IACR Transactions on Symmetric Cryptology
ISSN XXXX-XXXX, Vol. 0, No. 0, pp. 1–18. DOI:XXXXXXXX

Formal Analysis and Verification of DigiLocker with Tamarin

Nayan Kakade, Aditya Mundada and Raghvendra Rohit

Department of Computer Science and Engineering, Indian Institute of Technology Roorkee, Roorkee, India
{nayan_rk,mundada_am,raghvendra.rohit}@cs.iitr.ac.in

Abstract. Digilocker is a key component of India's Digital Public Infrastructure, enabling secure digital storage, retrieval, and sharing of government-issued documents. Given its large-scale deployment and reliance on protocols such as OAuth 2.0 with PKCE, HMAC-based API authentication, digital signatures, and encrypted storage, rigorous security assurance is essential which can not be be ensured through conventional testing alone. In this work, we present a formal modeling and verification of DigiLocker's authentication and document-handling workflows using the Tamarin Prover. We model the OAuth 2.0 authorization code flow with PKCE, issuer-based document retrieval mechanisms (PullURI and PullDoc), document push workflows, and self-upload with encrypted storage under the Dolev–Yao adversary model. We formally specify and verify key security properties, including token secrecy, authorization code uniqueness, PKCE binding, document authenticity, integrity, confidentiality, and key management. Our results show that DigiLocker's protocol design satisfies these properties under ideal assumptions. However, controlled credential-leak scenarios demonstrate that compromise of sensitive values such as API keys or `digilockerid`s can lead to exploitable attack traces. This study highlights the value of formal verification in strengthening security guarantees for large-scale e-governance systems.

Keywords: Formal Verification · DigiLocker · OAuth 2.0 · PKCE · Tamarin Prover · Cryptographic Protocols

1 Introduction

Research on e-governance and digital public infrastructure (DPI) has accelerated rapidly over the past decade, with initiatives like Digital India aiming to provide paperless, secure access to essential citizen services [Min15]. The rapid adoption of interconnected platforms, notably the Unified Payments Interface (UPI) for real-time financial transactions [TC20] and Aadhaar for biometric identity, has cemented India's position as a leader in DPI deployment [Wor23]. Over time, the security community's understanding of the threat models and security claims for these large-scale digital document exchange systems has evolved. Modern e-governance platforms must now offer strong security guarantees and also securely handle highly sensitive data even in the presence of active network adversaries who may compromise different endpoints during the protocol's execution. This is befitting given that strong adversaries are capable of targeting interconnected APIs and authorization gateways to orchestrate identity theft or privacy violations on a national scale.

More recently, the reliance on authorization frameworks, such as OAuth 2.0, combined with cryptographic document verification, has become a critical focal point for security audits [LBLF23]. This requires rigorous protection against adversaries who exploit implementation flaws, misconfigurations, or credential leaks to bypass authentication, hijack

Licensed under Creative Commons License CC-BY 4.0.

sessions, or forge document retrieval requests [FKS16].

DigiLocker is a foundational component of India's DPI ecosystem, designed to provide citizens with secure, legally recognized digital equivalents of physical documents [Minb]. The protocol takes a hybrid approach to security and combines standard authentication frameworks such as OAuth 2.0 with the Proof Key for Code Exchange (PKCE) extension [SBA15], with various cryptographic primitives, including AES-GCM for secure document storage [Dwo07], HMAC for request authentication [KBC97], and digital signatures to ensure document authenticity. The integration of these diverse mechanisms provides a robust defense, but the complexity of multi-party workflows between users, clients, authorization servers, and document issuers necessitates rigorous formal verification to ensure that intended security properties indeed hold.

In this paper, we present formal analysis of DigiLocker protocol. We analyze DigiLocker's security in detail using the Tamarin Prover, a state-of-the-art security protocol model checker [MSCB13, BCDS26]. We report on our formal model of DigiLocker, the adversary assumptions under the Dolev-Yao model [DY83], and the protocol's desired properties. We use Tamarin's specification language to model the complete messaging flow and cryptographic operations of DigiLocker's core workflows: user login via Aadhaar [ABS18], document retrieval (via PullURI and PullDoc mechanisms), document push by issuers, and document self-upload. We also specify forms of adversary compromise, including the deliberate injection of controlled vulnerabilities, such as the leakage of a user's `digilockerid` or an issuer's API key. Using Tamarin's property language, we formalize and prove critical theorems including document authenticity, integrity, confidentiality, token secrecy, and replay resistance.

Our analysis establishes that DigiLocker provides strong security guarantees against an active network adversary that controls the communication channel, provided that essential identity credentials remain uncompromised. For example, the protocol successfully protects against token interception and session hijacking by binding authorization codes to secret verifiers via PKCE. Moreover, our fine-grained analysis of compromise possibilities demonstrates that while the standard protocol design is resilient, the leakage of a single credential (such as an issuer API key) directly violates confidentiality by allowing an attacker to forge PullURI requests. Such insights are highly valuable for guiding secure implementations and emphasizing the necessity of secure credential management in API-driven governance architectures [OWA23].

Our Contributions. Our first contribution is the formalization and machine-checked verification of the DigiLocker protocol to prove its core security claims. Namely, we use Tamarin to prove that DigiLocker offers strong security guarantees—such as document authenticity, integrity, and single-use authorization code enforcement—against a powerful active network adversary. These guarantees are comprehensive, and omitting any of the modeled adversary compromise cases helps explicitly define the boundary of the protocol's resilience. Our verification thereby provides a formal, machine-checked proof that DigiLocker meets the high expectations for a modern, national-scale digital public infrastructure. This high assurance is important given the prominent role of this protocol, which serves millions of citizens in India, and its limited prior formal symbolic analysis.

Our second contribution is to show that symbolic security protocol model checkers, in particular Tamarin, can successfully verify the complex, multi-party workflows of large-scale e-governance systems in their full complexity. This is non-trivial as it entails reasoning about the interleaved parallel sessions of multiple distinct entities (clients, authorization servers, resource servers, and external issuers) exchanging various tokens, hashes, and signatures. By modeling workflows ranging from PKCE-based OAuth exchanges to encrypted document storage, our work provides a general methodology for carrying out symbolic proofs on government-scale document handling ecosystems.

Organization of the Paper. In Section 2, we survey related work on DPI, formal verification, authentication protocols and Tamarin. Afterwards, in Section 3, we describe the DigiLocker protocol's workflows, threat model, and security requirements. In Section 4, we present our Tamarin model of DigiLocker, the adversary capabilities and the formalized security properties. Section 5 presents our detailed analysis of results. We draw conclusions and discuss future work in Section 6.

