AuthSaber: Automated Safety Verification of OpenID Connect Programs

Tamjid Al Rahat
University of California, Los Angeles
Los Angeles, California, United States
tamjid@ucla.edu

Yu Feng
University of California, Santa Barbara
Santa Barbara, California, United States
yufeng@cs.ucsb.edu

Yuan Tian
University of California, Los Angeles
Los Angeles, California, United States
yuant@ucla.edu

Abstract

Single Sign-On (SSO)-based authentication protocols, like OpenID Connect (OIDC), play a crucial role in enhancing security and privacy in today's interconnected digital world, gaining widespread adoption among the majority of prominent authentication service providers. These protocols establish a structured framework for verifying and authenticating the identities of individuals, organizations, and devices, while avoiding the necessity of sharing sensitive credentials (e.g., passwords) with external entities. However, the security guarantees of these protocols rely on their proper implementation, and real-world implementations can, and indeed often do, contain logical programming errors leading to severe attacks, including authentication bypass and user account takeover. In response to this challenge, we present *AuthSaber*, an automated verifier designed to assess the real-world OIDC protocol implementations against their standard safety specifications in a scalable manner. *AuthSaber* addresses the challenges of expressiveness for OIDC properties, modeling multi-party interactions, and automation by first designing a novel specification language based on linear temporal logic, leveraging an automaton-based approach to constrain the space of possible interactions between OIDC entities, and incorporating several domain-specific transformations to obtain programs and properties that can be directly reasoned about by software model checkers. We evaluate *AuthSaber* on the 15 most popular and widely used OIDC libraries and discover 16 previously unknown vulnerabilities, all of which are responsively disclosed to the developers. Five categories of these vulnerabilities also led to new CVEs.

CCS Concepts

• **Security and privacy** → **Web application security**; **Logic and verification**; **Security requirements**; • **Theory of computation** → **Program verification**; **Verification by model checking**.

Keywords

OpenID Connect security; single sign-on; safety verification; automated analysis; authorization; authentication;

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA.

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3670318

ACM Reference Format:
Tamjid Al Rahat, Yu Feng, and Yuan Tian. 2024. *AuthSaber*: Automated Safety Verification of OpenID Connect Programs. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3658644.3670318

1 Introduction

Single Sign-On (SSO) protocols such as OpenID Connect (OIDC) [41] have gained widespread adoption across various industries [27, 39] 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. Moreover, 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. Security guarantees of these protocols are only as strong as the implementation that enforces them. Unfortunately, the current security of the protocol holds mostly at the specification level [41], whereas real protocol implementations in the wild often have logical errors leading to severe attacks [18, 37, 45]. Recent years have witnessed a growing number of security attacks across the web stemming from these implementation errors. For instance, in 2023, a bug [38] 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 [24] 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 [17]. 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 [26, 46, 47] 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 [22, 25] 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 [36] or Alloy [29]. 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 [41] is written in dozens of pages in plain English, and incorporates more than ten other protocols (e.g., OAuth [48], JWT [32], 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 [46, 47] 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 [5, 26, 46] or the providers [47]), 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 [15, 34]. Therefore, devising an automated approach that accommodates complex protocol features implemented in various programming languages is both critical and challenging.

Our approach. 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 [15]. For OIDC implementations, we first compile them to WalaIR [53] and then perform a transpilation from WalaIR to Boogie IR [7], 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[1]). Additionally, *AuthSaber* also effectively verified the correctness of patched versions addressing the known violations previously reported.

Contributions. In summary, we make the following contributions:

- We design a domain-specific specification language AuthLTL that enables developers to express complex safety properties of OIDC protocol, including the temporal properties that are originally described in plain English.
- We design a specialized automated verification approach tailored for the analysis of real-world OIDC programs. Our approach adeptly models the multi-party interactions inherent in the OIDC protocol, thereby mitigating the complexity of the verification problem. Additionally, it transforms safety

[1] "The record creation date may reflect when the CVE ID was allocated or reserved and does not necessarily indicate when this vulnerability was discovered"– www.cve.mitre.org

properties into standard LTL, facilitating further analysis by software model checkers.
- We implement the proposed concept in a tool named *AuthSaber*, and evaluate its performance across the 15 most popular and widely adopted OIDC libraries. We identify 16 previously unknown violations, all of which are reported to the developers for responsive disclosure. Five categories of these violations also resulted in new CVEs.

