Securing Millions of Decentralized Identities in Alipay Super App with End-to-End Formal Verification

Ziyu Mao
Zhejiang University, Ant Group
Hangzhou, China
maoziyu@zju.edu.cn

Xiaolin Ma
Zhejiang University
Hangzhou, China
XiaolinMa@zju.edu.cn

Lin Huang
Ant Group
Beijing, China
linyu.hl@antgroup.com

Huan Yang
Ant Group
Hangzhou, China
qiaoyi.yh@antgroup.com

Wu Zhang
Ant Group
Hangzhou, China
jeff.zw@antgroup.com

Weichao Sun
Ant Group
Beijing, China
tudou.swc@antgroup.com

Yongtao Wang
Ant Group
Beijing, China
mengju@antgroup.com

Jingling Xue
UNSW Sydney, Ant Group
Sydney, Australia
j.xue@unsw.edu.au

Jingyi Wang*
Zhejiang University
Hangzhou, China
wangjyee@zju.edu.cn

***Abstract*—Decentralized Identity (DID) enhances authentication and privacy by empowering individuals to control their own digital identities, which has gained traction globally. To our knowledge, this paper presents the first end-to-end verification effort (from design to implementation) of a real-world Decentralized Identity (DID) protocol following the IIFAA DID standard, which has been deployed within the widely used super app Alipay and issued millions of DIDs in practice. We integrate formal verification into the development lifecycle of such industrial security protocol to systematically enhance its reliability from two levels: (1) At the design level, we utilized state-of-the-art protocol design verifier TAMARIN to formally model the IIFAA DID standard under a realistic threat model tailored for super apps. We then formulated and performed automated verification of desired security properties using TAMARIN. We identified several design flaws that could lead to a security breach. These issues were reported to the design team and have been addressed in the updated design. (2) At the implementation level, we first extract the desired specification derived from the verified symbolic model of protocol design in the form of a set of intermediate I/O specifications. Subsequently, we translate the I/O specifications into a set of functional specifications at the implementation level, which can then be verified by the automated tool VERIFAST. We identified several inconsistencies between the implementation and the verified design which are fixed by the development team and led to verified implementation faithfully obeying the verified design, together offering an end-to-end verified secure DID protocol in Alipay super app. Our work showcases how an industrial security protocol development team can design and implement a practical verified secure Decentralized Identity (DID) protocol with the help of end-to-end formal verification.**

***Index Terms*—Formal method, Security protocol, DIDs**

I. INTRODUCTION

Decentralized Identity (DID) enhances authentication and security by shifting control from centralized entities to individuals, ensuring greater privacy and autonomy. This model relies on two principles: (1) users independently manage their cryptographic keys, eliminating third-party reliance, and (2) secure verification occurs through *verifiable credentials* (`VCs`) [13] issued by trusted issuers and *verifiable presentations* (`VPs`) [41] enabling selective disclosure. A DID document serves as a unique identifier, functioning without centralized registries and promoting user-centric identity management. Since 2017, real-world DID adoption has gained traction globally, with pilot application and standardization efforts such as Microsoft's Entra Verified ID [32], BSN DID in China [12], and the European Digital Identity Regulation [1].

Mobile-based services, especially within super apps like Alipay and WeChat, are rapidly growing. By hosting diverse services via mini-programs, these apps enable seamless interactions and robust private identity management, making them well-suited for DID integration. Notably, Alipay and WeChat each serve over one billion active users, with Alipay reportedly reaching nearly 1.4 billion users worldwide as of 2025. Their extensive user base and ecosystem integration make super apps a highly impactful and scalable environment for deploying real-world DID protocols. In a mobile application context, the user of a super app (e.g., Alipay) can act as the Holder. The Holder's long-term keys are generated and securely stored within the mobile device's trusted execution environment (TEE). The Issuer, such as a bank or government agency, performs identity proofing and issues a `VC`, which is kept locally on the phone. When a service requires verified attributes, the Holder creates a `VP` for the Verifier, disclosing only the necessary information.

IIFAA [27], a trusted authentication alliance in China, has played a key role in securing DID implementations. Its technical standard, compatible with TEE and Secure Elements (SE), can adopt across 1.6 billion devices, 43 phone brands, and 900 models worldwide [29]. This wide hardware adoption makes

* Corresponding author

the IIFAA DID standard particularly suitable for integration into super apps at scale. Motivated by its wide deployment and practical relevance, *we design and implement the DID protocol within Alipay Super App according to the IIFAA DID standard*.