The complete Tamarin models, scripts, and instructions to reproduce the formal verification results presented in this paper are available on GitHub at https://github.com/nayan-1210/Formal-Verification-of-Digilocker-using-Tamarin-Prover.

2 Preliminaries

In this section, we give an overview of Digital Public Infrastructure (DPI), formal verification, authentication protocols and Tamarin prover in the context of DigiLocker.

2.1 E-Governance and Digital Public Infrastructure

Over the past decade, governments worldwide have fastened the deployment of e-governance systems and DPI to democratize access to resources. Platforms like DigiLocker facilitate paperless document exchange and rely on standard cryptographic mechanisms—such as OAuth 2.0 for user authorisation, HMAC for request authentication, and public-key infrastructure (PKI) for digital signatures. While these underlying primitives are robust, the sheer scale and complexity of API-driven integrations in DPIs introduces substantial attack surfaces [Minb].

Several studies have proposed formally verified frameworks for secure personal data sharing using technologies such as blockchain [D+18] and trusted execution environments (TEEs) [S+21]. These studies provide security guarantees such as confidentiality and integrity under the Dolev-Yao threat model. At the same time, research in e-governance systems has explored event-driven architectures. It has also used formal methods to analyze asynchronous workflows, API integrations, and message communication patterns. However, despite the large-scale real-world use of DigiLocker, there is still no work on formally verifying its document retrieval and document push workflows. This results in a lack of formally validated end-to-end security guarantees.

2.2 Authentication Protocols

OAuth 2.0 is widely used in many modern web and mobile applications, such as DigiLocker [LBLF24]. As it is adopted widely and has complex authorization flows, it has undergone extensive formal analysis. Researchers have modeled different configurations of OAuth and analyzed them with automated tools like ProVerif. These studies revealed many practical vulnerabilities such as cross-site request forgery (CSRF), open redirectors, and logic problems in the authorization code flow [FKS16, pro]. There exist other approaches, such as knowledge flow analysis using the Alloy framework, which have also been used to find logic problems in the early versions of OAuth.

To address vulnerabilities in mobile and single-page applications, the Proof Key for Code Exchange (PKCE) extension was introduced as a best current practice. PKCE mitigates authorization code interception attacks by dynamically binding the authorization code to a secret verifier known only to the legitimate client. While standalone OAuth and PKCE flows have been rigorously analyzed, evaluating their secure integration into broader document-handling ecosystems like DigiLocker requires specialized modeling that accounts for the system's unique state transitions and API constraints.

2.3 Symbolic Verification and the Tamarin Prover

Security protocols are typically evaluated in either the computational or symbolic model. Computational proofs offer granular cryptographic guarantees by bounding an adversary's probabilistic advantage, but they are often highly complex and rely on pen-and-paper arguments. In contrast, the symbolic model abstractly represents messages as terms in a term algebra and adversaries as entities with absolute control over the network (the Dolev-Yao model). This abstraction enables the use of automated verification tools.

ProVerif and Tamarin-Prover are two advanced tools for symbolic verification. ProVerif models protocols using Horn clauses and is highly effective at verifying properties such as secrecy and authentication even for an unlimited number of sessions [Bla14]. However, it is less suitable for protocols that have rich mutable state or complex global data structures. On the other hand, Tamarin uses trace-based semantics and multiset rewriting rules. These help in modeling complex systems having non-monotonic global states and unbounded loops. Properties in Tamarin are expressed using a form of first-order temporal logic [Tam].

Tamarin has been successfully used to verify many complex real-world protocols [MSCB13]. These include the verification of the TLS 1.3 standard [CHSvdM17], 5G-AKA authentication in cellular networks [BDH+18], and Apple's iMessage PQ3 protocol [LSB25]. In the analysis of PQ3 and TLS 1.3, Tamarin was very useful in modeling the protocols' nested loops, rich mutable state, and post-compromise security claims in their full complexity. Building on these capabilities, our work uses Tamarin's trace-based analysis and stateful modeling to formally verify DigiLocker's multi-stage workflows and provide security guarantees.

2.4 Background in Tamarin

The Tamarin Prover supports the automated, unbounded, symbolic analysis of security protocols. It features expressive languages for specifying protocols, adversary models, and properties, and support for efficient deduction and equational reasoning.

2.4.1 Cryptographic Primitives and Equational Theory

Tamarin models cryptography using symbolic terms and an equational theory. For example:

- **Hashing:** A one-way function $h(m)$
- **Asymmetric Encryption:** Defined by the equation $adec(aenc(m, pk(sk)), sk) = m$
- **Signatures:** Defined by $verify(sign(m, ltk), m, pk(ltk)) = true$.

2.4.2 Rule

In Tamarin, the protocol's execution and the adversary's capabilities are defined via Labeled Multiset Rewriting Rules. A rule represents an atomic transition from one system state to another. A rule is formally defined as a triplet (L, A, R) written as:

$$[l_1, \ldots, l_n] \xrightarrow{[a_1,\ldots,a_k]} [r_1, \ldots, r_m]$$

- **Left Hand Side(LHS):** The LHS consists of a multiset of pre-condition facts. For a rule to fire, all the facts in LHS should be available in the current system state.
  - **Consumption:** When the rule fires, all linear facts in L are consumed (removed from the state). This models the "use once" nature of session states or one-time tokens.

  - **Persistence:** Facts prefixed with an exclamation mark (!) are persistent. They are not consumed and remain available for subsequent rule applications (for example, !Ltk($A, lk) representing a long-term key).

- **Action Labels (A):** Action Labels mark the transition. They record that a specific event occurred at a specific point of time.
- **Right Hand Side(RHS):** The RHS define the post-condition facts generated by the transition. These facts are added to the system state, which become available for the future rules.

An example of rule is given in Listing 1.

Listing 1: Send Message rule

```
rule Send_Message:
    [ State(A, ~secret), !Pk(B, pkB) ]
       --[ Send(A, ~secret) ]->
    [ Out(aenc(~secret, pkB)) ]
```

The rule models the transition where agent A retrieves a local secret and agent B's persistent public key to generate an asymmetric ciphertext for network transmission. The *Send* action label is the temporal marker.

In Table 1, we provide the key components in Tamarin modeling.