2 Background

We first provide a background of single sign-on (SSO) service, followed by its most popular representative OpenID Connect (OIDC) protocol. To illustrate, we present an example of one of the authentication flows supported by OIDC.

2.1 Single Sign-On (SSO)

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 [41], OAuth [48], and SAML [49]. 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.

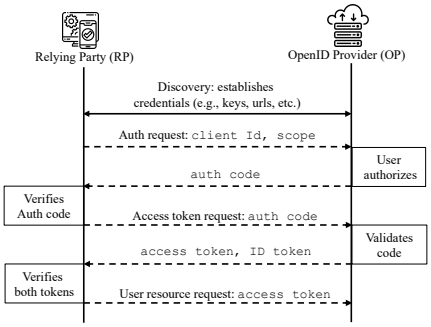

Figure 1: Interactions between the Relying Party (RP) and OpenID Provider (OP) during the authentication process of *Authorization Code Flow* in OIDC.

2.2 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 [41] 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) [32] 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 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 1 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.

3 Overview

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

3.1 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 [41] 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 2: Code example of an OIDC relying party validating the `id token` received as a response from the provider during the authentication flow. When validating the `id token` (line 11), it accepts 'none' as a valid algorithm value for the token, which enforces the verify function (line 19) to skip the signature verification for the token. This enables attackers to bypass the signature verification by injecting a maliciously crafted `id token` with an altered algorithm value in its unprotected header.

3.2 Running example

Figure 2 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 [43], 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 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 [32] library to verify the signature component (line 16-24) of the token.

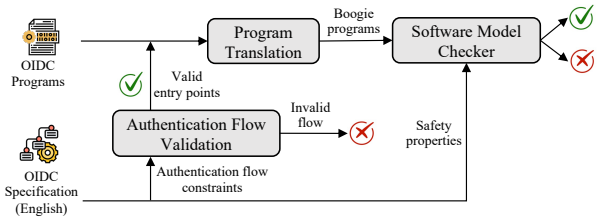

Figure 3: Schematic workflow of *AuthSaber*.

Safety Properties. When processing Id Tokens, an important safety property is: ($_1$) '*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 $_1$. 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: ($_2$) '*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 2 violates the property $_2$, 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 3 illustrates the schematic workflow of our safety verification approach. 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` ($_1$) can be expressed as follows in AuthLTL:

$$(\texttt{response}(s, arg(\text{access\_token}) = t \wedge arg(\text{id\_token}) = i) \Rightarrow \neg\, \texttt{request}(c, arg(\text{access\_token}) = t)\ \mathcal{U}\ \texttt{verifyToken}(i, ret = \text{true}))$$

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 [32] 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 ($_2$) can be expressed in AuthLTL as follows:

$$(\texttt{response}(s, arg(\mathsf{access\_token}) = t \wedge arg(\mathsf{id\_token}) = i \wedge i.alg = \mathsf{none}) \Rightarrow \neg\ \texttt{request}(c, arg(\mathsf{access\_token}) = t))$$

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 [15]. 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 [33] – 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.

4 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.

4.1 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 4 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.

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

Figure 4: 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 [32] 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 ( ), eventually ( ), next ($\bigcirc$) and until ($\mathcal{U}$). φ signifies a universal assertion, indicating that condition φ holds true at all steps during the program execution. Conversely, φ 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 1. *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:*

$$\varphi \quad := \quad (\mathsf{request}(s, arg(\texttt{response\_type}) = \texttt{code} \wedge \mathit{session} = u) \Rightarrow \neg\, \mathsf{response}(c, arg(\texttt{code}) = \mathit{gencode}(u, \_)) \;\mathcal{U}\; \mathsf{authUser}(u, \mathit{ret} = \texttt{true}))$$

4.2 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 [33]. Specifically, we begin by compiling the OIDC programs to WalaIR [53], 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 2. *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.

