Automated Repair of OpenID Connect Programs

Tamjid Al Rahat*, Yanju Chen†, Yu Feng‡, Yuan Tian*

*University of California, Los Angeles, tamjid@ucla.edu, yuant@ucla.edu
†University of California, San Diego, yanju@ucsd.edu
‡University of California, Santa Barbara, yufeng@cs.ucsb.edu

***Abstract*—OpenID Connect has revolutionized online authentication based on single sign-on (SSO) by providing a secure and convenient method for accessing multiple services with a single set of credentials. Despite its widespread adoption, critical security bugs in OpenID Connect have resulted in significant financial losses and security breaches, highlighting the need for robust mitigation strategies. Automated program repair presents a promising solution for generating candidate patches for OpenID implementations. However, challenges such as domain-specific complexities and the necessity for precise fault localization and patch verification must be addressed. We propose *AuthFix*, a counterexample-guided repair engine leveraging LLMs for automated OpenID bug fixing. *AuthFix* integrates three key components: fault localization, patch synthesis, and patch verification. By employing a novel Petri-net-based model checker, *AuthFix* ensures the correctness of patches by effectively modeling interactions. Our evaluation on a dataset of OpenID bugs demonstrates that *AuthFix* successfully generated correct patches for 17 out of 23 bugs (74%), with a high proportion of patches semantically equivalent to developer-written fixes.**

I. Introduction

Single-Sign-On (SSO) protocols like OpenID Connect are a cornerstone of modern online authentication, supporting billions of accounts and millions of applications across web and mobile platforms [8]. The OIDC ecosystem is broad and vibrant: widely used open-source projects such as ory/hydra [13], dexidp/dex [11], and oauth2-proxy/oauth2-proxy [14] form critical infrastructure, while major technology providers maintain official SDKs for developers [12], [10]. This level of adoption makes OIDC both indispensable and high impact, offering seamless login experiences for users, helping organizations manage access and compliance, and mitigating password-related security risks.

Due to its complexity, critical security bugs [44], [45] in OpenID Connect can lead to severe consequences, including financial losses and widespread breaches [6], [5], [2]. In 2022, a high-severity flaw (with a CVSS score of 8.7) was reported in Google's authentication flow [4], and in 2023, an OpenID-specific bug [7] in Microsoft's Azure authentication allowed attackers to forge tokens and compromise over two dozen organizations. These incidents illustrate the difficulty of timely repair, as even well-scoped protocol-level issues can remain unresolved for years. For example, the race condition issue [15] in OAuthLib has been open for *over six years*. Therefore, it is crucial to develop approaches that automate patch generation with formal assurance, ensuring vulnerabilities are fixed promptly and reliably.

A natural way to automate the above process is to leverage *program repair* [34], [31], [53], [52] for generating candidate patches of OpenID implementations. However, this approach faces several *challenges*. First, OpenID is a special domain with few studies in the verification and repair community, which makes it difficult to leverage prior heuristics and data-driven approaches for both fault localization and patch synthesis. Second, the OpenID domain is extremely complex in terms of specification and implementation, which goes beyond the capabilities of existing patch synthesis. For instance, OpenID specification involves hundreds of pages of documentation, making it difficult for developers to conform to all the logical requirements when fixing OpenID bugs. The implementation typically uses many complex APIs (e.g., cryptographic APIs) from external libraries. Finally, an incorrect fix can compromise the system's security. For instance, unsigned ID tokens are only supported in OpenID for low-power devices (e.g., IoT devices). Accepting unsigned tokens for regular clients like web applications may seem like a convenient shortcut for developers, but it can leave the system exposed to severe attacks like authentication bypass. Therefore, for each candidate patch, OpenID developers need oracles to verify correctness of the patch. While the OpenID Foundation [8] offers a certification process, it only evaluates the correctness of protocol endpoints, neglecting the actual implementations. This limitation leaves a critical gap in ensuring the overall functionality and reliability of the repaired programs. Therefore, it is of paramount importance to automate the repair of OpenID bugs while ensuring the correctness of the patches.

Our approach. To address the above-mentioned challenges, we propose *AuthFix*, a counterexample-guided repair engine powered by large language models (LLMs) [50], [30]. Specifically, *AuthFix* takes three inputs: a buggy OpenID program, a specification, and a domain-specific language (DSL), and then returns a fixed version that is guaranteed to conform to the specification. Internally, *AuthFix* is composed of three major components: *fault localization*, *patch synthesis*, and *patch verification*. To address the data sparsity problem due to our specialized domain (i.e., OpenID Connect), both fault localization and patch generation are guided by an LLM (e.g., ChatGPT). Motivated by prior work in component-based synthesis [21], [22], to address the large space during patch synthesis, *AuthFix* carefully decomposes the problem into two separate phases, namely, *sketch generation* and *sketch enumeration* where invalid partial patches are pruned early.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00187

Finally, to ensure the correctness of the candidate patch and the efficiency of the verification, *AuthFix* designs a novel Petri net based model checker, which effectively models the complex interactions among multiple components of an OpenID program. Here, the output of the verification algorithm will be used as the feedback to the *patch synthesis* module to prevent making similar mistakes.

We evaluate our proposed approach on a dataset of OpenID bugs that we compiled based on prior works [44], [45], [18] and other online platforms. 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 *AuthFix* generates correct patches for 17 OpenID bugs out of 23 that were found in popular OpenID libraries. In addition, our manual assessment shows that 71.4% of the generated patches are semantically equivalent to the manual patches written by OpenID developers.

In summary, this paper makes the following contributions:

- We design a counterexample-guided program repair approach augmented with LLMs to fix real-world bugs in OpenID Connect protocols.
- We utilize a novel Petri net based approach to validate the patches against the standard OpenID specification.
- We implement the proposed approach in a tool named *AuthFix*[1], and evaluate it over 23 benchmarks of real-world OpenID Connect bugs. Our evaluation shows that *AuthFix* can generate patches for 74% (17) of bugs that were discovered and reported in previous work.

II. Background

We provide a background of OpenID Connect (OIDC) protocol and explain the security impacts of OIDC bugs.

A. OpenID Connect

Among all single sign-on (SSO) protocols, OIDC is the most popular and supported by nearly all major identity service providers, including Google, Microsoft, and Amazon. OIDC is an authentication protocol [3] allowing users to authenticate themselves across different web and mobile applications. 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 JWT-formatted string [1] 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.

OIDC 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. Fig. 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.

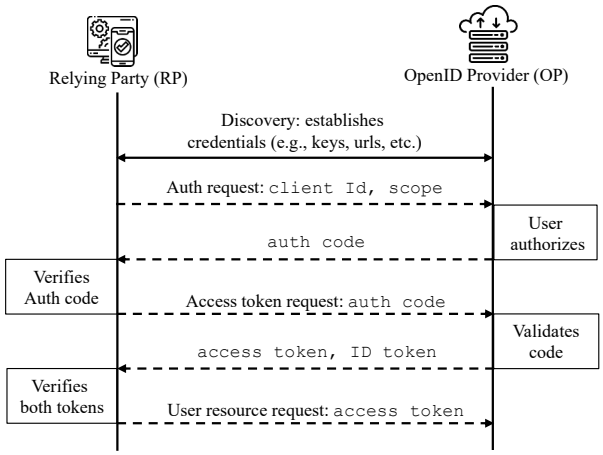

Fig. 1: Interactions between the Relying Party (RP) and OpenID Provider (OP) during the authentication process of *Authorization Code Flow* in OIDC.

