Copyright by
Tamjid Al Rahat
2024

ABSTRACT OF THE DISSERTATION

Automated Detection and Mitigation of Vulnerabilities
in Single Sign-on Implementations

by

Tamjid Al Rahat
Doctor of Philosophy in Electrical and Computer Engineering
University of California, Los Angeles, 2024
Professor Christina Panagio Fragouli, Chair

Single Sign-on (SSO) protocols play a critical role in operating secure systems, granting access and privileges to authenticated entities. However, logical mistakes in their implementation that violate the standard specification expose them to various security threats that can compromise the confidentiality, integrity, and availability of sensitive user resources. SSO protocols are complex as they involve interactions between multiple parties (e.g., service providers, relying parties, and users) and such interactions result in diverse security requirements for different platforms. Unfortunately, the standard specification is written in hundreds of pages of English documents and the security properties are often not well-defined, leading the protocol developers to make severe mistakes. These mistakes can completely compromise the security of the protocol and they have been exploited many times in recent years to conduct severe attacks, including identity theft and complete account or system takeover.

In this dissertation, we aim to identify, analyze, and mitigate security threats and vulnerabilities that impact the SSO-based authorization and authentication protocols. To achieve this goal, we incorporate static analysis, formal verification, and program synthesis techniques to design and implement a toolchain to 1) identify logical security errors in the client and server-side implementations of the SSO protocols, 2) provide safety guarantee with respect to the standard specification, and 3) automatically generate patches for the

detected buggy implementations. We further propose specification languages to allow developers to formally express the SSO security requirements that can be checked across large-scale SSO implementations on diverse platforms.

We evaluate our proposed approaches on 600 widely used relying party applications and the 25 most popular SSO service providers. Our experiments uncover 40 previously unknown vulnerabilities (i.e., zero-days), which have been acknowledged and fixed by the developers after our responsive disclosure. Furthermore, eight classes of these vulnerabilities lead to new entries in the Common Vulnerabilities and Exposures (CVE) database.

The dissertation of Tamjid Al Rahat is approved.

Todd D Millstein

Nader Sehatbakhsh

Yuan Tian

Christina Panagio Fragouli, Committee Chair

University of California, Los Angeles

2024

To my mother...

Rest in peace, Ammu (1971-2019)

TABLE OF CONTENTS

1 Introduction **1**

1.1 Contributions 3

1.2 Dissertation Structure 6

2 Background and Related Work **7**

2.1 Preliminaries on Single Sign-On (SSO) protocols 7

2.1.1 OAuth 1.0 8

2.1.2 OAuth 2.0 9

2.1.3 OpenID Connect 12

2.2 Related Work 13

3 Security Analysis of OAuth Relying Party **21**

3.1 Introduction 21

3.2 Overview 24

3.2.1 System Overview 24

3.2.2 Threat Model 28

3.3 Analysis 28

3.3.1 Datalog Preliminaries 28

3.4 Anti-protocols in OAuth 30

3.4.1 An Anti-protocol Language for OAuth 30

3.4.2 Common Anti-protocols in OAuth 30

3.5 Experimental Evaluation 37

3.5.1 Data Collection 37

3.5.2 Evaluation Results 37

3.5.3 Case Studies . . . 40
3.6 Best Security Practice for OAuth Relying Party . . . 42
3.7 Limitations . . . 43
3.8 Summary . . . 43

4 Security Analysis of OAuth Authorization Server . . . 44
4.1 Introduction . . . 44
4.2 Overview . . . 49
4.3 System Design . . . 51
4.3.1 Code Representation . . . 52
4.3.2 Facts and Inference Rules for SDG . . . 53
4.3.3 Query Language . . . 57
4.4 Evaluation . . . 61
4.4.1 Experimental Setup . . . 62
4.4.2 Evaluation Results . . . 64
4.5 Discussion . . . 72
4.6 Summary . . . 73

5 Safety Verification of OpenID Connect . . . 75
5.1 Introduction . . . 75
5.2 Overview . . . 79
5.2.1 Threat Model . . . 79
5.2.2 Motivating Example . . . 80
5.3 System Design . . . 84
5.3.1 Specification Language . . . 84
5.3.2 Program Translation . . . 86

5.3.3 AuthLTL to Regular LTL . . . 88
5.3.4 Safety Verification . . . 90
5.3.5 Implementation . . . 96
5.4 Evaluation . . . 97
5.4.1 Experimental Setup . . . 98
5.4.2 Experimental Results . . . 100
5.4.3 Developer Acknowledgment . . . 104
5.5 Summary . . . 105

6 Automated Repair of OpenID Connect Bugs . . . 107
6.1 Introduction . . . 107
6.2 System Design . . . 110
6.2.1 Overview . . . 110
6.2.2 Bug Localization . . . 111
6.2.3 Candidate Generation . . . 114
6.2.4 Candidate Validation . . . 117
6.3 Evaluation . . . 123
6.3.1 Experimental Setup . . . 124
6.3.2 Experimental Results . . . 125
6.4 Summary . . . 127

7 Conclusion . . . 128

References . . . 130

LIST OF FIGURES

2.1 Overview of OAuth 1.0. 8

2.2 Flow of authorization code (AC) grant for OAuth. 10

2.3 Overview of implicit flow of OAuth 2.0. 10

2.4 Proof Key for Code Exchange (PKCE), an extension of authorization code grant to provide more security benefits for client applications. 12

2.5 Interactions between the Relying Party (RP) and OpenID Provider (OP) during the authentication process of Authorization Code Flow in OIDC. . . 13

3.1 Overview of *OAuthlint*. 25

3.2 Rules for computing the `FlowTo` predicate. Here, the `alias` predicate is directly obtained from Flowdroid. 29

3.3 Rules for computing the OAuth-specific predicates. 30

3.4 The Anti-protocol language in *OAuthlint*. 31

3.5 Vulnerability of using Webview for OAuth transactions. 33

3.6 Number of applications with up to 5 vulnerabilities. 40

4.1 Code example for an authorization endpoint that is vulnerable to the redirect URI manipulation during the authorization flow. 51

4.2 Overview of *Cerberus*. Here, Automata-based slicing is an optimization step discussed in Sec. 4.3. 52

4.3 (A) System Dependence Graph (SDG) representation for the code example in Figure 4.1 and (B) graphical representation of the query for the Redirect URI property (P1). 53

4.4 Performance comparison for libraries in $\texttt{DataSet}_U$ (Table 4.5) based on average query execution time (in log scale) between *Cerberus*. 70

5.1 Motivating code example of an OIDC relying party incorrectly validating the `id token` received as a response from the provider during the authentication flow. 80

5.2 Schematic workflow of *AuthSaber*. 82

5.3 Syntax of AuthLTL specification. 84

5.4 Transformation of AuthLTL formula φ to its equisatisfiable regular LTL φ' while transforming the original program $\mathcal{P}$ to $\mathcal{P}'$. 89

5.5 Automata representation of the authentication flow constraints for the implicit flow defined (in English) by the specification. 91

5.6 Authentication flow validation approach implemented by *AuthSaber* which models the interaction between the entrypoints of each participating OIDC entities as IDG. 92

6.1 Bugs in line 3 and line 15 in a popular OpenID Connect library Pac4j. Fixes are shown in line 4, which adds a new clause in the conditional expression, and in line 16 where the method invocation is replaced with a new one. . . . 110

6.2 Overview of our repair approach, which takes a buggy OpenID code and specification (in English and Petri net format) as inputs, and generates a valid repaired program as output. 112

6.3 An example of an LLM query for bug localization in a partial OpenID Connect program. This query has four text components Q = [q_1, q_2, q_3, q_4], where $q1$ and q_4 are static components that provide instructions to the LLM agent and q_2 and q_3 are dynamic components that provide the specification and buggy program, respectively. 113

6.4 Output from LLM agent in response to the query in Figure 6.3 113

6.5 Simplified grammar used for the repair expressions. 116

6.6 A simple Petri net with four places (circles) and two transitions (rectangles). 117

6.7 Result of Petri net in Figure 6.6 when transition T_1 is fired. 118

6.8 (A) A Petri net example (simplified) for the code example in Figure 6.1, and (B) corresponding specification Petri net provided by users. 119

LIST OF TABLES

3.1 Top Android applications using OAuth 38

3.2 OAuth vulnerabilities in top android applications 40

3.3 Statistics of top OAuth APIs in our evaluation 41

4.1 Comparison with existing OAuth analysis tools. 46

4.2 We define ten security properties to detect the most commonly observed logical flaws on OAuth servers based on the standard OAuth specifications 56

4.3 Properties (P1-P5) encoded in our query language 58

4.4 Properties (P6-P10) encoded in our query language 59

4.5 Statistics for the open-source OAuth server libraries in $\texttt{DataSet}_U$. Here, AC and I indicate the OAuth's authorization code grant and implicit grant, respectively. 63

4.6 Evaluation results of *Cerberus* on $\texttt{DataSet}_K$. 65

4.7 Evaluation results of *Cerberus* on $\texttt{DataSet}_U$ whose vulnerabilities are unknown. 66

4.8 Comparing *Cerberus* against *OAuthlint* in terms of average running time using the $\texttt{DataSet}_U$ and $\texttt{DataSet}_K$. 68

5.1 OIDC components and relevant details that we consider to gather the safety properties for our evaluation. 98

5.2 Properties within the scope of ID Token validation of OpenID Connect. . . . 99

5.3 Popular OIDC libraries selected for evaluating *AuthSaber*. Among these, eleven libraries are implemented in Java and four are implemented in JS. 100

5.4 Verification results for the popular OpenID Connect implementations. In all 15 OIDC libraries, we successfully verified 134 properties and found 16 confirmed security violations that were previously unknown. We validated and responsively disclosed all violations to the respective developers. 100

5.5 Verification results for the collected safe benchmarks that were patched by the developers based on previously reported violations. For this evaluation, we exclude the libraries for which no previous violations are found, or the reports are no longer publicly accessible. 101

5.6 Comparison of *AuthSaber* with existing tools in SSO domain in terms of the number of OIDC safety properties that can be expressed and checked. 102

6.1 Statistics of our dataset of OpenID bugs which are collected from libraries studied in Chapter 5 and from publicly available bug reports for other OpenID projects (OpenID, 2023). 124

6.2 Evaluation results of *AuthFix* for OpenID bugs repair in our dataset. Column #Fix shows the correct and plausible fixes generated by *AuthFix* and column $\#\text{Fix}_{dev}$ shows the number of generated fixes that are semantically equivalent to the manual patches written by the developers (if available). 125

6.3 Comparison between the repairs generated by *AuthFix* (column $\#\text{Fix}_{AuthFix}$) and repairs generated by using only LLM (column $\#\text{Fix}_{LLM}$). 126

ACKNOWLEDGMENTS

Chapter 3 of this dissertation is a version of Tamjid Al Rahat, Yu Feng, and Yuan Tian. "Oauthlint: An empirical study on OAuth bugs in Android applications." In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 293-304. IEEE, 2019. **Chapter 4** is a version of Tamjid Al Rahat, Yu Feng, and Yuan Tian. "Cerberus: Query-driven scalable vulnerability detection in OAuth service provider implementations." In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, pp. 2459-2473. 2022. **Chapter 5** is a version of Tamjid Al Rahat, Yu Feng, and Yuan Tian. "Automated Safety Verification of OpenID Connect Programs." which is in preparation for publication (i.e., *accepted* to be published) in the Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security.

VITA

2022–2024 Ph.D. Candidate in Electrical and Computer Engineering, University of California Los Angeles

2018–2022 M.S. in Computer Science, University of Virginia

2011–2016 B.S. in Computer Science and Engineering, Bangladesh University of Engineering and Technology

PUBLICATIONS

Tamjid Al Rahat, Yu Feng, Yuan Tian, AuthSaber: Automated Safety Verification of OpenID Connect Programs. *(To appear in) ACM Conference on Computer and Communications Security (CCS)*, 2024

Tamjid Al Rahat, Yu Feng, Yuan Tian, Cerberus: Query-driven Scalable Vulnerability Detection in OAuth Service Provider Implementations. *ACM Conference on Computer and Communications Security (CCS)*, 2022

Tanujay Saha, **Tamjid Al Rahat**, Najwa Aaraj, Yuan Tian, Niraj K. Jha, ML-FEED: Machine Learning Framework for Efficient Exploit Detection. *IEEE International Conference on Trust, Privacy and Security (TPS)*, 2022

Tamjid Al Rahat, Minjun Long, Yuan Tian, Is Your Policy Compliant? A Deep Learning-based Empirical Study of Privacy Policies' Compliance with GDPR. *ACM Workshop on Privacy in the Electronic Society (WPES)*, 2022

Tamjid Al Rahat, Yu Feng, Yuan Tian, OAuthLint: An Empirical Study on OAuth Bugs in Android Applications. *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, 2019

Tamjid Al Rahat, Arif Arman, Mohammed Eunus Ali, Maximizing Reverse k-Nearest Neighbors for Trajectories (MaxRkNNT). *29th Australasian Database Conference*, 2018

CHAPTER 1

Introduction

Single Sign-On (SSO) protocols such as OAuth (OAuth2, 2012) and OpenID Connect (OIDC) (OpenID, 2021) have gained widespread adoption across various industries (Naik and Jenkins, 2017; Hussain et al., 2020) in recent years, as organizations recognize their value in bolstering security, streamlining access management, and enhancing interoperability. These protocols play a pivotal role in the current digital ecosystem as they eliminate the need for users to manage multiple usernames and passwords while providing enhanced security through centralized authentication and authorization.

SSO protocols involve communication among multiple entities, including the relying party, service provider, and users. The protocol follows various flows, such as authorization code and implicit flow, depending on the platform and available resources to each entity. Upon successful authorization and authentication, the service provider issues an access token to the relying party, which can be exchanged with resource servers to access users' resources. However, due to the complexity of the specification, developers often make security mistakes when implementing each party's role in the protocol execution, leaving vulnerabilities in the system.

However, the security guarantees of these protocols are only as strong as the implementation across the multiple parties that enforce them. Unfortunately, the current security of the protocol holds mostly at the specification level (OAuth2, 2012; OpenID, 2021), whereas real protocol implementations in the wild often have logical errors leading to severe attacks (Microsoft, 2023a; Blackdirect, 2021; Facebook, 2018a). These attacks have resulted in significant financial losses (Microsoft, 2023b) for companies and organizations, totaling billions of dollars, while also exacerbating the privacy and security risks of their

users (Twitter, 2021), thereby putting them at greater risk. Thus, developing a toolchain that can automatically detect and mitigate security vulnerabilities in SSO protocol implementations is crucial to ensure robust security of identity and access management systems.

Prior research in analyzing the security of SSO protocols can be, in general, divided into three categories: 1) Network traffic analysis, 2) protocol design verification, and 3) client API analysis. Researchers have developed semi-automated tools, such as AuthScan (Bai et al., 2013) and SSOscan (Zhou and Evans, 2014) intercept the traffic between the relying party (RP) and Service Provider (SP), and analyzes the traffic to identify security issues in the protocol flows. However, traffic monitoring-based approaches often need an extensive manual setup and do not scale for large-scale analysis. Prior research further extends SSO protocol's security analysis using formal verification (Fett et al., 2017; Hammann et al., 2020) that concentrate on protocol design rather than its implementation. These approaches require manual translation of both protocol interactions and specifications into corresponding formal models, which can then be consumed by off-the-shelf verifiers like Tamarin Prover (Meier et al., 2013) or Alloy (Jackson, 2012). However, manually translating complex interactions within the real-world protocols' programs can be error-prone and inefficient, rendering them impractical for large-scale evaluations. Moreover, program analysis tools like S3kVetter (He et al., 2015a) analyzes the SSO APIs provided for the client-side applications while treating the SP blackbox. Since SSO protocols involve interactions between multiple parties, unfortunately, analyzing only the client applications prevents checking many security critical properties (e.g., redirect uri manipulation).

This dissertation delves into analyzing the implementations of most commonly used SSO protocols, examining the interactions and behavior between participating entities during protocol execution. The primary objective is to empower protocol developers and users with the ability to conduct thorough, end-to-end security analyses and mitigate any potential vulnerabilities with minimal manual effort.

Towards this goal, we design a series of automated and scalable methods to precisely reason about the security behavior of two most common and popular SSO protocols:

OAuth (OAuth2, 2012) and OpenID Connect (OpenID, 2021) that are implemented in diverse platforms. Specifically, we first propose automated static analysis methods to reason about the information flow across the RP applications and check compliance against the standard specifications (Chapter 3). We further validate the data- and control flow within the implementations of the authorization server of SSO protocols and propose a novel scalable analysis method to reason about the complex properties of SSO (Chapter 4). To provide a stronger guarantee of safety and security in both RP and SP implementations, we introduce an automated safety verification (Chapter 5) for SSO implementations across multiple parties. We propose an automated program repair approach (Chapter 6) to generate fixed SSO programs that satisfy all the requirements enforced by the standard specification.

1.1 Contributions

We now describe the contributions we make in this dissertation.

Analyzing OAuth relying parties against anti-protocols. We devise a simple but effective query language for expressing common anti-protocols that violate the OAuth specification. Given an anti-protocol expressed in our query language, we design a demand- and query-driven static analysis for checking violations in the mobile applications from the relying parties. We propose five anti-protocols that widely appear in the implementations of relying parties and implement our proposed approach in a tool called *OAuthlint*. Using *OAuthlint*, we further present an empirical study on the usage of OAuth APIs in mobile applications and their security implications.

Scalable security analysis of OAuth authorization servers. We formalize the OAuth specifications and security best practices for authorization servers. We design *Cerberus*, an automated static analyzer, to find logical flaws and identify vulnerabilities in the implementation of OAuth authorization server libraries. To efficiently detect security

violations in a large codebase of service provider implementation, Cerberus employs a demand-driven algorithm for answering queries about OAuth specifications. Our proposed algorithm is 20× faster and more efficient in detecting complex security bugs in popular authorization server implementations of OAuth.

Automated safety verification of OpenID Connect protocol. We design AuthLTL, a highly expressive specification language that is extended from the standard Linear Temporal Logic (LTL). AuthLTL augments standard LTL with more expressive OIDC predicates that allow developers to formally express the desired safety properties of the protocol, including the temporal relations between interactions and events involving multiple parties during the authorization and authentication flows. We propose a staged approach that iteratively constrains the space of valid candidates without compromising the assurance of the verifier. We design several customized transformations that convert OIDC implementations and their specifications to corresponding forms that can be analyzed by existing software model checkers. We implement our methods in a tool called *AuthSaber* which is capable of performing safety verification of real-world OpenID Connect implementations against their standard safety specification.

Mitigation of OpenID Connect bugs with automated repair. We present an automated program repair approach that generates fixed programs for identified bugs. Our approach leverages a combination of large language models (LLMs) and logical reasoning to identify faulty program statements and produce potential repair candidates. To address the limited availability of comprehensive test cases, we also propose a Petri net-based approach to validate the generated repair candidates against their specifications. Our repair approach successfully generates correct patches for 15 out of 23 OpenID bugs, and a manual assessment reveals that 44% of the generated patches are semantically equivalent to the manually written patches by developers.

Improving the security of real-world SSO implementations. We evaluate our approaches on the most popular and widely used implementations of SSO protocols and uncover many critical zero-day vulnerabilities. We use *OAuthlint* to evaluate 600 popular apps that have more than 10 millions of downloads. Our evaluation shows that 101 applications make at least one security mistake. We use *Cerberus* to identify 47 vulnerabilities from ten classes of logical flaws in OAuth, 24 of which were previously unknown. In addition, *AuthSaber*, our proposed safety verification method for OpenID Connect, discover 16 previously unknown vulnerabilities in popular libraries. We responsively disclosed all identified vulnerabilities to the respective developers and helped them to fix the vulnerabilities. Our findings across multiple real-world SSO implementations also led to eight new CVE entries.

Contributions summary. In summary, we make the following contributions:

- To analyze insecure authorization flows implemented by SSO relying parties, we first develop a query-driven static analyzer that detects security violations in relying party applications.
- We identify the security properties required to implement the authorization server of SSO service providers and devise a scalable technique to check logical security flaws in the extensive codebase of the authorization server.
- We develop a domain-specific specification language, AuthLTL, and create a customized automated verification approach specifically designed for ensuring safety across multi-party interactions in SSO-based protocols.
- To mitigate the identified security vulnerabilities, we leverage LLM to locate faulty statements in buggy SSO programs and develop an automated program repair method to ensure that the underlying protocol execution (after repairing) conforms to standard SSO specifications.

- Finally, our evaluation of popular SSO implementations uncovered 40 previously unknown security vulnerabilities and led to 8 new CVEs. Consequently, our efforts have strengthened the security of popular SSO libraries and applications, benefiting millions of their users.

1.2 Dissertation Structure

This dissertation is organized as follows. Chapter 2 provides an overview of Single Sign-on authorization and authentication protocols, as well as a review of related works in the field. Chapter 3 introduces *OAuthlint*, a static analysis-based solution for detecting security threats in OAuth protocol's relying party programs. Chapter 4 presents a scalable solution for identifying security threats in large authorization server codebases. In Chapter 5, we propose an automated formal verification method to ensure the safety and security of OpenID Connect protocol implementations. Chapter 6 details an approach for automatically generating repaired programs for identified OpenID Connect bugs. Finally, Chapter 7 concludes the dissertation with directions for future research directions.

CHAPTER 2

Background and Related Work

In this section, we first provide a background of SSO-based protocols, followed by an overview of existing works in SSO space and related fields.

2.1 Preliminaries on Single Sign-On (SSO) protocols

Single Sign-On (SSO) is a security and authentication mechanism that enables users to access multiple applications or services with a single set of login credentials. The purpose of SSO protocols is to provide a secure and reliable means of identifying individuals and ensuring that they are who they claim to be in the digital space. These protocols are used in various contexts, such as online banking, e-commerce, social media, and government services. Some common examples of SSO protocols include OpenID Connect (OpenID, 2021), OAuth (RFC6749, 2021), and SAML (SAML, 2023). SSO protocols typically involve the exchange of digital tokens or credentials that a trusted authority, such as a certification authority or identity provider, can verify. The protocol, in general, specifies how these tokens are created, transmitted, and validated in a secure fashion, as well as how access to resources or services is granted based on the authorization obtained from the end users. In this dissertation, we concentrate on OAuth and OpenID Connect as key representatives of SSO protocols due to their popularity and widespread adoption in SSO services.

The concept of OAuth protocol was proposed in 2007 and was designed as an authorization protocol which contains two major OAuth versions that are currently deployed: OAuth 1.0 and 2.0. The first version of the OAuth protocol (OAuth 1.0) was published

in April 2010 (Hammer-Lahav, 2010). Since then, the protocol has gone through a few revisions. The most notable changes to the protocol were released as the OAuth 2.0 framework in October 2012 (Hardt, 2012).

2.1.1 OAuth 1.0

We illustrate the OAuth 1.0 protocol flow in Figure 2.1. The dashed lines in our figures for OAuth represent redirection and solid lines represent direct server-to-server API calls (e.g., a REST API call). Also, parameters inside square brackets are signed using shared secrets.

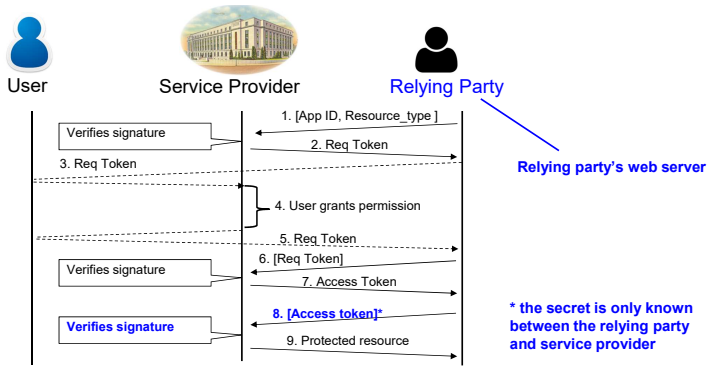


Figure 2.1: Overview of OAuth 1.0.

We summarize the OAuth 1.0 flow in the following:

- Relying party developers will need to register with the service provider and obtain shared secrets (consumer secret and consumer key). The shared secrets will be used in the following steps to sign packets and verify the signatures.
- First, the relying party gets a request token from the service provider using a direct server-to-server call (Step 1,2).
- Then, the relying party redirects the user to the service provider (in mobile devices, it is done by inter-process communication such as Intent in Android) with the request token as a URI parameter. Then, the user grants the relying party access to their protected resource and is redirected back to the relying party (Step 3-5).

- Access With the request token authorized, the relying party can exchange the request token for an access token using another direct server-to-server call with the service provider. Note that these two steps are very critical for the security of OAuth 1.0. The relying party needs to sign the packet with the shared secret (consumer secret/key) and the service provider needs to verify the signature to check the identity of the relying party (Step 6,7).

- Finally, the relying party can use the access token to obtain the user's protected resource (Step 8,9).

2.1.2 OAuth 2.0

Instead of building on top of the existing OAuth 1.0 protocol, the authorities changed the specification completely to create a different protocol, known as OAuth 2.0.

Compared to OAuth 1.0, OAuth 2.0 removed obtaining the shared secrets and providing signatures as mandatory processes. Instead, OAuth 2.0 introduced the concept of *bearer token* (Jones and Hardt, 2012). For the bearer token, a user's access token was no longer bound to a relying party; any party with this token could access the user's protected resource.

2.1.2.1 Authorization Grants

OAuth specification defines four types of grants–(1) authorization code grant (AC), (2) implicit grant (I), (3) resource owner password credentials grant, (4) client credentials grant. Two grants are most widely used in practice – authorization code grant and implicit grant. In the following, we explain the details of these two grants.

Authorization Code Grant. Authorization code grant can be used by both web apps and native apps to obtain the access token. The flow of authorization code grant is illustrated in Figure 2.2. The authorization code grant augments the implicit grant by adding a step for authenticating the relying party (client). After user grants permission to the relying party, OAuth server redirects the user back to the relying party. Instead of

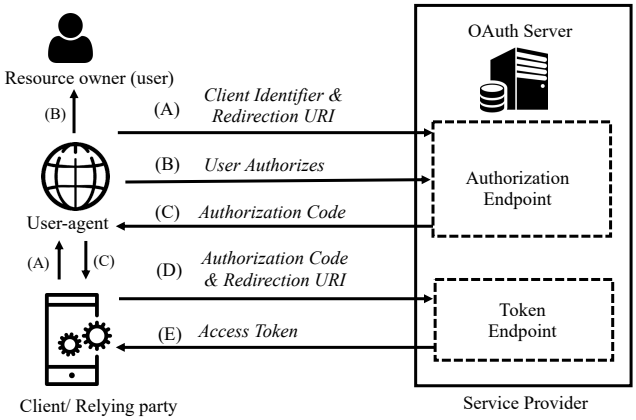


Figure 2.2: Flow of authorization code (AC) grant for OAuth.

providing the access token directly to the relying party, the server sends an authorization code this time. Then the relying party can use the authorization code to exchange for the access token by making a new request at the token endpoint. To get the access token, the relying party needs to include its identity in this request so that the server can verify if the authorization code is granted to the same party.

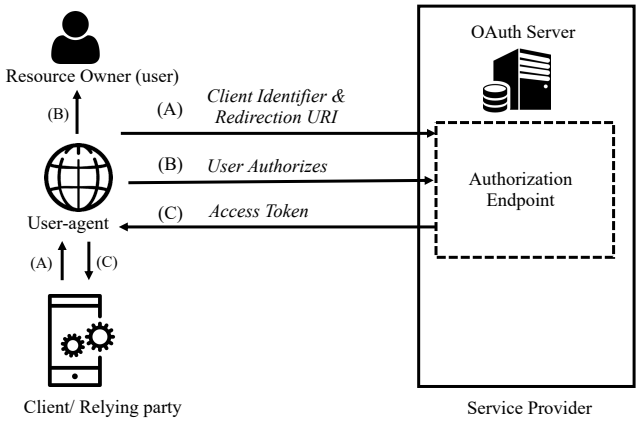


Figure 2.3: Overview of implicit flow of OAuth 2.0.

Implicit grant. The implicit grant is the shortest of all OAuth 2.0 flows. Figure 2.3 illustrates the transactions during the implicit flow grant. It consists of two steps. First, the user is redirected to the service provider to grant the relying party access to his/her protected resource. After the user grants the permission, the service provider redirects the user back to the relying party along with an access token as a parameter in the URI. The relying party can then use this access token to exchange for the user's resource. Besides

the implicit grant, we find that the authorization code grant is also used in the mobile applications we studied. Compared to the implicit grant, the authorization code grant has additional steps for the relying party to obtain an authorization code and then use the authorization code to exchange for the token.

There are two important differences in the implicit grant compared to other OAuth flows. First, with the exception to the final protected resource request, every message in the protocol is exchanged through the user agent (e.g., using browser redirection or Android Intent). Second, the implicit grant does not require the relying party to present a shared secret to the service provider. This is ideal for the mobile environment, where the relying party resides on an untrusted device.

2.1.2.2 PKCE

Although the authorization code grant provides better security benefits than the other grants, it is still vulnerable to code interception attacks, especially for public clients (e.g., native desktop apps), where the attackers intercept the authorization code returned from the authorization endpoint (step C in Figure 2.2) and obtain the access token by exchanging the code at the token endpoint (step D). To mitigate the risk of code interception attack, OAuth specification introduces an extension of the authorization code grant called Proof Key for Code Exchange (PKCE) (RFC7636, 2021) and requires all OAuth servers to support PKCE for public clients. It is worth mentioning that, although PKCE was originally designed to protect public clients, it is recommended (IETF, 2021) to use PKCE for all kinds of clients, including web applications.

Since public clients cannot maintain confidentiality and cannot securely store the client's secret, using PKCE allows OAuth server to authenticate clients without the secret key. PKCE utilizes a dynamically created cryptographic random key called *code verifier*. A unique *code verifier* is generated by the client for every authorization request. The transformed value of the *code verifier*, called *code challenge*, is sent to the OAuth server to obtain the authorization code. When a client makes a new request at the token endpoint

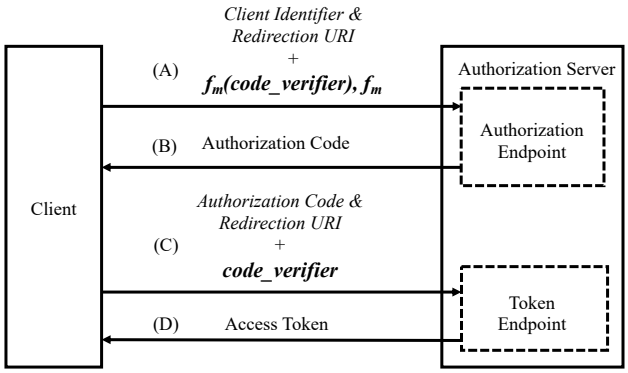


Figure 2.4: Proof Key for Code Exchange (PKCE), an extension of authorization code grant to provide more security benefits for client applications.

to obtain the access token, it also sends the *code verifier* along with the authorization code it received from the previous request. To validate the proof of possession of the *code verifier* by the client, OAuth server transforms the *code verifier* and validates it with the previously received *code challenge*. This approach helps to mitigate the authorization code injection attack as an intercepted authorization code from the authorization endpoint cannot be exchanged for an access token without the one-time key of *code verifier*.

2.1.3 OpenID Connect

Among all SSO protocols, OpenID Connect (OIDC) is the most popular and supported by nearly all major identity service providers, including Google, Microsoft, and Amazon. OIDC is an authentication protocol (OpenID, 2021) allowing users to authenticate themselves across different web and mobile applications. It is built on top of the OAuth 2.0 protocol, which provides authorization for accessing resources. OIDC utilizes an identity layer added on top of the authorization flows, which allows the clients to authenticate end users using their existing accounts. Precisely, a user authenticates with the OIDC provider's authorization server and receives an ID Token, a JSON Web Token (JWT) (JWT, 2015) formatted string that contains information about the user's identity, such as their name and email address as payload along with a signature component that can be cryptographically verified by the recipient. The ID token is then used to authenticate the user to a web or mobile application, which can then authorize the user to access

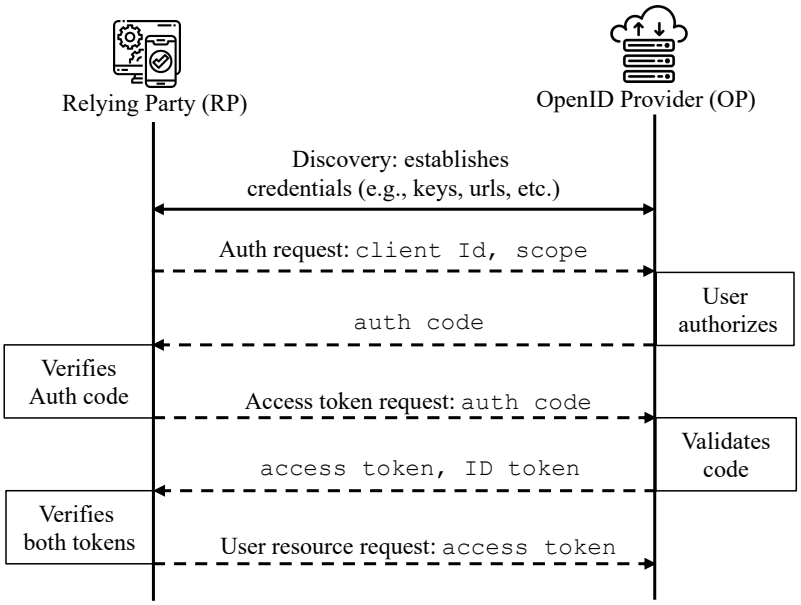


Figure 2.5: Interactions between the Relying Party (RP) and OpenID Provider (OP) during the authentication process of Authorization Code Flow in OIDC.

resources or services. The protocol relies on communication between multiple parties, such as the Relying Party (RP), OIDC Provider (OP), and users. It provides support for three authentication flows that can be implemented by the participating parties: (1) Authorization Code Flow, (2) Implicit Flow, and (3) Hybrid Flow. Each flow defines the transactions that occur between multiple parties during the authentication process. Figure 2.5 illustrates an example of an Authorization Code Flow, where upon establishing the discovery of credentials, RP initiates an authentication request. The OP then issues an authorization code upon successful authorization from end users. RP then verifies the code and exchanges it for an access token along with an ID token. Finally, RP verifies the authenticity of the tokens and exchanges the access token for user resources.

2.2 Related Work

In this section, we present related works on SSO and relevant areas, along with a justification for our contributions.