4.3 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 4.1). 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

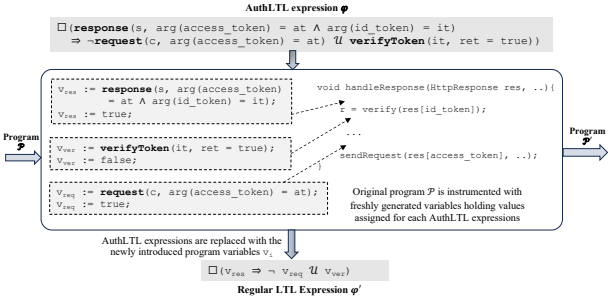

Figure 5: Transformation of AuthLTL formula φ to its equisatisfiable regular LTL φ' while transforming the original program $\mathcal{P}$ to $\mathcal{P}'$.

equisatisfiable in that $\mathcal{P}'$ satisfies φ' if and only if $\mathcal{P}$ satisfies φ. Intuitively, this transformation 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 3. Figure 5 illustrates the above-mentioned transformation using an AuthLTL formula discussed in Section 3. 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 request$(_, \phi)$ predicate, we assign true the corresponding v_{req} variable if the condition ϕ holds right *before* the request is sent. Similarly, since 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 verifyToken(t, ϕ) 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.

4.4 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.

4.4.1 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.

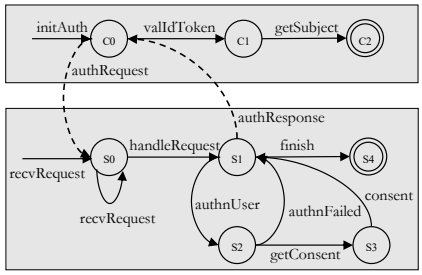

Figure 6: Automata representation of the authentication flow constraints for the OIDC *implicit flow* defined (in English) by the specification.

Key observation. In addition to safety properties that can be expressed in our language in Section 4.1, the OIDC specification [41] 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

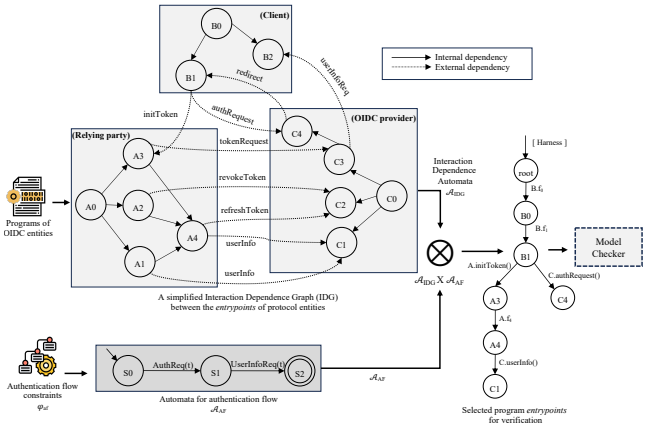

Figure 7: Authentication flow validation approach implemented by *AuthSaber* which first models the interaction between the *entrypoints* of each participating OIDC entities as IDG. The IDG model is then validated against the flow constraints defined by the standard OIDC specification and the resulting (valid) endpoints are then processed for safety verification using model checking.

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 6 shows a finite state machine that corresponds to the flow constraint at the beginning of Section 4.4.1. 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 7 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 [1] to preserve a subset of external dependencies that are consistent with the flow constraints ϕ_{af}. As shown in Figure 7, *AuthSaber* first transforms the IDG to a finite automaton $\mathcal{A}_{IDG}$ $(Q_i, \ \ _i, q_{0i}, F_i, \delta_i)$, where states Q_g corresponds to the nodes of IDG and $\ \ _i$ includes the entry functions associated with the graph nodes. Transition function $\delta_i : Q_i \times \ \ _i \rightarrow Q_i$ corresponds to the external and internal edges (E_{in} and E_{ex}) of IDG. Finally, $q_{0i} \in Q_i$ 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 [1]) 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}.