B. Security Impacts of OpenID Bugs

OIDC is a widely used protocol for authentication and authorization in modern software. The prevalence of logical bugs in the OIDC implementations is due to several factors such as implementation mistakes, lack of adherence to the best practices, and complexities of the specification. Following are some of the most common categories of bugs in OIDC:

- Authorization Code Interception: The authorization code flow is susceptible to interception attacks if the code is processed without proper validation, or if the redirect URI is not adequately protected. Attackers can intercept the authorization code and use it to obtain access tokens.
- Cross-Site Request Forgery (CSRF): CSRF attacks occur when an attacker tricks a user into unknowingly executing actions on a website on which the user is authenticated. Incorrect use of the "*state*" parameter during the protocol execution can lead to unauthorized actions being performed using the authenticated session.
- Token Expiry and Revocation: Failure to enforce token expiration policies or properly handle token revocation can lead to unauthorized access. Tokens should be regularly refreshed, and revoked tokens should be invalidated promptly.
- Unauthorized Access to Account: If an attacker successfully injects a malicious token into the OIDC flow, they can gain unauthorized access to protected resources or user accounts. They might forge an access token with elevated privileges or modify an ID token to impersonate a legitimate user.

[1]The artifact is available at https://github.com/tamjidrahat/authfix.

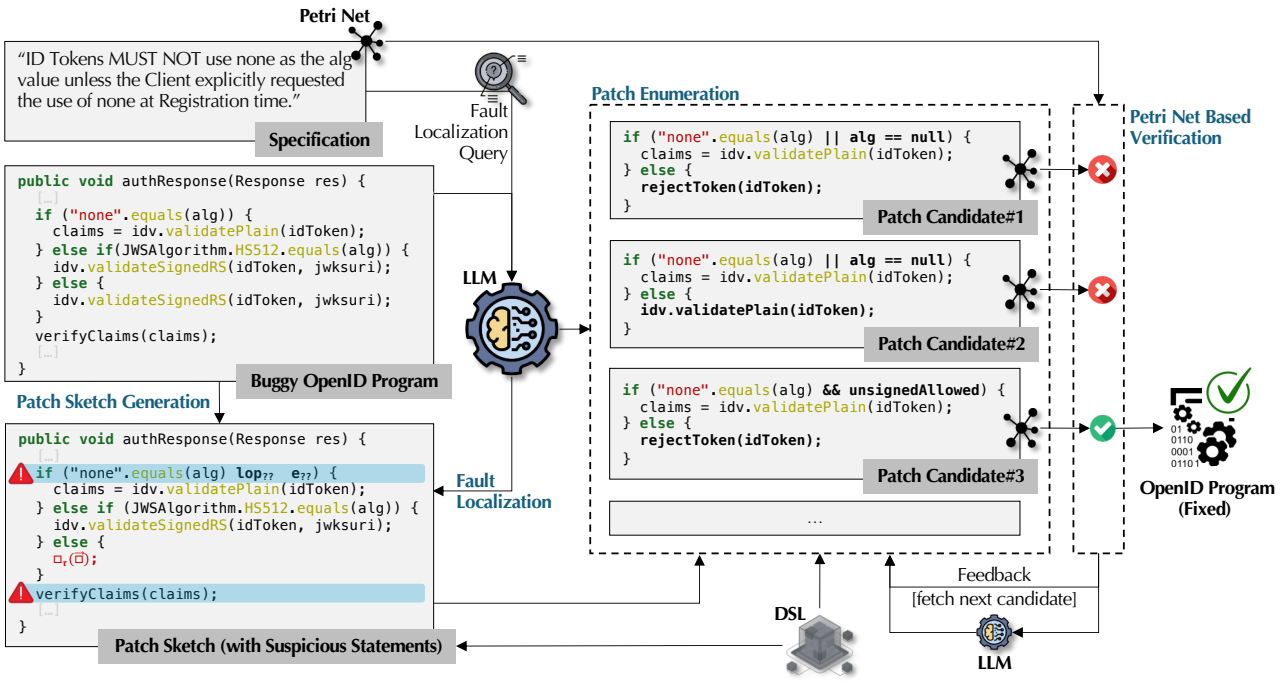

Fig. 2: Workflow of *AuthFix* on a concrete example. Here, *AuthFix* first takes a buggy OpenID program and its corresponding English specification as inputs and then generates repair sketches based on the fault localization by an LLM agent. The tool then performs patch enumeration while transforming the repair candidates into their corresponding Petri-net models, which are verified against the Petri-net specification constructed (manually) from the English specification.

III. Overview

Here, we present an overall workflow of *AuthFix* with a concrete example presented in Fig. 2. The example represents bugs inspired by a recently reported vulnerability (CVE-2021-44878) from Pac4j library which provides Java-based security framework for web applications. The library utilizes OIDC's ID token to provide secure authorization and authentication support for a wide variety of platforms. This example demonstrates a flaw in the ID token validation process, potentially allowing attackers to bypass signatures and inject malicious payloads, such as access token, into OpenID entities. Specifically, the library incorrectly handles the algorithm (*alg*) parameter when verifying the signature component of the ID token. Since the *alg* parameter is encoded in the header component and can be altered by the attacker, it can be leveraged to bypass the signature-based authentication. Despite being classified as a high-severity bug, developers took nearly a month to patch it due to the inherent complexity of correctly validating ID token according to the specifications.

The example includes a set of validity checks, including the validation of the signature and certain OpenID-specific claims (e.g., issuer, audience, etc.). Specifically, to perform the signature validation for ID token, this example utilizes the value of *alg* parameter in the header component of the token. Unfortunately, as shown in line 3, the code does not perform a check required by the OpenID specification [3], which states that "*ID tokens must not use* `none` *as the* `alg` *value unless it is explicitly requested by the client during the registration time*." Therefore, `validatePlain` method should not be invoked unless the boolean `unsignedTokenAllowed` in configuration is `true`. In addition, the code incorrectly invokes the `validateSignedRS` method (line 8) when the `alg` value does not match any of the expected values (e.g., HS256, RS256.) specified by the protocol. The specification requires rejecting the ID token in such a case. ID token validation is critical for OpenID as it is the cornerstone of authentication in OIDC. 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 would lead to unauthorized access or even user account takeover by attackers. For example, in the context of the above code example, using `none` as `alg` value forces the protocol to *skip the signature validation*, attackers can alter and inject an ID token with malicious claims and bypass the signature validation. However, automatically fixing this buggy code is highly non-trivial as it requires a comprehensive understanding of the OpenID specification and exploring a large space of candidate patches that involve multiple locations. Consequently, it takes months for developers to fix such highly sensitive bugs. For instance, a vulnerability (CVE-2021-22573) in the ID token validation of Google's authorization library remained unpatched for nearly four months, leaving both the library and the applications that depended on it exposed to risk for an extended period.