Table 1: Key Components in Tamarin Modeling

Component	Tamarin Syntax	Research Interpretation
Variables	`~x`, `$A`, `m`	Represents fresh values (nonces), public identities, and generic messages respectively.
Fresh Fact	`Fr(~x)`	Models the generation of a cryptographically secure, unique random number.
Network	`Out(m)`, `In(m)`	Interfaces with the Dolev-Yao network; `Out` grants knowledge to the adversary.

2.4.3 Lemma

A Lemma is a mathematical specification of the security property that we intend to prove. Tamarin evaluates these properties over Traces — sequences of action labels generated by rule applications.

Consider a standard **Aliveness Lemma** in Listing 2.

Listing 2: Aliveness Lemma

```
lemma Aliveness:
    "All A B #i. Commit(A, B) @i ==> (Ex #j. Create(B) @j & j < i)"
```

- Lemmas are written in first order temporal logic.
- Quantifiers such as *(All, Ex)* are used. In case of *All*, the property must hold for all possible execution traces.
- Timepoint Association *(@i)* binds an action to a specific point in the protocol history.
- The implication symbol *(==>)* sets the condition. For Example, if the event *Commit* happens at time i, then event *Create* must have occured at time j before the event *Commit*.

2.4.4 Proof by Constraint Solving

Tamarin's engine runs backwards from the Lemma. To prove a security property, it tries to reconstruct a valid chain of rules (a dependency graph) that could lead to that state. This chain is also called a trace. If a valid trace exists, then the lemma is falsified and a counter example graph is generated in interactive mode. If no such chain is found, then the lemma is verified.

3 The DigiLocker Protocol

DigiLocker is a multi-entity protocol enabling users to asynchronously fetch, store, and share verified digital documents [Min23]. In this section, we describe the protocol's core workflows at a high level, focusing on the cryptographic operations that ensure secure document handling.

3.1 Client and Issuer Onboarding

Before interacting with the system, client applications and document issuers must register via the API-Setu Portal [Mina]. Upon successful onboarding, clients are provisioned with a `client_id` and, if they are confidential server apps, a `client_secret` [Minc]. Issuers configure specific endpoints and are issued API keys, which are heavily utilized for HMAC authentication during document retrieval requests.

3.2 User Login and OAuth 2.0

To access protected documents, a citizen must securely log into DigiLocker client application using their government-authorized credentials. Because mobile applications are considered public clients and cannot inherently store static client secrets securely, DigiLocker employs the OAuth 2.0 Proof Key for Code Exchange (PKCE) extension [LBLF24].

The authentication and authorization sequence are split into two distinct phases to ensure that even if an authorization code is intercepted, it cannot be exchanged for an access token by a malicious actor.

3.2.1 Phase 1: Authorization Request and Authentication

Figure 1 illustrates the initial authorization request and the user authentication phase. The process begins when the user initiates a login request. To secure the upcoming transaction, the client application locally generates a high-entropy random string called the `code_verifier`. It then computes a `code_challenge` by hashing the verifier using SHA-256 and encoding it with Base64URL [Minc].

The client sends this `code_challenge` to the DigiLocker Authorization Server (AS) via an HTTPS GET request to the `/authorize` endpoint [Min24]. The AS temporarily stores this challenge and initiates the authentication process by querying the UIDAI (Aadhaar) authentication gateway [Uni22]. Once the user successfully authenticates and grants consent, the AS redirects the user back to the client application with a temporary `auth_code` [Min24].

3.2.2 Phase 2: Token Exchange and API Access

Figure 2 details the critical token exchange phase where the cryptographic binding of PKCE is enforced. The client application makes a POST request to the `/token` endpoint, providing the temporary `auth_code` it just received along with the original, unhashed `code_verifier` [Minc].

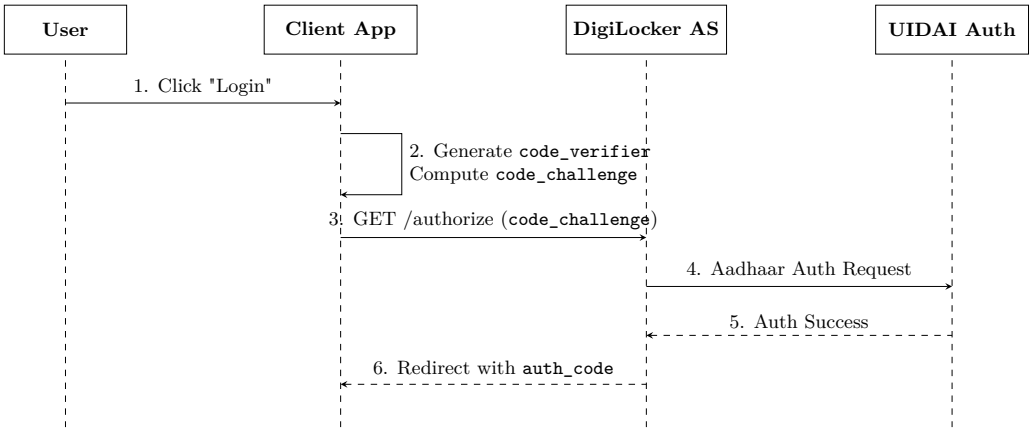

Figure 1: Phase 1: PKCE Authorization Request and UIDAI Authentication.

The AS independently calculates the SHA-256 hash of the received `code_verifier` and compares it against the `code_challenge` it stored during Phase 1. If the values match, the AS is mathematically assured that the entity redeeming the code is the exact same entity that requested it, successfully preventing interception attacks. The AS then issues the `access_token`, which the client subsequently uses as a Bearer token to fetch protected user documents from the resource APIs [Mina].