4.4.2 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 [15]. 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 $\mathcal{A} = (\ \ , Q, q_0, \rightarrow, \mathcal{F})$ is a finite-state automaton which consists a finite alphabet $\ \ $, a finite set of states Q, an initial state $q_0 \in Q$, a transition relation $\rightarrow$ which is a function $Q \times \ \ \rightarrow Q$, and a set of accepting states $\mathcal{F} \subseteq Q$. A *word* is an infinite sequence $w = e_0 e_1 \ldots$ such that $e_i \in \ \ $ 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 [6]. 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 $\delta_{\mathcal{P}} : \mathcal{L} \times \mathcal{S} \rightarrow \mathcal{L}$. On the other hand, $\mathcal{A}_{\varphi} = (\ , Q, q_0, \rightarrow, \mathcal{F})$ 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 $\mathcal{B} = (\ _{\mathcal{B}}, Q_{\mathcal{B}}, q_{0\,\mathcal{B}}, \delta_{\mathcal{B}}, \mathcal{F}_{\mathcal{B}})$ such that $_{\mathcal{B}}$ 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.,

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

$Q_{\mathcal{B}}$ is the cartesian product of program locations and Büchi automaton states (i.e., $\mathcal{L} \times Q$), 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,

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

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 $\mathcal{B}$.

4.5 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) [53] 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 [33] programs. Finally, the CEGAR framework of our model checking is built upon the UltimateAutomizer model checker tool [15].

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 [41], 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 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?

5.1 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.

Selection of properties. We define the safety properties based on the standard specifications outlined in OpenID Connect Core 1.0 [41] and established security best practices [28]. The scope of

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

Table 1: OIDC components and relevant details that we consider to gather the safety properties for our evaluation. Here, we consider the components required for all three possible authentication paths in OIDC specification [41]: (1) authorization code flow (AC), (2) implicit flow (I), and (3) hybrid flow (H). These components are supported by the relying party (RP) and OIDC provider (OP) to perform the authentication flow.

these properties is illustrated in Table 1. Additionally, Table 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, (2) implicit flow, and (3) hybrid flow. 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 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 [42] 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.

5.2 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.

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 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 benchmarks 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, *AuthSaber* successfully verifies 22 out of 25 safe benchmarks and incorrectly flags (i.e., false positive) three benchmarks.

#No.	Description of safety properties for the component of 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 2: Properties within the scope of ID Token validation (*OpenID ref. §2, §3.1.3.7, §3.2.2.11*).

OpenID libraries	Language	Popularity (#github stars)	#LOC
1) Keycloak	Java	15.2k	38.2k
2) Spring Security	Java	7.5k	19.5k
3) Connect2id	Java	N/A	23.2k
4) Mitre OpenID	Java	1.4k	21.3k
5) Google Client	Java	600	4.7k
6) Pac4j	Java	2.2k	29.8k
7) Oracle Cordova	Java	10	7.5k
8) AppAuth	Java	2.4k	8.8k
9) OxAuth	Java	400	34.4k
10) Quarkus	Java	11.4k	7.8k
11) Auth0	Java	257	23.1k
12) Node OIDC	JS	2.6k	13.6k
13) AppAuth-JS	JS	912	5.2k
14) Auth0 Express	JS	360	2.7k
15) Asgardeo	JS	25	3.6k

Table 3: Popular OIDC libraries selected for evaluating *AuthSaber*. Among these, eleven libraries are implemented in Java and four are implemented in JS.

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 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.

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

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: 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.

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 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 [47] and S3KVetter [26], 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 [48] 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 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. 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 [15], 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 4, 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.

5.3 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 paper, 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.

5.3.1 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 [23], 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 [3] 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 [41]. 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.*"

6 Related work

SSO verification. Recent years have seen several efforts to verify the security of the design, 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 [20–22] to prove authorization and session integrity properties against the model. Additionally, Lu et al. [35] use applied PI calculus to model OIDC protocol and provide security analysis using ProVerif [9]. Similarly, Hamman et al. [25] proves privacy properties in OIDC model using Tamarin prover [36]. 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. In 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.

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 [54] uses symbolic execution to check security properties in OAuth client's SDK. In addition, Oauthlint [46] performs static data-flow analysis to check six OAuth's anti-protocol patterns in Android applications, and Cerberus [47] also uses static analysis to check the security of OAuth service providers. 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. Moreover, WPSE [10] and Bulwark [52] study network-traffic-based security monitoring systems for different SSO entities (e.g., browsers, web apps, etc.). However, network traffic analysis requires heavy manual setup and cannot be applied or extended to uncover all possible execution behavior SSO protocols.