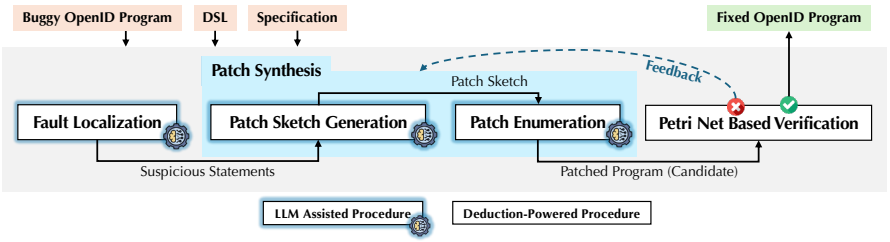

Fig. 3: An overview of *AuthFix*.

Motivation and our solution. Repairing OpenID Connect programs requires addressing both the complexity of the specification and the semantic correctness of fixes. OIDC protocols involve concurrent interactions across multiple parties, with correctness conditions spanning sequential steps and synchronization points. Test-based or heuristic repair cannot ensure compliance in this setting, since many bugs arise from protocol-level violations rather than local code errors. A formal model is therefore needed to capture both sequencing and concurrency in authentication flows.

AuthFix builds on a Petri-net–based model of OIDC, which represents protocol states, concurrent transitions, and forbidden behaviors specified in the standard, enabling validation of candidate repairs against the full specification. On this basis, AuthFix follows a counterexample-guided inductive synthesis (CEGIS) [24], [36] paradigm, where candidate patches are generated, verified against the Petri-net model, and iteratively refined with counterexamples until compliance is achieved. Concretely, the workflow contains the following steps:

- *Preparation*. The specification of security-sensitive OIDC components (e.g., authorization flow, ID token verification) is first obtained from the official standard [3] and then manually transformed into an equivalent Petri-net representation, enabling modeling of the OIDC flow together with its concurrent requirements that must hold during protocol execution. During the repair process, the buggy program and its corresponding specification are embedded into an LLM query to identify suspicious statements.
- *Sketch Generation*. Based on these statements, a set of schemas is applied to generate repair sketches containing partial expressions represented as holes (denoted by $\square$). For example, in Fig. 2, line 3 of the buggy program is transformed into the *partial* expression `none.equals(alg)` $\square_\sigma$ $\square$, where $\square_\sigma$ denotes a hole for logical operators and $\square$ denotes a hole for expressions. Similarly, the function call in line 8 is transformed into $\square_\tau(\vec{\square})$, which includes a hole for a method $\square_\tau$ of type τ and its arguments $\vec{\square}$.
- *Patch Synthesis*. Guided by a domain-specific grammar optimized by an LLM agent, AuthFix then generates concrete repair candidates. For instance, the operator hole $\square_\sigma$ may be instantiated as $\&\&$ or $||$, producing multiple candidates.
- *Verification*. Each candidate is translated into an equivalent Petri net and validated against the specification-derived Petri net. This process iterates until a valid patch is found or the search space is exhausted.
- *Counterexample-Guided Refinement*. When verification fails, the counterexample trace provided by the Petri-net checker is fed back into the synthesis loop, guiding the LLM to refine subsequent candidates and converge toward a specification-compliant repair.

Together, these stages constitute a CEGIS-style repair framework that ensures specification-compliant fixes, and the following section elaborates on each component in detail.

IV. Methodology

Fig. 3 presents the overall architecture of *AuthFix*. Given a buggy OpenID program, a specification, and a DSL, *AuthFix* executes three main procedures: fault localization, patch synthesis (including sketch and candidate enumeration), and Petri-net–based model checking, producing a repaired version certified by the verifier. In the first step, suspicious statements are identified through fault localization, from which a *sketch*—a partial program with unfilled holes—is derived. These holes are then completed through best-first enumeration to construct concrete repair candidates. To ensure correctness, *AuthFix* translates each candidate into a Petri net and validates it against the specification using the verification procedure. Verification yields two outcomes: (1) the patched program is accepted if it conforms to the specification; or (2) otherwise, the procedure returns feedback (e.g., counterexamples) that exposes the root cause of inconsistency, prompting patch synthesis to generate new candidates until a verified repair is found or the search times out.

To avoid ad-hoc heuristics that are hard to generalize across benchmarks, *AuthFix* interacts with large language models (LLMs) for prioritizing choices in fault localization and patch synthesis. For example, suspicious statements are labeled by LLMs during the invocation of the fault localization procedure, and patches are ranked within the patch synthesis procedure before they get checked by the verification procedure.

In what follows, we elaborate on different components of *AuthFix*. Because patch verification is one of our major contributions, we defer its detailed discussion to Section V.

A. Fault Localization

To ensure the correctness of the patch, existing fault localization methods [16], [17], [42] rely on a comprehensive set of test cases, which are not available and hard to construct

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

```
public void validate() {
  if (idToken.getAlg() == "HS256") {
    idToken.verify(idToken.getAlg(), config.client_secret);
  } else if (idtoken.getAlg() == "none") {
    idToken.verify(idToken.getAlg(), null);
  }
  if (idToken.getAlg() != config.expected_alg) {
    sendError();
  }
  if (idToken.getAud() != null && idToken.getAud() != config.client_id) {
    sendError();
  }
  idToken.validateClaims();
}
```

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).

Fig. 4: An example of an LLM query for fault 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.

for most OpenID implementations. To mitigate this limitation, *AuthFix* utilizes an LLM agent, which contains a *query process* and a *response process* modules, to locate suspicious statements that might cause the bug.

1) Construction of LLM Query: Our fault localization approach begins with a text-formatted query Q to the LLM agent. An LLM query Q is a sequence of text-based components $Q = [q_0, q_1, \ldots, q_n]$, where each component q_i is one of the following:

- *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.
- *Dynamic query component*. Dynamic query components may differ across the benchmarks, i.e., there may exist benchmark program p and p' for which $q_i(p) \neq q_i(p')$.

In particular, a dynamic component is one of the following:

- *Buggy code*. This component includes the body of the function containing the OpenID bugs.
- *Specification*. This component describes the expected behavior of the buggy code extracted from the standard OpenID Connect specification [3], which is written in plain English.

2) Processing of Query Response: The LLM queries produce output in JSON format, as specified in the query. Fig. 5 illustrates an example of the output generated for the LLM query in Fig. 4. The generated output consists of a JSON object containing an array of fault localization results, where each object provides detailed information about the bug, including its specific location (line number), the code content of the

```
{"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."
  }
]}
```

Fig. 5: An LLM response for the query described in Fig. 4.

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 (denoted by as $\square$, or *holes*) that need to be filled in with concrete repair expressions.

B. Patch Synthesis

To generate potential repair candidates, we reduce the problem of program repair to program synthesis by transforming the buggy program into the patched version. To avoid state explosion in synthesis, Depending on the type of suspicious statements, we carefully design a set of schemas to generate the sketches. In the following, we elaborate on the details.