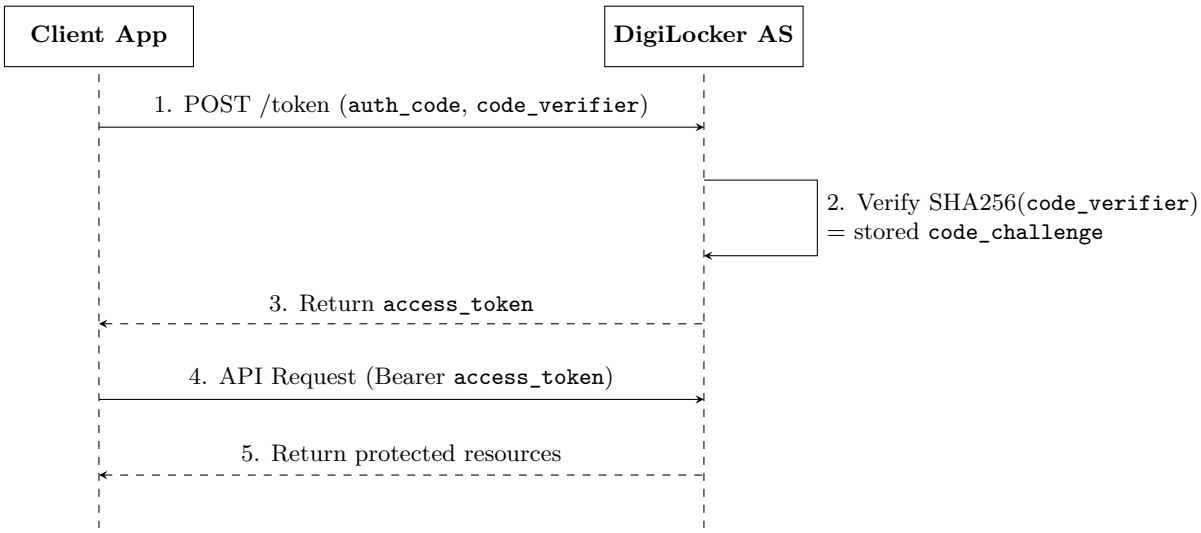

Figure 2: Phase 2: Cryptographic Token Exchange and Subsequent API Access.

3.3 Document Retrieval: PullURI and PullDoc

Using a valid access token, the client application fetches documents linked to the user's account. DigiLocker handles retrieval via two primary mechanisms, namely PullDoc and PullURI. The execution of these workflows is depicted in Figure 3 [Minc] and explained in detail as follows.

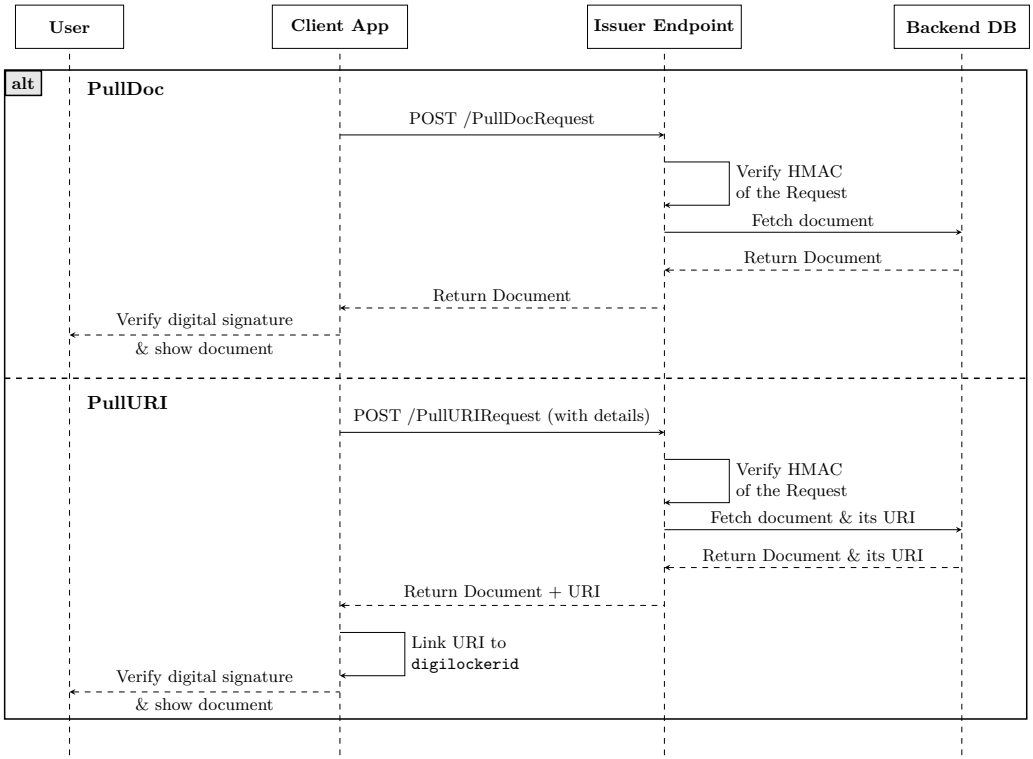

Figure 3: Sequence diagram illustrating the PullDoc and PullURI retrieval mechanisms.

3.3.1 PullDoc Mechanism

This mechanism is invoked when the Document URI is already known and linked to the user's `digilockerid` . As shown in the top half of the `alt` block in Figure 3, the client application sends an HTTP POST request containing the document URI and the user's ID in the body, along with a hash of the body data in the header. The issuer endpoint performs HMAC authentication by comparing the received data hash to the header hash. Upon successful validation, the issuer fetches the document from the backend repository and returns it to the client, providing the PDF data within the `<DocContent>` element and XML metadata within the `<MetadataContent>` element [Minc].

3.3.2 PullURI Mechanism

When the URI of the document is unknown, the user must explicitly search for it using identifying parameters such as Name, Date of Birth, or a specific Roll Number. The client issues an HTTP POST `PullURIRequest` with these search parameters embedded in the body, again utilizing a body hash in the header for HMAC authentication. If the HMAC validation is successful, the issuer retrieves the document based on the search parameters, generates the corresponding URI, and returns both the document bytes and the URI to the client application. The client subsequently links this newly discovered URI to the user's `digilockerid` locally for future access [Minc].

3.3.3 Digital Signature Verification

To ensure authenticity, documents returned by issuers contain an embedded digital signature. This signature is created by the issuer by computing the SHA-256 hash of the

original XML data followed by encryption. Upon receiving the document, the client application performs a local verification process before displaying it to the user. It extracts the signature block, decrypts it and compares the resulting value against a fresh SHA-256 hash of the remaining document bytes. Additionally, the client checks that the metadata matches the expected fields, such as the user's `name` and `digilockerid`, accepting the document only if all verification steps pass [Minc].

3.4 Document Push by the Issuer (PushURI)

In addition to the pull mechanisms, DigiLocker provides a proactive method for document delivery known as the Document Push or `PushURI` mechanism. This allows an authorized issuer, such as an educational board or government department, to directly link freshly generated documents to a user's `digilockerid` without waiting for the user to initiate a search.

This process bypasses the need for the `PullURI` search mechanism, saving user effort and ensuring immediate delivery of critical documents. To execute a push, the issuer sends an HTTP POST request directly to the DigiLocker `PushURI` endpoint . This request contains issuer identification information (such as the `client_id`) along with the specific document metadata, including the `uri`, `doctype`, `issuedate`, and the target user's `digilockerid`. To ensure the security and authenticity of the transaction, the entire push request is validated using an HMAC-256 hash authentication system [Minc].