CEGAR-based verification. Our verification technique is based on the counterexample-guided abstraction refinement (CEGAR) [11], which has been used to provide formal security guarantees in many problem domains, including blockchain security [44, 51], cryptography [40], IoT protocols [2, 50] and autonomous systems [16, 31]. However, these tools are designed for specific domains and cannot be used or extended to check complex multi-party protocols like OIDC.

Automated protocol analysis. Automated protocol analysis has also been the subject of extensive security research, resulting in a wide range of works in recent years. Automated tools like Tamarin prover [36] and Proverif [9] have been used to prove properties in various protocols, including TLS [13, 14], 5G [8, 12] and single-party authentication [4, 19, 30]. 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.

7 Conclusion

In this paper, 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.

8 Acknowledgements

We are grateful to the anonymous reviewers for their insightful and constructive feedback and suggestions. This work is supported in part by *National Science Foundation* under the award numbers 2320903, 2323105, 2325369, 2317184, and 1908494, by *DARPA* under the agreement number N66001-22-2-4037, by *Google Faculty Research*, and *Ethereum Foundation* awards. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the funding agencies.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley. https://www.worldcat.org/oclc/12285707

[2] Mahmoud Ammar, Bruno Crispo, Bart Jacobs, Danny Hughes, and Wilfried Daniels. 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 (2019), 885–901.

[3] AppAuth. 2023. AppAuth-Android. https://openid.github.io/AppAuth-Android/.

[4] Linard Arquint, Felix A Wolf, Joseph Lallemand, Ralf Sasse, Christoph Sprenger, Sven N Wiesner, David Basin, and Peter Müller. 2023. Sound verification of security protocols: From design to interoperable implementations. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1077–1093.

[5] Guangdong Bai, Jike Lei, Guozhu Meng, Sai Sathyanarayan Venkatraman, Prateek Saxena, Jun Sun, Yang Liu, and Jin Song Dong. 2013. Authscan: Automatic extraction of web authentication protocols from implementations. (2013).

[6] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking*. MIT press.

[7] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. 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*. Springer, 364–387.

[8] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. 2018. A formal analysis of 5G authentication. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 1383–1396.

[9] Bruno Blanchet. 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* (2014), 54–87.

[10] Stefano Calzavara, Riccardo Focardi, Matteo Maffei, Clara Schneidewind, Marco Squarcina, and Mauro Tempesta. 2018. WPSE: fortifying web protocols via browser-side security monitoring. In *27th USENIX Security Symposium*. 1493–1510.

[11] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-guided abstraction refinement. In *Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings 12*. Springer, 154–169.

[12] Cas Cremers and Martin Dehnel-Wild. 2019. Component-based formal analysis of 5G-AKA: Channel assumptions and session confusion. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society.

[13] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. 2017. A comprehensive symbolic analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1773–1788.

[14] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. 2016. Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 470–485.

[15] Daniel Dietsch, Matthias Heizmann, Vincent Langenfeld, and Andreas Podelski. 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*. Springer, 49–66.

[16] Yizhak Yisrael Elboher, Justin Gottschlich, and Guy Katz. 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*. Springer, 43–65.

[17] Expo. 2023. Critical OAuth Vulnerability in Expo Framework Allows Account Hijacking. https://thehackernews.com/2023/05/critical-oauth-vulnerability-in-expo.html.

[18] Facebook. 2018. Facebook Security Update. https://about.fb.com/news/2018/09/security-update.

[19] Haonan Feng, Hui Li, Xuesong Pan, Ziming Zhao, and T Cactilab. 2021. A Formal Analysis of the FIDO UAF Protocol.. In *NDSS*.

[20] Daniel Fett, Pedram Hosseyni, and Ralf Küsters. 2019. An extensive formal security analysis of the openid financial-grade api. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 453–471.

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

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

[23] Google. 2023. Google Oauth Client. https://cloud.google.com/java/docs/reference/google-oauth-client/latest/overview.

[24] Grammarly. 2023. Critical OAuth Flaws Uncovered in Grammarly, Vidio, and Bukalapak Platforms. https://thehackernews.com/2023/10/critical-oauth-flaws-uncovered-in.html.