1) Schemas for Generating Repair Sketch: *AuthFix* applies the following set of schemas to generate program sketches for repairing based on the identified bug locations in the input program.

a) **Conditional statement:** Given a buggy conditional statement e, *AuthFix* applies the following schemas *in the presented order* for sketch generation:

- **(S1)** transforms any arithmetic, relational or logical operator σ into an operator hole $\square_\sigma$ of the same type.
- **(S2)** transforms any variable, constant or field dereference of type τ into an expression hole $\square_\tau$ of the same type.
- **(S3)** introduces a new top-level computation for e, yielding $e\ \square_\sigma\ \square$, where $\square_\sigma$ denotes a logical operator and $\square$ corresponds to an expression of compatible type.

b) **Method invocation:** Given a buggy method invocation (e.g., $m(\ldots)$), *AuthFix* applies the following set of schemas for sketch generation:

- **(S4)** transforms the object reference o of type τ with an expression hole $\square_\tau$.
- **(S5)** replaces the argument expression e of type τ with an expression hole $\square_\tau$.
- **(S6)** transforms the method invocation $m(\vec{a})$ of type τ into $\square_\tau(\vec{\square})$ where $\square_\tau$ corresponds to methods of the same type and $\vec{\square}$ corresponds to its arguments.
- **(S7)** adds a *conditional guard* $\square_{\tau=\mathsf{bool}}$ to the method invocation $m(\ldots)$, yielding $\mathsf{if}\ \square_{\tau=\mathsf{bool}}\ \mathsf{then}\ m(\ldots)$, which enables additional pre-condition check before invocation.

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

$\mathcal{C} \in$ **OpenID Constants** $\quad \mathcal{F} \in$ **OpenID Functions**

Fig. 6: A domain-specific language for repair synthesis.

c) **Return statement (S8):** Given a buggy return statement e of type τ, *AuthFix* transforms it into $\square_\tau$, an expression hole of the same type.

2) Synthesis of Repair Expression: *AuthFix* leverages 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 near location l, relevant expressions used for repair are determined by: (a) relevant program elements in scope at l and (b) 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. The relevant classes extracted are based on the classes that are instantiated, whose fields or methods are accessed within the buggy method.

In addition to the program elements within the scope, *AuthFix* further extracts all the OpenID-specific elements that can be accessed within the context of the underlying OpenID *authentication flow* for the buggy program. For instance, if the buggy program is part of an authentication flow such as implicit flow or hybrid flow of OpenID Connect, we 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.

a) Syntax of repair expressions: Fig. 6 illustrates the grammar 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.

The grammar of the repair expressions is then used by *AuthFix* to instantiate a program transformation at each location

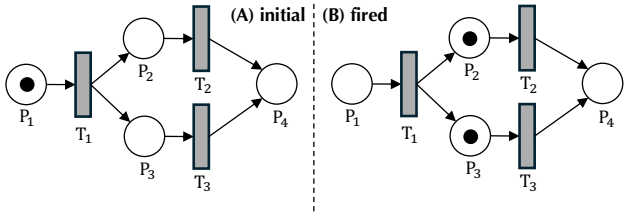

Fig. 7: (A) A Petri net with four places (circles) and two transitions (rectangles); (B) The same Petri net after firing T_1.

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.

b) LLM-driven search space pruning and prioritization: Using the grammar in Fig. 6 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 [26], 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.

V. Petri-Net–Based Verification

In this section, we introduce the Petri net based verification framework of *AuthFix*. The goal is to formally check whether a candidate patch conforms to the OpenID specification by modeling both the program and the specification as *guarded Petri nets* and analyzing their reachable states. We first review the basics of Petri nets and reachability (Section V-A), then extend them with guarded transitions (Section V-B) and show how to construct nets from both code (Section V-C) and specification (Section V-D), followed by the validation procedure (Section V-E) and the use of counterexamples to refine patches (Section V-F).

A. Petri Nets and Reachability

A Petri net is a bipartite graph $\langle P, T, A, M_0 \rangle$: places P hold *tokens*, transitions T move tokens along arcs A, and M_0 is the

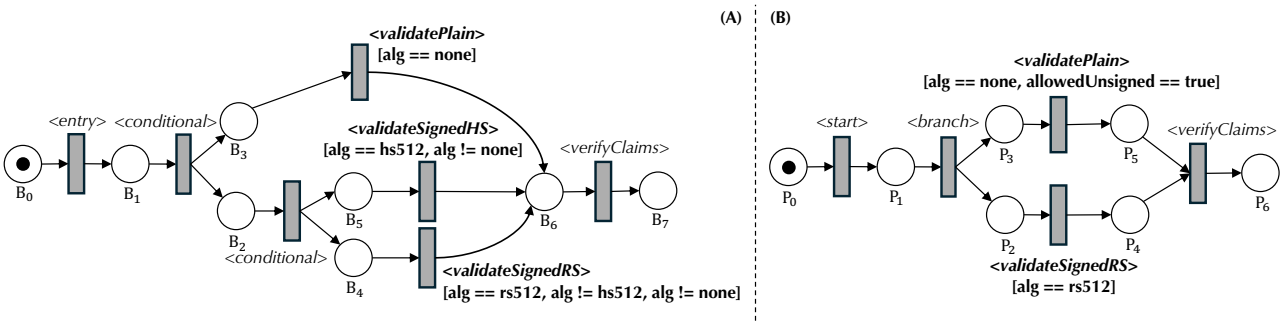

Fig. 8: (A) A Petri net example (simplified) for the example in Fig. 2, and (B) Petri net specification provided by users.

initial marking. A transition fires when every input place has at least one token, after which it consumes those tokens and produces its outputs.

A marking $M \in R(M_0)$ is *reachable* from M_0 if there exists a transition sequence $\sigma \in T^*$ such that $M_0 \xrightarrow{\sigma} M$, where $R(M_0)$ denotes the *reachable set*.

Example 1. Fig. 7(A) starts with $M_0 = (1\,0\,0\,0)$. Firing T_1 yields Fig. 7(B), where $M_1 = (0\,1\,1\,0)$, and then T_2 yields $M_2 = (0\,0\,0\,1)$. Here, we have $M_1, M_2 \in R(M_0)$.

B. Petri Net with Guarded Transitions

We extend the Petri net described above with guarded conditions, which allows imposing 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.).

C. Construction of Petri Nets from Codes

Given a method's control-flow graph $\mathsf{CFG} = \langle N, E \rangle$ we build a guarded Petri net $\mathcal{N}_P = \langle P, T, A, M_0 \rangle$ as follows:

- **Places** P**.** Create one place p_n for every basic block $n \in N$; a token in p_n denotes that the program counter is currently in that block.
- **Transitions** T**.** For each conditional edge $(n_i, n_j) \in E$ add a branch transition $t_{i \to j}$ guarded by the edge predicate (e.g., `alg==none`). For every call site generate a transition labelled with the callee name; its guard is `true` unless a value condition is present (e.g., `role.equals("admin")`). Two distinguished transitions t_{entry} and t_{exit} mark the beginning and end of the method so that inter-procedural composition is possible.
- **Arcs** A**.** Insert an arc $p_{n_i} \to t_{i \to j}$ and an arc $t_{i \to j} \to p_{n_j}$ for every control-flow edge (n_i, n_j); all arc weights are one.
- **Initial marking** M_0**.** Place a single token in the entry block's place. One token suffices because we analyse one dynamic execution of the method at a time; recursion or concurrency is handled at the call-graph level.