SSO security analysis. Researchers have studied the implementation of OAuth and discovered many prominent attacks for web applications (Sun and Beznosov, 2012a; Yang et al., 2016a; Shernan et al., 2015; Calzavara et al., 2018; Ghasemisharif et al., 2018). Several recent studies focus on OAuth and OpenID Connect at the protocol and implementation level for various platforms such as mobile (Wang et al., 2015a), web (Sun and Beznosov, 2012b; Cao et al., 2014), and IoT (Sciancalepore et al., 2017; Cirani et al., 2015; Fernandes et al., 2018; Emerson et al., 2015). Yang et al. (Yang and Manoharan, 2013) study the attacker models to perform the common OAuth attacks (e.g., impersonation attacks, CSRF attacks) in web applications.In recent years, mobile developers also use OAuth to build authentication or authorization schemes, and they have many misunderstandings of the security implications of OAuth. Researchers also did field studies to identify the vulnerabilities due to the implementation errors of OAuth in mobile apps. Chen et al. (Chen et al., 2014a) present a comprehensive study on implementation errors and misunderstandings regarding OAuth protocols in mobile applications. Their study also shows that many OAuth implementations are vulnerable to various attacks such as token injection and impersonation attacks. Also, Shehab et al. (Shehab and Mohsen, 2014) analyze source codes of Android applications and demonstrate possible attacks in OAuth implementation. However, these studies depend on dedicated manual analysis by security experts such as inspecting the network traffic and inferring the protocol flows and cannot be scaled to large-scale studies. Furthermore, researchers also studied user's privacy risks during OAuth transactions (Sucasas et al., 2018). Comparing to these works, our system is designed to automatically identify the vulnerabilities of OAuth and OpenID Connect implementations efficiently and effectively. Although, the manual approach can be effective for checking particular flaws, it is not scalable and might miss vulnerabilities. In comparison, *Cerberus* is the first automated and scalable tool to systematically check the logical flaws on OAuth server implementation. In addition, *Cerberus* defines new security properties and detects novel vulnerabilities in real-world OAuth server.

SSO bug finding. There has also been substantial interest in detecting SSO bugs based on vulnerable patterns, which can be categorized into two classes: 1) static bug analysis and 2) network traffic analysis. For instance, S3kvetter (Yang et al., 2018a) uses symbolic execution to check security properties in OAuth client's SDK. The scope of these works is limited to OAuth protocol, and they require the bug patterns are already known to the tool users. Additionally, these works focus on the properties of one entity (i.e., client or server) and assume the other entities are correctly implemented whereas *AuthSaber* uses an entity-agnostic verification against the properties from the standard specification. Researchers also propose to run automatic traffic analysis to identify OAuth implementation errors (Zuo et al., 2017), however, this kind of approach will fail to point out the details of the implementation errors. Compared to their work, our approaches in this work can identify the details of the implementation errors, which is very helpful for the developers to fix the security issues. Wang et al. (Wang et al., 2015b) propose a tool that combines static analysis and network analysis to identify OAuth bugs, however, their tool is semi-automatic and requires manual work to identify the vulnerabilities.Moreover, WPSE (Calzavara et al., 2018) and Bulwark (Veronese et al., 2020) study network-traffic-based security monitoring systems for different SSO entities (e.g., browsers, web apps, etc.). While these tools are effective for checking flaws on client applications, as we show in our evaluation, they are not designed to analyze the large scale codebase on OAuth server. Instead of analyzing the entire application, our tool is designed to automatically extract and reason about the OAuth-specific partial programs to identify security critical logical mistakes. Moreover, network traffic analysis requires heavy manual setup and cannot be applied or extended to uncover all possible execution behavior SSO protocols.

SSO verification. Recent years have seen several efforts to verify the security of the design (Pai et al., 2011; Ye et al., 2015), especially the authorization and authentication flows supported by the SSO protocols. For instance, Fett et al. use a Dolev-Yao style generic formal model of several SSO protocols (Fett et al., 2016, 2017, 2019) to prove authorization and session integrity properties against the model. Additionally, Lu et

al. (Lu et al., 2017) use applied PI calculus to model OIDC protocol and provide security analysis using ProVerif (Blanchet, 2014). Similarly, Hamman et al. (Hammann et al., 2020) proves privacy properties in OIDC model using Tamarin prover (Meier et al., 2013). However, unlike *AuthSaber*, these works verify the properties against a manually constructed model based on the protocol design and are not equipped to reason about complex programming language semantics used in OIDC or other SSO implementations. Moreover, the level of abstraction in these papers make it difficult to detect implementation errorsIn addition, manual construction of models for SSO programs can be both error-prone and time-consuming, preventing them from being used for large-scale analysis.

Automated analysis of security protocols. In recent years, researchers have been studying the problem of implementation errors in many security applications such as SSL, and crypto libraries. For example, CryptoLint (Egele et al., 2013a) did a systemic analysis of the incorrect usage of crypto API, and SSLLint (He et al., 2015c) did program analysis to identify the errors of implementing SSL. Moreover, researchers also utilized demand-driven static analysis for other domains. Arzt et al. present Flowdroid (Arzt et al., 2014b) - a static taint-analysis tool designed to identify data leaks in Android applications. Feng et al. (Feng et al., 2015) implement a tool to answer a class of interprocedural control flow queries about Java programs. Martin et al. (Martin et al., 2005b) present a query language (PQL) that uses context-sensitive pointer alias analysis to mine information (e.g., bugs) from a program. Sridharan et al.(Sridharan et al., 2005) introduce a demand-driven points-to analysis algorithm that outperforms the previous techniques. Feng et al.'s(Feng et al., 2014) work implements a semantic-based approach that identifies an Android malware that leaks the user's private information. Compared to these papers, *OAuthlint* needs to overcome more challenges because OAuth is a multi-party protocol and we don't have access to the data from all parties.

Automated tools like Tamarin prover (Meier et al., 2013) and Proverif (Blanchet, 2014) have been used to prove properties in various protocols, including TLS (Cremers et al., 2016, 2017), 5G (Basin et al., 2018; Cremers and Dehnel-Wild, 2019) and single-party

authentication (Feng et al., 2021; Jacomme et al., 2023; Arquint et al., 2023). However, these verifications, unlike *AuthSaber*, mostly work at the protocol's external communication level and cannot uncover fine-grained security violations caused by programming errors at their implementation level.

Query-based security analysis. Graph-based query approaches (Martin et al., 2005a; Goldsmith et al., 2005; Rahat et al., 2018) have been applied to solve a wide variety of security issues such as API misuse detection (Zhang et al., 2014; Aafer et al., 2018; Staicu et al., 2018; Sven et al., 2019; Rahaman et al., 2019; Lv et al., 2020), tainted data-flow analysis (Yang and Yang, 2012; Li et al., 2014; Yamaguchi et al., 2015; Cheng et al., 2018; Fass et al., 2021) and threat detection (Gao et al., 2021; Park et al., 2021). He et al. implemented SSLint (He et al., 2015b) that uses static analysis and graph signature matching to identify incorrect API usage for SSL. Egele et al. (Egele et al., 2013b) studied cryptographic API misuse and used static program slicing to detect incorrect cryptographic operations in Android applications. These works focus on high-level API usage patterns, whereas *Cerberus* is designed to identify fine-grained properties (e.g., expressions). Graph traversal using different graph representations(Aho et al., 2007; Ferrante et al., 1987; Yamaguchi et al., 2012) of source code has also been used for checking code properties in various applications (Fass et al., 2019; Shoshitaishvili et al., 2015; Cao et al., 2015). In addition to these classic representations, combined graphs (Li et al., 2022) have also been used for vulnerability detection. For example, Yamaguchi et al. (Yamaguchi et al., 2014) use a combined graph representation called Code Property Graph to discover vulnerabilities. However, though these implementations are effective for checking common vulnerabilities (e.g., buffer overflows) in specific languages (e.g., C/C++), we cannot evaluate or compare against them as they do not support the languages (Java and Javascript) we target in this dissertation. In addition, these works eagerly construct the graphs for the entire program, which, as our evaluation demonstrates (Figure 4.4), does not scale for large-scale programs such as OAuth server implementation.

CEGAR-based verification. Our verification technique is based on the counterexample-guided abstraction refinement (CEGAR) (Clarke et al., 2000), which has been used to provide formal security guarantees in many problem domains, including blockchain security (Permenev et al., 2020; Stephens et al., 2021), cryptography (Nejati et al., 2017), IoT protocols (Ammar et al., 2019; Souri and Norouzi, 2019) and autonomous systems (Elboher et al., 2020; Jin et al., 2022). However, these tools are designed for specific domains and cannot be used or extended to check complex multi-party protocols like OIDC.

Search-based program repair. GenProg (Le Goues et al., 2011), a pioneering work in the area of automated program repair utilizes genetic programming to explore a search space of potential repairs generated by reusing code snippets from within the program. PAR (Kim et al., 2013) utilizes GenProg's search approach with a set of repair templates manually derived from human-written patches. Le et al. (Le et al., 2016) employs an extensive collection of templates sourced from GenProg, PAR, and mutation testing to generate a diverse array of potential repairs. These repairs are subsequently organized and pruned based on the frequency of similar (human-written) patches. Later, ACS (Xiong et al., 2017) introduces a technique for accurate condition synthesis by instantiating variables within commonly occurring predicates across a designated code corpus, employing various heuristics to prioritize and select the most suitable variables. In contrast, RSRepair (Weimer et al., 2013) employs a random search approach, while AE (Qi et al., 2014) leverages deterministic search, enhanced by analytical techniques to eliminate redundant patches and optimize the search process. Our proposed approach tackles the repair problem specifically for OpenID Connect protocol that requires a richer domain-specific repair expressions to fix bugs. Moreover, existing repair techniques rely on comprehensive test cases or human-written manual patches, which are not often available for complex protocols like OpenID Connect.

Repair using constraint solving. Program repair techniques like SemFix (Nguyen et al., 2013), MintHint (Kaleeswaran et al., 2014), and NOPOL (Xuan et al., 2016) employ

symbolic execution to construct an oracle-like representation of an expression undergoing repair, and then utilize program synthesis to produce a repaired statement that satisfies the constraints of the oracle. DirectFix (Mechtaev et al., 2015) focuses on generating concise repairs, ensuring comprehensibility by formulating the issue as a partial maximum satisfiability problem with SMT formulas. Angelix (Mechtaev et al., 2016), on the other hand, addresses the scalability challenges encountered by DirectFix by implementing a lightweight repair constraint mechanism. S3 (Le et al., 2017) extends the semantics-based works like SemFix and Angelix with a set of ranking criteria based on the execution traces. However, these works require translating the extracted constraints to SAT/SMT, which runs into the risk of incomplete translation (Mechtaev et al., 2016; Le et al., 2017) due to the complex program semantics and usage of external libraries which is very common in API-based protocols like OpenID Connect. Additionally, these works exclusively reason about boolean or integer types in conditions which limits their applicability to repair bugs related to complex APIs such as cryptographic function calls in OpenID Connect. Moreover, they may suffer from the limitations of the symbolic execution engines (e.g., KLEE (Mechtaev et al., 2016)) to extract constraints. Since our proposed approach explores the actual program behaviors (i.e., at AST-level) in presence of the libraries, and does not require encoding or translating the entire program semantics to SAT/SMT, it can handle repair of expressions and handle a whole range of variable types, function invocation and APIs. Moreover, our approach is not limited by complex data structures as long as they can be executed.

AI for program repair. In recent years, AI-based methods, especially Convolutional Neural Networks (CNNs), Neural Machine Translation (NMT), LLMs and their combination have shown success in program repair. Lutellier et al. (Lutellier et al., 2020) proposes CoCoNuT which leverages ensemble learning, combining CNN and NMT to generate patches. DLFix (Li et al., 2020) adopts a two-tier approach, where the first layer learns the context of bug fixes, and the second layer generates the corresponding patch. Notably, CURE (Jiang et al., 2021) has recently achieved state-of-the-art results on the

Defects4J (Just et al., 2014) and QuixBugs (Ye et al., 2021) datasets, outperforming NMT-based APR techniques by utilizing a pre-trained programming language model, code-aware search, and sub-word tokenization. Elixir (Saha et al., 2017) uses machine learning to rank potential repair candidates to reduce the search space for object oriented programs. InferFix (Jin et al., 2023) combines LLM and static analyzer to fix critical security and performance bugs. AI-based approaches rely on extensive datasets of patched programs, tailored to specific bugs. However, the scarcity of reported and patched bugs in the context of the OpenID Connect makes it impractical to develop a robust, generalized learning model for repairing such bugs.

CHAPTER 3

Security Analysis of OAuth Relying Party

3.1 Introduction

Our study in this dissertation begins with the relying party of the OAuth protocol, as it initiates the protocol's execution and performs crucial operations. In this chapter, we concentrate on mobile applications as the relying party, given the widespread adoption of OAuth by mobile developers for implementing Single Sign-On (SSO) services. For instance, numerous mobile game developers utilize OAuth for authentication with users' Google, Twitter, or Facebook accounts. However, it is important to note that OAuth was initially designed for authorizing third-party websites in 2009, not for authenticating users in third-party mobile apps. This distinction is significant, as the security implications for authentication and authorization differ fundamentally. Moreover, mobile platforms have distinct security schemes compared to browsers, making it challenging for developers to implement OAuth protocols securely in mobile environments.

Recent years, many widespread attacks for the OAuth implementation in popular mobile applications have been reported, causing users to lose their accounts or information. For example, Facebook has a bug for the insecure storage of OAuth tokens, which exposes a large number of user accounts to attackers (Facebook, 2018a). Studying such a critical and popular multi-party protocol will also provide insight for building other multi-party protocols securely.

Due to the complexity of the OAuth protocol, it is challenging to implement it in a

secure way. In particular, there are three parties in OAuth: the user who owns protected resources, the service provider that hosts the user's services, and the relying party that uses the service provider to get access to the user's resources or authenticate the user. During authorization, the three parties need to verify each other's identity and synchronize their secure states. However, despite wide deployment, the OAuth protocol is still too complicated for most developers to follow. For example, as we mentioned earlier, OAuth was first designed for *authorization*, but not for *authentication*. Therefore, the use case for authentication is not clearly defined in the original specification. As a result, developers may misunderstand the threat model and security implications in OAuth, which leads to vulnerabilities that compromise user privacy. In the meantime, the original OAuth 1.0 protocol (OAuth1, 2010) was first proposed for websites, and then got widely used in mobile applications. However, many security schemes cannot be directly mapped from the web to mobile platforms. Therefore, mobile application developers make many mistakes (Chen et al., 2014a) for using the OAuth protocol.

Although an OAuth protocol typically involves multi-parties (i.e., user, relying party, and service provider), in this dissertation, we mainly focus on studying bugs in the relying parties and developing techniques for automatically finding those bugs. The ultimate goal is to provide a tool for helping developers to write secure authentication using the OAuth protocol. We focus on the relying parties for several reasons. First, comparing to the very few identity providers such as Google or Facebook, there are tens of thousands of relying parties in Android platforms and the developers of the relying parties contribute a majority of the common mistakes in using the OAuth protocol (Chen et al., 2014a). As a result, it is important to develop tools for helping numerous relying parties to check the security of their OAuth implementations. Second, mobile applications of most of the relying parties are publicly available through Google Play store. In contrast, the implementations of most of the identity providers are not available to us and we can only approximate their behaviors through black-box testing (Yang et al., 2018b).

Previous studies (Chen et al., 2014a; Li and Mitchell, 2014; Shehab and Mohsen, 2014) have identified many security issues of the mobile OAuth implementations by

manual or semi-automatic analyses. However, the entire process is very time-consuming and error-prone as it requires security experts with domain knowledge in OAuth to inspect the implementations manually. On the other hand, there are several challenges for automatically finding OAuth vulnerabilities in Android platforms. First, verifying OAuth implementation requires precisely modeling the data- and control-properties of mobile applications which are highly asynchronous and interactive. Second, to precisely reason about the OAuth vulnerabilities in mobile applications, our system has to consider the interactions among multiple parties (i.e., Service Provider, Relying Party, SDK, etc) and the majority of their implementations are not available. Finally, a system would need a specification for finding the OAuth vulnerabilities, however, the specification of the latest OAuth 2.0 protocol (OAuth2, 2012) has over 75 pages which are extremely difficult to digest by developers.

To understand the OAuth vulnerabilities in relying parties and help mobile developers to write secure code using OAuth, we perform the first empirical and systematic study on 600+ popular Android applications from Google Play. In particular, we propose *OAuthlint*, the first static analyzer for checking anti-protocols (i.e., vulnerabilities) in OAuth implementation. First, we present a simple but effective query language for describing common anti-protocols in OAuth implementations. Here, each anti-protocol is a class of vulnerabilities in OAuth implementation. Second, we perform a thorough study on specifications of the existing OAuth protocols and summarize five common *anti-protocols*. Third, given an anti-protocol expressed in our query language, we leverage a fully-automatic and demand-driven static analysis to identify the anti-protocols that appear in the Android applications from the relying parties. Formally speaking, an anti-protocol π for program P is true iff π matches a trace of P's possible executions. Our system will return a counter-example if the specified anti-protocol configuration is feasible in the application.

Most of the anti-protocols that we check are logical flaws due to the misunderstanding of the OAuth protocol. Because of the misunderstanding of the security implications of the OAuth protocol, the developers can make logical errors that compromise user privacy.

For example, in the OAuth 2.0 implicit flow, the *bearer token* represents that a user grants a set of permissions to an application. By its design, the *bearer token* is bound to the user and the set of permissions, but not to the application. Therefore, if developers just verify the *bearer token*, it's not secure to authenticate the user because a malicious application might get the bearer token and reuse it in the OAuth process of another application to gain access to the user's account.

To evaluate the effectiveness of our approach, we perform a systematic study on 600+ popular applications which have more than 10 millions of downloads. Our evaluation shows that for those popular apps that use OAuth API, more than 32% of them contain at least one anti-protocol. For those anti-protocols identified by *OAuthlint*, we also reported them to the developers of corresponding mobile applications and get acknowledged by companies such as WordPress and GoFundMe.

3.2 Overview

In this section, we give an overview of our approach with the aid of a motivating example, and then summarize the threat model in our system.

3.2.1 System Overview

Figure 3.1 gives a high-level overview of *OAuthlint*'s architecture. In particular, *OAuthlint* incorporates standard pointer and taint analyses to explicitly track sensitive information (e.g., secret keys, access tokens, etc.) in OAuth protocols.

Pointer analysis Given the source code or bytecode of an Android application, *OAuthlint* leverages Flowdroid (Arzt et al., 2014a) to perform (field- and object-sensitive) pointer analysis to build a precise call graph and identify all variables that may be an alias to each other. The call graph and alias information are further used by the subsequent taint analysis for generating additional program facts that are relevant to answering the *anti-protocols* in OAuth.

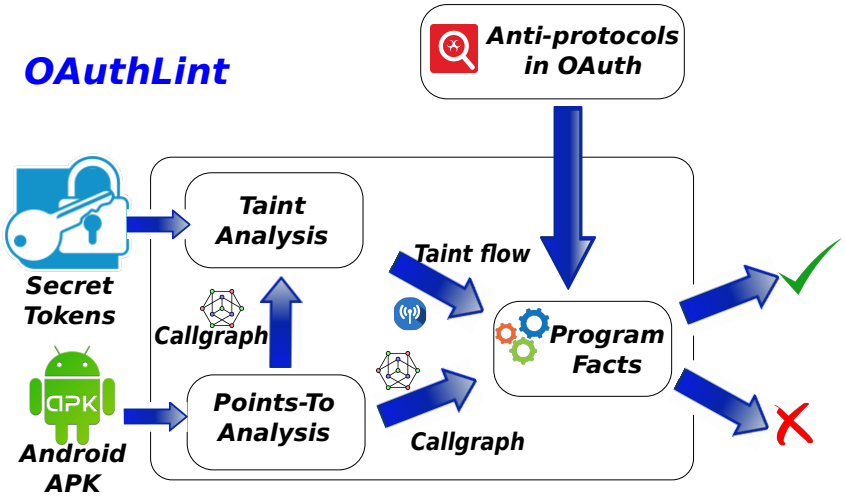


Figure 3.1: Overview of *OAuthlint*.

Taint analysis The taint analyzer leverages the annotations in *OAuthlint*'s configuration file to determine taint sources (i.e., secret keys and access tokens) and propagates them using a field- and object-sensitive analysis. Intuitively, a taint flow encodes that a sensitive field (e.g., secret keys) may flow to an untrusted resource (e.g., local storage, WebView, etc.).

Anti-protocols for OAuth To find logical flaws due to the misunderstanding of the OAuth protocol, *OAuthlint* first proposes a query language that enables developers to specify *anti-protocols*. These anti-protocols represent the OAuth-implementation mistakes by developers that may result in severe vulnerabilities such as impersonation attack and users' privacy violation. Here, each anti-protocol is expressed as a logical formula which encodes a class of vulnerabilities that compromise user's security and privacy. Section 3.4 includes detailed explanation and security impacts of five anti-protocols that widely appear in existing Android apps in Google Play Store. While we propose five anti-protocols using the query language in *OAuthlint*, a security expert can come up with more anti-protocols based on her insight on the standard OAuth specifications. After that, using the program facts generated by our previous analyses, *OAuthlint* leverages a fully-automatic and demand-driven static analysis for checking whether there exists an execution trace such that the anti-protocol holds. If so, a violation will be reported to

the developer.

For instance, *TikTok*, a very popular app for creating and sharing short videos, has over 500 million installs by the time of our submission. To integrate the user accounts from service providers such as Facebook, Instagram, and Google, *TikTok* implements the standard authentication flow using the OAuth 2.0 protocol (Hardt, 2012). However, after running *OAuthlint* on *TikTok*, we found the application contains multiple logical flaws in their implementation for the OAuth protocol.

Firstly, *TikTok* bundles its consumer key and secret within the application code. The following code snippet from *TikTok* represents one of the most common ways that developers use to bundle their secrets:

```
final String CONSUMER_KEY = "YYWjeT***...";
final String CONSUMER_SERECT = "w981H5bEd***...";
```

According to the specification in OAuth 1.0 (Hammer-Lahav, 2010), the consumer secrets or keys should never be bundled in the clients. The reasons are as follows: first, the consumer secrets are highly sensitive information shared between service providers and relying parties, and service providers will use consumer keys to verify identities of the relying parties. Second, all mobiles devices are technically untrusted, which means that an attacker could extract the consumer secrets through reverse engineering and impersonate a benign replying parties to get access to users' information.

Moreover, *TikTok* embeds a WebView to retrieve the access token attached with the redirect URL from Twitter and Instagram.

```
String url = "provider.com/..?
    client_id=".."
    &redirect_url=".."
    &response=token";
webView.loadUrl(url);
void onPageStarted(String url){
  String token = parseToken(url);
}
```

When an application hosts service provider's website inside a Webview, it gets full access to the information such as users' cookies. Therefore, using WebView for OAuth enables a malicious relying party app to log into a victim user's account with the service provider. This is due to the fundamental design of isolation in WebView and there is no way for the service provider to protect herself when it was loaded in WebView. Often service providers use long term cookies, which makes such vulnerability persistent.

Furthermore, *OAuthlint* also detects that *TikTok* exchanges its access token with service providers to fetch the corresponding userId for login purpose.

```
1  String user_id = fetchUserInfo(token, consumer_secret);
2  storeInSharedPreferences(user_id, token);
3  authorizeUserLogin(user_id);//login
```

This approach is also vulnerable because any requests made from a client could be potentially tampered by an attacker. Hence, for authentication using OAuth, client devices should not be trusted. According to OAuth 1.0 and 2.0, a secure authentication should be made through server-to-server calls.

Finally, after obtaining the userId, *TikTok* stores both userId and access token as plain text in the SharedPreferences. Here, there are multiple security issues. First, information stored in SharedPreferences is insecure, since they can easily be accessed by another malicious application in a rooted device or emulator. Hence, any sensitive information should not be stored in SharedPreferences. Second, storing sensitive information such as access token as plain text is insecure. According to the suggestion from OAuth 1.0 and 2.0, the access tokens should be encrypted and stored on the server side of the relying party.

We have reported all the above-mentioned flaws to *TikTok*'s security team and they are still working on those issues by the time of this submission.

3.2.2 Threat Model

In this dissertation, we focus on the vulnerabilities in the design and implementation of the relying parties for using OAuth protocols. We assume the service providers are trusted and correctly implement the security schemes for OAuth. For the attacks to authentication, we consider the case where an attacker has the capability of accessing the mobile apps on behalf of the victims. For the attacks to authorization, we consider the case where malicious relying parties seek to access user's data without consensus.

3.3 Analysis

This section describes *OAuthlint*'s static analyses for computing an over-approximation of some built-in predicates that encode the data- and control-dependency of a program. In what follows, we first give some background on Datalog, and then describe the syntax and semantics of *OAuthlint*'s built-in predicates.

3.3.1 Datalog Preliminaries

A Datalog program consists of a set of *rules* and a set of *facts*. Facts simply declare predicates that evaluate to true. For example, parent("Bill", "Mary") states that Bill is a parent of Mary. Each Datalog rule is a Horn clause defining a predicate as a conjunction of other predicates. For example, the rule:

```
ancestor(x, y) :- parent(x, z), ancestor(z, y).
```

says that `ancestor(x, y)` is true if both `parent(x, z)` and `ancestor(z, y)` are true. In addition to variables, predicates can also contain constants, which are surrounded by double quotes, or "don't care", denoted by underscores.

Datalog predicates naturally represent relations. Specifically, if tuple (x, y, z) is in relation A, this means the predicate $A(x, y, z)$ is true. In what follows, we write the type of a relation $R \subseteq X \times Y \times \ldots$ as $(s_1 : X, s_2 : Y, \ldots)$, where $s1$, $s2$, ... are descriptive

texts for the corresponding domains.

Base Facts. The base facts of our inference engine describe the instructions in the application's control-flow graph (CFG). The base facts take the form of $\mathrm{A}(y, x_1, ..., x_n)$, where `A` is the instruction name, `y` is the variable storing the instruction result (if any), and $x_1, ..., x_n$ are variables given to the instruction as arguments (if any). For example, the instruction $r_1 = 0$ is encoded to $\mathrm{assign}(r_1, 0)$. Further, the special local store instruction $\mathrm{lstore}(d, v)$ denotes that the value of v is stored in location d.

Flow-to Predicates. The `FlowTo` predicate is defined over pairs of variables and is inferred from the application's CFG. The intuitive meaning (also summarized in Figure 3.2) is: $\mathrm{flowTo}(x_1, x_2)$ holds for x_1 and x_2 if the value of variable x_2 depends on the value of x_1.

$$
\begin{array}{lll}
\mathsf{FlowTo}(x, y) & :- & \mathsf{alloc}(y, x) \\
\mathsf{FlowTo}(x, y) & :- & \mathsf{assign}(y, x) \\
\mathsf{FlowTo}(x, z) & :- & \mathsf{assign}(y, x), \mathsf{FlowTo}(y, z) \\
\mathsf{FlowTo}(x, z) & :- & \mathsf{alias}(y, z), \mathsf{FlowTo}(x, y)
\end{array}
$$

Figure 3.2: Rules for computing the `FlowTo` predicate. Here, the `alias` predicate is directly obtained from Flowdroid.

OAuth-specific Predicates. In addition to base facts from programs, *OAuthlint* also defines a list of predicates that are specific to the OAuth domain. As shown in Figure 3.3, `isToken(x)` denotes that x may be an access token. Since it is difficult to precisely pinpoint strings that corresponds to access tokens, we use both pattern matching (i.e., regular expressions) and domain-specific knowledge (i.e., API that may return an access token) to over-approximate the domain of access tokens. Similarly, the client secret is encoded as `isSecret(x)`. Furthermore, statement `y = new SecretKeySpec(x)`, which is used to construct a client secret, is represented as `SecretKeySpec(y, x)`. Finally, `isLocalStore(x)` represents a location for a local store, which can be Android `SharePreferences` or file systems.

$$
\begin{aligned}
\mathsf{isToken}(x) &:- \text{ } x \text{ may be an access token} \\
\mathsf{isSecret}(x) &:- \text{ } x \text{ may be a secret} \\
\mathsf{webView}(x) &:- \text{ } x \text{ is the URL of a WebView} \\
\mathsf{secretKey}(y, x) &:- \mathsf{SecretKeySpec}(y, x) \\
\mathsf{Http}(y, x, _) &:- \text{ } y \text{ is an HTTP object of which arguments contain } x \\
\mathsf{Istore}(x, z) &:- \mathsf{isLocalStore}(x), \mathsf{FlowTo}(z, x) \\
\mathsf{login}(x, y) &:- \text{ Login with user x and password y}
\end{aligned}
$$

Figure 3.3: Rules for computing the OAuth-specific predicates.

3.4 Anti-protocols in OAuth

This section describes *OAuthlint*'s anti-protocol language, which is a Datalog program augmented with builtin predicates. We propose five anti-protocols that widely appear in existing mobile apps and formalize them using our language. To identify the pattern of these anti-protocols, we manually reverse engineered 43 Android apps from Google Play Store. The anti-protocols are designed with Android and Java APIs, and therefore are not biased with any particular application. For each anti-protocol, the user defines a unique predicate that serves as the signature for this anti-protocol. In what follows, we first describe the syntax of *OAuthlint*'s built-in predicates, and then discuss five common anti-protocols in OAuth.

3.4.1 An Anti-protocol Language for OAuth

We first introduce our anti-protocol language for OAuth, shown in Figure 3.4. Here, `arg`, `reg`, and `mem` are variables from function arguments, registers, and memory, respectively. The predicate `flowTo` determines the data dependency between two variables, as specified in Section 3.3. Finally, we can express more complex queries by composing simple expressions with logical operators (i.e., $\neg, \wedge, \vee$, etc.).

3.4.2 Common Anti-protocols in OAuth

In the following, we describe the most common anti-protocols in OAuth.

$$
\begin{array}{rcl}
v_i & ::= & \mathsf{arg} \mid \mathsf{reg} \mid \mathsf{mem} \mid \ldots \\
\phi & ::= & \mathsf{flowTo}(v_1, v_2) \mid \mathsf{isToken}(v_1) \mid \mathsf{isSecret}(v_1) \\
& \mid & \mathsf{secretKey}(v_1, v_2) \mid \mathsf{Http}(v_1, v_2, v_3) \mid \mathsf{Istore}(v_1, v_2) \\
& \mid & \mathsf{webView}(v_1) \mid \mathsf{login}(v_1, v_2) \\
& \mid & \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi
\end{array}
$$

Figure 3.4: The Anti-protocol language in *OAuthlint*.

3.4.2.1 Locally Bundled Client Secrets

Relying party secret, which is often referred to as the *client secret* by OAuth 2.0 and *consumer secret* by OAuth 1.0, is used by service providers to authenticate the relying party. Developers can obtain the relying party secret from the developer's console of the service providers when they register their application. Many developers misunderstand the purpose of the relying party secrets and store them locally on client-side applications. If a developer bundles the relying party secret with her mobile application, an attacker can easily retrieve it through reverse engineering, and use this secret to get her own application to be authenticated by the service providers as a benign application. Relying party gets the client secret from the service provider when it registers for the OAuth service, and the client secret is then used to generate the signature from the token in step 8. Since the service provider verifies the signature to check the identity of the relying party, if the relying party bundles the secret in their mobile app, the attacker can do simple reverse engineering to extract the client secret and impersonate the relying party (i.e., victim) to access the user's protected resources.

In practice, *OAuthlint* identifies that many developers bundle their relying party secrets as field variables, resource files, or constants in their application code. For instance, here is an example from the GoFundMe application (GoFundMe, 2019) which hard-codes its secret for communicating with Twitter:

```
1  String consumer_secret = "QfMu9***...";
2  mac = Mac.getInstance("HmacSHA1");
3  key = encodeParam(consumer_secret);
4  mac.init(new SecretKeySpec(key.getBytes(), "HmacSHA1"));
```

```
5  HttpURLConnection c = new URL(url).openConnection();
6  params = new ArrayList();
7  params.add("signature="+getSignature(mac,token,..));
8  [...]
9  r = c.getInputStream();
```

The locally store client secrets can be encoded as the following anti-protocol in *OAuthlint*:

$$\mathsf{isSecret}(x) \wedge \mathsf{secretKey}(y, z) \wedge \mathsf{flowTo}(x, z) \\ \wedge \mathsf{Http}(_, u, _) \wedge \mathsf{flowTo}(y, u)$$

In addition to the above scenario, developers also store relying party secrets in Android *SharedPreferences*, which is also insecure:

```
1  String appSecret = getAppSecret();
2  SharedPreferences sf = getActivity().getPreferences(mode);
3  SharedPreferences.Editor editor = sf.edit();
4  editor.putString("app_secret", appSecret);
5  editor.commit();
```

Since the data in *SharedPreferences* are stored in the file system, an attacker can access the secret by other malicious applications if the device is rooted. *OAuthlint* uses the following anti-protocol for checking this variant:

$$\mathsf{isSecret}(x) \wedge \mathsf{lstore}(_, z) \wedge \mathsf{flowTo}(x, z)$$

where `lstore` represents all untrusted locations.

3.4.2.2 Using WebView for OAuth Transactions

Using WebView for OAuth transactions is insecure because WebView breaks the isolation between the service provider and the relying party. When a malicious relying party hosts the service provider's website in the WebView of their mobile applications, the malicious relying party can get the user's cookies to log in on behalf of the user. Figure 3.5 illustrates

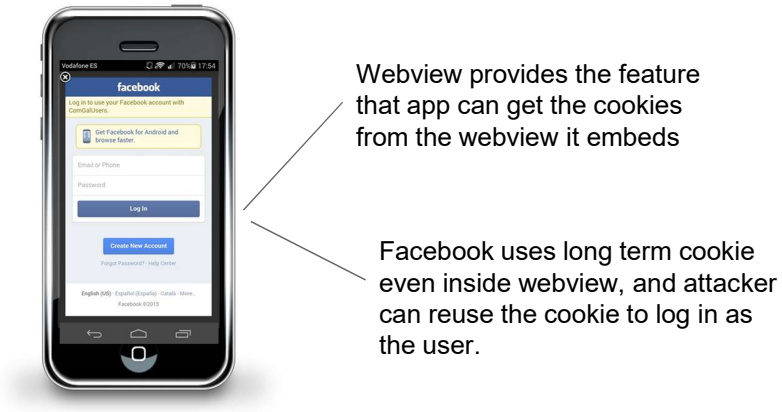


Figure 3.5: Vulnerability of using Webview for OAuth transactions.

an example of using Webview for Facebook login. Webview enables the hosting app (malicious relying party) to access the cookies of the service provider such as Facebook. With the cookies, the malicious relying party can log in on behalf of the user. This attack can be persistent if the service providers such as Google and Facebook use long term cookie for better user experiences, which is quite common.