[25] Sven Hammann, Ralf Sasse, and David Basin. 2020. Privacy-preserving openid connect. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 277–289.

[26] Boyuan He, Vaibhav Rastogi, Yinzhi Cao, Yan Chen, VN Venkatakrishnan, Runqing Yang, and Zhenrui Zhang. 2015. Vetting SSL usage in applications with SSLint. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 519–534.

[27] Fatima Hussain, Rasheed Hussain, Brett Noye, and Salah Sharieh. 2020. Enterprise API security and GDPR compliance: Design and implementation perspective. *IT Professional* 22, 5 (2020), 81–89.

[28] IETF. 2021. OAuth 2.0 Security Best Current Practice. https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics.

[29] Daniel Jackson. 2012. *Software Abstractions: logic, language, and analysis*. MIT press.

[30] Charlie Jacomme, Elise Klein, Steve Kremer, and Maïwenn Racouchot. 2023. A comprehensive, formal and automated analysis of the EDHOC protocol. In *USENIX Security'23-32nd USENIX Security Symposium*.

[31] Peng Jin, Jiaxu Tian, Dapeng Zhi, Xuejun Wen, and Min Zhang. 2022. Trainify: A cegar-driven training and verification framework for safe deep reinforcement learning. In *International Conference on Computer Aided Verification*. Springer, 193–218.

[32] JWT. 2015. Json Web Token. https://datatracker.ietf.org/doc/html/rfc7519.

[33] K Rustan M Leino. 2008. This is boogie 2. *manuscript KRML* 178, 131 (2008), 9.

[34] K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*. Springer, 348–370.

[35] Jintian Lu, Jinli Zhang, Jing Li, Zhongyu Wan, and Bo Meng. 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*. Springer, 209–220.

[36] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. 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*. Springer, 696–701.

[37] Microsoft. 2023. Azure B2C – Crypto Misuse and Account Compromise. https://securityboulevard.com.

[38] Microsoft. 2023. 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.

[39] Nitin Naik and Paul Jenkins. 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)*. IEEE, 163–174.

[40] Saeed Nejati, Jia Hui Liang, Catherine Gebotys, Krzysztof Czarnecki, and Vijay Ganesh. 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*. Springer, 120–131.

[41] OpenID. 2021. OpenID Connect Core 1.0. https://openid.net/specs/openid-connect-core-1_0.html.

[42] OpenID. 2023. OpenID Certification. https://openid.net/certification/.

[43] Pac4j. 2023. Pac4J. https://www.pac4j.org.

[44] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. Verx: Safety verification of smart contracts. In *2020 IEEE symposium on security and privacy (SP)*. IEEE, 1661–1677.

[45] Portswigger. 2023. Authentication bug that enabled unauthorized access to client applications. https://portswigger.net.

[46] Tamjid Al Rahat, Yu Feng, and Yuan Tian. 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*. 293–304.

[47] Tamjid Al Rahat, Yu Feng, and Yuan Tian. 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*. 2459–2473.

[48] RFC6749. 2021. The OAuth 2.0 Authorization Framework. https://tools.ietf.org/html/rfc6750.

[49] SAML. 2023. SAML protocol. http://saml.xml.org/saml-specifications.

[50] Alireza Souri and Monire Norouzi. 2019. A state-of-the-art survey on formal verification of the internet of things applications. *Journal of Service Science Research* 11, 1 (2019), 47–67.

[51] Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu Lahiri, and Isil Dillig. 2021. SmartPulse: automated checking of temporal properties in smart contracts. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 555–571.

[52] Lorenzo Veronese, Stefano Calzavara, and Luca Compagna. 2020. Bulwark: Holistic and Verified Security Monitoring of Web Protocols. In *European Symposium on Research in Computer Security*. Springer, 23–41.

[53] WALA. 2023. T.J. Watson Libraries for Analysis (WALA). https://sourceforge.net/projects/wala.

[54] Ronghai Yang, Wing Cheong Lau, Jiongyi Chen, and Kehuan Zhang. 2018. Vetting Single Sign-On SDK Implementations via Symbolic Reasoning. In *27th USENIX Security Symposium*. 1459–1474.