Fig. 8(A) shows the resulting net for our running example. Blocks $B_0 \ldots B_6$ map to places; transitions such as <validatePlain> carry the guard `alg==none`. Section V checks and rejects any candidate patch with markings that are not reachable.

D. Construction of Petri Nets from Specification

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,

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

To derive $\mathcal{N}_\phi$ from the natural-language description of each OpenID Connect authentication flow (Authorization Code, Implicit, and Hybrid), we follow a systematic mapping procedure:

- **Execution states as places.** Each major step in the OIDC protocol (e.g., discovery, authorization request, issuance of tokens, ID token validation, and claim verification) is mapped to a place in P_ϕ. A token in a place denotes that the execution has reached the corresponding stage of the protocol.
- **Events as transitions.** Each message exchange or logical step prescribed by the specification is represented as a transition in T_ϕ. For example, moving from "authorization granted" to "token exchange" is modeled as a transition consuming the token from one place and producing it in the next.
- **Guards from specification constraints.** Conditional requirements in the specification (e.g., "ID tokens must not use none as the alg value unless ..." [3]) are encoded as guard conditions on the relevant transitions. This ensures that the Petri net only admits state progressions that satisfy the logical rules of OIDC.
- **Initial and terminal states.** The initial marking $M_{0\phi}$ places a token in the <start> state of the protocol.

Terminal markings correspond to successful completion, such as after <verifyClaims>. This enables reachability analysis to check whether a candidate implementation can faithfully execute the entire flow.

Through this mapping process, each of the three OIDC authentication flows is transformed into a Petri net model that captures both the sequential ordering of protocol steps and the logical conditions imposed by the specification. This specification Petri net is then used as the reference model against which candidate program nets are validated.

Example 2. Fig. 8(B) illustrates the specification Petri net constructed from the specification (as standard Petri net input formats [20]) provided by the user. Each place in $\{P_1, P_2, \ldots, P_n\}$ 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 to guard conditions using the relevant protocol variables and operators.

E. Validation

We regard a patch as *correct* if the behavior of the patched program is consistent with the OpenID specification (i.e., semantically equivalent), meaning that every program trace is admitted by the specification (i.e., reachable).

Once the Petri net $\mathcal{N}_P$ is constructed for the candidate program, we validate it against the specification Petri net $\mathcal{N}_\phi$ using the VALIDATE procedure described in Algorithm 1. The procedure takes as input both nets and outputs either $\top$ if the candidate program conforms to the specification, or $\bot$ together with a counterexample set Φ if the program violates the specification.

The algorithm initializes the search space Φ with the initial markings of the two nets (line 5). In each iteration, it selects a pair of markings (M, M_ϕ) to explore (line 7). If the program net has reached one of its final markings while the specification net has not (line 8), the procedure terminates immediately and returns $\bot$ along with the counterexample trace Φ.

For each enabled transition T in the program net (line 9), the algorithm checks whether there exists a corresponding enabled transition T_ϕ in the specification net (line 10). If such a transition exists and both the event label ϵ and the guard condition γ match (line 11), then both transitions are fired simultaneously, and the resulting markings (M', M'_ϕ) are added to the search space (lines 12–13). Otherwise, the algorithm fires only the program transition T while leaving the specification marking unchanged (lines 17–18).

The procedure continues until the search space is exhausted. If no violating case is found, the algorithm terminates with $\top$ and an empty counterexample set $\varnothing$ (line 21), indicating that the candidate program satisfies the specification.

F. Counterexample-Guided Solution Refinement

Once a candidate patch is verified against the Petri-net specification, the verification procedure VALIDATE may either succeed ($\top$) or fail ($\bot$). In the case of failure, *AuthFix* leverages the verification feedback in two complementary ways to refine the repair process:

- First, the binary pass/fail outcome directly eliminates the current candidate from the search space, ensuring that invalid patches are not re-suggested in subsequent iterations.
- Second, beyond this coarse-grained signal, the Petri-net verifier also provides a counterexample trace Φ that captures the precise execution path or sequence of inputs leading to the violation of the specification. This counterexample is then reformulated as part of a structured follow-up prompt to the LLM, guiding it toward avoiding similar errors in the next round of patch synthesis.

Concretely the prompt to the LLM includes: (i) the buggy code with the previous candidate patch, (ii) the relevant specification excerpt and (iii) the counterexample trace highlighting the mismatch between the program and the specification. By incorporating the counterexample into the LLM's context, *AuthFix* effectively directs the model to prioritize alternative repairs that eliminate the violation. This iterative loop of *verification / counterexample extraction / LLM refinement* allows *AuthFix* to converge more efficiently toward a correct and specification-compliant patch, while pruning away unproductive search directions.

Algorithm 1 Petri Net Based Program Validation

```
1:  procedure VALIDATE(N_P, N_φ)
2:      input: Petri net N_P of the candidate program P, and Petri
        net N_φ of the specification φ
3:      output: ⊤ if valid, otherwise ⊥ with counterexample Φ
4:      assume: N_p(P, T, A, M_0) ∧ N_φ(P_φ, T_φ, A_φ, M_0φ)
5:      Φ = {⟨M_0, M_0φ⟩}
6:      while Φ ≠ ∅ do
7:          choose ⟨M, M_φ⟩ ∈ Φ
8:          if M ∈ final(N_P) ∧ M_φ ∉ final(N_φ) then return ⊥, Φ
9:          for all T ∈ enabled(M) do
10:             if ∃T_φ ∈ enabled(M_φ) then
11:                 if T.ϵ = T_φ.ϵ ∧ T.γ ⊨ T_φ.γ then
12:                     M', M'_φ = fire(M, T), fire(M_φ, T_φ)
13:                     Φ = Φ ∪ ⟨M', M'_φ⟩
14:                     continue
15:                 end if
16:             end if
17:             M' = fire(M, T)
18:             Φ = Φ ∪ ⟨M', M_φ⟩
19:         end for
20:     end while
21:     return ⊤, ∅
22: end procedure
```

VI. EVALUATION

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

- **RQ1**: Is *AuthFix* effective in fixing real-world bugs in OpenID implementations?
- **RQ2**: How effective is our approach when compared to other LLM-based program repair methods?
- **RQ3**: Are the generated repairs as correct as the manual patches written by the developers?

TABLE I: Average time for fault localization, sketch generation, enumeration, and validation in our dataset.