In practice, we found many developers use embedded WebView or mobile browsers for user redirection in OAuth, which endangers user privacy. For example, here is a code snippet from the Waplog application (Waplog, 2019):

```
String url = "provider.com/..?
    client_id="..."
    &redirect_url="..."
    &response=code";
public void onCreate() {
    webview.loadUrl(url);
}
```

Using WebView for OAuth transactions can be encoded as the following anti-protocol:

$$\mathsf{isSecret}(x) \wedge \mathsf{isToken}(y) \wedge \mathsf{webView}(z)$$
$$\wedge \mathsf{flowTo}(x, z) \wedge \mathsf{flowTo}(y, z)$$

3.4.2.3 Client-side API Call

Client devices should not be trusted during API calls that are involved in the flows for OAuth authentication. In other words, relying parties should always assume that any requests made from client devices could be tampered by the attackers. Unfortunately, in reality, developers often misunderstand the security implication and assume that the access token granted by the service providers are only bounded to the relying party. As a result, an attacker could leverage the access token granted for some malicious applications and login as a user for other benign applications to access other sensitive information. For example, the Topface application (Topface, 2019) exchanges access token for user id by doing API call from the application to authenticate users.

```
1  String url = "api.provider.com/..";
2  HttpURLConnection c = new URL(url).openConnection();
3  c.setRequestMethod("POST");
4  [...]
5  params = new ArrayList();
6  params.add("oauth_consumer_key="+client_id);
7  params.add("oauth_token="+access_token);
8  params.add("oauth_signature="+getSignature(client_secret));
9  [...]
10 c.setRequestProperty("Authorization",createHeaders(params));
11 String user_id = parseJSON(c.getInputStream(),"id");
12 newUserLogin(user_id);
```

We express the Client-side-API-Call vulnerability using the following anti-protocol:

$$\begin{aligned}&\mathsf{isToken}(x) \wedge \mathsf{isSecret}(y) \wedge \mathsf{Http}(r, u, _) \wedge \mathsf{login}(v, _)\\ &\wedge \mathsf{flowTo}(x, u) \wedge \mathsf{flowTo}(y, u) \wedge \mathsf{flowTo}(r, v)\end{aligned}$$

3.4.2.4 Storing Access Tokens on Client Devices

By the end of a typical OAuth transaction, the relying party receives an access token, which is a raw string that can be used to make API calls to retrieve protected resources

from the service provider. Stealing access tokens provides an ideal vector through which an attacker can compromise user accounts and harvest confidential data such as email and contacts. Some service providers (e.g., Google) also allow access to user's files stored in the cloud via access tokens. More importantly, an access token does not require user's password and is capable of bypassing any two-factor authentication. To make things worse, the only way to revoke an attacker's access is to explicitly revoke access to the malicious application that uses the access token to launch attacks. Thus, having access tokens obtained by attackers could have an adverse impact on users. During the U.S. Presidential Election in 2016, one of the tactics attackers used was collecting OAuth access tokens, as reported (FireEye, 2019) by security experts from FireEye. Our analysis finds that many developers often do not encrypt their raw access tokens before storing them on client devices, using *SharePreferences* or files in the external storage. This is insecure, as data in *Sharedpreferences* or filesystem and can easily be accessed from any rooted device or an emulator. For example, the Chatous application (Chatous, 2019) stores the access token received from Instagram as follows:

```
1  String aToken = getAccessToken();
2  SharedPreferences sf = getActivity().getPreferences(mode);
3  SharedPreferences.Editor editor = sf.edit();
4  editor.putString("access_token", aToken);
5  editor.commit();
```

Storing access tokens on client devices can be encoded as the following anti-protocol:

$$\mathsf{isToken}(x) \wedge \mathsf{Istore}(_, u) \wedge \mathsf{flowTo}(x, u)$$

3.4.2.5 Sending Raw Access Token to Server

To make server-to-server API calls during OAuth transactions, developers send the access token to the relying party server. However, if a raw (i.e., unsigned) access token is sent to the backend server, a modified client application can send arbitrary access token and initiate an impersonation attack.

A client device is assumed to be untrusted when OAuth is used for authentication. Thus, instead of making API call directly, a mobile application should communicate with its own backend server and pass the access token to the server. The server would then use the access token to make API calls to communicate with the resource server of the corresponding service provider. However, it is important for developers to understand that access tokens are portable. Once an access token is received, it can be used from both applications and server to fetch users' resources. Thus, sending an unsigned access token to the server can lead to *Token Hijacking*. An example from the Wish application (Wish, 2019) that sends the raw access token to the backend server is given below:

```
1  String aToken = getAccessToken();
2  HttpClient httpClient = new DefaultHttpClient();
3  HttpPost httpPost = new HttpPost("/backend.com/tokensignin");
4  params = new ArrayList(1);
5  params.add(new BasicNameValuePair("access_token", aToken));
6  [...]
7  httpPost.setEntity(new UrlEncodedFormEntity(params));
8  httpClient.execute(httpPost);
```

Sending raw access token to server can be encoded as the following anti-protocol:

$$\mathsf{isToken}(x) \wedge \mathsf{Http}(_, u, _) \wedge \mathsf{flowTo}(y, u)$$

Google recently added a security notice to address this vulnerability in their official documentation (Google, 2019). To authenticate users on Google's backend servers, they recommend developers to send ID-token ((returned by *GoogleSignInAccount.getIDToken()*)) which is signed by Google's public keys. If backend server receives sensitive information such as access token in plain text, a modified application can send an arbitrary token to the server and thereby, initiate an impersonation attack.

3.5 Experimental Evaluation

To evaluate the effectiveness of *OAuthlint*, we conduct a series of experiments to answer the following research questions:

- **RQ1**: How effective is *OAuthlint* at identifying real-world vulnerabilities in OAuth implementations?
- **RQ2**: How prevalent are those anti-protocols discussed in section 3.4?
- **RQ3**: How do the real-world OAuth vulnerabilities look like?

3.5.1 Data Collection

To find the current scenario of vulnerable implementation of OAuth, we have analyzed 600 latest Android applications (shown in Table 3.1) collected from Google Play Store. Our unbiased selection of applications includes top 300 free applications from all categories, top 200 free social applications, and top 100 free communication applications. The reason why we include more applications in the social and communication category is that these applications are usually more likely to use OAuth. Most applications use OAuth SDKs from service providers such as Google and Twitter. Table 3.3 shows the detail of the statistics. All applications were collected in April 2019.

Our analysis found that 316 out of the 600 applications use OAuth with at least one Service Provider. We built a Soot-based sanitizer that automatically analyzes the Dalvik byte-codes to filter out applications not using any relevant classes or APIs provided by the Service Providers. We included 20 most popular Service Providers for this sanitization step.

3.5.2 Evaluation Results

OAuthlint successfully analyzed, in total 316 applications that use at least one OAuth service provider. Among the remaining 284 applications, 273 applications did not use

Category	**#Apps**	**#OAuth apps**	**#Vulnerable apps)**
Top Free	300	178	39
Social	200	109	48
Communication	100	29	14
Total	600	316	101

Table 3.1: Top Android applications using OAuth

any OAuth implementation and 11 applications ran out of memory during the analysis. In average, total runtime for each application was 282.6 seconds and maximum memory consumption was 1931.06 MB during the analysis. All results mentioned here are in reference to the 316 successfully analyzed applications.

Table 3.2 lists the number of vulnerable applications for different OAuth vulnerabilities. Total vulnerable applications with distinct number of vulnerabilities are illustrated in Figure 3.6. We discuss the results for each vulnerability below:

Locally Bundled Client Secrets. *OAuthlint* successfully identified 29 applications that bundle the consumer key/secret of at least one service provider within the application code. We found 18 applications bundled with Twitter consumer key and secret, 7 applications bundled with the client id and client secret from Instagram, 3 applications bundled with the Facebook app secret, and 12 applications bundled with the consumer keys/secrets from other service providers in the source code.

Using WebView for OAuth Transactions. This vulnerability is a common scenario for the application that implements OAuth in an embedded WebView for web-based service providers. By hosting the service provider's website in the WebView of the relying party's mobile applications, the relying party can interact with the service provider easily. In addition, a malicious relying party can access the user's cookies on the service provider's website to log into the user's account. *OAuthlint* successfully identified 24 applications who use WebView for the authorization and authentication transactions of OAuth. 9 applications choose to implement the web-based OAuth version instead of the app-based version, even though corresponding service providers provide official SDK, particularly for

app-based OAuth implementation.

Client-side API Call. When using OAuth for authentication, developers should pass the encrypted access token received from the relying party application to the backend server of the same relying party and then, should verify the access token with the service provider by doing a server-to-server API call. *OAuthlint* identified 69 applications that violated this secure approach and performed the authentication by making client-to-server API calls. For authentication, the client device should not be trusted as an API call made from the client side could be tampered with by a malicious user.

Storing Access Token on Client Device. *OAuthlint* identified 21 applications that store plain (i.e., unencrypted) access tokens in Android *SharedPreferences*. Access tokens are portable, which means an access token obtained from a relying party application can also be used from any client, server, or otherwise. Hence, if malicious attackers retrieve the access token from a client device, they can use it from a different machine to launch an impersonation attack. Storing sensitive access tokens in insecure *SharedPreferences* gives partial security, as data stored in *SharedPreferences* can be easily accessed by any application or user from a rooted device or emulator. Android provides a *KeyStore* system to enhance the security for storing such sensitive information such as access tokens.

Sending Raw Access Token to Server. Right after obtaining the access token at the authorization endpoint, the relying party application should send the access token to their backend server in order to exchange it for the user's resources (i.e., user ID). However, to this end, the relying party server should not trust data received from the application. A malicious attacker could modify the application to tamper with the access token and hence, initiate an impersonation attack to the relying party server. *OAuthlint* identified 17 applications that send the plain access token to the relying party server.

Finally, we note that the analysis in *OAuthlint* is precise. As shown in Table 3.2, the average false positive rate is 10%. We manually inspected the false alarms and confirmed

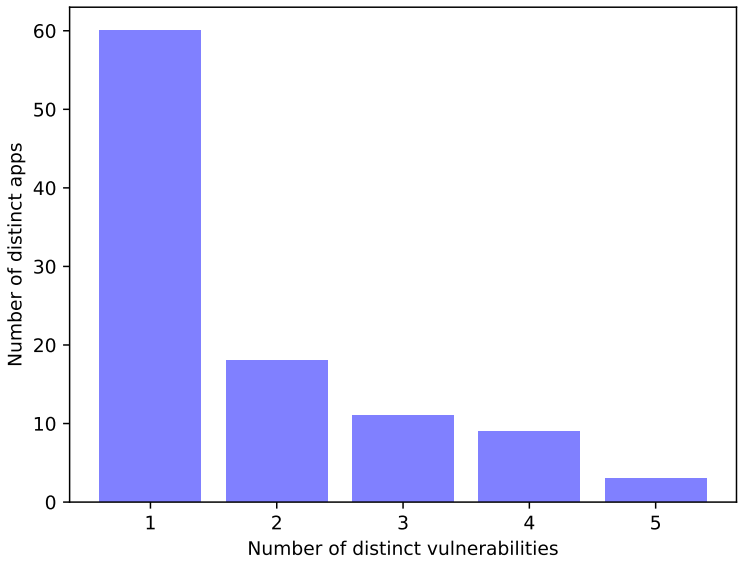


Figure 3.6: Number of applications with up to 5 vulnerabilities.

that most of them are caused by the imprecision of the pointer analysis and call-graph construction.

Vulnerability	#**apps**	#**FP**
Locally Bundled Client Secrets	29	2
Using WebView for OAuth Transactions	24	3
Client-side API Call	69	7
Storing Access Token on Client Device	21	3
Sending Raw Access Token to Server	17	1

Table 3.2: OAuth vulnerabilities in top android applications

3.5.3 Case Studies

CBS Sports. To estimate the impact of vulnerable implementations of OAuth, we manually inspected the applications that were reported as vulnerable by *OAuthlint*. *CBS Sports*(CBS, 2019) is one of the most popular applications in the U.S. for top sports news, scores, and videos. According to Google Play, this application was installed more than 10 million times. This application got flagged by *OAuthlint* for three reasons. First, it bundles the Twitter consumer key and consumer secret within the application code,

Provider	Total apps	Vulnerable apps	OAuth version
Facebook	301	52	2.0
Google	295	11	2.0
Twitter	34	23	1.0a, 2.0

Table 3.3: Statistics of top OAuth APIs in our evaluation

which makes it vulnerable to impersonation attacks with the authorization transactions of OAuth. Secondly, it uses an embedded WebView for OAuth transactions between relying party applications and service providers. Currently, there exists no secure way for the web-based service provider to deliver the sensitive OAuth data to the honest relying party application. Hence, using WebView for OAuth transactions makes it very difficult for the service provider to determine the identity of the *CBS Sports* application. Thirdly, the application uses client-side API calls to authenticate new users. Client devices must not be trusted during the OAuth transactions for authentication since corresponding API calls could be tampered with by malicious users.

Topface. We also performed a thorough inspection with *Topface* (Topface, 2019), one of the most popular social applications in Google Play. This application allows users to meet new people online and was installed more than 10 million times. However, *Topface* provides authentication using OAuth for various service providers including Google, Facebook, Instagram, and Vkontakte. The application was flagged by *OAuthlint* for several reasons. Even though it provides secure intent-based OAuth transactions for Google and Facebook, it uses WebView for both authorization and authentication transactions for Instagram and Vkontakte. More importantly, it performs all API call for OAuth transactions from client-to-server instead of server-to-server call, which allows a malicious user to tamper with sensitive information such as *access token* and *user id*. The application also stores a plain access token, user ID, and email address in the *SharedPreferences* without doing any encryption. Storing these sensitive information in *SharedPreferences* gives partial security since any data stored in *SharedPreferences* can be easily accessed from any rooted devices and emulators.

3.6 Best Security Practice for OAuth Relying Party

According to our analysis, 101 of the OAuth relying party implementations suffer from at least one vulnerability. We hope this study can help to provide more guidelines for mobile OAuth developers, especially for the relying party developers.

For securing the OAuth protocol, there are two general major points : (1) be aware that the security of OAuth partially lies in its access token delivery methodology; (2) never trust the mobile client because it might belong to a malicious user who can access the secret data and temper the verification results or data. For example, if developers are developing OAuth 2.0 in Android, instead of using the default Intent scheme to deliver the access token, they should use the developer key hash in order to check the identity of the party that receives the token. If developers are developing OAuth 1.0, they also need to make sure they can verify the token receiver's identity correctly. The relying party should never bundle the developer's consumer secret and consumer key into its mobile app because a malicious user can just extract the secret and key from the app and pretend to be the relying party to access user data.

For using OAuth to do authentication, the relying party needs to be more careful about dealing with the users. First, the relying party must not bundle any security related protocol logic (e.g., security checks) or any sensitive information (e.g., the token) into its own mobile application. Second, the relying party must assume that the attacker could tamper with any data sent from the user's device. Because of this, the relying party must check that the relying party receiving the user's ID in the last step of the protocol is the same relying party that the user intends to authenticate to. For example, instead of using the default implicit flow of OAuth 2.0 for authentication, the relying party should do an additional verification step such as described in the *appsecret_proof* flow of Facebook OAuth guidelines (Facebook, 2019).

3.7 Limitations

Like any other program analysis tool, *OAuthlint* has a number of limitations. *OAuthlint* proposes a query language that helps developers to define OAuth-based anti-protocols. In this dissertation, we explore five anti-protocols, each of which represents a class of vulnerabilities in OAuth implementation that widely appears in Android apps. However, defining a new anti-protocol that covers a wide range of vulnerable implementation scenarios may seem tedious for a security expert. Also, applications having complex logical implementation flaws may still be undetected by *OAuthlint*. In addition to that, service providers typically allow developers to make certain changes in the security settings of the developer's panel. Cross-checking between program facts in relying party apps and security settings in the developer's panel is beyond the limit of *OAuthlint*.

3.8 Summary

In this chapter, we propose *OAuthlint*, the first static analyzer for checking anti-protocols (i.e., vulnerabilities) in mobile OAuth implementation. We propose five anti-protocols that widely appear in mobile apps and formalize them using our query language. Furthermore, we leverage a fully automatic and demand-driven static analysis to identify anti-protocols that appear in the Android apps from the relying parties. To evaluate the effectiveness of our approach, we perform a systematic study on 600+ popular apps which have 10 million downloads. Our evaluation shows that for those popular apps that use OAuth API, more than 32% of them contain at least one anti-protocol. For those anti-protocols identified by *OAuthlint*, we also reported them to the developers of corresponding mobile apps.

CHAPTER 4

Security Analysis of OAuth Authorization Server

4.1 Introduction

In Chapter 3, we address the security behavior of the relying party (RP) applications, assuming a correctly implemented authorization server. However, in reality, this assumption does not always hold, as numerous attacks have exploited logical errors in authorization servers' implementation. Thus, this chapter shifts its focus to the OAuth authorization servers implemented by the service provider (SP), which typically supports multiple authorization flows to cater to a diverse range of relying party platforms.

OAuth defines a process for end-users (resource owners) to grant a third-party website/application (also noted as relying party or client applications (RFC6749, 2021)) access to their private resources stored on a service provider (SP), without sharing their passwords for the service providers with the client applications. As a multi-party protocol, the security of OAuth depends on the service providers (also noted as authorization server or *OAuth server* (RFC6749, 2021)), the client applications, and the resource owners. In particular, vulnerabilities in OAuth server implementations could lead to severe consequences because that would impact all the client applications (along with all the resource owners of the applications) that work with the service provider. For example, a vulnerability discovered in 2019 in the Microsoft OAuth server's redirect URI validation mechanism allowed the attackers to take over Microsoft Azure Accounts (Microsoft, 2020). Also, in 2018, a vulnerability in Facebook OAuth server allowed the attackers to steal access

tokens (issued by authorization server) of almost 50 million users (Facebook, 2018b).

Even well-known service providers such as Microsoft and Facebook still make severe mistakes in their authorization server implementation – which leads to a pressing question, *how to vet OAuth service provider implementation for security?* Since OAuth is very high-profile and critical for user security and privacy, many previous works have been on OAuth security. However, as far as we know, there is no automatic tool to identify the security-sensitive logical flaws in the service provider implementation.

Researchers have devoted significant effort to investigating OAuth security by analyzing the protocol (Yang and Manoharan, 2013) and proposing formal verification tools (Fett et al., 2016; Pai et al., 2011). However, these tools analyze the security at the protocol level and do not consider the diversified implementation details of OAuth server. Chen et al. (Chen et al., 2014b) perform the first in-depth study on the security issues of OAuth by running a manual analysis of mobile apps and monitoring the network traffic. Later on, researchers develop semi-automated tools (Bai et al., 2013; Zhou and Evans, 2014) to report various security issues with OAuth client applications. However, these analyses often need an extensive manual setup and do not scale for large programs. Most recently, *S3kvetter* (Yang et al., 2018a) performs symbolic execution, and *OAuthlint* leverages whole-program data-flow analysis to check client-side properties. As we show later in the evaluation, these approaches are not applicable for checking OAuth server properties for the following reasons: first, symbolic execution suffers from path explosion (Xie et al., 2009) and whole-program analysis also does not scale well, especially for large codebases like OAuth server programs. Second, data-flow predicates alone are not expressive enough to describe crucial properties that require control-flow predicates.

In addition, most existing works assume that the OAuth servers are securely implemented, which clearly is not the case according to our study in this dissertation. As a result, it is urgent to help developers check the security of the OAuth server implementations. Unfortunately, as far as we know, there is no systematic security study of OAuth server implementations. Most previous works in OAuth security either focus on client applications or report security issues on an ad hoc basis.

We compare the representative OAuth tools from previous work using five relevant criteria: (1) *Automated:* ability to identify vulnerabilities automatically, (2) *Coverage:* the ability to cover and reason about the behavior of OAuth implementation, (3) *Server-side flaws:* the ability to analyze logical flaws of service provider implementations, (4) *Extensibility:* the ability to provide an interface to define and check new security properties, (5) *Scalability:* scalable for extensive programs. We summarize the comparison in Table 4.1, which clearly shows the gap.

To bridge the gap, we propose systematically checking the security of OAuth server implementations of authorization flows. We start by investigating popular open-source OAuth server libraries. We find many developers use these libraries to implement their service provider applications instead of starting from scratch due to their complexity. For example, Node OAuth2 Server (NodeOAuth2, 2021) library is used by more than 2,400 repositories on Github. Since developers widely use these open-source libraries to integrate OAuth in their implementations, checking the security of OAuth server implementations in these libraries also provides us insights about many implementations built on top of these libraries.

Table 4.1: Comparison with existing OAuth analysis tools.

Tool	**Automated**	**Coverage**	**Server**	**Extensible**	**Scalability**
AuthScan (Bai et al., 2013)	◐	◐	○	○	○
SSOScan (Zhou and Evans, 2014)	●	◐	○	○	○
OAuthTester (Yang et al., 2016b)	●	○	○	○	○
S3kVetter (Yang et al., 2018a)	●	●	○	●	○
Cerberus	●	●	●	●	●

● Full support ◐ Partial support ○ No support.

Challenges: However, developing an automatic tool for finding logical flaws of OAuth server implementations is quite challenging due to the following reasons:

- *Expressiveness.* The original specifications for OAuth protocols are written in plain English, which are difficult to be turned into checkable invariants consumed by the analyzers. Consequently, properties for implementing a secure and effective

server-side implementation for authorization using OAuth are not well-defined.

- *Scalability.* Implementing the protocols on OAuth server is typically complex and large and often relies on many third-party libraries. Therefore, as shown in Sec. 4.4, a naive whole-program analysis will not scale well.
- *Generality.* Checking security properties in OAuth server implementations is not straightforward as they depend on various factors such as the client's platform and types of grants (i.e., authorization flows) used by the client during the authorization process. Thus, it is non-trivial to devise a framework that unifies all properties enforced by the OAuth specifications.

Our solution: To address the challenges mentioned above, we design and implement *Cerberus*, a scalable and automated static analysis tool for detecting logical flaws in large-scale OAuth server programs written in server-side languages like Java and Javascript. In particular, we first design a domain-specific language (DSL) to enable security analysts to express the security properties of OAuth server that are recommended by OAuth specification (RFC6749, 2021) and OAuth current security best practices (IETF, 2021).

Second, given an OAuth property Q encoded in our query language, we represent both the desired property Q and the OAuth server programs as *System Dependence Graphs* (SDGs). Therefore, *Cerberus* converts the problem of checking OAuth properties into a graph query problem, which an off-the-shelf Datalog engine (Soufflé Developers, 2020) can answer. Note that our graph representation aims to strike a good balance between expressiveness and scalability by capturing temporal sequencing of API calls, data flows between arguments and returns of a procedure, data flows between various program objects, etc.

However, there is a steep trade-off between the precision of an SDG and the cost of constructing it. For example, SDGs constructed using a context-insensitive pointer analysis tend to grossly overapproximate the targets of virtual method calls, which leads to unacceptable false alarms. On the other hand, more precise SDGs obtained using context-sensitive pointer analysis can take hours to construct. Currently, analyses that rely on system dependence graph information must implement their own ad hoc analysis (He

et al., 2015b; Egele et al., 2013b) to answer application-specific queries. To mitigate this challenge and only reason about program fragments relevant to the query, we introduce a query-driven approach based on automata theory. Since the original system dependence graph is obtained by stitching all control flow graphs from the methods, our key intuition is to keep track of the methods relevant to the OAuth property. In particular, we define the OAuth request endpoint, which is the precondition of the OAuth property, specifying the entry and exit points of the relevant code snippet. After that, we leverage the request endpoints to pinpoint a sub-callgraph between the request endpoints. Since the sub-callgraph is typically small, its corresponding system dependence graph will also be small.

Findings: We evaluate our tool with the ten most popular libraries that use the standard OAuth protocol for implementing OAuth server and find pervasive vulnerabilities in these popular libraries. All the ten popular libraries we studied have at least one security-critical property violation (i.e., vulnerability). In total, we identified 47 vulnerabilities, 24 of which were previously unknown. We study ten security-critical logical flaws in OAuth server, five of which (P4, P5, P6, P7, and P9 in Table 4.2) are studied by us for the first time. Violation of these properties may lead to severe attacks that are commonly observed on OAuth servers. We got acknowledged by the developers of eight libraries, among which five libraries, including Spring Auth Server (Spring, 2021) and Node OAuth2 Server (NodeOAuth2, 2021)), immediately took actions to fix the vulnerabilities. Three classes of new vulnerabilities in these libraries also lead us to have new CVE entries (CVE-2020-26877, CVE-2020-26937, and CVE-2020-26938). These vulnerabilities can lead to account breaches and confidential information leakage for both clients and resource owners. Eight libraries immediately acknowledged our findings; five have fixed the issues after we reported them, and the rest of the libraries are currently taking action. Finally, *Cerberus* is effective in that it significantly outperforms *OAuthlint*'s approach in terms of running time ($> 20\times$ faster) and expressiveness.

4.2 Overview

This section briefly explains how our tool detects vulnerabilities in OAuth server implementations using a motivating example. In what follows, we first describe the threat model our system, and then with a real-world example, we explain how insufficient security checks in the OAuth server allow malicious clients to steal sensitive OAuth credentials and how our tool is designed to detect such logical flaws.

Threat Model: We aim to detect vulnerabilities in the OAuth server at the implementation level. We assume attackers can be a malicious relying party or a malicious user (resource owner) who interacts with the victim OAuth server, also known as service provider. We assume the attackers cannot directly modify the source code or logic of the service provider but can initiate attacks by sending requests to their server. The relying party attackers control their own malicious relying party apps. For example, the relying party attackers might send malicious requests to the victim service provider to access the user's information without the user's approval. The resource owner attackers use their own devices to communicate with the benign service provider to log in on behalf of the victim user.

Vulnerable implementation. The industry-standard authorization protocol of OAuth is well designed for access delegation, but a wrong implementation or incorrect usage can have a colossal impact. During the authorization process, client gets an access token with specific permissions to take actions on behalf of the user which can even be used to control user's account. When responding to an authorization request, the OAuth server passes the authorization code or tokens to the client application using a redirect URI, which describes the destination where the code or tokens are passed. The client application sets an allowed list of trusted URIs to receive the OAuth tokens during the registration process. However, many OAuth servers do not appropriately validate redirect URI, leading to the possibility of passing the tokens to a malicious URI under the attackers' control. In recent years, many attacks exploiting the redirect URI have been observed in the OAuth servers, including the popular ones like Microsoft (Microsoft, 2020) and Twitter (Twitter, 2021).

Similar attacks have been observed in the open-source OAuth servers, as they also utilize the same OAuth protocol.

For example, ApiFest (ApiFest, 2021) is a popular open-source OAuth server implementation that uses OAuth protocol to provide a secure API management service. We identify a new vulnerability of incorrect redirect URI validation in this library (CVE-2020-26877). Figure 4.1 demonstrates a simplified implementation of their authorization request endpoint. ApiFest server makes mistakes in the critical step of checking the redirect URI submitted by the client, which allows an attacker to obtain the authorization code by using a maliciously crafted redirect URI during an authorization request. In particular, this implementation makes two severe security mistakes at this step (line 6-9 in Figure 4.1): (1) it does not check whether the redirect URI is registered to the corresponding client, (2) it omits the required format checking for the redirect URI. Since the server does not match the submitted redirect URI with the client's registered URI, it allows the attacker to craft the redirect URI parameter with attacker's own redirect URI and thereby, steal the authorization code of a legitimate client through the redirected URI. Additionally, attackers can also leverage open redirectors of the user agent to steal the authorization code. These mistakes also violate the standard OAuth specification as described in RFC-6749 (RFC6749, 2021).

To detect such security vulnerabilities caused by incorrect implementation of logical properties for OAuth, in this dissertation, we design and implement an automated and scalable tool, called *Cerberus*, to analyze the large codebase that implements OAuth server. We first identify the security-sensitive properties based on the standard OAuth specification (RFC6749, 2021) and security best practices (IETF, 2021; RFC6819, 2021). Then, we meticulously design a query language to formally express the properties so that developers can easily define them and they are understandable by the analysis tool as well. *Cerberus* then represents OAuth server programs at the statement level using system dependence graphs (SDGs), while maintaining the control and data flow relationship between the statements. However, as OAuth server programs can be huge, running analyses on the statement-level representation for the whole program do not scale

```
1  public void (authRequest)(HttpRequest request, HttpResponse response) {
2      String clientId = request.client_id;
3      Client client = getClient(clientId);
4      String redirectUri = request.redirect_uri;
5      [...]
6      //incorrect Redirect URI validation
7      if (redirectUri == null) {
8          redirectUri = client.redirect_uri;
9      }
10     [...]
11     AuthCode authCode = generateCode(clientId, scope, ...);
12     authCode.setClient(clientId);
13     authCode.setRedirectUri(redirectUri);
14     DB.storeAuthCode(authCode);
15     [...]
16     QueryStringEncoder encUri = new QueryStringEncoder(redirectUri);
17     encUri.addParam("code", authCode.getCode());
18     //redirects with auth code
19     response.sendRedirect(encUri.toString());
20 }
```

Figure 4.1: Code example for an authorization endpoint that is vulnerable to the redirect URI manipulation during the authorization flow.

well. To overcome the scalability challenge, *Cerberus* automatically identifies the program component corresponding to OAuth query. Finally, after *Cerberus* pin-points its scope in the OAuth relevant implementation in the program, it executes the query to identify the violation of properties that might expose the server to security attacks.

4.3 System Design

In this section, we discuss the design and implementation of *Cerberus*, our end-to-end static analysis tool for systematically checking the security issues of OAuth server implementation.

Figure 4.2 shows an overview of the *Cerberus* approach for checking OAuth properties for Java or Javascript programs. *Cerberus* takes two inputs: (1) the source or byte code of an application, and (2) a user-provided OAuth property Q specifying the correct behavior using our query language. Given these two inputs, *Cerberus* performs signature matching by checking whether there exists an *embedding* of the application with respect to property

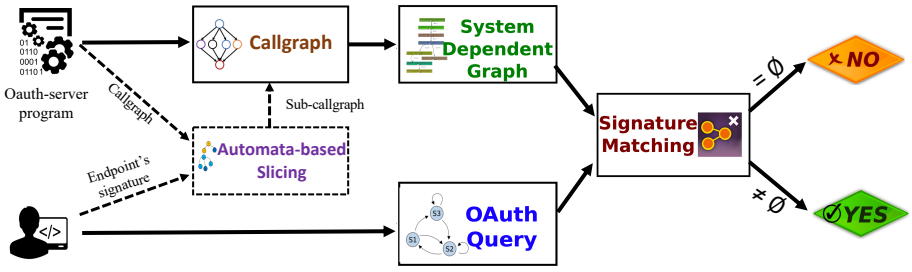


Figure 4.2: Overview of *Cerberus*. Here, Automata-based slicing is an optimization step discussed in Sec. 4.3.

Q.

4.3.1 Code Representation

Given a program, *Cerberus* first generates its abstract representation using static analysis. In particular, we leverage *System Dependence Graph (SDG)*, which summarizes both data- and control-dependencies among all the statements and predicates in the program.

More formally, `SDG` for an application A is a graph (V, X, Y) where:

- V is a set of vertices, where each $v \in V$ is a program statement of A.
- X encodes control-dependency edges. Specifically, $(v, v') \in X$ indicates that during execution, v can directly affect whether v' is executed. Precisely, `SDG` creates three additional edges to handle function calls: (1) call edge, (2) parameter-in edge, and (3) parameter-out edge. Call edge connects the node at callsite (in caller) to the entry node of the called procedure (i.e., callee). Parameter-in edges connect the actual-in (caller) parameter nodes to the formal-in (callee) parameter nodes of the called procedure, and parameter-out edges connect the formal-out (callee) nodes to the actual-out (caller) nodes.
- Y is a set of data-dependency edges. In particular, $(v, v', d) \in Y$ indicates that statement v and v' are related by metadata d. Here, we use metadata d to denote *taint sources* that will be propagated by the data-flow analysis. Also, X is data dependent on Y if Y is an assignment and the value assigned in Y can be referenced from X.

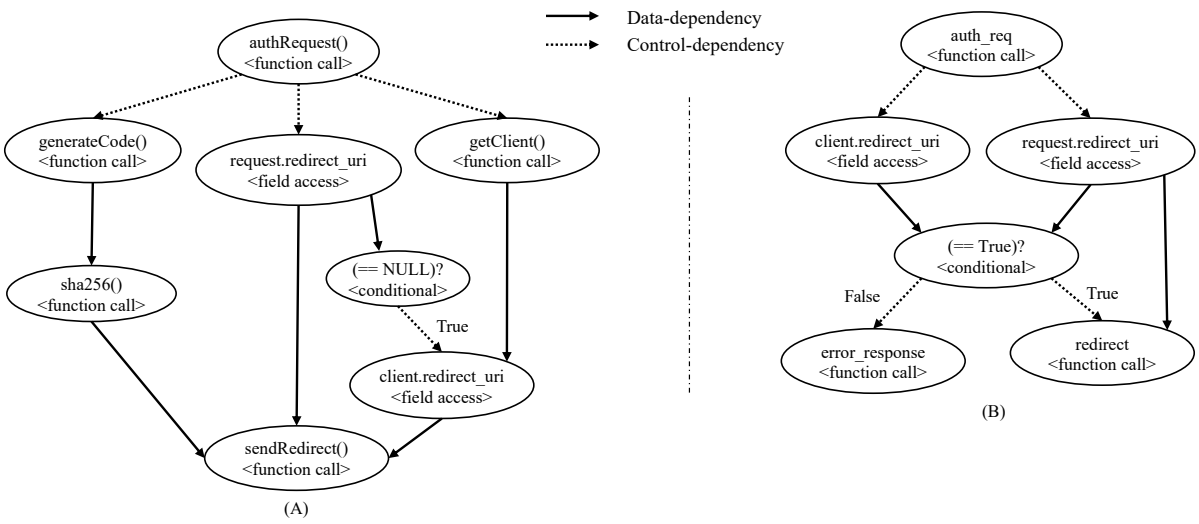


Figure 4.3: (A) System Dependence Graph (SDG) representation for the code example in Figure 4.1 and (B) graphical representation of the query for the Redirect URI property (P1).

Example 1. *Figure 4.3(A) shows a simplified SDG constructed from the code example in Figure 4.1. In the* `SDG`*, the nodes represent statements of the program such as function calls, field access, etc. The edges represent the control- and data-dependencies between the statements. For example, since the if condition (line 7) depends on the value of redirect_uri field (line 4), the conditional node (==Null?) has a data-dependency edge from the field access node (request.redirect_uri). On the other hand, control-dependency between the function call nodes, GetClient() and AuthRequest(), implies the GetClient() is invoked after the AuthRequest() is invoked.*

4.3.2 Facts and Inference Rules for SDG

Inspired by *OAuthlint*'s formalization, *Cerberus* converts the application's SDG into its corresponding facts and rules using Datalog. We first give some preliminaries on the Datalog program, and then describe the syntax and semantics of *Cerberus*'s built-in predicates.

A Datalog (Datalog, 2022) program is a set of *facts* and *rules* written in a declarative logic language. Facts correspond to predicates that evaluate to *true*. For example, in

SDG's context, edge($Stmt_1$, $Stmt_2$) implies that statement node $Stmt_1$ and $Stmt_2$ are connected by an edge. Each rule is a Horn clause (Van Emden and Kowalski, 1976) defining a predicate as a conjunction of predicates. For example, the following program:

```
path(x, y) :- edge(x, y).
path(x, y) :- path(x, z), edge(z, y).
```

says that path(x, y) is true if edge(x, y) is true, or both path(x, z) and edge(z, y) are true. In addition to variables, predicates can also contain constants (surrounded by double quotes), or "don't care", denoted by '_'.