3.5 Document Self-Upload

DigiLocker also provides users with the capability to upload and securely store scanned copies or digital versions of their physical documents. These user-uploaded files are maintained separately and can be accessed within the "Uploaded Documents" section of the platform.

The self-upload workflow is comprehensive, involving secure data transit, rigorous policy checks, and robust at-rest encryption. We illustrate this process in two parts: the initial upload and encryption phase (Figure 4), followed by the document listing and retrieval phase (Figure 5).

3.5.1 Phase 1: Upload Request and Secure Storage

The upload process begins with the client application creating an HTTP POST request directed at the `/file/upload` endpoint. To ensure authenticity and integrity during transit, the request header includes the user's access token, file metadata (such as its MIME type), and an HMAC tag. The body of the request contains the actual file data, which is transferred over a standard TLS connection.

Upon receiving the payload, the Resource Server validates the access token and performs a consolidated file scanning and policy check to ensure the file is safe and conforms to size limits. Before the file is written to storage, the Resource Server encrypts the document locally using AES-GCM mode. The encrypted document is then sent to the Object Storage, while the accompanying metadata is saved in the Metadata Database, and an internal URI is generated to track the file . Finally, the server returns an HTTP 200 OK response containing the path of the file, allowing the client to confirm a successful upload to the user [Minc].

3.5.2 Phase 2: Listing and Retrieving Uploaded Documents

Once documents are successfully stored, users can view and retrieve them at any time. As shown in Figure 5, when a user navigates to their uploaded documents, the client

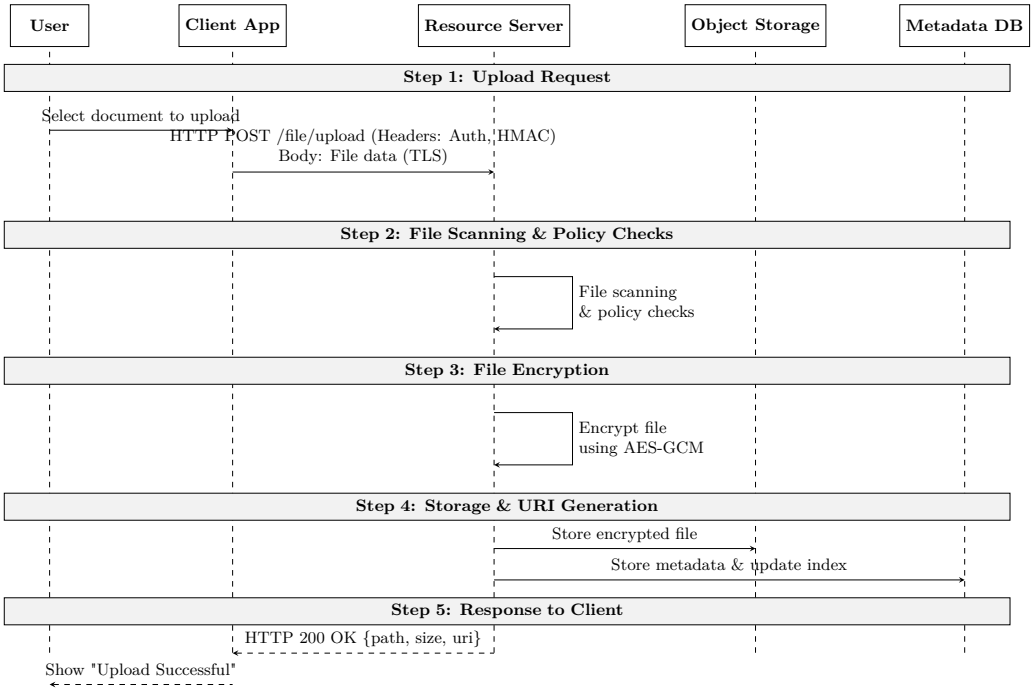

Figure 4: Phase 1 of Document Self-Upload: Upload request, scanning, encryption, and storage.

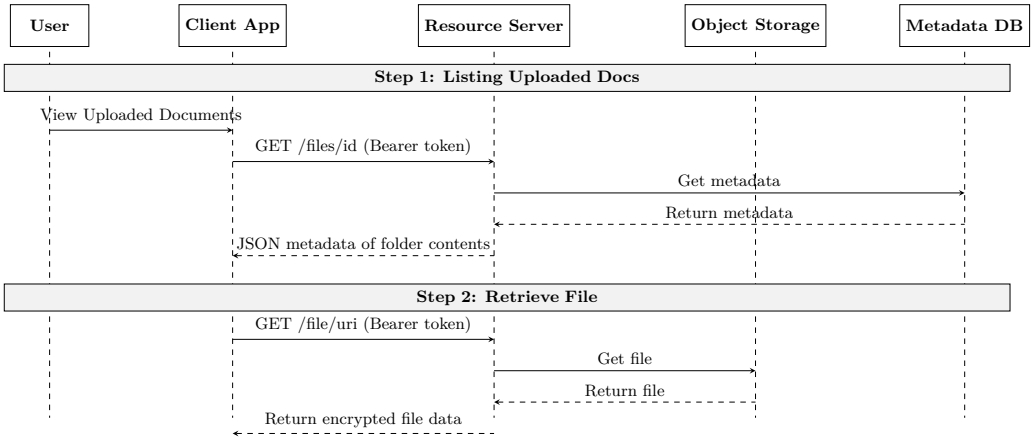

Figure 5: Phase 2 of Document Self-Upload: Listing existing user uploads and retrieving file data.

app issues a GET request to the `/files/id` endpoint. The Resource Server queries the Metadata Database and returns a JSON array containing the file's URI, name, size, and checksum.

If the user wishes to view a specific document, the client issues a GET request to the `/file/uri` endpoint. The Resource Server fetches the corresponding encrypted file from the Object Storage and returns the encrypted data back to the client app, provided the user's access token is valid and they possess the correct authorization.

4 Modeling DigiLocker in Tamarin

In this section, we provide our modeling of the DigiLocker protocol in Tamarin. First, we give an overview of the required security properties. Second, we discuss the Dolev-Yao model in brief. Finally, we give the concrete modeling of DigiLocker's Document Retrieval Mechanism as an example.