OpenID Bug Type	#Bug	Fault Localization (s)	Sketch Generation (s)	Enumeration & Validation (s)	Total Time (s)
Incorrect auth flow	2	2.3	38.1	294.0	334.4
Signature verification	7	5.1	88.7	583.2	677.0
`alg` validation	3	2.0	25.8	191.5	219.3
`aud` validation	3	2.7	53.2	249.8	305.7
`iss` validation	2	3.1	47.4	145.3	195.8
`nonce` check	1	4.5	78.2	265.4	348.1
Access token validation	2	4.5	84.5	363.7	452.7
CSRF protection	3	3.6	88.3	282.5	374.4

A. Implementation

We implement the technical concepts discussed above in the *AuthFix* tool, which takes OIDC programs and their specification (in English and the corresponding Petri net representation in PNML format [23]) as input and generates a fixed program that satisfies the specification. The *AuthFix* tool, consisting of approximately 5,400 lines of Java code, uses the popular ChatGPT (GPT-3.5) [49] LLM agent for the localization of bugs and the generation of optimized repair expressions. Additionally, we utilize IBM T.J. Watson Libraries for Analysis (WALA) [9] 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 [19] to check the equivalence of guard conditions associated with Petri net events in both programs and specifications.

B. Experimental Setup

We describe in the following the setup of our evaluation, covering benchmarks, specifications, and environment.

a) Benchmark collection: To evaluate the performance of our repair approach, we compile a dataset of OpenID bugs that consists of confirmed security bugs in OpenID libraries in previous papers [44], [45], [18], as well as public records (e.g., CVE reports) for other projects based on OpenID. Our collected dataset of OpenID bugs is representative as they are collected from the most popular OpenID libraries and cover all three authentication flows in the OpenID protocol: 1) authorization code flow (AC), 2) implicit flow (I), and 3) hybrid flow (H). Depending on the flow, the impact of each bug is either the relying party (RP), OpenID provider (OP), or both. In our dataset, each benchmark contains at least one bug that violates the OpenID specification. Along with the bugs, we further collect the patches written by human developers to fix the bugs. Specifically, our dataset contains 23 bugs from eight categories of OpenID bugs.

b) Specification: We follow OpenID Connect Core 1.0 [3] to get the specification for each benchmark in our dataset. The standard specification describes the required behavior of the authentication flows that are supported by OpenID. In addition, it details the protocol parameters and how each party involved in the protocol execution should process them. Once we collect the English specification for each benchmark, we manually construct its equivalent Petri-net model, as described earlier in the paper.

TABLE II: Evaluation results of *AuthFix* for OpenID bugs repair in our dataset. Column #Fix_{A} shows the correct 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).

OpenID Bug Type	#Bug	#Fix_{A}	#Fix_{dev}
Incorrect auth flow	2	1	1
Signature verification	7	4	2
`alg` validation	3	3	3
`aud` validation	3	3	3
`iss` validation	2	1	1
`nonce` check	1	1	-
Access token validation	2	2	-
CSRF protection	3	2	0
Overall	23	**17**	10

c) Alignment of programs with specifications: For each buggy program, we adopt a semi-automatic procedure to determine the relevant Petri-net specification. We keep a library of Petri-net models for different authentication flows and security properties, and use an LLM-based retrieval-augmented generation (RAG) [35] method to suggest the closest match. A lightweight manual confirmation then finalizes the minimal alignment between the buggy program and the specification. In this way, the Petri-net model is not global but contextually selected, and *AuthFix* operates in a *modular* fashion on the function or code region associated with the bug, which can be identified by existing bug localization or vulnerability detection tools [45].

C. 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. In total, 23 Petri nets were constructed for the benchmark programs, and 12 are constructed from the specification OpenID that covers the security properties for the three authentication flows. We delve into the details of the results of our experiments as follows.

a) Effectiveness of AuthFix (RQ1): To evaluate the effectiveness of *AuthFix* in repairing OpenID bugs, we use it to repair 23 benchmarks in our dataset and manually check the repaired bugs to validate their correctness with respect to the corresponding specification. As shown in Table II, it generated correct fixes for 17 out of 23 bugs. Among these 17 bugs, ten 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. In addition, the average time for *AuthFix* to fix

a bug is 362 seconds (6 minutes), whereas the minimum and maximum times are 78 seconds and 14 minutes, respectively.

We further examined the reasons behind the six instances where *AuthFix* failed to find a repair and identified the following root causes: (1) *AuthFix* was unable to repair expressions that required variables that are exchanged during the dynamic registration (e.g., `id_token_signed_response_alg`), which also allows passing a response as a JWT object. As extracting parameters from these objects requires complex operations such as decryption and base64 decoding, our repair algorithm could not synthesize expressions where values from these objects are required. (2) *AuthFix* also allows the clients to obtain keys to validate ID token from an external URI set by the `jwks_uri` parameter. Our repair algorithm was unable to synthesize the API calls (e.g., HTTP calls) needed to obtain the values from these external sources.

b) Comparison with other LLM-based repair approaches (RQ2): Given the recent success of LLM in various program repair tasks [27], [51], we further investigate how our approach performs compared against them, as follows.

1) ThinkRepair [55]: To enhance LLM's repair capabilities, ThinkRepair [55] utilizes a two-phase process involving a curated knowledge pool for bug fixing and a bug-fixing phase with Chain of Thoughts (CoT) prompting and few-shot learning. Specifically, given a corpus of buggy functions, ThinkRepair first builds a knowledge pool for chains of thoughts on fixing the buggy functions. To repair a bug, it then uses examples from the knowledge pool to guide LLMs in understanding and fixing bugs while adjusting prompts iteratively with test feedback to refine solutions. To compare *AuthFix*, we utilize the knowledge pool collected by ThinkRepair for Defects4J dataset [28] and use them to select examples of bugs and CoTs for similar repairs to guide the LLM to repair our OpenID benchmarks. For this selection step, we choose the contrastive-based selection as it utilizes a contrastive learning framework to further fine-tune UniXcoder for better semantic embedding. As shown in Table III, ThinkRepair was able to repair 7 bugs (out of 23) when evaluated on our OpenID benchmarks. Since ThinkRepair utilizes bug examples from its knowledge pool to guide the repair process, we found it was most effective in repairing bugs in conditional statements. However, it performs poorly for bugs (e.g., signature validation) that require understanding the semantics of the OpenID protocol and specification.

2) LLM inference: We further compare the results with LLM inference only (i.e., zero-shot completion) program repair. Following the common prompt construction approach in prior works [27], we construct prompts to repair the bugs in each of our benchmarks. 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. We submit an LLM query $Q = \{q_i, q_{code}, q_{spec}\}$ where q_i is a static repair instruction for the LLM agent, q_{code} is the buggy code and q_{spec} is the OpenID specification for the corresponding bug. As shown in Table III, LLM inference approach was able to generate repairs for only four bugs. We observe that LLM can fix commonly observed programming bugs [46], [53] such as missing clauses for null checking. However, in this work, we focus on the logical bugs relevant to the incorrect implementation of OpenID protocol, for which LLM agents struggle to generate a valid repair.

TABLE III: Comparison between the repairs generated by *AuthFix* (**#Fix$_A$**), ThinkRepair (**#Fix$_T$**) [55], and LLM inference approach (**#Fix$_L$**) for the OpenID bugs.