4.3.2.1 Base Facts in *Cerberus*

Unlike *OAuthlint* whose analysis is flow-insensitive, many crucial properties in OAuth server require a flow-sensitive analysis. Therefore, *Cerberus*'s facts take the form of $A(L, y, x_1, ..., x_n)$, where A is the instruction name, L is the instruction's label, y is the variable storing the instruction result (if any), and $x_1, ..., x_n$ are variables given to the instruction as arguments (if any). For example, the instruction $l_1 : r_1 = 0$ is encoded as `assign`$(l_1, r_1, 0)$. Additionally, `alloc`(l_1, y, x) means that variable x may point to abstract location y and `alias(x,y)` denotes that variable x and y may point to the same abstract location. Furthermore, the branch instruction `branch`(L_1, X, L_2, L_3) denotes that if X is evaluated to `true`, then the next instruction will be L_2, otherwise $L3$. Using the base facts described above, *Cerberus* computes the semantic facts of: (i) control-dependency predicates, which capture instruction dependencies according to the application's CFG, and (ii) data-dependency predicates.

4.3.2.2 SDG Predicates

Cerberus provides built-in predicates to encode SDG generated from Sec. 4.3.1. In particular, the `depOn`(y, x) predicate indicates that the value of variable x has data-dependence on y. Similarly, the `followBy` predicate is inferred from the application's

CFG. Intuitively, $\texttt{followBy}(L_1, L_2)$ holds for L_1 and L_2 if both are in the same basic block and L_2 follows L_1, or there is a path from the basic block of L_1 to the basic block of L_2. The SDG predicates are computed using the following datalog rules:

$$
\begin{aligned}
\mathsf{depOn}(x, y) &:- \mathsf{alloc}(_, y, x) \\
\mathsf{depOn}(x, y) &:- \mathsf{assign}(_, y, x) \\
\mathsf{depOn}(x, z) &:- \mathsf{assign}(_, y, x), \mathsf{depOn}(y, z) \\
\mathsf{depOn}(x, z) &:- \mathsf{alias}(y, z), \mathsf{depOn}(x, y) \\
\mathsf{followBy}(x, y) &:- \mathsf{follow}(x, y) \\
\mathsf{followBy}(x, z) &:- \mathsf{followBy}(y, z), \mathsf{follow}(x, y)
\end{aligned}
$$

Here, we use the $\texttt{follow}(L_1, L_2)$ as the base case which holds if L_2 immediately follows L_1 in the CFG.

4.3.2.3 OAuth Predicates

In addition to basic facts from the SDG, *Cerberus*'s query language also defines a list of predicates specific to the OAuth domain. The predicate $\texttt{OAuthTag}(L, T)$ defines that, the value at label L is assigned tag T. Here, tag T are associated with program statements that hold OAuth-specific resources such as *redirect_uri*, *access_token*, etc. For instance, the field access at line 8 in Figure 4.1 is denoted as $\texttt{OAuthTag}(8, \texttt{client_redirect_uri})$.

Prop.	Grants	Description
P1	AC, I	OAuth server must validate the redirect URI parameter in the authorization request exactly matches with the client's registered URI before sending the redirected response to the URI. (RFC-6749)
P2	AC, I	Redirect URI parameter in authorization request must be an *absolute* URI. OAuth server implementations utilize pattern matching APIs to perform this validation. (RFC-6749, RFC-6819).
P3	AC	Authorization `code` must be single-use, meaning the `code` must be revoked once it is exchanged for token. If a revoked `code` is used again, OAuth server must deny the request with an error message. (RFC-6749).
P4	AC	Authorization `code` must be bound to a certain client. Before issuing the token from the token request endpoint, OAuth server must check the `code` is the same as the one issued to the client. (RFC-6749, OAuth Security Best Practices).
P5	AC	Authorization `code` must be bound to the client's redirect URI to where it was issued to. Before issuing the token, the server must check that the `code` is associated with the client's redirect URI. (RFC-6749).
P6	AC	OAuth server must store the PKCE parameters (i.e., code challenge and code challenge method) at the authorization endpoint to be validated at token endpoint (RFC-7636, OAuth Security Best Practices).
P7	AC	OAuth server must verify the authenticity of PKCE parameters at token endpoint. The transformed (i.e., sha256) value of `code_verifier` parameter must be same as the `code_challenge` value (RFC-7636, OAuth Security Best Practices).
P8	AC, I	OAuth server must provide CSRF protection by handling the `state` parameter in authorization request. Value of the `state` parameter must be added to the redirected response of authorization request. (RFC-6749, OAuth Security Best Practices).
P9	AC, I	Access tokens issued by the OAuth server should be constrained to a certain client. *m-TLS* is a standardized and widely used client-constrained mechanism, in which the server first obtains the client's certificate from TLS stack, decodes and hashes the certificate and finally, associates it with the access token. (RFC-8705, OAuth Security Best Practices).
P10	AC, I	OAuth server should not store access tokens as clear-text and should store access token hashes only (RFC-6819).

Table 4.2: We define ten security properties to detect the most commonly observed logical flaws on OAuth servers based on the standard OAuth specifications

The `OAuthTag` predicate is computed through a standard data flow analysis in WALA. Specifically, our analysis first marks a set of APIs that could return OAuth resources as the sources. All variables assigned (either direct or transitive) by the sources will also point to the corresponding sources. In particular, since it is difficult to precisely pinpoint strings that correspond to redirected URLs, we use both pattern matching (i.e., regular expressions) and domain-specific knowledge (i.e., API that may return a redirected URI or object values accessed by the keys defined in the specification) to over-approximate the domain of URI. Furthermore, `invoke`(L, m) predicate implies that method m is invoked at a location with label L. Here, m can be both OAuth-specific methods (e.g., `gen_token`) and programming language-specific methods (e.g., `matches`). `sload`(L, q) predicate defines the load data operation for server's storage model q (e.g., database). Similarly, `sstore`/`sdelete` defines the store/delete operations. Finally, `error`(L, e) implies that an OAuth exception is thrown with an error message e.

4.3.3 Query Language

In this section, we show how to express OAuth properties over semantics facts of an application. We begin by defining the query language for expressing security patterns. This construction enables us to determine whether an application complies with a given security property. Specifically, for each OAuth property, the user defines a unique predicate that serves as the signature for the property. The user may also define additional helper predicates used by the signature.

Syntax. The syntax of the query language is given by the following Backus–Naur form (BNF):

$$
\begin{aligned}
\phi \;&::=\; \mathsf{depOn}(v_1, v_2) \mid \mathsf{followBy}(v_1, v_2) \mid \mathsf{branch}(...) \\
&\mid\; \mathsf{OauthTag}(\mathsf{x}, \mathsf{y}) \mid \mathsf{invoke}(\mathsf{l}, \mathsf{m}) \mid \mathsf{sload}(\mathsf{l}, \mathsf{q}) \mid ... \\
&\mid\; \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi
\end{aligned}
$$

We now formally define our *property signatures* and state what it means for an app

Prop.	*Cerberus* query
P1	`invoke(L1, auth_req),OAuthTag(L2,req_URI), OAuthTag(L3,client_URI), invoke(L4,redirect), error(L5,_), branch(L6,X,L4,L5), followBy(L1,L2), followBy(L1,L3), depOn(L2,X), depOn(L3,X).`
P2	`OAuthTag(L1, auth_req), OAuthTag(L2, req_URI), OAuthTag(L3,abs_URI), invoke(L4,matches), invoke(L5,redirect), error(L6,_), branch(L7,X,L5,L6), followBy(L1,L2), depOn(L2,L4), depOn(L3,L4), depOn(L4,X).`
P3	`invoke(L1,token_req), OAuthTag(L2,code), sload(L3,db),invoke(L4, gen_token), sdelete(L5,db), error(L6,_), branch(L7,X,L4,L6), followBy(L1,L2), depOn(L2,L3), depOn(L3,X), followBy(L4,L5), depOn(L2,L5).`
P4	`invoke(L1,token_req), OAuthTag(L2,code), OAuthTag(L3,client_code), invoke(L4, gen_token),error(L5,_), branch(L6,X,L4,L5), followBy(L1,L2),followBy(L1,L3), depOn(L2,X), depOn(L3,X).`
P5	`invoke(L1,token_req), OAuthTag(L2,code_URI), OAuthTag(L3, client_URI), invoke(L4,gen_token), error(L5, _), branch(L6,X,L4,L5), followBy(L1,L2), followBy(L1,L3), depOn(L2,X), depOn(L3,X).`

Table 4.3: Properties (P1-P5) encoded in our query language

to *match* a property. Intuitively, a signature for a property $\mathcal{F}$ is an SDG (V_0, X_0, Y_0) that captures semantic properties. Ideally, $G_0 = (V_0, X_0, Y_0)$ would satisfy the following: G_0 occurs as a subgraph (defined below) of the SDG $G_S = (V_S, X_S, Y_S)$ of an OAuth application.

By "occurs as a subgraph", we mean there exists an embedding $F_S : V_0 \to V_S$ such that the following properties hold:

- **One-to-one.** For every $v, v' \in V_0$ where $v \neq v'$, F_S cannot map both v and v' to the same vertex, i.e., $F_S(v) \neq F_S(v')$.
- **Type preserving.** For every $v \in V_0$, F_S must map v to a vertex of the same type, i.e., $T(v) = T(F_S(v))$

Prop.	*Cerberus* query
P6	`invoke(L1,auth_req), OAuthTag(L2, code_challenge), OAuthTag(L3, code_challenge_method), sstore(L4,db), followBy(L1,L2), followBy(L1,L3), depOn(L2,L4), depOn(L3,L4).`
P7	`invoke(L1,token_req),OAuthTag(L2, code_challenge), OAuthTag(L3, code_verifier), invoke(L4, sha256), invoke(L5, gen_token), error(L6, _), branch(L7,X,L5,L6), followBy(L1,L2), followBy(L1,L3), depOn(L3,L4), depOn(L4,X), depOn(L2,X).`
P8	`invoke(L1,auth_req), OAuthTag(L2,state), invoke(L3,redirect), error(L4,_), branch(L5,X,L3,L4), followBy(L1,L2), depOn(L2,X), depOn(L2,L3).`
P9	`OAuthTag(L1,access_token), OAuthTag(L2, client_cert), invoke(L3,b64_decode), invoke(L4, b64_encode), invoke(L5,sha256), invoke(L6,add_cert), depOn(L2,L3),depOn(L3,L5),depOn(L5,L4), depOn(L1,L6),depOn(L4,L6).`
P10	`OAuthTag(L1, access_token), invoke(L2, sha256), sstore(L3,db), depOn(L1,L2), depOn(L2,L3).`

Table 4.4: Properties (P6-P10) encoded in our query language

- **Edge preserving.** For every $v, v' \in V_0$, F_S must map an edge $(v, v') \in X_0$ to an edge in X_S:

$$(v, v') \in X_0 \Rightarrow (F_S(v), F_S(v')) \in X_S.$$

Given property $G_0 = (V_0, X_0, Y_0)$ and app S with `SDG` $G_S = (V_S, X_S, Y_S)$, we say that G_0 *exactly matches* (or simply *matches*) S if G_0 occurs as a subgraph of G_S. In other words, given a signature (V_0, X_0, Y_0) and a sample S with `SDG` (V_S, X_S, Y_S), we can check whether (V_0, X_0, Y_0) matches S. If so, we have determined that $S \in \mathcal{F}$; otherwise, $S \notin \mathcal{F}$.

4.3.3.1 Expressing OAuth properties.

Table 4.3 and Table 4.4 shows the list of properties that we express using our query language. We obtain these properties from the documentation of standard OAuth specifications (RFC6749, 2021; RFC6819, 2021; RFC7636, 2021) and security best practices (IETF, 2021). In what follows, we briefly discuss the properties and their formal representation.

OAuth requires that the redirect URI submitted during the authorization request match the client's registered URI (P1). Specifically, we tag these OAuth parameters by `OAuthTag(_,req_URI)` and `OAuthTag(_,client_URI)`, respectively. In addition, since this URI is used to transfer sensitive information like authorization code, the specification also requires the URI to be an *absolute URI* (P2), preventing attackers from utilizing the open redirector of the user agent to intercept the sensitive information. Server programs commonly use pattern matching APIs (e.g., '`matches`') to match the redirect URI value.

To prevent code replay attacks during token request, OAuth also requires the authorization code to be single-use (P3), meaning the code can be used only once (to generate token) by the client. Therefore, once the code is used, OAuth server removes it from the storage (e.g., database) and rejects requests (with error message) when the code is not found in storage. We use `OAuthTag(_,code)` to tag authorization code and `invoke(_,gen_token)` to track the OAuth-specific API call for generating token.

OAuth security best practices require OAuth server supporting PKCE (RFC7636, 2021) to protect public clients (e.g., native desktop apps). To securely implement PKCE, servers first receive and store the PKCE parameters (e.g., `code_challenge`) at authorization endpoint (P6), and transforms (by '`sha256`') the `code_verifier` parameter to match with the `code_challenge` (P7) at the token endpoint.

OAuth also requires access token to be constrained to a certain client to prevent token injection attacks (P9). Mutual-TLS (RFC8705, 2021) (also noted as m-TLS) is a standardized and widely used client-constrained mechanism that allows the clients to demonstrate the proof of possession when using the access token. In m-TLS, the server first obtains the client's certificate from TLS stack and associates it with the token before sending to

the clients. The certificate is decoded and encoded using standard Base64 decode-encode APIs. In our formalization, we use `OAuthTag(_, client_cert)` to tag statements holding client's certificate, and `invoke(L4, b64_encode)`, `invoke(L4, b64_encode)`, etc. to track the corresponding domain-specific APIs.

Once a query expressing a property from the OAuth specification is submitted, *Cerberus* checks it against the SDG representation of the input program.

Example 2. *Figure 4.3(B) shows a graphical representation of the query that* over-approximates *the redirect URI property (P1) in Table 4.2. It represents a signature where a redirect URI received during an authorization request (request.redirect_uri) is matched with the redirect URI field of a client instance (client.redirect_uri) before making a redirect function call, which must be used for a successful response to the authorization request. Therefore, this signature can be used to check that a successful redirection occurs only after the check for redirect URIs is performed. Otherwise, an error response is generated. On the other hand, the SDG (Figure 4.3(A)) of the program for processing authorization request retrieves the 'redirect_uri' from the 'request' instance, but the 'redirect_uri' is matched only against the NULL value before sending the redirection using 'sendRedirect' function call. However, OAuth specification requires that the server must check whether the 'redirect_uri' of the authorization request is the same as the 'redirect_uri' registered to the client, which is not done in this program. Therefore, Cerberus reports a violation.*

4.4 Evaluation

To determine the effectiveness of *Cerberus*, we evaluate it on popular OAuth-server libraries to answer the following research questions:

- **RQ1**: Can *Cerberus* identify real-world vulnerabilities?
- **RQ2:** Is our query-driven approach efficient and effective?

4.4.1 Experimental Setup

We conduct all experiments on a Quad-Core Intel Core i5 computer and 16GB of memory running on the macOS 12.1 operating system. In what follows, we elaborate on the details of the setup.

4.4.1.1 Dataset

We consider high-profile OAuth-server libraries written in Java or Javascript. With extra engineering effort, our techniques can technically be applied to applications in other languages as well. To answer the research questions, we consolidate two datasets of open-source OAuth server libraries that implement the standard OAuth specification (RFC6749, 2021).

- $\texttt{DataSet}_U$: This dataset contains 10 popular open-source libraries of OAuth server (i.e., service provider) where it is *unknown* whether the libraries satisfy the security properties. Table 4.5 shows the key statistics for the selected libraries. To select the libraries, we first consider their popularity among the web developers. For example, *Node Oauth2 Server* (NodeOAuth2, 2021) library has more than 200k downloads each month. Similarly, *Oauth2orize* (Oauth2orize, 2021) and *Node oidc provider* (NodeOIDC, 2021) have approximately 179k and 65k downloads each month. Additionally, we also consider the number of dependant repositories (i.e., repositories that use the APIs of the library) as an indicator of the popularity of our selected libraries. All of the libraries support the widely used authorization code grant (RFC6749, 2021), and three also support the implicit grant. Four of these libraries are implemented in Javascript, and six are implemented in Java. Some libraries (e.g., *OxAuth*) are considerably larger than others as they also support additional endpoints (e.g., token introspection) and platform-specific custom grants for their authorization server.
- $\texttt{DataSet}_K$: This dataset contains 12 OAuth libraries with 49 OAuth-specific logical flaws that are confirmed by Github issues or online forums. Among the 49 logical flaws, 13 were caused for improper handling of authorization code, 12 for mishandling access

token, 7 for redirect URI validation, 8 for missing or incorrect PKCE validation, 3 for state parameter, and 6 for other issues such as incorrect client validation. While the selection criteria for these libraries are similar to $\texttt{DataSet}_U$, we exclude libraries whose ground truths are unknown. Six of these libraries are implemented in Javascript, and six are implemented in Java. All libraries follow the standard OAuth specifications and support all the commonly used grants.

OAuth-server libraries	**Ver.**	**Language**	**Grants**	**#LOC**
1) Node oauth2 server (NodeOAuth2, 2021)	3.1.1	JS	AC	3,936
2) Oauth2orize (Oauth2orize, 2021)	1.11.0	JS	AC, I	3,483
3) Node oidc provider (NodeOIDC, 2021)	6.29.5	JS	AC, I	15,493
4) Oauth2 server (OAuthServer, 2021)	1.0	JS	AC	2,946
5) Spring auth server (Spring, 2021)	1.0	Java	AC	28,374
6) Clouway server (Clouway, 2021)	1.0.6	Java	AC	8,462
7) Jobmission server (OAuthServerJob, 2021)	1.0	Java	AC	6,480
8) Apifest (ApiFest, 2020)	0.3.1	Java	AC	14,371
9) Yoichiro server (Oauth2Server, 2021)	1.0	Java	AC	6,764
10) OxAuth (OXAuth, 2021)	3.0.2	Java	AC, I	62,857

Table 4.5: Statistics for the open-source OAuth server libraries in $\texttt{DataSet}_U$. Here, AC and I indicate the OAuth's authorization code grant and implicit grant, respectively.

4.4.1.2 State-of-the-arts.

Existing tools focus on OAuth properties by analyzing the flows observed from client-side applications while treating the server-side as blackbox. Therefore, there is no prior work that directly performs whitebox analysis on OAuth server programs. *OAuthlint* is the closest to our work as they also statically analyze source code of client applications to check OAuth properties. Unlike *Cerberus*, *OAuthlint* only supports OAuth properties expressed via data-flow predicates. Furthermore, *OAuthlint* performs whole-program analysis over the client application. *S3kvetter* (Yang et al., 2018a) leverages symbolic execution to check security properties in OAuth SDKs, which provides APIs for implementing client applications. Similar to *OAuthlint*, *S3kvetter* also consider server-side implementation as blackbox. In fact, *S3kvetter* implements it's own *model* of the server to analyze the

client-side flows for all of the SDKs. Its properties are defined in terms of the request-response behavior observed from client apps. Therefore, we cannot compare against *S3kvetter* because of unsupported properties (e.g., checking expressions, API calls, etc.) and unsupported languages (e.g., *S3kvetter* only supports Python).

4.4.1.3 User inputs

Cerberus takes as inputs a target program P and a property Q. For each benchmark, we check it against all properties defined in Table 4.2. To quantify the manual effort of writing the queries before running the evaluation, we recruited 18 independent students from a graduate-level security class – who were provided with the OAuth properties (i.e., from specification written in English) and documentation for our query language. We provide five randomly selected properties to each participant and measure the time they spent expressing the properties in our query language. We found participants spent on average 10.6 minutes to specify the given properties using our query language.

To speed up the analysis, *Cerberus* provides built-in OAuth endpoints (as defined in the specification) to leverage the query-driven slicing method (Sec. 4.3.3). Meanwhile, we allow users to provide (as query) their own custom endpoints in regex format to check any additional endpoints that is not covered by the standard specification. Thanks to the library documentation, these endpoints' format are clearly specified for most libraries and can be easily defined using regex for our tool. This flexibility also allows the analysts to use *Cerberus* to efficiently check security properties in extended or custom endpoints (e.g., token introspection) as well as new OAuth extensions (e.g., DPoP token (IETF, 2022)).

4.4.2 Evaluation Results

4.4.2.1 Discovered Vulnerabilities

To answer **RQ1**, we use *Cerberus* to identify logical flaws from both known and unknown datasets.

Detection of known vulnerabilities. We first evaluate *Cerberus* using $\texttt{DataSet}_K$ which consists of 12 popular OAuth libraries with 49 known flaws. As shown in Table 4.6, using the properties from Table 4.2, *Cerberus* identifies 43 out of 49 logical flaws.

4.4.2.2 Unknown vulnerabilities

We next evaluate *Cerberus* using $\texttt{DataSet}_U$ with ten popular open-source libraries of OAuth service providers.

OAuth-server libraries	#Known bugs	#Detected bugs	#FN
1) OxAuth	9	8	1
2) Mitre Server	7	6	1
3) Spring Auth Server	6	6	0
4) Node OAuth2 Server	6	4	2
5) Loopback OAuth	4	4	0
6) OAuth Provider	4	3	1
7) Egg OAuth2 Server	4	4	0
8) ApiFest OAuth	1	1	0
9) Clouway Server	3	3	0
10) OAuth2orize	3	2	1
11) Java OAuth Server	1	1	0
12) Connect OAuth2	1	1	0
Total	**49**	**43**	**6**

Table 4.6: Evaluation results of *Cerberus* on $\texttt{DataSet}_K$.

Ground truth determination. To determine the ground truth for $\texttt{DataSet}_U$, we perform both dynamic analysis and source code inspection with the OAuth flows simulated for each library. We deploy each library on a local server and implement client-side programs to simulate the OAuth flows according to the specification. In particular, for each property, we manually construct the parameters to initiate the corresponding OAuth requests and analyze the responses from the server. For example, to verify the result reported by *Cerberus* for redirect URI property (P1), we first deploy the library on a local server and create an authorization server instance. We create two client instances–one with a benign redirect URI and another with a malicious one. We initiate an authorization request from the benign client, but replace the '*redirect_uri*' parameter with the redirect URI from the malicious client. We observe the traces generated by the server. If the server redirects the authorization response (with *authorization code*) to the malicious URI,

we mark a violation to the redirect URI property. Even if *Cerberus* reports no property violation, we still investigate the library, in the same way to find if the library actually violates the property.

OAuth-server libraries	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**	**P9**	**P10**	**#Vuln.**
1) Node oauth2 server	✓	×	✓	✓	✓	×	×	✓*	×	×	5
2) Oauth2orize	✓	×	×	✓	✓	×	×	×*	×	×	6
3) Node oidc provider	✓	×	✓	✓	✓*	✓	✓	✓	✓*	✓	1
4) Oauth2 server	×	×	✓	×*	✓	×	×	×	✓	✓	5
5) Spring auth server	✓	×	×	✓	✓	×	×	✓	✓	✓	4
6) Clouway server	✓	×	×	×*	✓	✓	✓	✓	×	×	4
7) Jobmission server	×*	×	×	×	×	×	×	×*	×	✓	7
8) Apifest	×	✓	✓	✓	×*	×	×	✓	×	×	5
9) Yoichiro server	×*	×	×	×	×	×	×	×	×	✓	8
10) OxAuth	✓	×	✓	✓	×	✓	✓	✓	✓	✓	2
Total vulnerabilities:	2	9	6	2	3	7	7	2	5	4	47

Table 4.7: Evaluation results of *Cerberus* on $\texttt{DataSet}_U$ whose vulnerabilities are unknown.

As presented in Table 4.7, we discover 47 confirmed vulnerabilities in the ten popular libraries of the authorization server, 24 of which were previously unknown. Here, ✓means that the library satisfies the property, and × means the library violates the property. Moreover, $\times^*$ indicates a false positive (FP), where library satisfies the desired property, but *Cerberus* marks it as a violation due to the limitation of the underlying static analysis. On the other hand, $\checkmark^*$ indicates false negative (FN), which could occur due to imprecisely resolved call-sites.Security impacts of these vulnerabilities can cause severe attacks on the authorization server. We discuss these vulnerabilities and their implications below:

Redirect URI manipulation. *Cerberus* successfully identifies two libraries violating the property that the redirect URI from authorization request must match the URI registered by the client (P1). In addition, *Cerberus* finds nine libraries violating the property that the redirect URI must be an *absolute* URI (P2). For example, to check if the redirect URI parameter contains an absolute URI, the Node oauth2 server library matches it with an incorrect URI pattern "[a-zA-Z][a-zA-Z0-9+.-]+:". Violation of these properties allows the attackers to manipulate the redirect URI while initiating the authorization request and obtain a legitimate client's authorization code or access token directly from the authorization server. Our finding indicates that developers often do not follow the

requirement for redirect URI, possibly because it may not seem security-critical from reading the specification.

Authorization code injection. *Cerberus* successfully identifies six libraries violating at least one property required to protect the clients against the authorization code injection attack. We find six libraries omitting or incorrectly implementing the property that the authorization code must be single-use (P3). This vulnerability allows the attackers to replay the authorization code intercepted from a legitimate client's request and obtain the access token. We also identify two libraries that do not bind the authorization code to any particular client (P4), allowing the attackers to exchange the code using a malicious client application. Additionally, *Cerberus* successfully identifies four libraries that do not bind the authorization code to the redirect URI (P5), which prevents the authorization server from cross-checking the redirect URI with the one submitted at authorization endpoint. *Cerberus* also finds seven libraries that do not support PKCE (P6 & P7)–a recommended mechanism designed for the authorization server to mitigate authorization code injection attacks for public clients.

Access token injection. *Cerberus* successfully identifies five libraries vulnerable to the access token injection. All five libraries do not issue client-constrained access tokens from the token endpoint (P9) which limits the applicability of the token to a particular client. It prevents the attackers from using an access token that is obtained through a malicious client application. *Cerberus* also identifies four libraries that store the access token in clear text format (P10). Storing access tokens as clear-text is vulnerable since it can expose the tokens to the attackers through accessing the local storage or launching a SQL injection attack.

CSRF attack. *Cerberus* successfully identifies two libraries that do not provide CSRF protection using OAuth's *'state'* parameter (P8). By design, libraries supporting the PKCE are CSRF-protected. Therefore, libraries without PKCE must handle the authorization request's *'state'* parameter to protect their clients against CSRF. *Cerberus* reports a library as vulnerable if it neither supports PKCE, nor handles *'state'* parameter.

Property	***OAuthlint*** (no slicing) Exec. time (min)		***Cerberus*** (with slicing) Exec. time (min)	
	$\texttt{DataSet}_U$	$\texttt{DataSet}_K$	$\texttt{DataSet}_U$	$\texttt{DataSet}_K$
P6	48.3	64.8	**2.1**	**3**
P9	55.8	71.6	**2.5**	**3.7**
P10	34	52.9	**1.7**	**2.2**

Table 4.8: Comparing *Cerberus* against *OAuthlint* in terms of average running time using the $\texttt{DataSet}_U$ and $\texttt{DataSet}_K$.

4.4.2.3 Analysis Accuracy

Cerberus successfully identifies 47 property violations (Table 4.7) and reports 7 false-positive (FP) cases and 3 false-negative (FN) cases in $\texttt{DataSet}_U$, and 6 FN cases (Table 4.6) in $\texttt{DataSet}_K$. Our assessment finds the primary reason for the FP and FN cases is the spurious points-to targets from the WALA framework. Additionally, unsoundness can also occur for a few cases during Javascript library analysis. We summarize the reasons for the FP and FN cases in the following:

- Semantics of complex string operations (e.g., `subString` or `concate`) cannot be precisely modeled by WALA–which leads our analysis to be unable to keep track of the precise values held by some string variables.
- Semantics of functions for executing dynamically generated code (e.g., Javascript's `eval`) are not currently modeled as we do not reason dynamic code in this work.
- For reflected calls where a reference to a target method is computed at runtime (e.g., Java's `invoke`), our static analysis may not accurately resolve the target method, which can result in FNs.
- Javascript libraries with dynamic features (e.g., dynamic module loading) may cause WALA to generate a broken and imprecise callgraph that leads to FPs and FNs.

4.4.2.4 Acknowledgments from Developers.

We reported all the unknown vulnerabilities to the developers of the libraries. So far, we have received confirmation from the developers of eight libraries, who have acknowledged our findings. By the time of writing this dissertation, five libraries had fixed the redirect URI, PKCE, and authorization code issues as we reported; the other remaining issues are undergoing a review process. Developers of the two libraries responded that they are taking action on our reports, and the issues will be addressed in their next release. Developer of one library confirmed the issues we reported, but declined to take action as they no longer work for the corresponding vendor. Three classes of unknown vulnerabilities in these libraries led us to have three new CVE entries (CVE-2020-26877, CVE-2020-26937, and CVE-2020-26938). However, we have not yet received feedback from two libraries. We are making our best effort to reach the developers through different channels and help them fix the vulnerabilities.

Result for RQ1: We identified 47 vulnerabilities in ten popular OAuth libraries, including developers' acknowledgment from eight libraries and three accepted CVE entries.

4.4.2.5 Performance Evaluation

To answer **RQ2** and evaluate the effectiveness of our query-driven algorithm discussed in Sec. 4.3.1, we compare *Cerberus* with its variant, which constructs the callgraph for the entire program eagerly (noted as $Cerberus_{eager}$). Unlike *Cerberus*, which constructs an on-demand callgraph based on the OAuth query and prunes away any program components that are irrelevant with respect to the query, $Cerberus_{eager}$ does not consider any relevance with the query. Instead, it constructs the callgraph eagerly for the whole program. We execute our queries with both $Cerberus_{eager}$ and *Cerberus* and compare their performances.

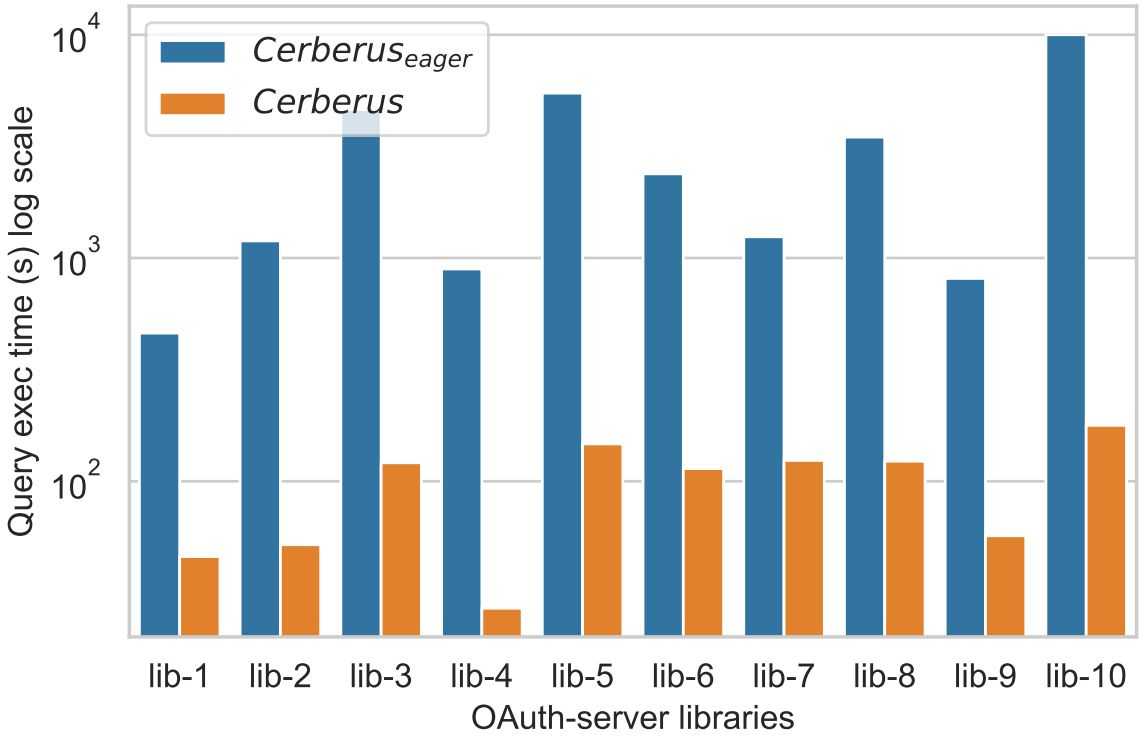


Figure 4.4: Performance comparison for libraries in $\texttt{DataSet}_U$ (Table 4.5) based on average query execution time (in log scale) between *Cerberus*.

Figure 4.4 shows the average query execution time for the two different design choices of callgraph construction by our tool: *Cerberus* and $Cerberus_{eager}$. Y-axis shows the average time in seconds (in log scale) to answer queries for each library (X-axis) selected for the evaluation. Our analysis shows that our tool, *Cerberus*, which executes queries over an on-demand callgraph, is on average 25× faster than $Cerberus_{eager}$, which eagerly constructs callgraph. For example, *Spring Authorization Server* and *Node Oidc Provider* libraries take more than one hour to answer an OAuth query using $Cerberus_{eager}$. In Figure 4.4, bars touching the maximum time (10^4) indicate the library not terminating or causing memory explosion within the maximum time limit (3 hours). On the other hand, the same query is answered by 147 and 121 seconds, respectively, using the on-demand approach by *Cerberus*. This significant query execution time with $Cerberus_{eager}$ is because, by design, the analysis also visits the program components that are not directly relevant to the OAuth protocol (e.g., database models and operations). On the other hand, *Cerberus*

only visits the program components relevant to the submitted OAuth query and therefore requires significantly less time to answer a given query.

However, we limit the execution time to 3 hours for executing the queries. In other words, the query execution automatically terminates if it can not find any result by 3 hours. We find that one of the large libraries, *OxAuth* ran into a memory explosion after running for more than 2 hours using $Cerberus_{eager}$. However, it takes 178 seconds to answer the same queries using *Cerberus*, which shows the effectiveness of our tool for analyzing large-scale implementations.

Comparison with existing tools. We further extend our effort to compare *Cerberus* against *OAuthlint*, which only supports OAuth properties with data-flow predicates. We managed to encode only three of our properties (out of ten) using their predicates with our best effort. As shown in Table 4.8, for the three properties supported by both tools, *Cerberus* is 22× and 21× faster than *OAuthlint* on $\texttt{DataSet}_K$ and $\texttt{DataSet}_U$, respectively.

Result for RQ2: Our query-driven analysis is 25× faster in checking OAuth security properties than the eager analysis.