4.1 Security Requirements

DigiLocker is designed to provide strong security in all authentication, authorization, and document-handling workflows. In our analysis, we summarize these protections into three main security requirements

- **Authentication and Authorization:** Only users whose identity is verified should be allowed to access to their locker must be strictly bound to their verified identity. The protocol must enforce single-use authorization codes and ensure correct cryptographic binding via PKCE during the OAuth 2.0 workflow to prevent session hijacking and token interception.
- **Document Authenticity and Integrity:** Any document accessed by a user should be verifiable in terms of its integrity. If a document is accepted as valid, the corresponding honest issuer must have previously created it, and the document's signature remains valid and is not forged.
- **Confidentiality and Secrecy:** All Access tokens, API keys, and document payloads should remain confidential. Unless the locker is compromised and the adversary gains access, all self-uploaded documents should remain encrypted and unaccessible to the adversary.

4.2 Dolev–Yao Model

We analyze DigiLocker under the standard Dolev–Yao adversary model [DY83]. The attacker is assumed to have full control over the communication network and can intercept, modify, block, and replay any message exchanged between the user, the client application, the authorization server, the resource server, and the issuer endpoints.

We also assume that the standard cryptographic primitves are secure against classical cryptanalysis. These primitives include, but not limited to AES-GCM, HMAC-SHA256, and RSA digital signatures. Unless the adversary get the keys, it cannot decrypt encrypted texts.

We further analyse the protocol in two deployment situations:

- Under ideal assumptions (long-term identity keys and issuer API keys remain secret), and
- Under compromised assumptions (specific secrets such as a user's `digilockerid` or an issuer's `apikey` are leaked).

This approach also allows us to identify the limits of Digilocker's security guarantees.

4.3 Modeling Digilocker Protocol in Tamarin

We model DigiLocker using Tamarin Prover, representing system evolution through labeled multiset rewriting rules. The **Document Retrieval** mechanism is shown in the below example. Protocol participants (Client, Authorization Server, Resource Server, and Issuer) are defined as distinct roles.

4.3.1 Equational Theory

Real-world cryptographic operations are abstracted symbolically:

- `sha256/1` for hashing,
- `senc/denc` for symmetric encryption/decryption,
- `sign/verify` for public-key signatures,
- `pkce/1` for PKCE code challenge computation.

4.3.2 Protocol Specification (Multiset Rules)

The document self-upload mechanism is modeled using the following rules,

- **The Client_Prepare_Upload** rule in Listing 3 models creation of a fresh local Data Encryption Key (DEK), symmetric encryption of the file contents, and finally the HMAC computation before constructing a upload request.

Listing 3: Client_Prepare_Upload

```
rule Client_Prepare_Upload:
  let
    file_data = <$filename, $content>
    hmac_key = client_secret
    client_dek = ~cdek
    ciphertext = senc(file_data, client_dek)
    hmac = h(<hmac_key, ciphertext>)
    upload_req = <'upload', access_token, $path, hmac, ciphertext>
  in
  [ !Client_Credentials($Client, client_id, client_secret),
    !Access_Token($Client, access_token, locker_id),
    Fr(~upload_id), Fr(~cdek)
  ]
  --[ Client_Upload($Client, ~upload_id, ciphertext, hmac),
      Honest($Client)
    ]->
  [ Upload_Request(upload_req),
    Client_Uploading($Client, ~upload_id, ciphertext,
                  access_token, locker_id)
  ]
```

- The **Pull_URI_Response** rule in Listing 4 models the creation of document when the server responds to the user's request.

Listing 4: Pull_URI_Response

```
rule Pull_URI_Response:
    [ St_Pull_URI_Pending($User, $Issuer, txn, ts, digilockerid, uid,
        docparams)
      , !Issuer_Info($Issuer, issuerid, apikey, sk_issuer)
```

```
      , Pull_URI_Request_Sent(<txn, ts, digilockerid, uid, docparams,
          hmac_received>)
      , Fr(~uri) // Document URI
      , Fr(~doccontent) // Document content
  ]
  --[ Eq(hmac_received, h(<apikey, txn, ts, digilockerid, uid, docparams>))
      , Pull_URI_Responded($Issuer, $User, txn, ~uri)
      , Document_Created($Issuer, ~uri, ~doccontent)
    ]->
  [ !St_URI_Received($User, $Issuer, ~uri,txn)
      , !URI_Mapping(~uri, $Issuer, ~doccontent)
      , Out(<txn, 'status_1', ~uri, ~doccontent>)
  ]
```

- The **Pull_Doc_Response** rule in Listing 5 models the signing of the document before sending response to the user.

Listing 5: Pull_Doc_Response

```
rule Pull_Doc_Response:
   [ St_Pull_Doc_Pending($User, $Issuer, uri, txn_pull, ts_pull,
       digilockerid)
      , !Issuer_Info($Issuer, issuerid, apikey, sk_issuer)
      , !URI_Mapping(uri, $Issuer, doccontent)
      , Pull_Doc_Request_Sent(<txn_pull, ts_pull, uri, digilockerid,
          hmac_received>)
   ]
   --[ Pull_Doc_Responded($Issuer, $User, uri, doccontent)
      , Document_Signed($Issuer, doccontent, sign(doccontent, sk_issuer))
      , Eq(hmac_received, h(<apikey, txn_pull, ts_pull, uri, digilockerid>)
         )
      ]->
   [ St_Doc_Received($User, $Issuer, uri, doccontent, sign(doccontent,
       sk_issuer))
      , Out(<txn_pull, 'status_1', doccontent, sign(doccontent, sk_issuer)
          >)
   ]
```

- The **Verify_Document** rule in Listing 6 verifies the signature of the document received.

Listing 6: Verify_Document

```
rule Verify_Document:
   [ St_Doc_Received($User, $Issuer, uri, doccontent, signature)
   , !Pk($Issuer, pk_issuer)
   ]
   --[ Eq(verify(signature, doccontent, pk_issuer), true)
      , Document_Verified($User, $Issuer, uri, doccontent)
      , Honest($Issuer), Honest($User)
    ]->
   [ St_Doc_Verified($User, $Issuer, uri, doccontent, 'verified')
      , !Verified_Doc($User, uri, doccontent, $Issuer)
   ]
```

All the rules and their implementation are available at https://github.com/nayan-1210/Formal-Verification-of-Digilocker-using-Tamarin-Prover.

4.3.3 Security Properties (Lemmas)

The security properties are modeled in Tamarin. One such is discussed here.

- **Document Integrity:** The `Document_Integrity` lemma in Listing 7 strengthens authenticity by enforcing correct signing order and untampered content. It ensures that if a document is verified, then it must have been created by the issuer before being digitally signed, and that signature must precede the verification event. This establishes a strict temporal chain—creation → signing → verification—thereby guaranteeing that the document content remains unchanged and that the signature corresponds to the originally issued document.