OpenID bug type	**#Fix$_A$**	**#Fix$_T$**	**#Fix$_L$**
Incorrect auth flow (2)	1	0	0
Signature verification (7)	4	1	1
`alg` validation (3)	3	2	1
`aud` validation (3)	3	1	1
`iss` validation (2)	1	1	0
`nonce` check (1)	1	1	0
Access token validation (2)	2	0	0
CSRF protection (3)	2	1	1
Overall	17	7	4

c) Quality of the generated repairs (RQ3).: 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 reported in prior works [45], [18] and their respective patches based on the GitHub commit history. However, for three bugs (from the categories of 'nonce check' and 'access token validation'), we were not able to obtain the manual patches as the developers had not published any patches by the time of our experiments. In total, we collected the 20 manually written patches corresponding to the bugs in our dataset. Upon our manual investigation, we found that 10 out of the 15 generated patches are semantically equivalent to the patches written by human developers (Table II). The other six patches are not exactly semantically equivalent just because our patches used hard-coded values (e.g., OpenID configured constants) whereas developers obtained the values from dynamic API calls or from sessions. These results show that *AuthFix* can generate high-quality patches.

D. Threats to Validity

This section outlines possible limitations of our study and their implications for the results.

a) Benchmark creation: Our benchmark doesn't include all the publicly reported bugs in OpenID implementations. As many reported bugs include the OpenID implementations that are not open-sourced, creating such a comprehensive dataset is quite challenging. However, we still tried our best efforts to select the most representative bugs and cover a wide variety of bugs in our benchmarks. Furthermore, since *AuthFix* is designed to cover all the flows in OpenID protocol and incorporate domain knowledge of the entire OpenID specification, we believe *AuthFix* can generalize well to new OpenID bugs.

b) Petri net creation: The Petri net models were manually constructed to capture the program's expected behavior according to the specification. On average, constructing the model for each of the three OpenID authentication flows took

about 3.5 hours and required expertise in formal modeling and the OIDC specification. Since this is a one-time effort per flow, the resulting models can be reused across all benchmarks following the standard. Although effective in practice, prior work [56] shows that formal specifications can be automatically extracted from natural language documentation, suggesting that future research could further reduce or even automate this effort using LLM-based techniques.

VII. Related work

In this section, we review related lines of work and highlight how our approach differs from and complements them.

A. Search-Based Repair

GenProg [34], 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 [31] utilizes GenProg's search approach with a set of repair templates manually derived from human-written patches. Le et al. [33] 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 [52] 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 [48] employs a random search approach, while AE [43] leverages deterministic search, enhanced by analytical techniques to eliminate redundant patches and optimize the search process.

Different from previous work, our proposed approach tackles the repair problem for complex logical bugs in OpenID Connect protocol that requires a richer domain-specific repair expression 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.

B. Repair Using Constraint Solving

MintHint [29], and NOPOL [53] employ symbolic execution to construct oracle-like representations and then use program synthesis to generate repairs. DirectFix [40] targets concise fixes via partial maximum satisfiability with SMT formulas, while Angelix [41] improves scalability with a lightweight constraint mechanism. S3 [32] further augments these semantics-based approaches with ranking criteria from execution traces, and CPR [47] leverages concolic path exploration to prune overfitting patches. More recently, SymlogRepair [38] combines repair with Datalog-defined static analysis, showing that constraint solving can capture richer program properties.

These methods, however, require translating constraints into SAT/SMT, which risks incompleteness [41], [32] due to complex semantics and external libraries common in API-based protocols like OpenID Connect. They also mainly reason about boolean or integer conditions, limiting their ability to handle bugs involving complex APIs such as cryptographic functions. Furthermore, symbolic execution engines (e.g., KLEE [41]) often fail to extract constraints at scale. In contrast, our approach operates directly at the AST level, avoiding heavy translation and enabling the repair of expressions, diverse variable types, function invocations, and APIs, while remaining applicable to complex data structures as long as they can be executed.

C. 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. [39] proposes CoCoNuT which leverages ensemble learning, combining CNN and NMT to generate patches. DLFix [37] 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 [25] has recently achieved state-of-the-art results on the Defects4J [28] and QuixBugs [54] datasets, outperforming NMT-based APR techniques by utilizing a pre-trained programming language model, code-aware search, and subword tokenization. Elixir [46] uses machine learning to rank potential repair candidates to reduce the search space for object-oriented programs. InferFix [26] 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.

VIII. Conclusion

OpenID Connect has significantly improved online authentication, but its complexity has led to critical security vulnerabilities causing substantial financial and data breaches. To address this, our proposed tool, *AuthFix*, leverages large language models to automate bug detection and patch generation, ensuring accurate and reliable fixes through a novel Petri-net-based model checker. Our evaluation of a dataset of OpenID bugs demonstrates that *AuthFix* successfully generates correct patches for 17 out of 23 bugs (74%), with a high proportion of patches semantically equivalent to developer-written fixes.

Acknowledgments

This work is supported in part by CISCO faculty award, Google Faculty Research Award, Ethereum Foundation Academic Award, NSF 1908494, NSF 2317184, and DARPA N66001-22-2-4037.

References

[1] "Json web token," https://datatracker.ietf.org/doc/html/rfc7519, 2015.

[2] "Facebook security update," https://about.fb.com/news/2018/09/security-update, 2018.

[3] "Openid connect core 1.0," https://openid.net/specs/openid-connect-core-1_0.html, 2021.

[4] "High vulnerability in google's oauth client library for java," https://it.ucsf.edu/high-vulnerability-googles-oauth-client-library-java, 2022.

[5] "Authentication bug that enabled unauthorized access to client applications," https://portswigger.net, 2023.

[6] "Azure b2c – crypto misuse and account compromise," https://securityboulevard.com, 2023.

[7] "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, 2023.

[8] "Openid certification," https://openid.net/certification/, 2023.

[9] "T.j. watson libraries for analysis (wala)," https://sourceforge.net/projects/wala, 2023.

[10] "cpprestsdk," https://github.com/microsoft/cpprestsdk, 2025.

[11] "dex," https://github.com/dexidp/dex, 2025.

[12] "googleapis," https://github.com/googleapis/google-api-nodejs-client, 2025.

[13] "hydra," https://github.com/ory/hydra, 2025.

[14] "oauth2-proxy," https://github.com/oauth2-proxy/oauth2-proxy, 2025.

[15] "Oauth2.0 authorization code - security issue - race condition," https://github.com/oauthlib/oauthlib/issues/618, 2025.

[16] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, ser. TAICPART-MUTATION '07. USA: IEEE Computer Society, 2007, pp. 89–98.