4.4.2.6 Extensibility

To show the extensibility of *Cerberus*, we use a recent property from DPoP (Demonstrating Proof-of-Possession) (IETF, 2022), which is a new draft describing a mechanism for sender-constrained OAuth token. When sending a token request, the client sends an additional DPoP token in JWT format (JWT, 2015) that allows the server to verify the proof of possession of the token. The server first verifies the token using the key embedded within the token. If verified, it adds the thumbprint of the key with the access token. Otherwise, the token request is rejected with an error response. One can use *Cerberus*'s predicates to express this query as follows:

```
1. P :- invoke(L1, token_req), OAuthTag(L2, dpop),
2.      OAuthTag(L3, key), invoke(L4, jwt_verify),
```

```
3.      OAuthTag(L5, access_token), invoke(L6, add_thumbprint),
4.      error(L7, _), branch(L8,X,L6,L7), followBy(L1,L2),
5.      followBy(L1,L5), depOn(L2,L4),depOn(L3,L4),
6.      depOn(L4,X), depOn(L5,L6), depOn(L3,L6).
```

As shown from the above query, it is straightforward for a domain expert to use *Cerberus*'s built-in predicates to define new properties. Specifically, given a new property, a domain expert only needs to encode the new domain-specific information using predicates `OAuthTag` and `invoke`. E.g., for the DPoP property, the extra taint sources will be `dpop`, `key` and `jwt_verify`.

4.5 Discussion

While designing *Cerberus*, we strived to achieve a good trade-off between expressiveness and scalability. For complex semantics (e.g., secure computation, storage modeling, etc.) that are difficult to model precisely, our current static analysis leverages data- and control-dependencies to over-approximate the actual semantics, which, in theory, can lead to spurious execution paths. However, as discussed in Sec. 4.4, our tool achieves a low false positive rate despite our modeling. Secondly, dynamic features of Java and Javascript, such as reflective calls, dynamic class loading, and exceptional handling can result in false negatives. Our current implementation of hybrid analysis currently provides limited support to handle dynamic features (e.g., reflective calls with string constants).

Our study in this chapter focuses on OAuth 2.0, a prevalent multi-party protocol for authorization. *Cerberus* is designed based on the standard OAuth specification (RFC6749, 2021). Therefore, any implementation that follows the standard specification can be analyzed using our proposed approach, which exhibits the generality of our work. Additionally, *Cerberus* can be extended to other relevant protocols like OpenID Connect (OpenID, 2021). Since OpenID Connect uses similar grants and flows as OAuth 2.0, the vulnerabilities we address in this chapter are also applicable for OpenID Connect supported servers. In particular, *Cerberus*'s predicates can be extended to check OpenID properties

using our query-based method. However, as our current analysis does not model the cryptographic APIs (e.g., RSA verification), some OpenID Connect flows (e.g., hybrid flow) that involve cryptographic operations cannot be checked by our tool. Additionally, it would be an interesting future work to apply *Cerberus* on a large scale of OAuth service provider implementations to check the prevalence of the issues we identified. The results might be similar because we study very popular libraries for OAuth service provider implementations, and developers usually just call these libraries' APIs instead of building their own implementations. Even worse, if developers start from scratch to build their service providers, they might make more security mistakes. In addition to OAuth, our proposed method can also be applied to cross-check applications against policy regulations such as GDPR (Rahat et al., 2021) and CCPA (Goldman, 2020).

Finally, our analysis in this dissertation is focused on the attacks that utilize incorrect or logical implementation mistakes made during the OAuth flow. Our tool is designed to find violations in the implementation with respect to the queries representing the violation pattern. While our properties cover the OAuth-specific attacks commonly observed on the OAuth servers, attackers might exploit other vulnerabilities to attack the server. For example, Attackers might leverage generic web/mobile vulnerabilities and perform web-based/mobile attacks (e.g., SQL-injection) to steal OAuth credentials or protected resources of the resource owner. Detecting those generic vulnerabilities is beyond the scope of this dissertation.

4.6 Summary

In this chapter, we presented *Cerberus*, an automated analyzer that can discover logic vulnerabilities in OAuth server libraries that service providers widely use. To efficiently detect OAuth violations in large codebases, *Cerberus* employs a query-driven algorithm for answering queries about security-critical OAuth properties. To demonstrate the effectiveness of *Cerberus*, we evaluate it on datasets of popular OAuth server libraries with millions of downloads. Among these high-profile libraries, *Cerberus* has discovered

47 vulnerabilities from ten classes of logic flaws, 24 of which were previously unknown and led to new CVEs.

CHAPTER 5

Safety Verification of OpenID Connect

5.1 Introduction

This chapter delves into the OpenID Connect (OIDC) protocol, a robust extension of OAuth that enhances security features. In addition, OIDC provides authentication capabilities, offering a comprehensive solution for secure authentication and authorization. OIDC contributes to a seamless and efficient user experience, leading to increased productivity and reduced support costs. Through integration with diverse authentication service providers, OIDC establishes a fundamental foundation for convenient, secure, and scalable identity management. Therefore, ensuring the security and correctness of the implementation of these protocols is of utmost importance.

While OIDC protocol offers numerous benefits, it can be vulnerable to severe security attacks if not implemented correctly. Recent years have witnessed a growing number of security attacks across the web stemming from these implementation errors. For instance, in 2023, a bug (Microsoft, 2023b) in Microsoft's OIDC implementation allowed hackers to breach over two dozen organizations via forged tokens used to access Azure service. Grammarly recently fixed a security flaw (Grammarly, 2023) in its implementation of access token verification which could allow attackers to hijack user accounts. Similar OIDC vulnerabilities have been discovered earlier in popular online services like Booking and Expo (Expo, 2023). The presence of errors in the protocol implementation of these

esteemed authentication service providers underscores the urgency of developing analysis tools to enhance the security of OIDC implementation.

Prior research efforts in developing security analysis tools for SSO protocols can be, in general, divided into two categories: bug detection and verification. Most bug detection techniques (He et al., 2015a; Rahat et al., 2019, 2022) look for syntactic or semantic patterns that are highly correlated with security vulnerabilities (e.g., *token leakage*) observed in the protocol flows. These patterns often require upfront knowledge about the lower-level details (e.g., specific data flows, source and sink APIs, etc.) of the programs, leading them to be error-prone and incomplete. In addition, most of these solutions rely on approximated static analysis and lack precision in practice. These limitations not only reduce the scope of checkable properties but also lead to significant false positive and false negative cases. On the other hand, previous efforts in formal verification within this domain (Fett et al., 2017; Hammann et al., 2020) concentrate on protocol design rather than its implementation. Moreover, these approaches require manual translation of both protocol interactions and specifications into corresponding formal models, which can then be consumed by off-the-shelf verifiers like Tamarin Prover (Meier et al., 2013) or Alloy (Jackson, 2012). However, manually or semi-automatically translating complex interactions within the real-world protocols' programs can be error-prone and inefficient, rendering them impractical for large-scale evaluations. Consequently, it is crucial to design an automated verification toolkit to ensure that the protocol implementation conforms to its safety specifications.

However, developing an automated verifier for checking OIDC programs against their standard safety specifications is quite challenging, primarily due to the following reasons: 1) **C1 (Expressiveness):** The core specification of OIDC (OpenID, 2021) is written in dozens of pages in plain English, and incorporates more than ten other protocols (e.g., OAuth (RFC6749, 2021), JWT (JWT, 2015), etc.), making the security behavior of the protocol challenging to express. Additionally, many safety properties require complex reasoning about *temporal relationships* that go beyond the expressiveness of prior works (Rahat et al., 2019, 2022) relying on simple data- and control-dependencies.

2) **C2 (Multi-party interactions):** Unlike traditional programs that have single or a few entry points, OIDC implementations typically involve complex interactions among multiple parties (e.g., identity providers, relying parties, etc.). Prior works in this domain only analyze a portion of the protocol (e.g., the relying parties (Bai et al., 2013; He et al., 2015a; Rahat et al., 2019) or the providers (Rahat et al., 2022)), and consider the remaining pieces as black boxes–which significantly limits the assurance guarantee and the scope of safety properties that can be verified. On the other hand, a naive approach can lead to prohibitive performance issues by enumerating all possible combinations of interactions or false negatives by choosing interactions in an ad-hoc manner. Therefore, it is non-trivial to automatically model the complex interactions among different parties into a unified model. 3) **C3 (Automation):** Mainstream OIDC implementations are coded in high-level programming languages like Java and Javascript. Consequently, neither OIDC programs nor their specifications (containing domain-specific information) can be directly analyzed by existing program verification tools (Leino, 2010; Dietsch et al., 2015). Therefore, devising an automated approach that accommodates complex protocol features implemented in various programming languages is both critical and challenging.

To address the above-mentioned challenges, we design and implement *AuthSaber*, an automated verifier for checking OIDC implementations against their security behavior (i.e., safety properties) defined by the standard specification. To address the challenge **C1**, we design AuthLTL, a highly expressive specification language that is extended from the standard Linear Temporal Logic (LTL). In particular, AuthLTL augments standard LTL with more expressive OIDC predicates that allow developers to formally express the desired safety properties of the protocol, including the temporal relations between interactions and events involving multiple parties during the authorization and authentication flows.

Additionally, to address **C2**, we propose a staged approach that iteratively constrains the space of valid candidates without compromising the assurance of the verifier. In particular, given an OIDC implementation together with its *Interaction Dependence Graph* (IDG) that over-approximates the interactions among different entry points, we leverage

an automaton-based algorithm to constrain the space of potential interactions among multiple parties using the *flow constraints* defined by the specification. The output of this step will lead to an effective harness that only considers interactions among different entry points that are consistent with the flow constraint enforced by the specification and ignores the spurious interactions.

Moreover, to address **C3**, we design several customized transformations that convert OIDC implementations and their specifications to corresponding forms that can be analyzed by existing software model checkers (Dietsch et al., 2015). For OIDC implementations, we first compile them to WalaIR (WALA, 2023) and then perform a transpilation from WalaIR to Boogie IR (Barnett et al., 2006), which can be consumed by the model checker. On the other hand, since the properties expressed in AuthLTL contain terms and predicates that are not accepted by the model checker, we devise a customized transformation that converts an AuthLTL formula into its *equisatisfiable* formula in standard LTL while instrumenting the transpiled program. The model checker then consumes the resulting new formula and program, and generates a counter-example if there is a violation.

Findings. We evaluate *AuthSaber* on the most popular implementations of OIDC protocol which is currently supported by all major authentication providers. We verify 24 safety properties across the 15 most popular OIDC implementations and uncover critical security violations. In total, we identify 16 previously undiscovered vulnerabilities, posing risks of severe attacks such as user account takeover and authentication bypass. We responsively disclosed all vulnerabilities and received acknowledgments for 10 vulnerabilities from the developers and prominent providers, including Google. Furthermore, the confirmed violations resulted in the assignment of five new CVEs (CVE-2023-33292, CVE-2023-35819, CVE-2023-35820, CVE-2021-44878, CVE-2021-22573). Additionally, *AuthSaber* also effectively verified the correctness of patched versions addressing the known violations previously reported.

5.2 Overview

We now outline the threat model followed by a running example of OIDC implementation to discuss the overview of *AuthSaber*.

5.2.1 Threat Model

Our objective in this work is to verify the OIDC protocol implementations (i.e., programs) against the standard security behavior expressed as safety properties. We assume that an attacker may attempt to impersonate legitimate users and/or other participating entities such as relying parties and service providers to gain unauthorized access to the resources of users and clients. Hence, malicious actors can be users, client applications, and even authorization servers (i.e., deployed by attackers) during the protocol execution. Attackers may exploit vulnerabilities in the authentication flows implemented by these entities, aiming to gain unauthorized access or privileges. In addition, attackers may intercept and modify communication exploiting any insecure client applications or devices running the applications. Precisely, attackers may utilize the supported OIDC endpoints (OpenID, 2021) to exploit the authentication access control flow during the protocol execution. This exploitation may also enable them to eavesdrop on, tamper with, or inject malicious parameters (e.g., tokens) into the protocol endpoints, thereby compromising the integrity and confidentiality of the authorization and authentication. It is essential to note that we assume attackers do not possess the capability to directly modify the source code or binaries of server-side implementations. Additionally, they are restricted from direct access to the internal storage or databases of the authorization servers for retrieving information.

```
1  void processTokenResponse(TokenResponse res) {
2    UserInfo info;
3    String atoken = res.getParam("access_token");
4    if(verifyToken(res)) {
5      info = userInfoRequest(atoken);
6      loginUser(info);
7    } else {
8      authError(); //Reject token
9    }
10 }
11 boolean verifyToken(TokenResponse res) {
12   Client c = Registration.client;
13   IdToken itoken = res.getParam("id_token");
14   Algorithm alg = itoken.getAlgorithm();
15   boolean v = false;
16   if(alg == "none") {
17     //Incorrect: using "none" algorithm
18     //forces to skip signature verification
19     v = JWT.verify(itoken, alg);
20   }
21   else if(alg == "hs256") {
22     //Verifies signature using client secret
23     v = JWT.verify(itoken, alg, c.client_secret);
24   }
25   [...]
26   if(!v) {
27     return false;
28   }
29   return true;
30 }
```

Figure 5.1: Motivating code example of an OIDC relying party incorrectly validating the `id token` received as a response from the provider during the authentication flow.

5.2.2 Motivating Example

Figure 5.1 illustrates a critical security mistake when validating the `id token` received from the OIDC provider during an authentication flow. Here, `id token` contains a cryptographically verifiable signature component that prevents attackers from injecting maliciously crafted tokens. Therefore, incorrect validation of `id token` could be exploited to *bypass authentication* during the execution of OIDC protocol. The example is motivated by a vulnerable implementation of OIDC protocol from the Pac4j (Pac4j, 2023), a popular library for identity access management. This example first processes the response (line 1) received from the token endpoint, which contains an `access token` and an `id token`.

The `id token` contains a signature that is generated by signing its header and payload component, which typically are not protected. The code calls the verifyToken method (line 4) to verify the `id token`. Once the `id token` is successfully verified, it uses the `access token` (line 5) at the *UserInfo* endpoint to obtain the user's information to login the user. If the `id token` is not verified, the token is rejected and the flow is terminated with an error response (line 8). In verifyToken function, it first extracts the header algorithm (line 14) of the `id token`, and based on the algorithm value, it uses the verify function provided by JWT (JWT, 2015) library to verify the signature component (line 16-24) of the token.

Safety Properties. When processing Id Tokens, an important safety property is: (Φ_1) '*once an* `access token` *is received along with an* `id token`*, it should not be used at the user info endpoint unless the* `id token` *is verified' (OIDC ref. §3.1.3.5).* At first glance, the code example seems to satisfy the property Φ_1. However, there is an edge case relevant to the `id token`'s header algorithm, which leads to a subtle bug due to the discrepancy between the original specification and the actual implementation. Particularly, using the '*none*' as algorithm value enforces the verification algorithm to skip the signature component of an `id token`. Even though the '*none*' value must not be used by OIDC for validating `id token`, it is accepted as a valid algorithm for JWT tokens. Therefore, another important safety property to check is: (Φ_2) '*if a received* `id token` *uses 'none' as the header algorithm value, the associated* `access token` *should not be used at the user info endpoint' (OIDC ref. §2, §3.1.3.7).*

Unfortunately, the code example depicted in Figure 5.1 violates the property Φ_2, as it accepts '*none*' as a valid algorithm when verifying the signature of the `id token`. Since using '*none*' as the algorithm value enforces the verify function to ignore the signature component, it eventually allows attackers to bypass authentication simply by modifying the algorithm value within the unprotected header component of the `id token`.

Verification with *AuthSaber*. *AuthSaber* can uncover such violations by verifying OIDC implementations against their safety properties that describe the unexpected security behavior. Figure 5.2 illustrates the schematic workflow of our safety verification approach.

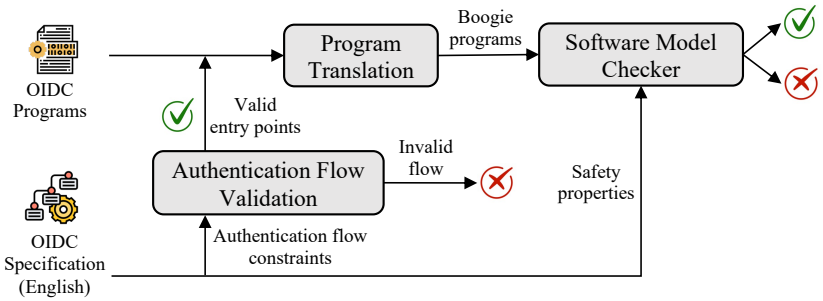


Figure 5.2: Schematic workflow of *AuthSaber*.

To use *AuthSaber*, the user first needs to provide the security properties expressed in AuthLTL, the specification language in *AuthSaber* that supports *linear temporal logic*. For example, the requirement to verify the `id token` (Φ_1) can be expressed as follows in AuthLTL:

$$\Box(\texttt{response}(s, arg(\mathsf{access_token}) = t \wedge arg(\mathsf{id_token}) = i) \Rightarrow \neg\, \texttt{request}(c, arg(\mathsf{access_token}) = t)\ \mathcal{U}\ \texttt{verifyToken}(i, ret = \mathsf{true}))$$

This AuthLTL property states that if a response from the authorization server s contains an `access token` t and an `id token` i, a request using the token t from client c should not be sent until the `id token` i is successfully verified. Since an `id token` is expected to contain a *signature* component that is usually verified by calling an external JWT (JWT, 2015) function (e.g., verify), a true value returned from the external call indicates the successful verification of the token. Similarly, the property of handling the '*none*' algorithm for `id token` (Φ_2) can be expressed in AuthLTL as follows:

$$\Box(\texttt{response}(s, arg(\mathsf{access_token}) = t \wedge arg(\mathsf{id_token}) = i \wedge i.alg = \mathsf{none}) \Rightarrow \neg\Diamond\texttt{request}(c, arg(\mathsf{access_token}) = t))$$

This property states that if `access token` t is received (as a response from the authorization server s) with an `id token` i whose alg header value is '*none*', a subsequent request (e.g., *user info* request) using the `access token` should not be sent.

AuthSaber then verifies the OIDC programs against the safety properties expressed

in AuthLTL. However, unlike regular programs, an OIDC program typically involves multiple entities (e.g., relying party, authorization server, etc.) and each of them has many entry points to interact with each other. Therefore, constructing a naive harness required by the model checker can result in both false negatives (i.e. invoking entry points in an ad-hoc manner) and scalability problems (i.e., enumerating all possible interactions). To mitigate these challenges, *AuthSaber* takes additional authentication flow constraints as input (defined by the standard specification) and preserves a subset of interactions that are consistent with authentication flow constraints, which dramatically reduces the complexity of verification without compromising soundness.

In particular, given an OIDC program, *AuthSaber* first validates the authentication flow constraints in the programs. If the programs violate the flow constraints, the verification terminates without checking the safety properties. Otherwise, the flow validation procedure returns a harness with a set of entry points that satisfy the flow constraints.

To verify the implementation of resulting valid entry points of OIDC protocol against their safety properties in AuthLTL, *AuthSaber* utilizes an off-the-shelf LTL model checker (Dietsch et al., 2015). However, mainstream OIDC programs are mostly written in high-level programming languages like Java and Javascript, and these programs are not directly accepted by existing LTL model checkers. Furthermore, to ensure expressiveness, AuthLTL introduces domain-specific constructs that go beyond the standard syntax of LTL and are not consumable by the model checker. To mitigate these challenges, *AuthSaber* first translates the input programs into Boogie (Leino, 2008) – a language-agnostic representation that makes the verification task tractable for model checking. Second, it automatically transforms the specification in AuthLTL into their *equisatisfiable* regular LTL formulas through our customized program transformation. Finally, *AuthSaber* verifies the programs against the transformed safety properties in LTL. If the programs are not verified, it returns a counter-example that implies the violated paths in the program.

5.3 System Design

This section outlines the design and implementation details of *AuthSaber*, an automated safety verification tool specifically crafted to address the scalability challenge posed by multi-party interactions within the OIDC protocol.

5.3.1 Specification Language

We begin with the details of our specification language that can be used for expressing the OIDC safety properties based on the standard specification.

Syntax. Figure 5.3 illustrates the formal syntax of our AuthLTL specification language. AuthLTL is designed to formally express the safety properties of OIDC that developers or security analysts intend to verify. Notably, AuthLTL comprises atomic propositions featuring logical and temporal connectives. However, unlike the standard LTL where the formulas are defined over propositional variables, AuthLTL formulas are defined over the OIDC-specific predicates.

$$
\begin{array}{lcl}
\varphi & ::= & \psi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \Rightarrow \varphi \mid \varphi \Leftrightarrow \varphi \mid \neg\varphi \\
& & \mid \Box\, \varphi \mid \Diamond\, \varphi \mid \bigcirc\, \varphi \mid \varphi\, \mathcal{U}\, \varphi \\
\psi & ::= & \phi \mid \mathsf{verifyToken}(\mathsf{t}, \phi) \mid \mathsf{authUser}(\mathsf{u}, \phi) \\
& & \mid \mathsf{request}(\mathsf{d}, \phi) \mid \mathsf{response}(\mathsf{s}, \phi) \mid \mathsf{check}(\mathsf{f}, \phi) \\
& & \mid \mathsf{call}(f, \vec{v}, \phi) \mid ... \\
\phi & ::= & e\ \mathtt{comp}\ e \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \\
e & ::= & v \mid c \mid e.f \mid f(\vec{v}) \mid e\ \mathtt{op}\ e \mid \mathsf{arg}(\mathsf{c}) \mid \mathsf{conf}(\mathsf{c})
\end{array}
$$

Figure 5.3: Syntax of AuthLTL specification.

Expressions. AuthLTL expressions include variables v, constants c, field variables $e.f$, binary operations *op*, and function calls $f(\vec{v})$. We provide an additional construct $\mathsf{args}(\mathsf{c})$ that returns the values mapped against a key constant c in the context of a request or response instance. Similarly, $\mathsf{conf}(\mathsf{c})$ returns the configured value against the key c. We found these constructs very useful as OIDC properties require reasoning about the

protocol-specific request and response parameters mapped against certain string constants. For instance, $\mathsf{arg}(\mathsf{state})$ returns the value of the "*state*" parameter used in a request used during the protocol interactions. Expressions can be composed using standard arithmetic and logical operators to construct more complex expressions ϕ.

OIDC predicates. Our domain-specific atomic predicates ψ work as the cornerstone of AuthLTL formulas as they refer to the events and actions that occur during the execution of the protocol implementation. For example, predicate $\mathsf{verifyToken}(\mathsf{t}, \phi)$ is true if the signature of an `id token` t is verified in a context that satisfies predicate ϕ. In the context of OIDC, $\mathsf{verifyToken}(\mathsf{t}, \phi)$ provides an abstraction of the external JWT (JWT, 2015) API calls that are commonly used to verify the signature component of the `id token` with respect to its header algorithm. Similarly, $\mathsf{userAuth}(\mathsf{u}, \phi)$ predicate is true if an authorization (from user u) satisfying the predicate ϕ is obtained during the protocol flow. Moreover, $\mathsf{request}(\mathsf{d}, \phi)$ and $\mathsf{response}(\mathsf{s}, \phi)$ predicates are true if an HTTP request is sent to an entity d and a response received from an entity s, respectively, in a context satisfying ϕ. Finally, $\mathsf{check}(\mathsf{f}, \phi)$ is true if a condition satisfying ϕ is explicitly checked by function f and $\mathsf{call}(f, \vec{v}, \phi)$ is true if function f is called with arguments $\vec{v}$ in a context satisfying ϕ. Along with variables, predicates also accept "_" (underscore) to represent "don't care".

Temporal operators. It is common for OIDC protocols to check properties that require reasoning about the temporal modality of protocol events. Temporal operators in our specification language include always ($\Box$), eventually ($\Diamond$), next ($\bigcirc$) and until ($\mathcal{U}$). $\Box\varphi$ signifies a universal assertion, indicating that condition φ holds true at all steps during the program execution. Conversely, $\Diamond\varphi$ asserts that φ will become true at some point in the future. $\bigcirc\varphi$ signifies an immediate successor, stating that φ must hold true in the very next time step. Finally, $\varphi_1 \mathcal{U} \varphi_2$ defines a relationship between two conditions, specifying that the first condition φ_1 remains true until φ_2 becomes true, encapsulating the concept of persistence and change over time. These operators enable precise specification and reasoning about the protocol's execution steps, making them invaluable tools for analyzing the security behavior and safety of OIDC programs. Additionally, AuthLTL formulas can

be combined with standard logical connectives (e.g., $\wedge$, $\vee$, $\Rightarrow$, and $\neg$) to express more complex properties.

Example 3. *When responding to the authorization code flow to perform authentication, the server returns an authorization* `code` *(OpenID spec. §3.1.2) that can be exchanged for an* `id token` *and (or) an* `access token`*. This flow prevents exposing any tokens directly to the user agent and other malicious client applications. However, the specification requires authenticating the user before sending the* `code` *to the client. Since the methods used by the server to authenticate the end user vary for different service providers, AuthLTL provides* authUser *predicate to model user authentication. In addition, gencode is an OIDC API that returns the* `code` *generated by the server. We encode this property in AuthLTL as follows:*

$$\begin{aligned}\varphi \;:=\; & \Box(\mathsf{request}(s, arg(\texttt{response_type}) = \texttt{code} \\ & \wedge\ session = u) \Rightarrow \neg\ \mathsf{response}(c, arg(\texttt{code}) = \\ & gencode(u, _))\ \mathcal{U}\ \mathsf{authUser}(u, ret = \texttt{true}))\end{aligned}$$

5.3.2 Program Translation

Real-world OIDC protocols are implemented in high-level programming languages like Java and Javascript which are not directly consumed by the model checkers. Hence, we first transform the original OIDC programs to its corresponding Boogie IR (Leino, 2008). Specifically, we begin by compiling the OIDC programs to WalaIR (WALA, 2023), a stackless static-single assignment (SSA) form consisting normal statements like assignments, function calls, conditional jumps, and so on. We then encode the WalaIR statements to their corresponding Boogie representation.

Boogie language. A Boogie program consists of a set of declarations, which can introduce *types, constants, functions, axioms, variables, procedure declarations, and procedure implementations.* Boogie *types* include both primitive data types like `bool`, `int`, as well as the user-defined data types and `map` types. Boogie *functions* are typically pure, meaning they do not have side effects, and *functions* are commonly used to define background theories and language properties. Additionally, Boogie procedure

implementations imperatively describe the behavior of a declared procedure and consist of standard statements like `assignments`, `call`, `return`, `assume` and `assert`.

Functions are *uninterpreted* by default and their semantics are quantified through extra axioms. In addition, functions without arguments are treated as *constants*. Boogie *axioms* are used to restrict the interpretation of *functions* and *constants*. *Axioms* may include common operations in programming languages, such as arithmetic, boolean, function, map, and first-order quantifiers. Unlike *functions*, Boogie *procedures* can have side effects, and they are commonly used to encode the executable components of input programs such as functions and constructors. Procedure declarations can have input and output parameters, and they are used to specify pre-conditions (`requires` construct) and post-conditions (`ensures` construct) of the encoded methods.

Memory modeling. Programming languages used for implementing OIDC protocols are typically object-oriented and an important design choice for translating such languages to Boogie is the representation of the memory (i.e., heap). We model the program's classes, objects, and fields with the help of a heap, which maps an object reference and field to values. Since the values can be of different types, depending on their corresponding field name, we use a polymorphic map for the heap variable.

Example 4. *Let's consider the following example of the Boogie program illustrating a common pattern of object-oriented memory modeling.*

```
type Obj;
type Field t;
type Heap = <t>[Obj,Field t]t;
const data: Field t;
const next: Field Obj;
var h: Heap;
```

This Boogie program defines a nullary type constructor `Obj` *for object references and a unary type constructor* `Field` *for instance fields. It then declares a constant* `data`*, a field of type* `t`*, and* `next`*, a field of type* `Obj`*. Variable* `h` *of type* `Heap` *maps from object-field pairs to value. Thus, given the field* `data` *in an object reference* `o`*, a field access expression*

`o.data` *in the source code can be translated into expression* `h[o, data]` *in Boogie.*

Encoding WalaIR to Boogie. With the help of the memory modeling described above, most of the WalaIR expressions and statements can be translated into Boogie in a syntax-directed way:

1. As WalaIR expressions are already simplified with SSA form, they can be directly translated to Boogie. We use Boogie's `int` and `bool` primitive types to represent the integer and boolean-typed constants in the source program. We model the arithmetic operations with Boogie functions along with assertion for operations like integer division.
2. WalaIR's assign statements can be directly translated to Boogie assignments. For field assignment, we use Boogie's map update operation, as explained above.
3. Method invocations are translated as Boogie procedure calls. We use the symbol table structure and pointer analysis methods to precisely infer the values of arguments. However, Boogie defines specific return variables in procedure declarations. Therefore, we translate the WalaIR return statements to assignments to the defined return variable.
4. Conditional branches in WalaIR take the form `if (expr) goto L`, which is translated by adding an assume statement (i.e., `assume(expr)` or `assume(¬expr)`) on the control-flow path of the Boogie program.

5.3.3 AuthLTL to Regular LTL

Our specification language AuthLTL is designed to achieve high expressiveness for complex OIDC properties, which involve domain-specific predicates such as verifyToken and authUser (Section 5.3.1). This leads us to a new challenge as these predicates are not recognizable by an off-the-shelf LTL model checker. To mitigate this challenge, *AuthSaber* automatically transforms the given AuthLTL formula φ into its *equisatisfiable* LTL formula φ' through instrumenting the original program $\mathcal{P}$ to $\mathcal{P}'$. Precisely, φ and φ' are equisatisfiable in that $\mathcal{P}'$ satisfies φ' if and only if $\mathcal{P}$ satisfies φ. Intuitively, this transformation

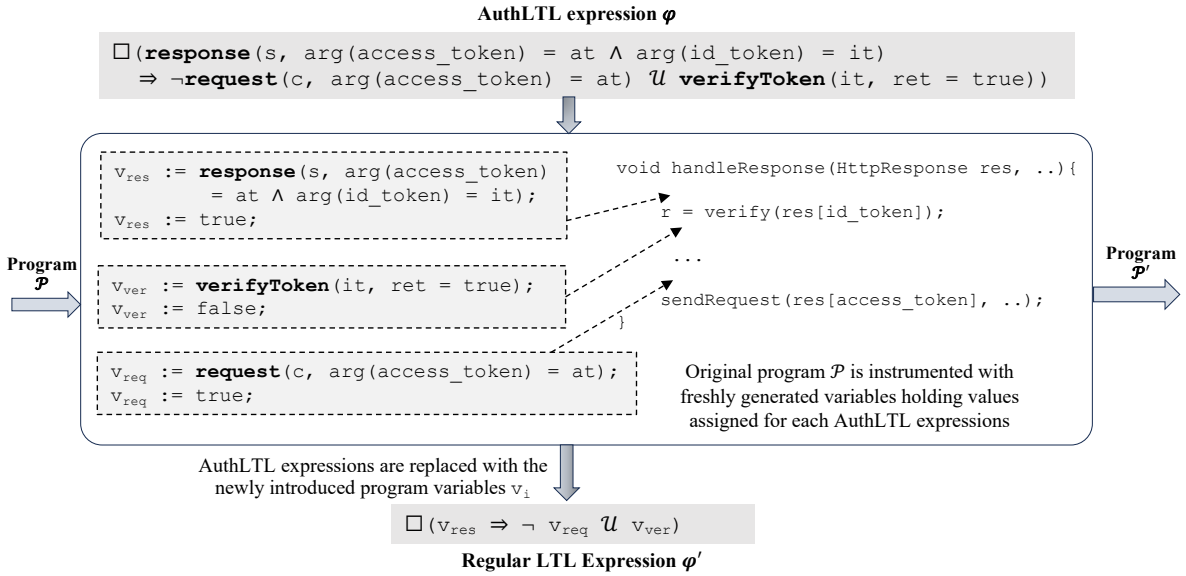


Figure 5.4: Transformation of AuthLTL formula φ to its equisatisfiable regular LTL φ' while transforming the original program $\mathcal{P}$ to $\mathcal{P}'$.

allows us to bridge the semantic gap between the AuthLTL properties and the ones in canonical LTL that can be directly consumed by model checkers.

To transform a given AuthLTL formula φ into an equisatisfiable LTL formula φ', we implement the following steps. First, we introduce a global mapping that maps AuthLTL's expressions to their corresponding program variables. Second, for every domain-specific predicate ψ in a given AuthLTL formula, we introduce a fresh variable v_ψ and update the original AuthLTL formula by replacing every occurrence of ψ with the fresh variable v_ψ. Finally, we establish the relation between expressions and their corresponding fresh variables through instrumentation. Specifically, we instrument the program by assigning the correct value of v_ψ, which is determined by evaluating the predicate ψ with respect to the program's execution environment.

Example 5. Figure 5.4 illustrates the above-mentioned transformation using an AuthLTL formula discussed in Section 5.2. The transformation takes an AuthLTL formula φ and a program $\mathcal{P}$ as input, and generates the transformed LTL formula φ' along with the instrumented program $\mathcal{P}'$. As the original AuthLTL formula consists of three atomic predicates response, request and verifyToken, we introduce three freshly generated boolean

variables v_{res}, v_{req} and v_{ver}, respectively, which are replaced by each predicate in the original AuthLTL formula. Finally, we instrument $\mathcal{P}$ by assigning the correct value to these variables. Since the final step is not trivial, we explain it in more detail below:

1. For $\mathsf{request}(_, \phi)$ predicate, we assign true the corresponding v_{req} variable if the condition ϕ holds right *before* the request is sent. Similarly, since $\mathsf{response}(_, \phi)$ predicate should be evaluated after a response is received, its variable v_{res} is set to true if ϕ holds *after* the response handler is called.
2. For the $\mathsf{verifyToken}(t, \phi)$ predicate, the boolean variable v_{ver} is set based on the result returned from external function verify. Hence, v_{ver} is assigned after verify function is called.

5.3.4 Safety Verification

In this section, we elaborate on the details of our safety verifier that is customized for the OIDC domain. Specifically, we propose a staged approach that first generates an effective harness to constrain the space of *valid communication* among multiple parties through *authentication flow validation* enforced by the standard specification, and then reduces our verification problem to an instance of LTL model checking.

5.3.4.1 Authentication Flow Validation

To ensure a sound verification, the OIDC implementation needs to be driven by a *harness* that simulates *all possible* interactions of different parties. However, unlike regular programs with single or a few entry points, OIDC implementations consist of dozens of entry points that can be invoked by the participating entities (e.g., relying party, OIDC providers, etc.) to finish the desired authorization and authentication services. In this case, a naive harness, which exhaustively enumerates all combinations of entry points to verify safety properties is technically sound but prohibitive in practice due to the unbounded search space. On the other hand, a harness that invokes entry points in an ad-hoc manner could result in false negatives.

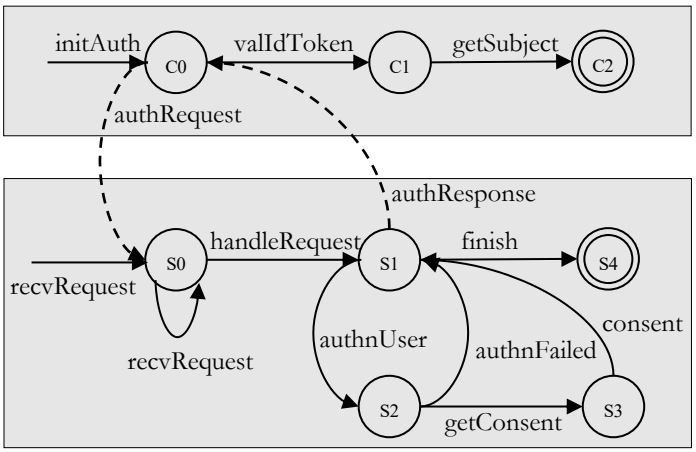


Figure 5.5: Automata representation of the authentication flow constraints for the implicit flow defined (in English) by the specification.