Listing 7: Document_Integrity Lemma

```
lemma Document_Integrity:
    "All User Issuer uri doc sig #i #j.
        Document_Verified(User, Issuer, uri, doc) @i & Honest(Issuer) @i
        & Document_Signed(Issuer, doc, sig) @j
        ==>
            (Ex #k. Document_Created(Issuer, uri, doc) @k
            & #k < #j & #j < #i)"
```

Note that we have discussed modeling of only one of the component of DigiLocker as discussed above. The modeling of other components and security properties such as *Document_Integrity, Document_Secrecy, Hmac_Authentication, User_Authentication* and their implementation are available at https://github.com/nayan-1210/Formal-Verification-of-Digilocker-using-Tamarin-Prover.

5 Results and Analysis

Our formal analysis using the Tamarin Prover in two assumptions resulted in distinct sets of results: verification of the protocol under standard, ideal assumptions, and the evaluation of the protocol's resilience under controlled injection of vulnerability.

5.1 Verification of Standard Protocol Design

As DigiLocker adopts standard, well-established security frameworks (such as OAuth 2.0 with PKCE and HMAC-based API authentication), the initial model did not naturally reveal any vulnerabilities during normal execution. But when we executed the Tamarin model under the assumption that long-term secrets (such as the issuer's API keys and the user's `digilockerid`) remain perfectly confidential, Tamarin successfully verified all primary security properties in first attempt.

Properties concerning document authenticity, integrity, and confidentiality were mathematically proven. Specifically, Tamarin verified the `No_Attacker_Pull_URI` lemma in Listing 8, showing that an attacker cannot forge a valid document retrieval request without getting access to the correct issuer API key.

Listing 8: No_Attacker_Pull_URI

```
lemma No_Attacker_Pull_URI:
  "not (Ex attacker Issuer txn docparams #i.
        Pull_URI_Requested(attacker, Issuer, txn, docparams) @i &
        Attacker_Forged_Pull_URI(txn) @i)"
```

5.2 Injection of a Controlled Vulnerability

To analyze the impact of sensitive information leakage, we deliberately injected controlled vulnerabilities into the model. We introduced "leak rules" (Listing 9) designed to simulate a real-world compromise, such as an insider threat or a misconfigured server exposing the issuer's API key and a user's `digilockerid` to the open network.

Listing 9: Leak rules

```
rule Leak_API_Key:
  [ !Issuer_Info(Issuer, issuerid, apikey, sk_issuer) ]
  --[ API_Key_Leaked(Issuer) ]->
  [ Out(apikey) ]

rule Leak_DigiLocker_ID:
  [ !User_Info(User, digilockerid, uid) ]
  --[ DigiLocker_ID_Leaked(User) ]->
  [ Out(digilockerid) ]
```

Observations. Figure 6 illustrates the dependency graph of the counter-example generated by Tamarin, which falsifies the No_Attacker_Pull_URI lemma. The trace reveals a multi-stage attack facilitated by the compromise of long-term credentials.

- **Credential Leakage:** The attack begins with the firing of the compromise rules, `#vr.2` (Compromise_API_Key) and `#vr.3` (Leak_DigiLocker_ID). These rules place the sensitive `apikey` and `digilockerid` onto the public network.
- **Adversary Derivation:** The nodes, `#vk` and `#vk.1` represent the adversary's knowledge derivation. The adversary intercepted the leaked terms from the network. The coerce and isend labels show the adversary's capability to manipulate these terms and prepare them as input for malicious requests.
- **Lemma Violation:** The presence of Attacker_Forged_Pull_URI(txn) action label at the final timepoint provides the formal proof that the lemma has been violated.

6 Conclusion

In this paper, we presented a comprehensive formal security analysis of the DigiLocker protocol using the Tamarin Prover. By systematically modeling the protocol's core workflows ranging from PKCE-secured OAuth 2.0 authentication to complex, document retrieval and document self-upload mechanisms, we established a mathematical foundation for evaluating its security claims. The complete implementation details can be found on Github at https://github.com/nayan-1210/Formal-Verification-of-Digilocker-using-Tamarin-Prover.

Our automated verification confirms that under standard assumptions, DigiLocker provides robust guarantees of document authenticity, integrity, token secrecy, and confidentiality against a Dolev-Yao active network adversary. However, our controlled vulnerability injection also shows that the security of this infrastructure is heavily dependent on keeping credentials safe. Our attack traces demonstrate that any leakage of API key or and a user's `digilockerid` can cause loss of document confidentiality which allows an attacker to forge requests. These insights emphasize the importance of continuous security auditing in DPIs.

With increasing focus on interconnected digital ecosystems, scalable formal verification has become extremely important to reduce frauds. Future work in this field should focus