[17] ——, "Spectrum-based multiple fault localization," in *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. USA: IEEE Computer Society, 2009, pp. 88–99. [Online]. Available: https://doi.org/10.1109/ASE.2009.25

[18] T. Al Rahat, Y. Feng, and Y. Tian, "AuthSaber: Automated safety verification of OpenID connect programs," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 2949–2962. [Online]. Available: https://doi.org/10.1145/3658644.3670318

[19] D. L. Berre and A. Parrain, "The Sat4j library, release 2.2: System description," *Journal on Satisfiability, Boolean Modelling and Computation*, vol. 7, no. 2-3, pp. 59–64, 2011. [Online]. Available: http://dx.doi.org/10.3233/SAT190075

[20] J. Billington, S. Christensen, K. Van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber, "The petri net markup language: concepts, technology, and tools," in *Proceedings of the 24th International Conference on Applications and Theory of Petri Nets*, ser. ICATPN'03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 483–505.

[21] Y. Feng, R. Martins, O. Bastani, and I. Dillig, "Program synthesis using conflict-driven learning," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 420–435. [Online]. Available: https://doi.org/10.1145/3192366.3192382

[22] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps, "Component-based synthesis for complex APIs," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '17. New York, NY, USA: Association for Computing Machinery, 1 Jan. 2017, pp. 599–612. [Online]. Available: https://doi.org/10.1145/3009837.3009851

[23] L. M. Hillah, F. Kordon, L. Petrucci, and N. Trèves, "PNML framework: an extendable reference implementation of the petri net markup language," in *Proceedings of the 31st International Conference on Applications and Theory of Petri Nets*, ser. PETRI NETS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 318–327. [Online]. Available: https://doi.org/10.1007/978-3-642-13675-7_20

[24] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 215–224. [Online]. Available: https://doi.org/10.1145/1806799.1806833

[25] N. Jiang, T. Lutellier, and L. Tan, "CURE: Code-aware neural machine translation for automatic program repair," in *Proceedings of the 43rd International Conference on Software Engineering*, ser. ICSE '21. Madrid, Spain: IEEE Press, 2021, pp. 1161–1173. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00107

[26] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "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*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, pp. 1646–1656. [Online]. Available: https://doi.org/10.1145/3611643.3613892

[27] H. Joshi, J. C. Sanchez, S. Gulwani, V. Le, I. Radiček, and G. Verbruggen, "Repair is nearly generation: multilingual program repair with LLMs," in *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence*, ser. AAAI'23/IAAI'23/EAAI'23. AAAI Press, 2023. [Online]. Available: https://doi.org/10.1609/aaai.v37i4.25642

[28] R. Just, D. Jalali, and M. D. Ernst, "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*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 437–440. [Online]. Available: https://doi.org/10.1145/2610384.2628055

[29] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso, "MintHint: automated synthesis of repair hints," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 266–276. [Online]. Available: https://doi.org/10.1145/2568225.2568258

[30] S. Kang, G. An, and S. Yoo, "A quantitative and qualitative evaluation of LLM-based explainable fault localization," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3660771

[31] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 802–811.

[32] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: syntax- and semantic-guided repair synthesis via programming by examples," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, pp. 593–604. [Online]. Available: https://doi.org/10.1145/3106237.3106309

[33] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 213–224. [Online]. Available: http://dx.doi.org/10.1109/SANER.2016.76

[34] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 54–72, Jan. 2012. [Online]. Available: https://doi.org/10.1109/TSE.2011.104

[35] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-T. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS '20. Red Hook, NY, USA: Curran Associates Inc., 2020.

[36] A. S. Lezama, "Program synthesis by sketching," Ph.D. dissertation, Citeseer, 2008.

[37] Y. Li, S. Wang, and T. N. Nguyen, "DLFix: context-based code transformation learning for automated program repair," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 602–614. [Online]. Available: https://doi.org/10.1145/3377811.3380345

[38] Y. Liu, S. Mechtaev, P. Subotić, and A. Roychoudhury, "Program repair guided by datalog-defined static analysis," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, pp. 1216–1228. [Online]. Available: https://doi.org/10.1145/3611643.3616363

[39] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "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*, ser. ISSTA 2020. New

York, NY, USA: Association for Computing Machinery, 2020, pp. 101–114. [Online]. Available: https://doi.org/10.1145/3395363.3397369

[40] S. Mechtaev, J. Yi, and A. Roychoudhury, "DirectFix: looking for simple program repairs," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Florence, Italy: IEEE Press, 2015, pp. 448–458.

[41] ——, "Angelix: scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 691–701. [Online]. Available: https://doi.org/10.1145/2884781.2884807

[42] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 609–620. [Online]. Available: https://doi.org/10.1109/ICSE.2017.62

[43] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 254–265. [Online]. Available: https://doi.org/10.1145/2568225.2568254

[44] T. A. Rahat, Y. Feng, and Y. Tian, "OAuthLint: an empirical study on OAuth bugs in android applications," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '19. San Diego, California: IEEE Press, 2020, pp. 293–304. [Online]. Available: https://doi.org/10.1109/ASE.2019.00036

[45] ——, "Cerberus: Query-driven scalable vulnerability detection in OAuth service provider implementations," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 2459–2473. [Online]. Available: https://doi.org/10.1145/3548606.3559381

[46] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "ELIXIR: effective object oriented program repair," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '17. Urbana-Champaign, IL, USA: IEEE Press, 2017, pp. 648–659.

[47] R. Shariffdeen, Y. Noller, L. Grunske, and A. Roychoudhury, "Concolic program repair," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, pp. 390–405. [Online]. Available: https://doi.org/10.1145/3453483.3454051

[48] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: models and first results," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '13. Silicon Valley, CA, USA: IEEE Press, 2013, pp. 356–366. [Online]. Available: https://doi.org/10.1109/ASE.2013.6693094

[49] T. Wu, S. He, J. Liu, S. Sun, K. Liu, Q.-L. Han, and Y. Tang, "A brief overview of ChatGPT: The history, status quo and potential future development," *IEEE/CAA Journal of Automatica Sinica*, vol. 10, no. 5, pp. 1122–1136, 2023. [Online]. Available: http://dx.doi.org/10.1109/JAS.2023.123618

[50] Y. Wu, Z. Li, J. M. Zhang, M. Papadakis, M. Harman, and Y. Liu, "Large language models in fault localisation," *arXiv [cs.SE]*, 2023. [Online]. Available: http://arxiv.org/abs/2308.15276

[51] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. Melbourne, Victoria, Australia: IEEE Press, 2023, pp. 1482–1494. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00129

[52] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 416–426. [Online]. Available: https://doi.org/10.1109/ICSE.2017.45

[53] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 34–55, Jan. 2017. [Online]. Available: https://doi.org/10.1109/TSE.2016.2560811

[54] H. Ye, M. Martinez, T. Durieux, and M. Monperrus, "A comprehensive study of automatic program repair on the QuixBugs benchmark," pp. 1–10, 2019. [Online]. Available: http://dx.doi.org/10.1109/IBF.2019.8665475

[55] X. Yin, C. Ni, S. Wang, Z. Li, L. Zeng, and X. Yang, "ThinkRepair: Self-directed automated program repair," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, pp. 1274–1286. [Online]. Available: https://doi.org/10.1145/3650212.3680359

[56] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language API documentation," in *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. USA: IEEE Computer Society, 2009, pp. 307–318. [Online]. Available: https://doi.org/10.1109/ASE.2009.94