Key observation. In addition to safety properties that can be expressed in our language in Section 5.3.1, the OIDC specification (OpenID, 2021) also suggests *authentication flow constraints* that should be conformed by OIDC implementations. For instance, here is an example of the original *constraint* of the Implicit Flow for authentication from the specification (in English):

1. Client prepares an authentication request and sends the request to the authorization server.
2. Server authenticates the user and obtains user's authorization.
3. Server sends back the user to the client with an ID Token.
4. Client validates the ID token and retrieves the user's subject identifier.

Note that the above flow constraint not only mentions relevant parties (i.e., client and server.) of the protocol but also specifies the key actions and their *temporal sequence* that should be performed by each party. Since a harness that violates the authentication flow is not recommended, our key insight is to decompose the original unbounded verification problem to a *staged approach*: 1) we first construct a finite state machine that formalizes the semantics of flow constraint, 2) construct an augmented state machine representation of OIDC implementation that encodes all possible paths of communication, and finally, 3) generate an effective harness that only invokes valid authentication flows by computing

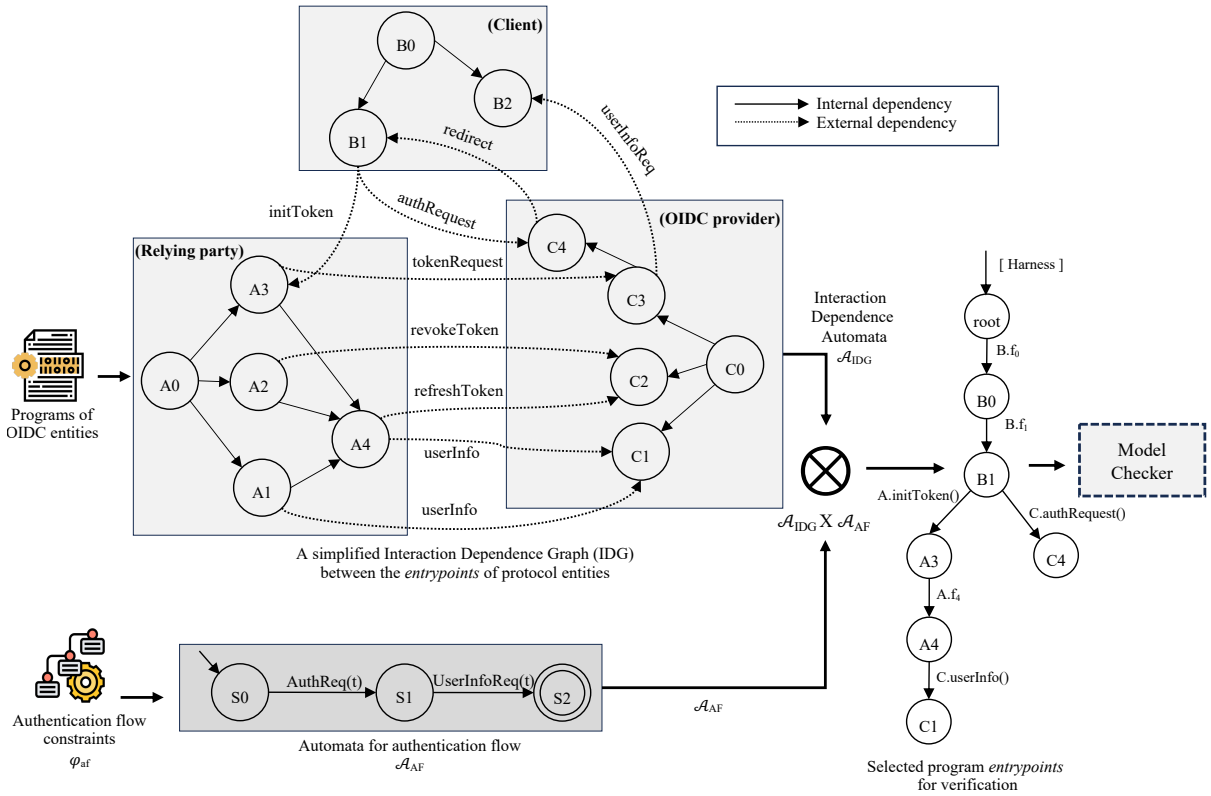


Figure 5.6: Authentication flow validation approach implemented by *AuthSaber* which models the interaction between the entrypoints of each participating OIDC entities as IDG.

the intersection between the flow constraints and the augmented program.

Step 1: Authentication flow constraints. Our first step is to convert the authentication flow constraint in English into its formal specification using *finite state machine* ϕ_{af}. Figure 5.5 shows a finite state machine that corresponds to the flow constraint at the beginning of Section 5.3.4.1. In particular, each node (s_i) denotes a representative state of a different party (i.e., client vs. server.). Each transition represents an authentication action that leads a party to a new state specified by the flow constraint. We use solid and dashed arrows to represent transitions within a single entity and multiple entities, respectively. Once the client receives an authentication response from the server at state c_0, according to the last step of the flow constraint from OpenID specification (i.e., "Client validates the ID token and retrieves the user's subject identifier"), the client should first check the validity of the token (state c_1) and then retrieves the user's subject identifier at

state c_2. We omit the formal definition of flow constraint since it can be easily translated from the original English description.

Step 2: Interaction Dependence Graph (IDG). Given the entities of the OIDC implementation, *AuthSaber* constructs its *Interaction Dependence Graph* (IDG) that *over-approximates* the interactions (i.e., communication via different entry points.) among the entities. Specifically, IDG is a graph $\mathcal{G}(V, E_{in}, E_{ex})$ where:

1. V is a set of vertices, where each $v \in V$ represents an entry function that communicates (i.e., via network APIs) with other entities of the protocol.
2. E_{in} are solid arrows that encode *internal* data dependencies among entry functions within the same entity. Precisely, $(v, v') \in E_{in}$ indicates that v' depends on the data from v during the protocol interaction.
3. E_{ex} are dashed arrows that encode the *external* dependencies among entry points across all participating entities. Specifically, $(v, v') \in E_{ex}$ indicates v' depends on an entry point v provided by an external entity.

To construct the IDG, we leverage the program dependency graph generated by the WALA framework to identify all entry functions in V as well as their internal data dependency in E_{in}. However, precisely constructing external dependency edges for E_{ex} is prohibitive since it requires whole-program analysis and global invariants. To mitigate this challenge, we over-approximate external data dependencies in E_{ex} by adding (dashed edges) between the entry points of all entities. Note that the current IDG is sound but encodes a lot of *spurious external dependencies*. Consequently, we leverage the flow constraint to further refine the external dependencies in step 3. Figure 5.6 illustrates the refinement technique of our interaction modeling.

Step 3: Refining IDG usng flow constraint ϕ_{af}. Since using the original IDG to explore all the possible interactions does not scale, *AuthSaber* automatically derives a subset of potentially valid interactions using flow constraints from step 1. Toward this goal, *AuthSaber* employs an automaton-based solution (Aho et al., 1986) to preserve a subset of external dependencies that are consistent with the flow constraints ϕ_{af}.

As shown in Figure 5.6, *AuthSaber* first transforms the IDG to a finite automaton $\mathcal{A}_{IDG}$ $(\mathcal{Q}_i, \Sigma_i, q_{0i}, F_i, \delta_i)$, where states $\mathcal{Q}_g$ corresponds to the nodes of IDG and Σ_i includes the entry functions associated with the graph nodes. Transition function $\delta_i : \mathcal{Q}_i \times \Sigma_i \rightarrow \mathcal{Q}_i$ corresponds to the external and internal edges (E_{in} and E_{ex}) of IDG. Finally, $q_{0i} \in \mathcal{Q}_i$ is a super root node that connects the main function of all entities and accepting states F_i are the nodes with no dependants (i.e., do not make any external request).

Meanwhile, *AuthSaber* already constructed the automaton $\mathcal{A}_{AF}$ from the flow constraint ϕ_{af} in step 1. In this case, *AuthSaber* computes the product automaton $\mathcal{A}_{(AF \times IDG)}$ of the two automatons (using the standard algorithm (Aho et al., 1986)) and the language recognized by the product automaton represents the intersection of the languages recognized by $\mathcal{A}_{AF}$ and $\mathcal{A}_{IDG}$. Therefore, the resulting product automaton $\mathcal{A}_{(AF \times IDG)}$ only preserves the external dependencies that are consistent with the flow constraint ϕ_{af}.

5.3.4.2 Model Checking

After *AuthSaber* derives the endpoints of valid interactions among OIDC entities, it proceeds to verify the safety properties within these endpoints. Our safety property verification technique for OIDC programs is an instance of the counter example guided abstraction refinement (CEGAR) framework which we adapt from (Dietsch et al., 2015). In the following, we begin with the necessary preliminaries followed by a brief overview of the verification using model checking.

Büchi automaton. A Büchi automaton $\mathcal{A} = (\Sigma, \mathcal{Q}, q_0, \rightarrow, \mathcal{F})$ is a finite-state automaton which consists a finite alphabet Σ, a finite set of states $\mathcal{Q}$, an initial state $q_0 \in \mathcal{Q}$, a transition relation $\rightarrow$ which is a function $\mathcal{Q} \times \Sigma \rightarrow \mathcal{Q}$, and a set of accepting states $\mathcal{F} \subseteq \mathcal{Q}$. A *word* is an infinite sequence $w = e_0 e_1 \ldots$ such that $e_i \in \Sigma$ for all $i \geq 0$. A *run* r is an infinite sequence of states $q_0 q_1 \ldots$ such that for all $e_i \in w$, there is a transition $q_i \times e_i \rightarrow q_{i+1}$ in the Büchi automaton $\mathcal{A}$. A word w is accepted by $\mathcal{A}$ if a run of w on $\mathcal{A}$ visits a set of final states infinitely many times. Finally, set of all words that are accepted by $\mathcal{A}$ are denoted by the *language* $\mathcal{L}(\mathcal{A})$.

LTL to Büchi. Any standard LTL formula can be expressed as a Büchi automaton (Baier and Katoen, 2008). Since *AuthSaber* has already transformed the AuthLTL formulas to standard LTL, our safety properties can also be converted into equivalent Büchi automaton, which is a more tractable form for automated analysis.

Büchi program product. To systematically explore the state space of the program for checking safety properties, a Büchi product representing the intersection of the Boogie program and the LTL automata is constructed. Let $\mathcal{P} = (\mathcal{S}, \mathcal{L}, \delta_{\mathcal{P}})$ be an OIDC program with a set of statements $\mathcal{S}$, program locations $\mathcal{L}$ and a transition function $\delta_{\mathcal{P}} : \mathcal{L} \times \mathcal{S} \to \mathcal{L}$. On the other hand, $\mathcal{A}_{\varphi} = (\Sigma, \mathcal{Q}, q_0, \to, \mathcal{F})$ be the Büchi automaton representation of a given AuthLTL safety property φ. Now, Büchi program product $\mathcal{P} \otimes \mathcal{A}_{\varphi}$ is in fact another Büchi automaton $\mathcal{B} = (\Sigma_{\mathcal{B}}, \mathcal{Q}_{\mathcal{B}}, q_{0\mathcal{B}}, \delta_{\mathcal{B}}, \mathcal{F}_{\mathcal{B}})$ such that $\Sigma_{\mathcal{B}}$ consists of all sequential compositions of two statements s_1 and s_2, where s_1 is a statement of program $\mathcal{P}$ and s_2 is a statement that assumes that a subset of atomic proposition ϕ in property φ is satisfied, i.e.,

$$\Sigma_{\mathcal{B}} = \{s_1; \texttt{assume}\ \phi \mid s_1 \in \mathcal{S}, \phi \in \Sigma\}$$

$\mathcal{Q}_{\mathcal{B}}$ is the cartesian product of program locations and Büchi automaton states (i.e., $\mathcal{L} \times \mathcal{Q}$), and $q_{0\mathcal{B}}$ is the pair of initial program location l_0 and initial state q_0 of $\mathcal{A}_{\varphi}$. Transition function $\delta_{\mathcal{B}}$ is a product of the program's transition and Büchi automaton's transition such that an edge in $\mathcal{B}$ is labeled by two sequential statements; one is from the program's edge label and another is an assume statement obtained from the transition of $\mathcal{A}_{\varphi}$. Formally,

$$\begin{aligned} \delta_{\mathcal{B}} \quad = \quad & \{(l_1, q_1),\ s; \texttt{assume}\ \phi,\ (l_2, q_2) \\ & \mid (l_1, s, l_2) \in \delta_{\mathcal{P}}, (q_1, \phi, q_2) \in \to\} \end{aligned}$$

Finally, $\mathcal{F}_{\mathcal{B}}$ is a pair (l, q) such as $q \in \mathcal{F}$ is an accepting state of $\mathcal{A}_{\varphi}$ and l is a location of accepting state of program $\mathcal{P}$ at which the protocol flow terminates.

Verification. The model checking technique first constructs a Büchi automaton $\mathcal{A}_{\neg\varphi}$ for a given OIDC safety property φ such that the language accepted by $\mathcal{A}_{\neg\varphi}$ violates the desired security behavior. Next, it constructs a Büchi product automaton $\mathcal{B} = \mathcal{P} \otimes \mathcal{A}_{\neg\varphi}$

which represents the input program together with the specification. Hence, $\mathcal{P}$ satisfies the safety property φ if and only if the language of $\mathcal{B}$ is empty or $\mathcal{B}$ does not have a trace that is feasible during the execution. To check this, the verification method iteratively finds a trace τ that is accepted by $\mathcal{B}$ and checks the feasibility of τ under the program's execution environment. Here, any trace τ accepted by $\mathcal{L}(\mathcal{B})$ always takes the lasso-shaped form $\tau_1\tau_2^\omega$. Hence, τ is feasible if and only if τ_2 can be executed infinitely many times after executing τ_1. This is done by first checking the feasibility of τ_1, the loop $\tau 2$, and then $\tau_1\tau_2$. If none of those are infeasible, the feasibility checking tries to find a ranking function to prove that the loop will eventually terminate. Finally, if τ is inferred as feasible, it represents a counterexample (i.e., a path that violates the safety property). Otherwise, τ is a spurious path and removed from the language of $\mathcal{B}$, and the verification method moves to the next trace accepted by $\mathcal{B}$.

5.3.5 Implementation

We implemented the technical concepts discussed above in *AuthSaber* tool, which takes OIDC programs along with the safety specification as input and verifies the programs against the specification. *AuthSaber* consists of approximately 9,500 lines of code in Java. For OIDC implementation in Java, *AuthSaber* takes *bytecode* as input, and for Javascript (JS), it accepts the source codes (i.e., scripts) as input. We use IBM T.J. Watson Libraries for Analysis (WALA) (WALA, 2023) to represent the input program as WalaIR. WALA supports both Java and JS as the front end and represents the programs in the unified structure of WalaIR. We further utilize WALA's pointer-analysis and callgraph APIs which allow us to reason about the semantics of the program instructions when translating into equivalent Boogie (Leino, 2008) programs. Finally, the CEGAR framework of our model checking is built upon the UltimateAutomizer model checker tool (Dietsch et al., 2015).

Manual effort for initial setup. To facilitate the automated verification of a diverse range of real-world OIDC programs, users may need to invest additional manual effort in

the initial setup of *AuthSaber*. For instance, *AuthSaber* utilizes a comprehensive ontological mapping of OIDC keywords to associated terms (such as function names and variables) used within the OIDC implementation. While the majority of OIDC-specific keywords (e.g., `id token`) in our AuthLTL can be seamlessly mapped using pattern matching to terms defined in the programs in accordance with the standard specification (OpenID, 2021), certain OIDC implementations feature custom authentication flows, necessitating manual effort to align predicates and variables in AuthLTL with their counterparts in the implementation. Furthermore, OIDC programs commonly use external JWT libraries to support their cryptographic functions, particularly for signature verification. Although *AuthSaber* offers abstract modeling for commonly used JWT APIs, users may be required to contribute additional modeling if OIDC implementation employs new JWT libraries or APIs. However, these manual tasks are a one-time requirement and can be universally applied to any OIDC implementations adhering to the standard specification.

5.4 Evaluation

This section begins by presenting the research questions, followed by an evaluation designed to address these questions.

Research questions. We design our evaluation scheme primarily to answer the following research questions:

1. **RQ1.** Can *AuthSaber* verify safety properties in real-world OIDC implementations?
2. **RQ2.** Can *AuthSaber* verify previously reported violations in OIDC implementations that were patched by the developers?
3. **RQ3.** How does *AuthSaber* perform when compared with other security analysis tools in the SSO domain?

5.4.1 Experimental Setup

Below, we first define the scope of OIDC properties considered in our evaluation. We then elaborate on the OIDC libraries and benchmarks selected for verification against these specified properties.

OIDC Components	Auth. Flows	RP	OP	OIDC References	#Properties
Authentication request validation	AC, I, H		✓	§3.1.2.2, §3.2.2.2, §3.3.2.2	1
Authentication response validation	AC, I, H	✓		§3.1.2.7, §3.2.2.8, §3.3.2.8	2
Token request validation	AC, H		✓	§3.1.3.2, §3.3.3.2	1
Token response validation	AC, H		✓	§3.1.3.5, §3.3.3.5	2
ID token validation	AC, I, H	✓	✓	§2, §3.1.3.7, §3.2.2.11	8
Access token validation	AC, I, H	✓	✓	§3.1.3.8, §3.2.2.9	1
Authorization code validation	AC, H	✓		§3.3.2.10	1
User info request and response	AC, I, H	✓	✓	§5.3.1, §5.3.2, §5.3.4	2
Client authentication	AC, I, H		✓	§9	6

Table 5.1: OIDC components and relevant details that we consider to gather the safety properties for our evaluation.

Selection of properties. We define the safety properties based on the standard specifications outlined in OpenID Connect Core 1.0 (OpenID, 2021) and established security best practices (IETF, 2021). The scope of these properties is illustrated in Table 5.1. Additionally, Table 5.2 provides details on the specific properties selected within the scope of Id Token Validation, a crucial component in OIDC's authentication flows.

In total, we have identified 24 safety properties, encompassing all three authentication flows supported in OIDC: (1) authorization code flow (AC), (2) implicit flow (I), and (3) hybrid flow (H). It is noteworthy that, for the current analysis, we exclude the *optional to implement* OIDC components, such as self-issued OIDC providers and passing requests as JWT objects. Our empirical study indicates that these components are not widely supported by most OIDC implementations.

Collection of OIDC implementations. To evaluate *AuthSaber*, we collect widely adopted open-source libraries of OIDC protocol. Statistical information for the selected

libraries is presented in Table 5.3, indicating their popularity based on metrics like GitHub stars. Out of these libraries, 11 are implemented in Java, and 4 in Javascript. Additionally, six of these libraries are certified (OpenID, 2023) by the official OpenID Foundation. It's worth noting that not all OpenID-certified libraries are open source or publicly accessible, which limits their inclusion in our evaluation.

Collection of labeled benchmarks. In order to evaluate *AuthSaber*'s performance on verifying *known* security issues in OIDC implementation, we further collect a dataset of OIDC programs and annotate them with associated safety properties that are violated or satisfied in the programs. To compile the dataset, we conduct a manual examination of publicly available security reports from the OpenID libraries and their online communities. We ensure that these reports have been verified and addressed by the developers of the respective libraries. Next, we annotate the incorrect program samples as *buggy* benchmarks and their corrected (i.e., patched) version as *safe* benchmarks. Libraries are excluded in cases where security reports are either not publicly available or not confirmed by the developers. Overall, our dataset comprises 28 buggy benchmarks and 25 safe benchmarks. For each benchmark, we further translate the corresponding security issues in AuthLTL expressions.

#No.	Description of safety properties for ID Token validation
1	Once a token request is received, a successful response should not be sent without obtaining end-user authentication.
2	If ID token with signature algorithm `none` is received as a response from the server, the token should be rejected.
3	If ID token is received with a `code` from the server, the `code` should not be used for access token unless the ID token is verified.
4	An ID token should not be verified without first checking if the `alg` header value matches the *expected* algorithm value.
5	Issuer claim of an ID token should not be empty and it should be checked against the configured issuer before making a token request.
6	Audience claim of an ID token should not be empty and it should be checked before making a token request.
7	If the authentication response type is '`id_token token`', the `at_hash` claim must be validated.
8	If the authentication response type is '`id_token token`', the `nonce` claim must be present and should match the nonce value sent with the authentication request.

Table 5.2: Properties within the scope of ID Token validation of OpenID Connect.

OpenID libraries	Language	#Github stars	#LOC
1) Keycloak	Java	15.2k	251k
2) Spring Security	Java	7.5k	61.6k
3) Connect2id	Java	N/A	97.4k
4) Mitre OpenID	Java	1.4k	53.2k
5) Google Client	Java	600	12.7k
6) Pac4j	Java	2.2k	76.1k
7) Oracle Cordova	Java	10	34.5k
8) AppAuth	Java	2.4k	22.9k
9) OxAuth	Java	400	179.6k
10) Quarkus	Java	11.4k	18.8k
11) Auth0	Java	257	41.7k
12) Node OIDC	JS	2.6k	50.4k
13) AppAuth-JS	JS	912	5.2k
14) Auth0 Express	JS	360	2.7k
15) Asgardeo	JS	25	3.6k

Table 5.3: Popular OIDC libraries selected for evaluating *AuthSaber*. Among these, eleven libraries are implemented in Java and four are implemented in JS.

5.4.2 Experimental Results

We carried out all experiments on a machine equipped with a Quad-Core Intel Core i5 processor and 32GB of memory, operating on macOS 14.0. We delve into the detailed insights regarding the results of our experiments as follows.

OIDC libraries	#Properties	#Verified	#Falsified	Avg. time(s)
1. Keycloak	15	12	1	3118
2) Spring Security	15	14	0	1835
3) Connect2id	11	11	0	2177
4) Mitre OpenID	8	6	1	1782
5) Google Client	9	7	2	638
6) Pac4j	13	11	1	940
7) Oracle Cordova	8	7	1	460
8) AppAuth	10	6	3	336
9) OxAuth	16	15	0	2348
10) Quarkus	10	7	3	430
11) Auth0	8	7	0	708
12) Node OIDC	15	13	0	1670
13) AppAuth-JS	7	6	1	218
14) Auth0 Express	8	6	1	510
15) Asgardeo	8	6	2	788
Overall	161	134	**16**	1316

Table 5.4: Verification results for the popular OpenID Connect implementations. In all 15 OIDC libraries, we successfully verified 134 properties and found 16 confirmed security violations that were previously unknown. We validated and responsively disclosed all violations to the respective developers.

OIDC libraries	#Collected benchmarks	#Verified by *AuthSaber*	#FP
1) Keycloak	6	5	1
2) Mitre OpenID	4	4	0
3) Google Client	3	3	0
4) Oracle Cordova	1	1	0
5) OxAuth	2	2	0
6) Quarkus	2	2	0
7) Auth0	3	2	1
8) AppAuth-JS	2	1	1
9) Asgardeo	2	2	0
Total:	25	22	3

Table 5.5: Verification results for the collected safe benchmarks that were patched by the developers based on previously reported violations. For this evaluation, we exclude the libraries for which no previous violations are found, or the reports are no longer publicly accessible.

Evaluation of OIDC implementations. We verify the popular OIDC libraries we collected against the 24 safety properties expressed in AuthLTL. However, given that libraries are often designed to support distinct authentication platforms, not all of them implement every authentication flow and feature supported in the OIDC protocol. Consequently, certain properties may be irrelevant for a particular library if it does not support the associated authentication feature or flows. In such cases, we exclude these properties from the library's evaluation. The verification results are presented in Table 5.4, showing the number of properties satisfied (i.e., verified) and did not satisfy (i.e., falsified) for each library. Here, falsified properties indicate a violation of properties, implying the existence of vulnerabilities in the input program. In the 15 popular OIDC libraries, *AuthSaber* successfully verified a total of 134 properties and reported 16 security violations that were previously unknown. *AuthSaber* also reported 11 false positives, as found during the manual validation of the generated counter-examples. We meticulously validate each violation before responsively disclosing them to the developers of the libraries.

RQ1: *AuthSaber* uncovered 16 previously unknown security violations in the most popular OIDC libraries, resulting in five new CVEs.

Evaluation of labeled benchmarks. We further evaluate *AuthSaber* using the bench-

marks we collect from publicly available security reports and relevant patches for the respective OIDC libraries. As outlined in the experimental setup earlier, we collect a total of 28 buggy OIDC programs (i.e., containing at least one violated property) and 25 safe programs (i.e., the patched version of the buggy programs). Since these programs are manually collected based on confirmed security issues from developers, they provide the known ground truths for all benchmarks, indicating whether a benchmark satisfies a given property. We evaluate each benchmark using *AuthSaber*, which successfully falsifies all 28 buggy benchmarks. We manually investigate the counter-examples generated by the tool for each benchmark and validate them against the labeled safety properties. On the other hand, as illustrated in Table 5.5, *AuthSaber* successfully verifies 22 out of 25 safe benchmarks and incorrectly flags (i.e., false positive) three benchmarks.

RQ2: *AuthSaber* successfully verified all 28 buggy benchmarks and 22 out of 25 safe benchmarks among the labeled benchmarks with known security issues.

OIDC Flow	**#OIDC Properties**	**#Checked Properties**		
		AuthSaber	**Cerberus**	**S3KVetter**
Auth code	12	11	4	3
Implicit	6	6	2	2
Hybrid	10	7	1	0
Total:	28	**24**	7	5

Table 5.6: Comparison of *AuthSaber* with existing tools in SSO domain in terms of the number of OIDC safety properties that can be expressed and checked.

Comparison with existing SSO analysis tools. Since there are no other formal verification tools available for OIDC programs, we compare *AuthSaber* with other available program analysis tools in this domain. Specifically, we compare *AuthSaber* against Cerberus (Rahat et al., 2022) and S3KVetter (He et al., 2015a), which use static analysis and symbolic execution, respectively, to analyze the security of OAuth protocol implementation. Although OIDC protocol is built on top of OAuth protocol, these tools are not designed to reason about the semantics of OIDC and therefore, this is not an apples-to-apples comparison. Hence, we first do our best to express 24 OIDC safety properties for three authentication flows: 1) authorization code flow, 2) implicit flow, and

3) hybrid flow. Among them, authorization code flow and implicit flows are inherited from the OAuth (RFC6749, 2021) protocol. Further, we evaluate the buggy benchmarks using Cerberus and S3kVetter tools against the safety properties. Since the hybrid flow and use of id tokens are not supported in OAuth, we add additional tags to specify the properties in Cerberus's query language. Additionally, as S3KVetter tool only supports Python programs, we manually translate our Java/JS benchmarks into Python language. Table 5.6 shows the detailed results of our evaluation. Out of 28 OIDC properties, *AuthSaber* was able to check 24 properties. Using Cerberus, we were able to check 7 out of 28 properties, and with S3KVetter tool, we were able to check 5 properties. Moreover, S3KVetter does not provide any specification language and expresses the properties over only request and response arguments, which restricts us from checking the properties of hybrid flow using their tool.

RQ3: *AuthSaber* is 3× more expressive in specifying and checking safety properties than other existing program analysis tools in SSO domain.

Limitations. When evaluating real-world OIDC programs, *AuthSaber* incorporates several conservative assumptions that could lead to false positive (FP) cases, as shown in Table 5.5. For instance, when dealing with dynamic programming features like dynamic class loading, reflected calls or signature-based function calls in Java or JS, *AuthSaber* currently relies on over-approximation. This reliance can introduce spurious paths in the downstream verification task, consequently leading to false positive cases. In addition, non-linear arithmetic and bitwise operations are currently modeled using uninterpreted functions which can also trigger false alarms during the evaluation. However, based on our evaluation, *AuthSaber* doesn't have many false alarms.

Moreover, although liveness properties are not common for expressing the unexpected security behavior of OIDC protocol and are not the focus of this work, our specification language AuthLTL can also be used to express the liveness properties. However, since the model checking technique of *AuthSaber* is based on the CEGAR framework provided by the UltimateAutomizer (Dietsch et al., 2015), which in general, does not give a termination

guarantee, our tool also inherits this limitation. However, a key aspect of our approach, as outlined in Section 5.3, uses the flow constraints to deduce a finite set of interactions. This allows *AuthSaber* to restrict the state space within a valid set of OIDC endpoints, ensuring a fair termination during the verification.

5.4.3 Developer Acknowledgment

After scrutinizing and validating the results obtained from *AuthSaber*, we responsively disclosed all 16 previously unknown violations to the developers and vendors of the respective OIDC libraries. As of the writing of this dissertation, we have received acknowledgments from the developers for 10 reported vulnerabilities, while the remaining reports are currently under review process. Six vulnerabilities were immediately fixed following our reports. Additionally, the confirmed vulnerabilities resulted in the assignment of five new CVEs (CVE-2023-33292, CVE-2023-35819, CVE-2023-35820, CVE-2021-44878, CVE-2021-22573). We intend to maintain ongoing communication with the developers, assisting them in promptly resolving the remaining vulnerabilities.

5.4.3.1 Case Studies

Here, we delve into a few case studies based on the evaluation results obtained from *AuthSaber* for popular OIDC libraries.

1) Google Client Authentication. We applied *AuthSaber* to verify the implementation of the Google Client Authentication library for Java (Google, 2023), a widely used library for facilitating authorized access to Google's APIs. In our investigation of the violated traces generated by *AuthSaber*, we identified that the library validates issuer (`iss`), audience (`aud`), and expiration (`exp`) claims upon receiving an `id token` from authorization service providers. Unfortunately, it failed to check the signature of the token and accepted it solely based on the validity of these claims received as the payload of the token. Since the payload of an `id token` is *unprotected* and cannot be trusted without proper verification of the signature component, attackers could manipulate the claims of a valid token or

forge a token received from another application, potentially gaining unauthorized access to clients.

Upon submission of our report, Google promptly acknowledged the vulnerability, with a statement from one of Google's Security Engineers stating, "*Nice catch! I've filed a bug with the responsible product team based on your report.*" The bug was labeled as "*priority-1*" and fixed within two weeks of our report. We received further acknowledgment from Google through their bug bounty program for our contribution to discovering and reporting this vulnerability.

2) AppAuth. AppAuth has gained popularity among developers utilizing the OIDC protocol due to its support for a wide range of mobile and web platforms. Using *AuthSaber*, we verified its OIDC implementation in Java (AppAuth, 2023) and uncovered two critical vulnerabilities. Firstly, while the library correctly validates the issuer claim of `id token`, it does so only if a 'discovery' document is present. However, utilizing a discovery document is optional in OIDC, and the issuer claim should be validated irrespective of the discovery method employed by the providers (OpenID, 2021). This vulnerability could allow attackers to inject an `id token` obtained from a malicious issuer. Secondly, the library neglects the signature verification of the token and incorrectly assumes that the tokens are exchanged using TLS communication – a practice also not enforced or validated by the library. We promptly reported all identified issues to the developers, and one of them acknowledged that "*these issues are not best security practice and should be fixed.*"

5.5 Summary

In this chapter, we propose *AuthSaber*, an efficient automated verifier for checking safety properties in the implementation of OIDC protocols. *AuthSaber* provides a highly expressive specification language to formally express the safety properties and incorporates an automated verification approach that is tailored to several scalability and automation challenges in the OIDC domain. We evaluate *AuthSaber* on 15 popular OIDC libraries

and discover 16 previously unknown security violations, leading to five new CVEs and further improvement in the security of popular OIDC implementations.

CHAPTER 6

Automated Repair of OpenID Connect Bugs

6.1 Introduction

In this chapter, we address the mitigation strategies for SSO bugs identified by our methods presented in Chapters 3–5. SSO protocols like OpenID Connect have made a significant impact on online authentication, offering a convenient and secure way to access multiple services. Unfortunately, as we demonstrate in the previous chapters, OpenID Connect has been marred by critical security bugs that have had far-reaching consequences. These vulnerabilities have resulted in significant financial losses for the industry, with estimates suggesting millions of dollars in damages. Moreover, these bugs have also led to numerous security breaches (Microsoft, 2023a; Portswigger, 2023; Facebook, 2018b) worldwide, compromising the sensitive information of countless individuals and organizations. For instance, in 2022, a high-severity flaw (severity rating 8.7 out of 10) was reported in Google's authentication flow implementation (Google, 2022). In the following year 2023, an OpenID-specific bug (Microsoft, 2023b) in Microsoft's Azure authentication allowed hackers to forge tokens to breach over two dozen organizations that utilize the Azure service. The severity of these breaches has not only eroded trust in the protocol but also highlighted the need for more robust mitigation to prevent such incidents in the future. Therefore, as the digital landscape continues to evolve, it is crucial to fix vulnerabilities in a timely manner and make the authentication process more secure and reliable.

However, manually fixing OpenID bugs can be a tedious and labor-intensive process for the developers, leading to prolonged exposure to security vulnerabilities and increased risk of breaches. For instance, it took on average 47 days for developers to fix the bugs

we reported for popular OpenID Connect libraries (Chapter 5). It is quite challenging for developers to fix these bugs mostly for three reasons: 1) **Complex specification:** OpenID specification is complex and involves hundreds of pages of documents, making it difficult for developers to conform to all the requirements from the specification when fixing OpenID bugs. 2) **Error-prone repairs:** Even when developers create patches to fix a bug, a naive fix can still compromise the overall system security. For instance, unsigned ID tokens are only supported in OpenID for low-computation-power devices (e.g., IoT devices). However, accepting and processing unsigned tokens for regular clients like web applications may seem like a convenient shortcut for developers, but it can actually leave the system exposed to severe attacks like authentication bypass. 3) **Lack of validation schemes:** Developers currently lack access to comprehensive Oracle or universal test cases to verify the functional correctness of protocol implementations. While the OpenID Foundation (OpenID, 2023) offers a certification process, it only evaluates the correctness of protocol endpoints, neglecting the implementation logic within the source code. This limitation leaves a critical gap in ensuring the overall functionality and reliability of the implementation. Therefore, it is of paramount importance to automate the repair of OpenID bugs while satisfying the requirements of the specification.