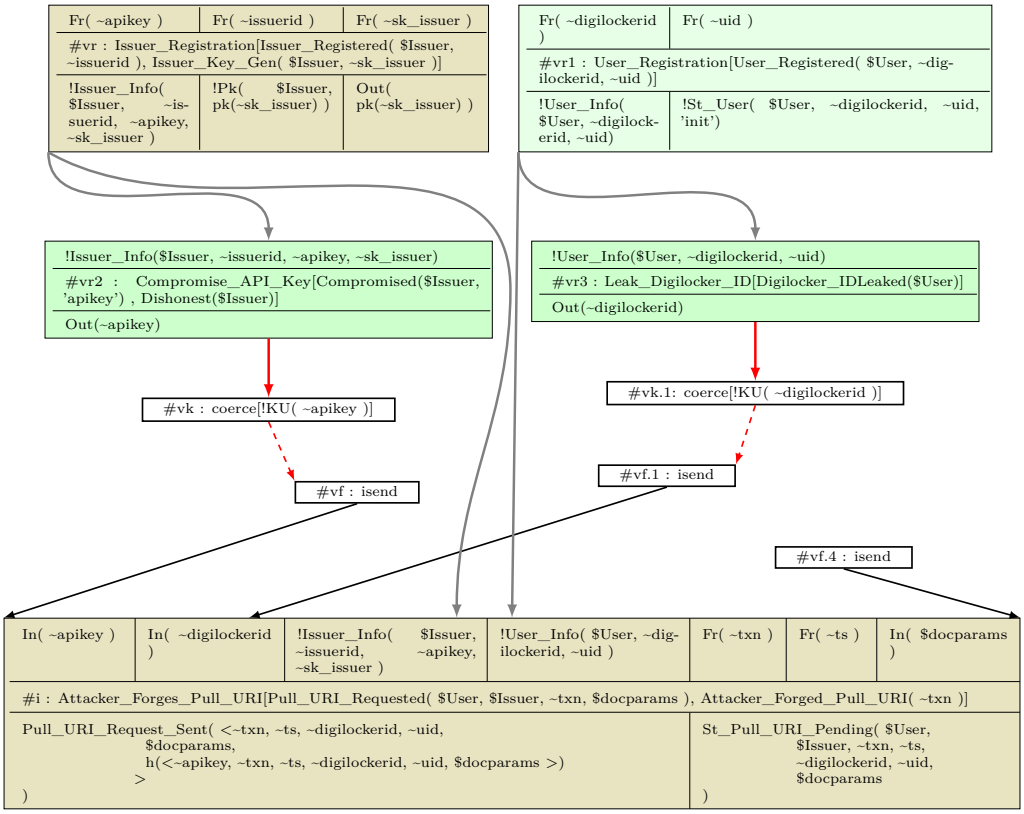

Figure 6: Trace in Pull_URI Workflow

on applying this methodology and extending our symbolic analysis to other important DPI systems, such as the Aadhaar authentication framework. Significant work also needs to be done in verifying security of underlying architecture and not just protocol.

Furthermore, since writing formal models for large systems is complex and takes too much time, we must explore the usage of Artificial Intelligence and Large Language Models (LLMs) in this field. They can be trained to autonomously generate Tamarin protocol rules directly from API specifications and documentation. This will significantly accelerate the process of security verification for large e-governance platforms.

References

[ABS18] R. Abraham, C. J. Bennett, and I. Sen. Aadhaar: Digital identity and socio-economic empowerment in India. *Information Technology for Development*, 24(4):694–713, 2018.

[BCDS26] David A. Basin, Cas Cremers, Jannik Dreier, and Ralf Sasse. *Modeling and Analyzing Security Protocols with Tamarin - A Comprehensive Guide*. Information Security and Cryptography. Springer, 2026.

[BDH+18] David Basin, Jannik Dreier, Lucca Hirschi, Sasa Radomirovic, Ralf Sasse, and Vincent Stettler. A formal analysis of 5G authentication. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1383–1396, 2018.

[Bla14] Bruno Blanchet. Automatic verification of security protocols in the symbolic model: The verifier ProVerif. In *Foundations of Security Analysis and Design VII*. Springer, 2014.

[CHSvdM17] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1773–1788, 2017.

[D+18] N. Diallo et al. Blockchain and smart contracts for e-government applications. In *IEEE International Conference on E-business Engineering (ICEBE)*, pages 254–259, 2018.

[Dwo07] Morris Dworkin. Recommendation for block cipher modes of operation: Galois/counter mode (GCM) and GMAC. Technical Report SP 800-38D, National Institute of Standards and Technology (NIST), 2007.

[DY83] Danny Dolev and Andrew Yao. On the security of public key protocols. In *IEEE Transactions on Information Theory*, volume 29, pages 198–208. IEEE, 1983.

[FKS16] Daniel Fett, Ralf Küsters, and Guido Schmitz. A comprehensive formal security analysis of OAuth 2.0. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1204–1215, 2016.

[KBC97] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. RFC 2104, RFC Editor, 1997.

[LBLF23] Torsten Lodderstedt, John Bradley, Andrey Labunets, and Daniel Fett. OAuth 2.0 security best current practice. Internet-draft, Internet Engineering Task Force, 2023.

[LBLF24] T. Lodderstedt, J. Bradley, A. Labunets, and D. Fett. Best current practice for OAuth 2.0 security. RFC 9700, Internet Engineering Task Force (IETF), 2024.

[LSB25] Felix Linker, Ralf Sasse, and David Basin. A formal analysis of Apple's iMessage PQ3 protocol. In *34th USENIX Security Symposium (USENIX Security 25)*. USENIX Association, 2025.

[Mina] Ministry of Electronics and Information Technology, Government of India. DigiLocker authorized partner API specification v1.11. https://img1.digitallocker.gov.in/assets/img/Digital%20Locker%20Authorized%20Partner%20API%20Specification%20v1.11.pdf.

[Minb] Ministry of Electronics and Information Technology, Government of India. DigiLocker ecosystem. https://www.digilocker.gov.in/web/ecosystem.

[Minc] Ministry of Electronics and Information Technology, Government of India. DigiLocker issuer API specification v1.12. https://img1.digitallocker.gov.in/assets/img/issuer_api/Digital%20Locker%20Issuer%20API%20Specification%20v1.12.pdf.

[Min15] Ministry of Electronics and Information Technology, Government of India. Digital India: A programme to transform India into a digitally empowered society and knowledge economy. https://www.digitalindia.gov.in/, 2015.

[Min23] Ministry of Electronics and Information Technology, Government of India. DigiLocker technology specification ver 2.3. https://img1.digitallocker.gov.in/assets/img/technical-specifications-dlts-ver-2.3.pdf, 2023.

[Min24] Ministry of Electronics and Information Technology, Government of India. DigiLocker requester-entity locker API specification. https://entity.digilocker.gov.in/assets/img/Requester%20-%20Entity%20Locker%20API%20Specification_28_11_24.pdf, 2024.

[MSCB13] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *Computer Aided Verification (CAV)*. Springer, 2013.

[OWA23] OWASP Foundation. OWASP API security top 10. https://owasp.org/API-Security/editions/2023/en/0x11-t10/, 2023.

[pro] ProVerif package on OPAM. https://opam.ocaml.org/packages/proverif/.

[S+21] V. Sharma et al. Secure data exchange in e-governance using trusted execution environments. In *ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 111–122, 2021.

[SBA15] N. Sakimura, J. Bradley, and N. Agarwal. Proof key for code exchange by OAuth public clients. RFC 7636, RFC Editor, 2015.

[Tam] Tamarin Prover Team. Tamarin prover documentation. https://tamarin-prover.com/.

[TC20] R. Thomas and A. Chatterjee. The digital financial inclusion in India: An impact analysis of UPI. *Journal of Digital Banking*, 5(2):145–158, 2020.

[Uni22] Unique Identification Authority of India (UIDAI). Aadhaar authentication API specification 2.5, revision 1. https://uidai.gov.in/images/resource/Aadhaar_Authentication_API-2.5_Revision-1_of_January_2022.pdf, Jan 2022.

[Wor23] World Bank Group. Digital public infrastructure: The India story. Technical report, World Bank, Washington, DC, 2023.