Existing works tackling the problem of automated bug repair such as GenProg (Le Goues et al., 2011), PAR (Kim et al., 2013), NOPOL (Xuan et al., 2016), and ACS (Xiong et al., 2017), require users to provide a set of test cases, with at least one failed case corresponding to the specific bug. Creating comprehensive test cases to cover all aspects of the OpenID specification is a daunting task, and unfortunately, there is no such datasets of test cases available currently, making it a significant challenge to adapt the existing solutions of automated program repair. Moreover, existing solutions are focused on very specific types of bugs and with tight restrictions, which are not well-fitted to handle OpenID-specific bugs. For instance, PAR (Kim et al., 2013) uses fix patterns learned from existing human-written patches and it replaces method call parameters with matching names or expressions from other calls within the same method, which is not applicable for fixing many OpenID bugs that require methods from external modules and

libraries. NOPOL and some other repair techniques (DeMarco et al., 2014; Mechtaev et al., 2015) extract constraints and synthesize repairs by translating the constraints to Satisfiability Modulo Theories (SMT) formulas. Such translation-based synthesis may involve incomplete translations and often require creating complex models for all involved libraries. Since OpenID Connect uses many complex APIs (e.g., cryptographic APIs) from external libraries, it is not practical to apply SMT-based translation for these programs.

In this dissertation, we propose an automated repair approach that takes a buggy OpenID program and its corresponding specification as input and generates a fixed program that satisfies the given specification. Being motivated by the recent success of utilizing LLM for fault localization (Wu et al., 2023b; Kang et al., 2023), we first invoke an LLM agent (White et al., 2023) to localize the suspicious statements in the buggy program. We then use a set of carefully designed schemas to generate program sketches that use partial expressions (e.g., *holes*) based on the suspicious statements. We then iteratively explore the search space of OpenID-specific repair expressions to fill the holes and generate concrete repair candidates. We utilize the LLM agent to reduce the search space and prioritize the candidates that are most likely to fix the underlying bugs. We validate the generated candidates using a novel Petri net-based model checker which checks the generated candidates against the OpenID specification. Our iterative search continues until a valid repair is generated or the search space is exhausted (i.e., no valid repair candidate is found).

We evaluate our proposed approach on a dataset of OpenID bugs that we compiled based on the confirmed bugs that we discovered in Chapter 5 and the bugs that have been reported in other OpenID libraries. We also collect the patches that are manually written by developers (if available) to fix the corresponding bugs in our dataset. Our evaluation shows that our proposed repair approach generates correct patches for 15 OpenID bugs out of 23 that were found in popular OpenID libraries. In addition, our manual assessment shows that 44% of the generated patches are semantically equivalent to the manual patches written by OpenID developers.

6.2 System Design

In this section, we first give an overview of our approach using a motivating code example, and then elaborate on the technical details of each component of our repair approach.

6.2.1 Overview

```
1  String jwksuri = config.getJWKSetURI();
2  IDTokenValidator idv = new IDTokenValidator(issuer, clientID);
3 -if ("none".equals(alg)) {
4 +if ("none".equals(alg) && config.unsignedTokenAllowed) {
5      claims = idv.validatePlain(idToken);
6  }
7  else if(JWSAlgorithm.HS512.equals(alg)) {
8      val secret = new Secret(config.getSecret());
9      claims = idv.validateSignedHS(idToken, secret);
10 }
11 else if(JWSAlgorithm.RS512.equals(alg)) {
12     claims = idv.validateSignedRS(idToken, jwksuri);
13 }
14 else {
15     -idv.validateSignedRS(idToken, jwksuri);
16     +rejectToken(idToken);
17 }
18 verifyClaims(claims);
19
```

Figure 6.1: Bugs in line 3 and line 15 in a popular OpenID Connect library Pac4j. Fixes are shown in line 4, which adds a new clause in the conditional expression, and in line 16 where the method invocation is replaced with a new one.

Figure 6.1 shows a buggy code that is inspired by a recently reported bug (CVE-2021-44878) in a popular OpenID Connect library called Pac4j. The example illustrates a partial code from the OpenID module of `id token` validation, which includes a set of validity checks, including the validation of the signature and certain claims. Specifically, it performs the signature validation for `id token` based on the value of *alg* parameter in the header component. In line 3, the code does not perform the necessary checks required by OpenID specification (OpenID, 2021) when the *alg* value is '`none`'. In addition, the code incorrectly calls the `validateSignedRS()` method when the *alg* value does not match any of the expected values specified by the protocol. `id token` validation is critical for OpenID

as it is the cornerstone of authentication benefits provided by the protocol. Specifically, `id token` issued by OP consists of a signed component that allows the RP to verify the authenticity of the information received from OP. Without proper validation of `id token`, the authentication process of OpenID can be vulnerable to access token injection which could eventually lead to unauthorized access or even user account takeover by attackers.

To mitigate these vulnerabilities that could have devastating consequences to the security of OpenID-based systems, we propose an automated repair technique, *AuthFix* that fixes the buggy statements to satisfy the standard OpenID specification. Figure 6.2 illustrates our repair approach which takes an example of the buggy code along with its OpenID specification and generates repaired code as output. Specifically, given a buggy OpenID program, we use an LLM agent to locate the suspicious statements that are responsible for the bugs. We then use a series of schemas to generate repair sketches that replace the buggy expressions with *holes*. For instance, we transform the conditional expression (“`none`”.`equals`(`alg`)) to (“`none`”.`equals`(`alg`) $\texttt{lop}_{??}$ $\texttt{e}_{??}$), where $lop_{??}$ and $e_{??}$ denote a hole for the logical operator and boolean type expression, respectively. We iteratively search over the space of OpenID expressions (e.g., `alg` == `null`, `config.unsignedAllowed` == `true`, etc.) to fill the holes and transform the sketches into concrete programs of potential repair candidates. We utilize LLM to reduce the search space of repair expressions and confine the generated candidates within the semantic space of the corresponding bugs. Finally, we apply a Petri net-based model checker to validate that the generated repair program satisfies the OpenID specification, which is also translated (manually) into a Petri net. In the following, we provide the detailed technical specifics of each component of our repair technique.

6.2.2 Bug Localization

This section delves into the details of our approach of utilizing LLMs to localize buggy statements in a given buggy program of OpenID implementation.

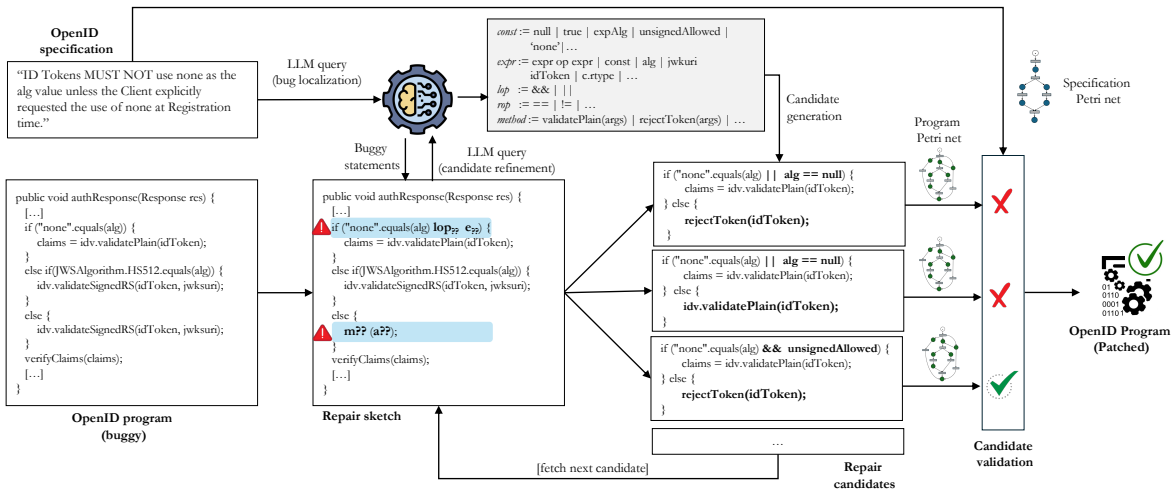


Figure 6.2: Overview of our repair approach, which takes a buggy OpenID code and specification (in English and Petri net format) as inputs, and generates a valid repaired program as output.

6.2.2.1 LLM Query

This section describes our approach to utilizing the LLM agent to locate the program's statements that might cause the bug. Our bug localization approach begins with submitting a text-formatted query Q to the LLM agent. An LLM query Q is a sequence of text components $Q = [q_0, q_1, \ldots, q_n]$, where each component q_i is one of the following:

a) Static query component: A static query component remains the same for all benchmarks, i.e., $q_i(p) = q_i(p')$ for all benchmark programs p and p'. We use static components to provide generic instructions (e.g., output format description) for the LLM agent.

b) Dynamic query component: Dynamic query components may differ across the benchmarks, i.e., there may exist benchmark program p and p' for which $q_i(p) \neq q_i(p')$. Dynamic components consist of the following parts:

1. *Buggy code:* This component includes the body of the function containing the OpenID bugs.

q_1	Analyze the following code from an OpenID Connect implementation against the specification that states the following:
q_2	"1. if alg value uses a MAC based algorithm such as HS256, client_secret should be used to validate the signature. 2. ID Tokens MUST NOT use none as the alg value unless the Client explicitly requested the use of none at Registration time."

q_3	1 public void validate() {
	2 if(idToken.getAlg() == "HS256") {
	3 idToken.verify(idToken.getAlg(),config.client_secret);
	4 } else if (idToken.getAlg() == "none") {
	5 idToken.verify(idToken.getAlg(), null);
	6 }
	7 if(idToken.getAlg() != config.expected_alg) {
	8 sendError();
	9 }
	10 if(idToken.getAud() != null && idToken.getAud() != config.client_id) {
	11 sendError();
	12 }
	13 idToken.validateClaims();
	14 }

q_4	Return the output in JSON format, consisting of a single JSON object with the "bug_location" array that contains JSON objects with three fields: "line_number" (indicating the buggy code) and "code_content" (showing the actual code statement), and "reason" (explaining why this location is identified as buggy).

Figure 6.3: An example of an LLM query for bug localization in a partial OpenID Connect program. This query has four text components Q = $[q_1, q_2, q_3, q_4]$, where $q1$ and q_4 are static components that provide instructions to the LLM agent and q_2 and q_3 are dynamic components that provide the specification and buggy program, respectively.


```
{
  "bug_location": [
    {
      "line_number": 5,
      "code_content": "idToken.verify(idToken.getAlg(), null);",
      "reason": "ID Tokens MUST NOT use 'none' as the alg value unless explicitly requested."
    },
    {
      "line_number": 7,
      "code_content": "if(idToken.getAlg() != config.expected_alg) {",
      "reason": "Algorithm verification does not handle 'none' algorithm correctly."
    }
  ]
}
```


Figure 6.4: Output from LLM agent in response to the query in Figure 6.3

2. *Specification (in English):* This component describes the expected behavior of the buggy code extracted from the standard OpenID Connect specification (OpenID, 2021), which is written in plain English.
3. *Log information:* The Dynamic part of the query may also include the log information extracted from the model checker that validates the potential repair candidates of the given buggy programs. This log information includes the program events and conditions that are not satisfied and allows the LLM agent to refine the bug localization outputs.

6.2.2.2 Processing query response

The LLM queries produce output in JSON format, as specified in the query. The generated output consists of a JSON object containing an array of bug localization results, where

each object provides detailed information about the bug, including its specific location (line number), the code content of the corresponding buggy statements, and the reason for the bug. After localizing the bug locations, we generate repair sketches that utilize a series of schemas to replace the buggy statements with partial expressions containing placeholders (represented as "??", or "holes") that need to be filled in with concrete repair expressions.

6.2.3 Candidate Generation

To generate potential repair candidates, we reduce the problem of program repair to program synthesis by transforming the buggy program into repair sketches. In the following, we describe the details of our schemas to generate the sketches based on the suspicious statements.

6.2.3.1 Schemas for Repair Sketch Generation

AuthFix applies the following set of schemas to generate program *sketches* for repairing based on the identified bug locations in the input program.

Conditional statement: Given a buggy conditional statement at location l, *AuthFix* applies the following schemas in the presented order to generate program sketches:

- **(S1)** transforms the arithmetic, relational, or logical operators of the conditional expression to an operator hole "$op_{??}$" which corresponds to any operators of the same operation type.
- **(S2)** transforms the variables, constants, or field dereferences with type τ of the expression with an expression hole "$e_{\tau??}$" which corresponds to any expression with type τ.
- **(S3)** introduces a new clause for the conditional expression e and transforms it into a new expression ($e\ lop_{??}\ e_{??}$), where $lop_{??}$ corresponds the logical operators and $e_{??}$ corresponds to any expression.

Method invocation: Given a buggy method invocation (e.g., $o.f(a, \dots)$) at location l

of the buggy program, *AuthFix* applies the following set of schemas for sketches:

- **(S4)** transforms the object reference o of type τ with an expression hole $e_{\tau??}$.
- **(S5)** replaces the argument expression of type τ with an expression hole "$e_{\tau??}$".
- **(S6)** replaces the method name with a method hole $m_{??}$ of the same signature.
- **(S7)** adds an *if guard* that transforms the statement with method invocation $f()$ to $\mathsf{if}(e_{??})\ \{f()\}$ to ensure the pre-condition for invocation of $f()$ that must be satisfied.
- **(S8)** transforms the complete method invocation by a new method hole $m_{??}(a_{??})$ where $m_{??}$ corresponds to the methods and $a_{??}$ corresponds to its arguments.

Return statement: (S9) if the buggy statement is a return statement, *AuthFix* applies this schema to replace the return expression with an expression hole $e_{\tau??}$ of type τ.

6.2.3.2 Synthesize Repair Expressions

Our approach uses a rich set of repair expressions to synthesize the *holes* in the generated sketches and thereby generate the potential repair candidates. Given a sketch with holes at location l, we use a) relevant program elements in scope at l, 2) OpenID-specific elements that are available in the context of the underlying authentication flow of the program. To synthesize the holes, *AuthFix* extracts all the local variables and literals in scope, fields in the same class, and public fields from other classes that are relevant to the buggy class of the program. We identify the relevant classes by analyzing those that are instantiated and whose fields or methods are accessed within the buggy method.

In addition to the program elements within the scope, we further extract all the OpenID-specific elements that can be accessed within the context of the underlying OpenID *authentication flow* for the buggy program. As the buggy programs are part of one of the authentication flows (e.g., implicit flow, hybrid flow, etc.) of OpenID Connect, we specifically extract the configured values (i.e., values determined during the registration process), constants, and external functions that are globally accessible during the execution of authentication flows.

Syntax of repair expressions. Figure 6.5 illustrates the grammar (simplified) of the repair expressions used by *AuthFix* to complete the *holes* in the sketches. We define the non-deterministic choices of program constructs that include expressions, operators, and method invocation. The atomic expression holes represent the program variables, constants, and field accesses. Constants include both program constants and OpenID-specific constants that are relevant within the scope of the holes. We define arithmetic operators, relational operators, and logical operators to complete the operator holes in the sketches. Moreover, we synthesize composite expression holes by combining expression holes with operator holes. Composite expressions can further be combined together to synthesize more complex expressions. Method invocation holes include OpenID-specific methods along with the list of arguments associated with the methods.

$$
\begin{array}{lrcl}
\text{constant} & const & := & \mathcal{C} \mid null \mid true \mid false \mid id \mid k \\
\text{arithmatic op} & aop & := & + \mid - \mid \times \mid / \mid \% \\
\text{relational op} & rop & := & == \mid != \mid > \mid < \mid \geq \mid \leq \\
\text{logical op} & lop & := & \&\& \mid || \\
\text{atomic expr} & expr & := & var \mid const \mid var.f \\
\text{composite expr} & expr & := & expr\ op\ expr \mid var[expr] \\
\text{arguments} & args & := & args, expr \mid expr \\
\text{method invocation} & m & := & \mathcal{F}(args) \mid var.\mathcal{F}(args)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C} & \in & \textbf{OpenID constants} \\
\mathcal{F} & \in & \textbf{OpenID functions}
\end{array}
$$

Figure 6.5: Simplified grammar used for the repair expressions.

The grammar of the repair expressions is then used by *AuthFix* to instantiate a program transformation at each location of the *holes* of the sketches to produce a set of concrete repair candidates. Specifically, *AuthFix* uses an enumeration-based search to synthesize generalizable repair expressions for each type of hole in the sketches. Instead of processing sketches and holes arbitrarily, we process them in the order of sketch generation schemas mentioned above. Specifically, the sketch generated by the schema $S1$ is processed before the sketch by $S2$. If the search procedure cannot find a solution for the sketch by a schema, it advances to the next. This approach tractably constrains the search space, especially for the bugs with simple fixes (e.g., alternative operator). While synthesizing a

hole in a given sketch, if the search fails, *AuthFix* iteratively chooses a new element from the grammar and generates a new repair candidate. Each candidates are then validated against the specification using our model checker.

Reduce search space using LLM. Using the grammar in Figure 6.5 may generate a significantly large number of candidates of repair expressions, especially for composite expressions and method invocation. Since LLM has shown success in understanding the semantics of a given program (Jin et al., 2023), we utilize the power of LLMs to prioritize the expressions that have a high likelihood of being valid for fixing the underlying bug. To do so, we include the program sketches and submit a query to LLM to suggest candidate expressions for each hole in the program. Therefore, in our iterative search for repair expressions for the holes, we first choose the expressions suggested by the LLM agent before other expressions generated from the grammar.

6.2.4 Candidate Validation

In the absence of comprehensive test cases for OpenID programs that cover all their specifications, we utilize Petri net-based model checking to ensure that the generated candidates satisfy the required OpenID specifications. To begin, we provide an overview of the essential concepts and properties of Petri net structures, which form the foundation of our candidate validation approach.

6.2.4.1 Petri net Preliminaries

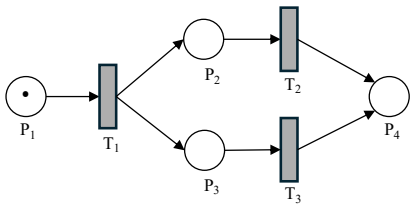


Figure 6.6: A simple Petri net with four places (circles) and two transitions (rectangles).

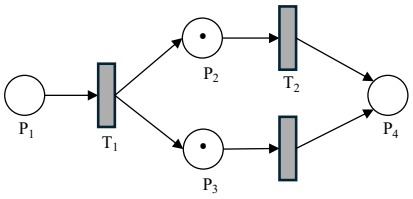


Figure 6.7: Result of Petri net in Figure 6.6 when transition T_1 is fired.

Petri net Definition Petri net is a powerful modeling language that has been successfully used to represent various aspects (Cortés et al., 2003; Billington et al., 2003; Grobelna et al., 2016) of modeled systems. A Petri net (Figure 6.6) is a particular kind of bipartite graph that consists of types of nodes: *places* and *transitions*. Places are represented with circles, and transitions are represented with long narrow rectangles. Places can contain *tokens*, which are drawn as dots. Tokens are used to define the execution of a Petri net. Directed *arcs* (i.e., edges) connect places to transitions or transitions to places. A *marking* of a Petri net is a mapping from each place p to the number of tokens at p. Transitions correspond to events that change the marking. Specifically, incoming arcs to a transition t represent the necessary conditions for t to fire, and outgoing arcs represent the outcome. The presence or absence of a token in a place can indicate whether a condition associated with a place is true or false. A necessary condition for a transition to fire is that there must be at least one token present in all incoming places connected to it. For example, transition T_1 in Figure 6.6 has at least one token in its input place P_1. Since the precondition to fire this transition is met, we say that T_1 is enabled and therefore, can fire. Once T_1 is fired, a token from P_1 is consumed and new tokens at the outgoing places P_2 and P_3 are produced.

Definition 1 (Petri net)**.** *A Petri net $\mathcal{N}$ is a tuple (P, T, A, W, M_0), where*

1. *$P = \{p_1, p_2, \ldots, p_m\}$ is a finite set of places;*
2. *$T = \{t_1, t_2, \ldots, t_n\}$ is a finite set of transitions, $P \cup T \neq \varnothing$, and $P \cap T = \varnothing$;*
3. *$A \subseteq (P \times T) \cup (T \times P)$ is the set of arcs (edges);*
4. *$W : E \to N$ is the weight function for edges; and*
5. *$M_0 : P \to N$ is the initial marking where N is a set of non-negative integers.*

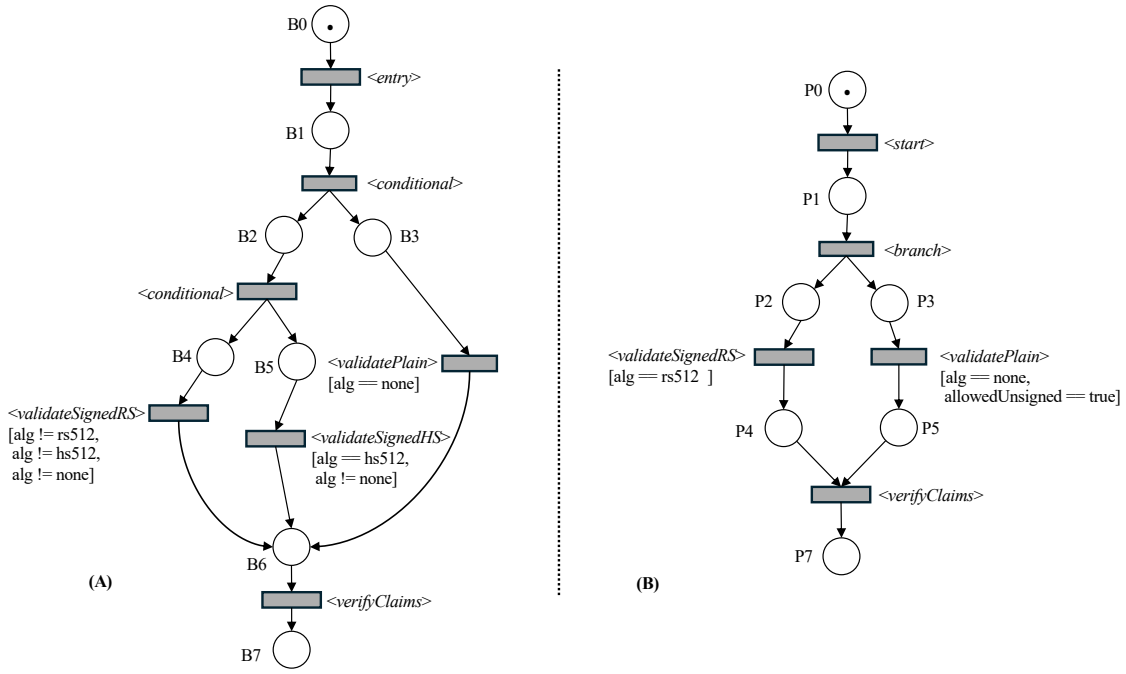


Figure 6.8: (A) A Petri net example (simplified) for the code example in Figure 6.1, and (B) corresponding specification Petri net provided by users.

Example 6. *Consider the example of a Petri net shown in Figure 6.6, where* $P = \{P_1, P_2, P_3, P_4\}$ *and* $T = \{T_1, T_2, T_3\}$*. We assume the weights for all edges* $e \in E$ *is one and therefore, the initial marking* M_0 *is* $(1\ 0\ 0\ 0)$*. Under this initial marking, only* T_1 *transition is enabled. Firing of* T_1 *results in a new marking, say* M_1 *is* $(0\ 1\ 1\ 0)$*, which enables the transitions* T_2 *and* T_3*.*

Petri net with guarded transitions We extend the Petri net described above with guarded conditions which allows to impose additional constraints on Petri net transitions. Specifically, we extend the transition T to a pair $\langle \epsilon, \gamma \rangle$, where ϵ denotes a transition event and γ denotes the guard condition corresponding to the transition. Therefore, in a given execution environment, a transition $T_i = \langle \epsilon_i, \gamma_i \rangle$ is enabled if and only if event ϵ_i occurs and the guard condition γ_i is satisfied. Guards are defined over the visible variables at the incoming places using binary operators (e.g., $>, >=, ==, \ldots, \&\&, ||$, etc.).

Reachability An important requirement in modeling event-driven systems or protocols like OpenID Connect using Petri net is whether a system can reach a specific state. Reachability of Petri net is defined as follows: Given the initial marking M_0 and a specific marking M_i of a Petri net $\mathcal{N}$, is it possible to fire a sequence of transitions to reach marking M_i starting from M_0? For example, given the Petri net in Figure 6.6, target marking $M_i = (0\ 0\ 0\ 1)$ is reachable as we can reach to M_i by firing the sequence of transitions T_1, T_2 or T_1, T_2, T_3.

We denote the reachable state space of Petri net $\mathcal{N}$ by $R(\mathcal{N})$, which represents the set of all markings that can be reached from the initial marking M_0.

We now illustrate how we validate the generated repair candidates. The candidate validation approach of *AuthFix* takes a repair candidate program P and its OpenID specification ϕ as input and validates if P satisfies the requirements of ϕ.

Construction of program Petri net Our approach requires to transform the candidate program P into a Petri net representation $\mathcal{N}_P$. Control Flow Graph (CFG). A CFG representation can be formally defined as $\mathcal{G}(N, E)$ where,

1. Each node $n \in N$ represent a basic block of the program; and
2. Each directed edge $(n_i, n_j) \in E$ indicates control flow from basic block n_i to n_j.

On the other hand, we define the Petri net constructed from the $\mathcal{G}$ as $\mathcal{N}_P(P, T, A, M_0)$ which can be constructed as follows.

1. **Places P:** Each basic block $n \in N$ corresponds to a place $p \in P$ in Petri net $\mathcal{N}_P$.
2. **Transitions T:** Each transition $t \in T$ represents a control point (e.g., branch, method invocation, etc.) of the program P. We create separate transitions for each possible branch outcome. For method invocation, we use the method name as the transition event, whereas conditional expressions are used as guards for the transition. We assign `true` as the guard value for the transitions with no guard condition.
3. **Arcs A:** For each edge $(n_i, n_j) \in E$ we create an arc from place p_i to transition t

and another arc from t to place p_j.

As tokens in the Petri net represent the state of execution or control flow, the initial marking M_0 represents a token at the starting place of the Petri net which corresponds to the entry block of the program.

Example 7. *Figure 6.8(A) illustrates a simplified program Petri net corresponding to a method associated with the code example in Figure 6.1. Here, each basic block in {B0, B1, ... Bn} corresponds to a place, and arcs connect these places to transitions and vice versa. Transitions symbolize program events, such as method invocations (e.g., validatePlain, verifyClaims, etc.) and branches that control the transition between the basic blocks. Conditional expressions of the branches are associated with the guard condition of the transition. For example, the guard condition [alg == none] for 'validatePlain' transition implies that the transition can fire only if validatePlain method is invoked and the value of alg is 'none' at the current state. Additionally, the program Petri net may include two special types of transitions: entry and exit, which mark the initial and final blocks of a method. A token is placed at the initial place to represent the program's starting state.*

Construction of specification Petri net *AuthFix* uses a Petri net representation of the OpenID Connect specification which defines the expected behavior during the protocol execution. Specification Petri net is defined as $\mathcal{N}_\phi(P_\phi, T_\phi, A_\phi, M_{0\phi})$ where,

1. P_ϕ is the set of protocol's execution state;
2. T_ϕ is the set of OpenID events that changes the protocol's state. These events can be constrained using *guard* conditions;
3. A_ϕ is the set of arcs that connect the execution states and transitions; and
4. $M_{0\phi}$ is the initial state of the protocol.

Example 8. *Figure 6.8(B) illustrates the specification Petri net constructed from the specification (as standard Petri net input formats (Weber and Kindler, 2003)) provided by the user. Each place in {P1, P2, ..., Pn} denotes a specification state and transitions denote the events that should be satisfied to enable firing. Similar to the program Petri net,*

specification transitions can also be associated with guard conditions using the relevant protocol variables and operators.

Validation algorithm Once the Petri net $\mathcal{N}_P$ is constructed for the candidate program, we validate $\mathcal{N}_P$ against the specification Petri net $\mathcal{N}_\phi$. Algorithm 1 illustrates the candidate validation algorithm, which takes two Petri nets, $\mathcal{N}_P$ and $\mathcal{N}_\phi$ as input and checks if $\mathcal{N}_P$ satisfies the $\mathcal{N}_\phi$. To do so, our algorithm starts with the initial marking of M_0 and M_ϕ for both Petri nets (line 5). In each iteration, it checks if an enabled transition T at M_0 *satisfies* the enabled transition T_ϕ at M_ϕ. This is done by checking the pre-condition of whether both transitions represent the same event ϵ and equivalent guard conditions γ (line 12). These guard conditions are transformed into Conjunctive Normal Form (CNF) and their satisfiability (denoted by $\models$) is checked by an off-the-shelf SAT solver (Le Berre and Parrain, 2010). If both transitions satisfy the pre-conditions, we fire both transitions and expand the search space to the resulting markings, M' and M'_ϕ, respectively for both nets (lines 13-15). However, if transition T does not satisfy the specification transition T_ϕ, we expand our search space on the program net while staying at the same marking in the specification net (lines 17-18). The algorithm terminates by returning `false` (line 9) if we reach the final marking at the program net $\mathcal{N}_P$ but the specification net $\mathcal{N}_\phi$ could not reach the final marking, meaning the candidate program did not satisfy the specification net. On the other hand, if we reach the final marking on both nets, the algorithm returns `true` (line 22), meaning the candidate program satisfies the specification.

6.2.4.2 Implementation

We implement the technical concepts discussed above in the *AuthFix* tool, which takes OIDC programs and their specification (in both English and corresponding Petri net representation in PNML format (Hillah et al., 2010)) as input and generates a fixed program that satisfies the specification. The *AuthFix* tool, consisting of approximately 5,400 lines of Java code, leverages the popular ChatGPT (GPT-3.5) (Wu et al., 2023a) LLM agent for bug localization and optimized repair expression generation. Additionally,

Algorithm 1 Candidate Validation

Input: $\mathcal{N}_P$: Candidate program net, $\mathcal{N}_\phi$: Specification net
Output: either `true` (valid) or `false` (invalid)
assume $\mathcal{N}_p(P, T, A, M_0)$
assume $\mathcal{N}_\phi(P_\phi, T_\phi, A_\phi, M_{0\phi})$
$\Phi = \{\langle M_0, M_{0\phi}\rangle\}$
while $\Phi \neq \emptyset$ **do**
    choose $\langle M, M_\phi\rangle \in \Phi$
    if $(M \in \texttt{final}(\mathcal{N}_P)) \wedge (M_\phi \notin \texttt{final}(\mathcal{N}_\phi))$ **then**
        return `false`
    end if
    for all $T \in \texttt{enabled}(M)$ **do**
        if $\exists T_\phi \in \texttt{enabled}(M_\phi).(T.\epsilon = T_\phi.\epsilon) \wedge (T.\gamma \vDash T_\phi.\gamma)$ **then**
            $M' = \texttt{fire}(M, T)$
            $M'_\phi = \texttt{fire}(M_\phi, T_\phi)$
            $\Phi = \Phi \cup \langle M', M'_\phi\rangle$
        else
            $M' = \texttt{fire}(M, T)$
            $\Phi = \Phi \cup \langle M', M_\phi\rangle$
        end if
    end for
end while
return `true`

we utilize the IBM T.J. Watson Libraries for Analysis (WALA) (WALA, 2023) toolkits to generate program representations (such as call graphs and control flow graphs) when transforming program semantics into Petri net representations. Furthermore, we employ the SAT4J tool (Le Berre and Parrain, 2010) to check the equivalence of guard conditions associated with Petri net events in both programs and specifications.

6.3 Evaluation

This section begins by presenting the research questions, followed by an evaluation designed to address these questions.

Research questions. We design our evaluation scheme primarily to answer the following research questions:

1. **RQ1.** Is *AuthFix* effective in fixing real-world bugs in OpenID implementations?

2. **RQ2.** Are the generated repairs as correct as the manual patches written by the developers?
3. **RQ3.** How effective is our approach when compared to the *LLM-only* repair method?

6.3.1 Experimental Setup

We first describe the datasets of OpenID bugs that we compile to evaluate *AuthFix*.

OpenID bug type	**Authentication Flows**	**Impacted parties**	**OpenID References**	**#Bugs**
Incorrect auth flow	AC, I, H	RP, OP	§3.1, §3.2, §3.3	2
Signature verification	I, H	RP, OP	§3.1.3.6, §3.1.3.7	7
`alg` validation	I, H	RP, OP	§3.1.3.7	3
`aud` validation	AC, I, H	RP	§3.1.3.7	3
`iss` validation	AC, I, H	RP	§3.1.3.7	2
`nonce` check	I, H	RP, OP	§3.1.3.7	1
Access token validation	AC, I, H	RP	§3.1.3.7, §3.2.2.9	2
CSRF protection	AC, I	RP, OP	§3.1.2.1	3

Table 6.1: Statistics of our dataset of OpenID bugs which are collected from libraries studied in Chapter 5 and from publicly available bug reports for other OpenID projects (OpenID, 2023).

6.3.1.1 Dataset collection

To evaluate the performance of our repair approach, we compile a dataset of OpenID bugs that consists of the confirmed security bugs in OpenID libraries as we reported in the prior chapters (Chapter 3-5), as well as the public records (e.g., CVE reports) for other OpenID-based projects. In our dataset, each benchmark contains at least one bug that violates the OpenID specification. Table 6.1 shows the statistics corresponding to the collected dataset of OpenID bugs. Specifically, our contains 23 bugs from eight categories of OpenID bugs. These bug types cover all three authentication flows in OpenID protocol: 1) authorization code flow (AC), 2) implicit flow (I), and 3) hybrid flow (H). Depending on the flow, each bugs impact either the relying party (RP), OpenID provider (OP), or both.

6.3.2 Experimental Results

We carried out all experiments on a machine equipped with a Quad-Core Intel Core i5 processor and 32GB of memory, operating on macOS 14.4. We delve into the detailed insights regarding the results of our experiments as follows.

Effectiveness of *AuthFix* (RQ1). To evaluate the effectiveness of *AuthFix* in repairing OpenID bugs, we use it to repair 23 benchmarks in our dataset. Among these bugs, 16 bugs require fixing conditional expressions and method invocations, three bugs involve fixing the return statement and four require fixing expressions used in the method's arguments. As shown in Table 6.2, *AuthFix* generated correct fixes for 15 bugs in our dataset. On the other hand, it could not generate any correct patches for eight bugs. By default, we consider the top 10 buggy statements localized by the LLM agent. As we increase the bound to 20, *AuthFix* generated only one more fix for a bug of 'nonce check' type. Moreover, the maximum time required by *AuthFix* to fix a bug is 92 minutes (for signature verification type bugs) whereas the minimum time is 1.2 minutes (for audience validation type bugs).

OpenID bug type	**#Bug**	**#Fix**	**#Fix$_{dev}$**	**Avg. repair time(m)**
Incorrect auth flow	2	1	1	44.7
Signature verification	7	4	2	85.4
`alg` validation	3	2	2	27.1
`aud` validation	3	3	3	16
`iss` validation	2	1	1	5.4
`nonce` check	1	1	-	18.4
Access token validation	2	1	-	38.7
CSRF protection	3	2	0	82.2
Overall	23	15	9	13.8

Table 6.2: Evaluation results of *AuthFix* for OpenID bugs repair in our dataset. Column #Fix shows the correct and plausible fixes generated by *AuthFix* and column #Fix$_{dev}$ shows the number of generated fixes that are semantically equivalent to the manual patches written by the developers (if available).

Quality of the generated repairs (RQ2). To answer our second research question, we manually investigate the generated patches and compare them against the patches written by the developers of the libraries we include in our benchmarks. These manually

written patches are based on the bugs we reported in our prior works that are presented in the previous chapters. However, for three bugs from 'nonce check' and 'access token validation' category, we were not able to obtain the manual patches as the developers did not respond to our reports. In total, we collected the 20 manually written patches corresponding to the bugs in our dataset. Upon our manual investigation, we found that 9 out of the 15 generated patches are semantically equivalent to the patches written by human developers (Table 6.2). These results show that *AuthFix* can generate high-quality patches to repair the OpenID bugs.

OpenID bug type	**#Bug**	**#Fix**$_{AuthFix}$	**#Fix**$_{LLM}$
Incorrect auth flow	2	1	0
Signature verification	7	4	1
`alg` validation	3	2	1
`aud` validation	3	3	1
`iss` validation	2	1	0
`nonce` check	1	1	0
Access token validation	2	1	0
CSRF protection	3	2	1
Overall	23	15	4

Table 6.3: Comparison between the repairs generated by *AuthFix* (column #Fix$_{AuthFix}$) and repairs generated by using only LLM (column #Fix$_{LLM}$).

Repair using only LLM agent (RQ3). Given the recent success of LLM in various program repair tasks (Joshi et al., 2023; Xia et al., 2023), we further investigate how our approach is compared against the repairs that are solely generated by LLM agents (e.g., ChatGPT (Wu et al., 2023a)). Specifically, we provide the LLM agent with the buggy code from the benchmarks in our dataset and a natural language specification, asking it to fix the corresponding bugs. As shown in Table 6.3, LLM agent were able to generate repairs for only four bugs. In this evaluation, we observe that LLM can fix commonly observed programming bugs (Saha et al., 2017; Xuan et al., 2016) such as missing clause for null checking. However, in this work, we focus on the bugs relevant to the incorrect implementation of OpenID protocol with respect to the it's specification, for which LLM agents struggle to generate valid repair.

6.3.2.1 Limitations

While *AuthFix* is capable of fixing many critical security bugs in OpenID implementations, it has some limitations. For example, *AuthFix* synthesizes the holes in repair sketches based on the expressions available within the codebase, including variables and methods that are implemented within the OpenID libraries. As a result, it cannot fix bugs that require introducing variables of new data types (such as certain OpenID parameters defined during registration) or creating new data structures (like a different structure for ID tokens). Additionally, when handling method invocations, our approach leverages available methods with consistent signatures or return types. However, our current approach cannot synthesize the complete implementation of new methods for processing OpenID parameters, which limits its ability to fix certain bugs. For example, some libraries lack implementations for handling JWT APIs for signature verification, and generating the necessary method bodies for fixing such bugs (like signature validation) is outside our current scope. Furthermore, our approach currently requires manual construction of the Petri net to model the program's expected behavior according to the specification. An exciting area for future exploration would be harnessing the natural language understanding capabilities of LLMs to automatically translate plain English specifications into their corresponding Petri net models.

6.4 Summary

In this chapter, we present an LLM-assisted automated program repair approach to fix real-world bugs in OpenID Connect protocol. We present an enumerative search-based repair technique that utilizes a novel Petri net-based approach to validate the repair candidates against the standard OpenID specification. Our evaluation shows that our proposed repair approach generates correct patches for 15 OpenID bugs out of 23, and 44% of generated patches are semantically equivalent to the manually written patches by the developers.

CHAPTER 7

Conclusion

This dissertation has addressed a critical issue in secure systems, namely the logical mistakes in the implementation of SSO protocols that violate the standard specification, leading to various security threats. Through a comprehensive approach incorporating static analysis, formal verification, and program synthesis techniques, we have designed and implemented automated and scalable techniques to identify, analyze, and mitigate security threats and vulnerabilities in SSO protocols.

Our toolchain has been evaluated on a large scale, covering over 20 popular SSO service providers and 600 relying party applications, revealing 40 previously unknown vulnerabilities, including eight that led to new CVE entries. This demonstrates the effectiveness of our approach in enhancing the security of SSO protocols and protecting sensitive user resources.

The significance of this research lies in its contribution to the development of secure and reliable SSO protocols, which are essential for operating secure systems. By providing an end-to-end framework that can identify and mitigate security threats, we have helped to bridge the gap between the standard specification and its implementation, reducing the risk of security breaches and attacks. Furthermore, our proposed specification languages can be extended to express properties for any possible extensions of the protocols in the future, ensuring that our proposed approaches are extensible and can be applied across diverse platforms.

In conclusion, this dissertation has made a crucial contribution to the field of secure systems, addressing a critical issue in SSO protocols and providing a comprehensive solution to identify, analyze, and mitigate security threats and vulnerabilities. Our

approach has the potential to enhance the security of SSO protocols, protecting sensitive user resources and preventing severe attacks. Future work includes extending our toolchain to support additional identity-based protocols and access management platforms, further enhancing the security of secure systems.

Aafer, Y., Tao, G., Huang, J., Zhang, X., and Li, N. (2018). Precise android api protection mapping derivation and reasoning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1151–1164. 17

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2007). *Compilers: Principles, Techniques, & Tools*. Pearson Education India. 17

Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley. 93, 94

Ammar, M., Crispo, B., Jacobs, B., Hughes, D., and Daniels, W. (2019). S μ v—the security microvisor: A formally-verified software-based security architecture for the internet of things. *IEEE Transactions on Dependable and Secure Computing*, 16(5):885–901. 18

ApiFest (2020). "apifest oauth2". https://github.com/apifest/apifest-oauth20. 63

ApiFest (2021). "apifest api security". http://www.apifest.org. 50

AppAuth (2023). Appauth-android. https://openid.github.io/AppAuth-Android/. 105

Arquint, L., Wolf, F. A., Lallemand, J., Sasse, R., Sprenger, C., Wiesner, S. N., Basin, D., and Müller, P. (2023). Sound verification of security protocols: From design to interoperable implementations. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1077–1093. IEEE. 17

Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Traon, Y. L., Octeau, D., and McDaniel, P. (2014a). Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI*, pages 259–269. ACM. 24

Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Traon, Y. L., Octeau, D., and McDaniel, P. D. (2014b). Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 259–269. 16

Bai, G., Lei, J., Meng, G., Venkatraman, S. S., Saxena, P., Sun, J., Liu, Y., and Dong, J. S. (2013). Authscan: Automatic extraction of web authentication protocols from implementations. 2, 45, 46, 77

Baier, C. and Katoen, J.-P. (2008). *Principles of model checking*. MIT press. 95

Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., and Leino, K. R. M. (2006). Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures 4*, pages 364–387. Springer. 78

Basin, D., Dreier, J., Hirschi, L., Radomirovic, S., Sasse, R., and Stettler, V. (2018). A formal analysis of 5g authentication. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 1383–1396. 16

Billington, J., Gallasch, G. E., and Han, B. (2003). A coloured petri net approach to protocol verification. In *Advanced Course on Petri Nets*, pages 210–290. Springer. 118

Blackdirect (2021). "blackdirect attack". https://black.direct. 1

Blanchet, B. (2014). Automatic verification of security protocols in the symbolic model: The verifier proverif. *Foundations of Security Analysis and Design VII: FOSAD 2012/2013 Tutorial Lectures*, pages 54–87. 16

Calzavara, S., Focardi, R., Maffei, M., Schneidewind, C., Squarcina, M., and Tempesta, M. (2018). Wpse: fortifying web protocols via browser-side security monitoring. In *27th USENIX Security Symposium*, pages 1493–1510. 14, 15

Cao, Y., Fratantonio, Y., Bianchi, A., Egele, M., Kruegel, C., Vigna, G., and Chen, Y. (2015). Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *NDSS*. 17

Cao, Y., Shoshitaishvili, Y., Borgolte, K., Kruegel, C., Vigna, G., and Chen, Y. (2014). Protecting web single sign-on against relying party impersonation attacks through a bi-directional secure channel with authentication. 14

CBS (2019). Cbs sports android application. https://play.google.com/store/apps/details?id=com.handmark.sportcaster. 40

Chatous (2019). Chatous android application. https://play.google.com/store/apps/details?id=com.chatous.chatous. 35

Chen, E. Y., Pei, Y., Chen, S., Tian, Y., Kotcher, R., and Tague, P. (2014a). Oauth demystified for mobile application developers. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 892–903. 14, 22

Chen, E. Y., Pei, Y., Chen, S., Tian, Y., Kotcher, R., and Tague, P. (2014b). Oauth demystified for mobile application developers. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 892–903. 45

Cheng, K., Li, Q., Wang, L., Chen, Q., Zheng, Y., Sun, L., and Liang, Z. (2018). Dtaint: detecting the taint-style vulnerability in embedded device firmware. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 430–441. IEEE. 17

Cirani, S., Picone, M., Gonizzi, P., Veltri, L., and Ferrari, G. (2015). Iot-oas: An oauth-based authorization service architecture for secure services in iot scenarios. *IEEE Sensors Journal*, 15(2):1224–1234. 14

Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. (2000). Counterexample-guided abstraction refinement. In *Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings 12*, pages 154–169. Springer. 18

Clouway (2021). "clouway: Oauth2 server". `https://github.com/clouway/oauth2-server`. 63

Cortés, L. A., Eles, P., and Peng, Z. (2003). Modeling and formal verification of embedded systems based on a petri net representation. *Journal of Systems Architecture*, 49(12-15):571–598. 118

Cremers, C. and Dehnel-Wild, M. (2019). Component-based formal analysis of 5g-aka: Channel assumptions and session confusion. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society. 16

Cremers, C., Horvat, M., Hoyland, J., Scott, S., and van der Merwe, T. (2017). A comprehensive symbolic analysis of tls 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1773–1788. 16

Cremers, C., Horvat, M., Scott, S., and van der Merwe, T. (2016). Automated analysis and verification of tls 1.3: 0-rtt, resumption and delayed authentication. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 470–485. IEEE. 16

Datalog (2022). "datalog". `https://en.wikipedia.org/wiki/Datalog`. 53

DeMarco, F., Xuan, J., Le Berre, D., and Monperrus, M. (2014). Automatic repair of buggy if conditions and missing preconditions with smt. In *Proceedings of the 6th international workshop on constraints in software testing, verification, and analysis*, pages 30–39. 109

Dietsch, D., Heizmann, M., Langenfeld, V., and Podelski, A. (2015). Fairness modulo theory: A new approach to ltl software model checking. In *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I 27*, pages 49–66. Springer. 77, 78, 83, 94, 96, 103

Egele, M., Brumley, D., Fratantonio, Y., and Kruegel, C. (2013a). An empirical study of cryptographic misuse in android applications. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 73–84. 16

Egele, M., Brumley, D., Fratantonio, Y., and Kruegel, C. (2013b). An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 73–84. 17, 48

Elboher, Y. Y., Gottschlich, J., and Katz, G. (2020). An abstraction-based framework for neural network verification. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I 32*, pages 43–65. Springer. 18

Emerson, S., Choi, Y., Hwang, D., Kim, K., and Kim, K. (2015). An oauth based authentication mechanism for iot networks. In *2015 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 1072–1074. 14

Expo (2023). Critical oauth vulnerability in expo framework allows account hijacking. https://thehackernews.com/2023/05/critical-oauth-vulnerability-in-expo.html. 75

Facebook (2018a). Facebook: Security update. https://newsroom.fb.com/news/2018/09/security-update. Accessed: 05-11-2019. 1, 21

Facebook (2018b). Facebook security update. https://about.fb.com/news/2018/09/security-update. 45, 107

Facebook (2019). Facebook: Securing graph api requests. https://developers.facebook.com/docs/graph-api/securing-requests. 42

Fass, A., Backes, M., and Stock, B. (2019). Hidenoseek: Camouflaging malicious javascript in benign asts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1899–1913. 17

Fass, A., Somé, D. F., Backes, M., and Stock, B. (2021). Doublex: Statically detecting vulnerable data flows in browser extensions at scale. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1804. 17

Feng, H., Li, H., Pan, X., Zhao, Z., and Cactilab, T. (2021). A formal analysis of the fido uaf protocol. In *NDSS*. 17

Feng, Y., Anand, S., Dillig, I., and Aiken, A. (2014). Apposcopy: semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 576–587. 16

Feng, Y., Wang, X., Dillig, I., and Lin, C. (2015). EXPLORER : query- and demand-driven exploration of interprocedural control flow properties. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 520–534. 16

Fernandes, E., Rahmati, A., Jung, J., and Prakash, A. (2018). Decentralized action integrity for trigger-action iot platforms. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society. 14

Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987). The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349. 17

Fett, D., Hosseyni, P., and Küsters, R. (2019). An extensive formal security analysis of the openid financial-grade api. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 453–471. IEEE. 15

Fett, D., Küsters, R., and Schmitz, G. (2016). A comprehensive formal security analysis of oauth 2.0. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1204–1215. 15, 45

Fett, D., Küsters, R., and Schmitz, G. (2017). 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. 2, 15, 76

FireEye (2019). M-trends 2017 by fireeye. https://content.fireeye.com/m-trends/rpt-m-trends-2017. 35

Gao, P., Shao, F., Liu, X., Xiao, X., Liu, H., Qin, Z., Xu, F., Mittal, P., Kulkarni, S. R., and Song, D. (2021). A system for efficiently hunting for cyber threats in computer systems using threat intelligence. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 2705–2708. IEEE. 17

Ghasemisharif, M., Ramesh, A., Checkoway, S., Kanich, C., and Polakis, J. (2018). O single sign-off, where art thou? an empirical analysis of single sign-on account hijacking and session management on the web. In Enck, W. and Felt, A. P., editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, pages 1475–1492. USENIX Association. 14

GoFundMe (2019). Gofundme android application. https://play.google.com/store/apps/details?id=com.GoFundMe.GoFundMe. 31

Goldman, E. (2020). An introduction to the california consumer privacy act (ccpa). *Santa Clara Univ. Legal Studies Research Paper*. 73

Goldsmith, S. F., O'Callahan, R., and Aiken, A. (2005). Relational queries over program traces. *ACM SIGPLAN Notices*, 40(10):385–402. 17

Google (2019). Authenticate with a backend server. https://developers.google.com/identity/sign-in/android/backend-auth. 36

Google (2022). High vulnerability in google's oauth client library for java. https://it.ucsf.edu/high-vulnerability-googles-oauth-client-library-java. 107

Google (2023). Google oauth client. https://cloud.google.com/java/docs/reference/google-oauth-client/latest/overview. 104

Grammarly (2023). Critical oauth flaws uncovered in grammarly, vidio, and bukalapak platforms. https://thehackernews.com/2023/10/critical-oauth-flaws-uncovered-in.html. 75

Grobelna, I., Wiśniewski, R., Grobelny, M., and Wiśniewska, M. (2016). Design and verification of real-life processes with application of petri nets. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 47(11):2856–2869. 118

Hammann, S., Sasse, R., and Basin, D. (2020). Privacy-preserving openid connect. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 277–289. 2, 16, 76

Hammer-Lahav, E. (2010). The oauth 1.0 protocol. *RFC*, 5849:1–38. 8, 26

Hardt, D. (2012). The oauth 2.0 authorization framework. *RFC*, 6749:1–76. 8, 26

He, B., Rastogi, V., Cao, Y., Chen, Y., Venkatakrishnan, V., Yang, R., and Zhang, Z. (2015a). Vetting ssl usage in applications with sslint. In *2015 IEEE Symposium on Security and Privacy*, pages 519–534. IEEE. 2, 76, 77, 102

He, B., Rastogi, V., Cao, Y., Chen, Y., Venkatakrishnan, V., Yang, R., and Zhang, Z. (2015b). Vetting ssl usage in applications with sslint. In *2015 IEEE Symposium on Security and Privacy*, pages 519–534. IEEE. 17, 47

He, B., Rastogi, V., Cao, Y., Chen, Y., Venkatakrishnan, V. N., Yang, R., and Zhang, Z. (2015c). Vetting SSL usage in applications with SSLINT. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 519–534. 16

Hillah, L.-M., Kordon, F., Petrucci, L., and Treves, N. (2010). Pnml framework: an extendable reference implementation of the petri net markup language. In *Applications and Theory of Petri Nets: 31st International Conference, PETRI NETS 2010, Braga, Portugal, June 21-25, 2010. Proceedings 31*, pages 318–327. Springer. 122

Hussain, F., Hussain, R., Noye, B., and Sharieh, S. (2020). Enterprise api security and gdpr compliance: Design and implementation perspective. *IT Professional*, 22(5):81–89. 1

IETF (2021). Oauth 2.0 security best current practice. https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics. 11, 47, 50, 60, 98

IETF (2022). Oauth 2.0 security best current practice. https://datatracker.ietf.org/doc/html/draft-ietf-oauth-dpop. 64, 71

Jackson, D. (2012). *Software Abstractions: logic, language, and analysis*. MIT press. 2, 76

Jacomme, C., Klein, E., Kremer, S., and Racouchot, M. (2023). A comprehensive, formal and automated analysis of the edhoc protocol. In *USENIX Security'23-32nd USENIX Security Symposium*. 17

Jiang, N., Lutellier, T., and Tan, L. (2021). Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1161–1173. IEEE. 19

Jin, M., Shahriar, S., Tufano, M., Shi, X., Lu, S., Sundaresan, N., and Svyatkovskiy, A. (2023). Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1646–1656. 20, 117

Jin, P., Tian, J., Zhi, D., Wen, X., and Zhang, M. (2022). Trainify: A cegar-driven training and verification framework for safe deep reinforcement learning. In *International Conference on Computer Aided Verification*, pages 193–218. Springer. 18

Jones, M. B. and Hardt, D. (2012). The oauth 2.0 authorization framework: Bearer token usage. *RFC*, 6750:1–18. 9

Joshi, H., Sanchez, J. C., Gulwani, S., Le, V., Verbruggen, G., and Radiček, I. (2023). Repair is nearly generation: Multilingual program repair with llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 5131–5140. 126

Just, R., Jalali, D., and Ernst, M. D. (2014). Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440. 20

JWT (2015). Json web token. `https://datatracker.ietf.org/doc/html/rfc7519`. 12, 71, 76, 81, 82, 85

Kaleeswaran, S., Tulsian, V., Kanade, A., and Orso, A. (2014). Minthint: Automated synthesis of repair hints. In *Proceedings of the 36th International Conference on Software Engineering*, pages 266–276. 18

Kang, S., An, G., and Yoo, S. (2023). A preliminary evaluation of llm-based fault localization. *arXiv preprint arXiv:2308.05487*. 109

Kim, D., Nam, J., Song, J., and Kim, S. (2013). Automatic patch generation learned from human-written patches. In *2013 35th international conference on software engineering (ICSE)*, pages 802–811. IEEE. 18, 108

Le, X.-B. D., Chu, D.-H., Lo, D., Le Goues, C., and Visser, W. (2017). S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 593–604. 19

Le, X. B. D., Lo, D., and Le Goues, C. (2016). History driven program repair. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, volume 1, pages 213–224. IEEE. 18

Le Berre, D. and Parrain, A. (2010). The sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(2-3):59–64. 122, 123

Le Goues, C., Nguyen, T., Forrest, S., and Weimer, W. (2011). Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72. 18, 108

Leino, K. R. M. (2008). This is boogie 2. *manuscript KRML*, 178(131):9. 83, 86, 96

Leino, K. R. M. (2010). Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*, pages 348–370. Springer. 77

Li, L., Bartel, A., Klein, J., Traon, Y. L., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., and McDaniel, P. (2014). I know what leaked in your pocket: uncovering privacy leaks on android apps with static taint analysis. *arXiv preprint arXiv:1404.7431*. 17

Li, S., Kang, M., Hou, J., and Cao, Y. (2022). Mining node. js vulnerabilities via object dependence graph and query. In *USENIX Security Symposium*. 17

Li, W. and Mitchell, C. J. (2014). Security issues in oauth 2.0 SSO implementations. In *Information Security - 17th International Conference, ISC 2014, Hong Kong, China, October 12-14, 2014. Proceedings*, pages 529–541. 22

Li, Y., Wang, S., and Nguyen, T. N. (2020). Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, pages 602–614. 19

Lu, J., Zhang, J., Li, J., Wan, Z., and Meng, B. (2017). Automatic verification of security of openid connect protocol with proverif. In *Advances on P2P, Parallel, Grid, Cloud and Internet Computing: Proceedings of the 11th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC–2016) November 5–7, 2016, Soonchunhyang University, Asan, Korea*, pages 209–220. Springer. 16

Lutellier, T., Pham, H. V., Pang, L., Li, Y., Wei, M., and Tan, L. (2020). Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pages 101–114. 19

Lv, T., Li, R., Yang, Y., Chen, K., Liao, X., Wang, X., Hu, P., and Xing, L. (2020). Rtfm! automatic assumption discovery and verification derivation from library document for api misuse detection. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1837–1852. 17

Martin, M., Livshits, B., and Lam, M. S. (2005a). Finding application errors and security flaws using pql: a program query language. *Acm Sigplan Notices*, 40(10):365–383. 17

Martin, M. C., Livshits, V. B., and Lam, M. S. (2005b). Finding application errors and security flaws using PQL: a program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 365–383. 16

Mechtaev, S., Yi, J., and Roychoudhury, A. (2015). Directfix: Looking for simple program repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 448–458. IEEE. 19, 109

Mechtaev, S., Yi, J., and Roychoudhury, A. (2016). Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*, pages 691–701. 19

Meier, S., Schmidt, B., Cremers, C., and Basin, D. (2013). The tamarin prover for the symbolic analysis of security protocols. In *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*, pages 696–701. Springer. 2, 16, 76

Microsoft (2020). "microsoft azure account takeover". https://www.cyberark.com/resources/threat-research-blog/blackdirect-microsoft-azure-account-takeover. 44, 49

Microsoft (2023a). Azure b2c – crypto misuse and account compromise. https://securityboulevard.com. 1, 107

Microsoft (2023b). Microsoft bug allowed hackers to breach over two dozen organizations via forged azure ad tokens. https://thehackernews.com/2023/07/microsoft-bug-allowed-hackers-to-breach.html. 1, 75, 107

Naik, N. and Jenkins, P. (2017). Securing digital identities in the cloud by selecting an apposite federated identity management from saml, oauth and openid connect. In *2017 11th International Conference on Research Challenges in Information Science (RCIS)*, pages 163–174. IEEE. 1

Nejati, S., Liang, J. H., Gebotys, C., Czarnecki, K., and Ganesh, V. (2017). Adaptive restart and cegar-based solver for inverting cryptographic hash functions. In *Verified Software. Theories, Tools, and Experiments: 9th International Conference, VSTTE 2017, Heidelberg, Germany, July 22-23, 2017, Revised Selected Papers 9*, pages 120–131. Springer. 18

Nguyen, H. D. T., Qi, D., Roychoudhury, A., and Chandra, S. (2013). Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE. 18

NodeOAuth2 (2021). Node oauth2 server. https://github.com/oauthjs/node-oauth2-server. 46, 48, 62, 63

NodeOIDC (2021). "node oidc provider". https://github.com/panva/node-oidc-provider. 62, 63

OAuth1 (2010). Oauth 1.0. https://oauth.net/1/. 22

OAuth2 (2012). Oauth 2.0. https://oauth.net/2/. 1, 3, 23

Oauth2orize (2021). "oauth2orize". https://github.com/jaredhanson/oauth2orize. 62, 63

Oauth2Server (2021). "oauth2 server". https://github.com/yoichiro/oauth2-server. 63

OAuthServer (2021). "oauth2 server node". https://github.com/af83/oauth2_server_node. 63

OAuthServerJob (2021). "oauth2 server". https://github.com/jobmission/oauth2-server. 63

OpenID (2021). Openid connect core 1.0. https://openid.net/specs/openid-connect-core-1_0.html. 1, 3, 7, 12, 72, 76, 79, 91, 97, 98, 105, 110, 113

OpenID (2023). Openid certification. https://openid.net/certification/. xiii, 99, 108, 124

OXAuth (2021). "gluufederation: oxauth". https://github.com/GluuFederation/oxAuth. 63

Pac4j (2023). Pac4j. https://www.pac4j.org. 80

Pai, S., Sharma, Y., Kumar, S., and Singh, S. (2011). Formal verification of oauth 2.0 using alloy framework. In *International Conference on Communication Systems & Network Technologies*. 15, 45

Park, K., Sahin, B., Chen, Y., Zhao, J., Downing, E., Hu, H., and Lee, W. (2021). Identifying behavior dispatchers for malware analysis. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 759–773. 17

Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D., and Vechev, M. (2020). Verx: Safety verification of smart contracts. In *2020 IEEE symposium on security and privacy (SP)*, pages 1661–1677. IEEE. 18

Portswigger (2023). Authentication bug that enabled unauthorized access to client applications. https://portswigger.net. 107

Qi, Y., Mao, X., Lei, Y., Dai, Z., and Wang, C. (2014). The strength of random search on automated program repair. In *Proceedings of the 36th international conference on software engineering*, pages 254–265. 18

Rahaman, S., Xiao, Y., Afrose, S., Shaon, F., Tian, K., Frantz, M., Kantarcioglu, M., and Yao, D. (2019). Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2455–2472. 17

Rahat, T. A., Arman, A., and Ali, M. E. (2018). Maximizing reverse k-nearest neighbors for trajectories. In *Australasian Database Conference*, pages 262–274. Springer. 17

Rahat, T. A., Feng, Y., and Tian, Y. (2019). OAUTHLINT: an empirical study on oauth bugs in android applications. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 293–304. 76, 77

Rahat, T. A., Feng, Y., and Tian, Y. (2022). Cerberus: query-driven scalable vulnerability detection in oauth service provider implementations. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, page 2459–2473. 76, 77, 102

Rahat, T. A., Le, T., and Tian, Y. (2021). Automated detection of gdpr disclosure requirements in privacy policies using deep active learning. *arXiv preprint arXiv:2111.04224*. 73

RFC6749 (2021). The oauth 2.0 authorization framework. `https://tools.ietf.org/html/rfc6750`. 7, 44, 47, 50, 60, 62, 72, 76, 103

RFC6819 (2021). Oauth 2.0 threat model and security considerations. `https://tools.ietf.org/html/rfc6819`. 50, 60

RFC7636 (2021). Proof key for code exchange by oauth public clients. `https://tools.ietf.org/html/rfc7636`. 11, 60

RFC8705 (2021). Oauth 2.0 mutual-tls client authentication and certificate-bound access tokens. `https://tools.ietf.org/html/rfc8705`. 60

Saha, R. K., Lyu, Y., Yoshida, H., and Prasad, M. R. (2017). Elixir: Effective object-oriented program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 648–659. IEEE. 20, 126

SAML (2023). Saml protocol. `http://saml.xml.org/saml-specifications`. 7

Sciancalepore, S., Piro, G., Caldarola, D., Boggia, G., and Bianchi, G. (2017). Oauth-iot: An access control framework for the internet of things based on open standards. In *2017 IEEE Symposium on Computers and Communications (ISCC)*, pages 676–681. IEEE. 14

Shehab, M. and Mohsen, F. (2014). Towards enhancing the security of oauth implementations in smart phones. In *IEEE Third International Conference on Mobile Services, Anchorage, AK, USA, June 27 - July 2, 2014*, pages 39–46. 14, 22

Shernan, E., Carter, H., Tian, D., Traynor, P., and Butler, K. R. B. (2015). More guidelines than rules: CSRF vulnerabilities from noncompliant oauth 2.0 implementations. In Almgren, M., Gulisano, V., and Maggi, F., editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings*, volume 9148 of *Lecture Notes in Computer Science*, pages 239–260. Springer. 14

Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., and Vigna, G. (2015). Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, volume 1, pages 1–1. 17

Soufflé Developers (2020). Soufflé - datalog. https://souffle-lang.github.io/index.html. 47

Souri, A. and Norouzi, M. (2019). A state-of-the-art survey on formal verification of the internet of things applications. *Journal of Service Science Research*, 11(1):47–67. 18

Spring (2021). "spring authorization server". https://github.com/spring-projects-experimental/spring-authorization-server. 48, 63

Sridharan, M., Gopan, D., Shan, L., and Bodík, R. (2005). Demand-driven points-to analysis for java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 59–76. 16

Staicu, C.-A., Pradel, M., and Livshits, B. (2018). Synode: Understanding and automatically preventing injection attacks on node. js. In *NDSS*. 17

Stephens, J., Ferles, K., Mariano, B., Lahiri, S., and Dillig, I. (2021). Smartpulse: automated checking of temporal properties in smart contracts. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 555–571. IEEE. 18

Sucasas, V., Mantas, G., Althunibat, S., Oliveira, L., Antonopoulos, A., Otung, I., and Rodriguez, J. (2018). A privacy-enhanced oauth 2.0 based protocol for smart city mobile applications. *Computers & Security*, 74:258–274. 14

Sun, S. and Beznosov, K. (2012a). The devil is in the (implementation) details: an empirical analysis of oauth SSO systems. In Yu, T., Danezis, G., and Gligor, V. D., editors, *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 378–390. ACM. 14

Sun, S.-T. and Beznosov, K. (2012b). The devil is in the (implementation) details: an empirical analysis of oauth sso systems. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 378–390. 14

Sven, A., Nguyen, H. A., Nadi, S., Nguyen, T. N., and Mezini, M. (2019). Investigating next steps in static api-misuse detection. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 265–275. IEEE. 17

Topface (2019). Topface android application. https://play.google.com/store/apps/details?id=com.topface.topface. 34, 41

Twitter (2021). "twitter redirect uri attack". https://hackerone.com/reports/110293. 2, 49

Van Emden, M. H. and Kowalski, R. A. (1976). The semantics of predicate logic as a programming language. In *Journal of the ACM (JACM)*, volume 23, pages 733–742. ACM New York, NY, USA. 54

Veronese, L., Calzavara, S., and Compagna, L. (2020). Bulwark: Holistic and verified security monitoring of web protocols. In *European Symposium on Research in Computer Security*, pages 23–41. Springer. 15

WALA (2023). T.j. watson libraries for analysis (wala). https://sourceforge.net/projects/wala. 78, 86, 96, 123

Wang, H., Zhang, Y., Li, J., Liu, H., Yang, W., Li, B., and Gu, D. (2015a). Vulnerability assessment of oauth implementations in android applications. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 61–70. 14

Wang, H., Zhang, Y., Li, J., Liu, H., Yang, W., Li, B., and Gu, D. (2015b). Vulnerability assessment of oauth implementations in android applications. In *Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA, December 7-11, 2015*, pages 61–70. ACM. 15

Waplog (2019). Waplog android application. https://play.google.com/store/apps/details?id=com.waplog.social. 33

Weber, M. and Kindler, E. (2003). The petri net markup language. In *Petri Net Technology for Communication-Based Systems: Advances in Petri Nets*, pages 124–144. Springer. 121

Weimer, W., Fry, Z. P., and Forrest, S. (2013). Leveraging program equivalence for adaptive program repair: Models and first results. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 356–366. IEEE. 18

White, J., Hays, S., Fu, Q., Spencer-Smith, J., and Schmidt, D. C. (2023). Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. *arXiv preprint arXiv:2303.07839*. 109

Wish (2019). Wish android application. https://play.google.com/store/apps/details?id=com.contextlogic.wish. 36

Wu, T., He, S., Liu, J., Sun, S., Liu, K., Han, Q.-L., and Tang, Y. (2023a). A brief overview of chatgpt: The history, status quo and potential future development. *IEEE/CAA Journal of Automatica Sinica*, 10(5):1122–1136. 122, 126

Wu, Y., Li, Z., Zhang, J. M., Papadakis, M., Harman, M., and Liu, Y. (2023b). Large language models in fault localisation. *arXiv preprint arXiv:2308.15276*. 109

Xia, C. S., Wei, Y., and Zhang, L. (2023). Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1482–1494. IEEE. 126

Xie, T., Tillmann, N., De Halleux, J., and Schulte, W. (2009). Fitness-guided path exploration in dynamic symbolic execution. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 359–368. IEEE. 45

Xiong, Y., Wang, J., Yan, R., Zhang, J., Han, S., Huang, G., and Zhang, L. (2017). Precise condition synthesis for program repair. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 416–426. IEEE. 18, 108

Xuan, J., Martinez, M., Demarco, F., Clement, M., Marcote, S. L., Durieux, T., Le Berre, D., and Monperrus, M. (2016). Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 43(1):34–55. 18, 108, 126

Yamaguchi, F., Golde, N., Arp, D., and Rieck, K. (2014). Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604. IEEE. 17

Yamaguchi, F., Lottmann, M., and Rieck, K. (2012). Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 359–368. 17

Yamaguchi, F., Maier, A., Gascon, H., and Rieck, K. (2015). Automatic inference of search patterns for taint-style vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*, pages 797–812. IEEE. 17

Yang, F. and Manoharan, S. (2013). A security analysis of the oauth protocol. In *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, pages 271–276. 14

Yang, F. and Manoharan, S. (2013). A security analysis of the oauth protocol. In *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, pages 271–276. IEEE. 45

Yang, R., Lau, W. C., Chen, J., and Zhang, K. (2018a). Vetting single sign-on sdk implementations via symbolic reasoning. In *27th USENIX Security Symposium*, pages 1459–1474. 15, 45, 46, 63

Yang, R., Lau, W. C., Chen, J., and Zhang, K. (2018b). Vetting single sign-on SDK implementations via symbolic reasoning. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, pages 1459–1474. 22

Yang, R., Li, G., Lau, W. C., Zhang, K., and Hu, P. (2016a). Model-based security testing: An empirical study on oauth 2.0 implementations. In Chen, X., Wang, X., and Huang, X., editors, *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, pages 651–662. ACM. 14

Yang, R., Li, G., Lau, W. C., Zhang, K., and Hu, P. (2016b). Model-based security testing: An empirical study on oauth 2.0 implementations. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 651–662. 46

Yang, Z. and Yang, M. (2012). Leakminer: Detect information leakage on android with static taint analysis. In *2012 Third World Congress on Software Engineering*, pages 101–104. IEEE. 17

Ye, H., Martinez, M., Durieux, T., and Monperrus, M. (2021). A comprehensive study of automatic program repair on the quixbugs benchmark. *Journal of Systems and Software*, 171:110825. 20

Ye, Q., Bai, G., Wang, K., and Dong, J. S. (2015). Formal analysis of a single sign-on protocol implementation for android. In *20th International Conference on Engineering of Complex Computer Systems, ICECCS 2015, Gold Coast, Australia, December 9-12, 2015*, pages 90–99. IEEE Computer Society. 15

Zhang, M., Duan, Y., Yin, H., and Zhao, Z. (2014). Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 1105–1116. 17

Zhou, Y. and Evans, D. (2014). Ssoscan: automated testing of web applications for single sign-on vulnerabilities. In *23rd USENIX Security Symposium*, pages 495–510. 2, 45, 46

Zuo, C., Zhao, Q., and Lin, Z. (2017). Authscope: Towards automatic discovery of vulnerable authorizations in online services. In *Acm Sigsac Conference*. 15