On the other hand, as DID adoption grows, security concerns (e.g., protecting users' personal data [18]) are increasingly relevant, especially as standards continue to evolve. Protocols like DIDComm v2.0 [19] enhance encryption and messaging, while OIDC4VC [13] expands support for credential issuance and selective disclosure. However, fragmented DID standards [11] pose security challenges. To address these, the W3C Threat Modeling Community Group (TMCG) launched in July 2024 to develop threat models. This document [42] applies Adam Shostack's four-question framework [38] for high-level security analysis. While insightful, these efforts provide only a broad overview, lacking the depth for systematic vulnerability assessment in our context.

Industry security audits rely on established methodologies and expert reviews. To improve rigor and consistency, we conduct the first in-depth formal analysis of a real-world IIFAA application, focusing on its mobile-based integration within Alipay. Using *symbolic analysis*, we clarify informal DID specifications, identify implicit security assumptions, and verify claimed security properties, despite the IIFAA standard's informal, native-language format, similar to other standards [1], [12], [32]. Moreover, as noted in [42], the threat model is a living document, requiring security analysis tailored to specific contexts. Therefore, we develop a super-app-specific threat model to ensure a more precise and relevant assessment. Furthermore, we also verified the consistency of low-level code implementation with the design-level protocol model, offering verified security assurance at both levels with end-to-end formal verification.

Recent studies have applied formal methods to analyze the security of DID-related protocols. Hauck [26] examined OpenID for `VC` Issuance [13] and `VPs` [41] using the Web Infrastructure Model (WIM). However, this pen-and-paper analysis relies on expertise, lacks automation, and covers limited aspects of communication protocols in DID applications. Badertscher et al. [6] conducted the first formal analysis of DIDComm's cryptographic foundations, confirming its anonymity and authenticity while proposing protocol improvements. Braun et al. [11] analyzed a self-constructed protocol based on SSI specifications. While comprehensive in modeling a DID application's workflow, it remains abstract, relying on a limited Dolev-Yao threat model [21] and overlooking the varied ways `VPs` can be derived from `VCs`, potentially missing real-world attack vectors.

This work bridges the gap between formal analysis and practical implementations by systematically analyzing IIFAA DID [27] within Alipay using TAMARIN [31] while preserving fine-grained details in our model. Two key challenges arise: (1) Like other DID standards [1], [12], [32], IIFAA DID lacks a detailed specification, providing only a high-level workflow [27], which complicates comprehensive modeling,

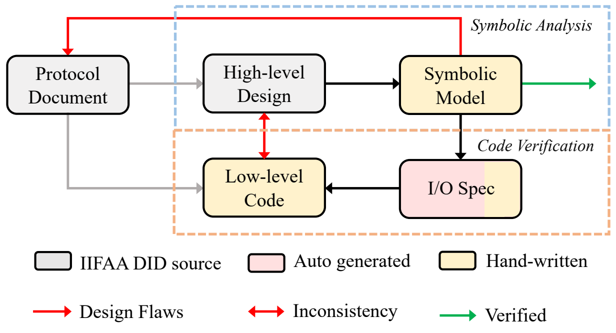


Fig. 1: Overview of our approach

(2) The complexity of super apps like Alipay poses significant challenges to both formal modeling and verification in TAMARIN, and (3) to enable code-level verification, we have to translate the security requirements in TAMARIN symbolic model to functional specifications in Java code to check the consistencies between design and implementation.

Our Approach. To address the above challenges, we apply an interactive two-level approach, which is showed in Figure 1[1]. First, we align the IIFAA standard [27] with customized design considerations in Alipay, adding the necessary details to construct a comprehensive formal model. Security goals are defined using first-order logic (FOL). We then guide TAMARIN's reasoning using auxiliary lemmas and oracles to enable efficient verification. Once potential design vulnerabilities are identified, they are reported back to the design team until all the design goals are met through formal verification. Next, to ensure the consistency between code implementation and protocol design model, which is critical to ensure the security properties can be inherited from symbolic design model, we apply code-level verification using VERIFAST against a set of functional specifications derived from intermediate I/O specification [5] extracted from the symbolic model.

Contributions. Super apps, likened to operating systems by Zhang et al. [44], integrate multiple services seamlessly. DID enables a unified, secure identity across these services. Within super apps, access control and cryptographic protocols interact with the DID infrastructure on behalf of mini-programs, introducing new security implications and the need for a dedicated threat model. This paper presents the first symbolic model capturing super app and mini-program features under a realistic threat model, enabling a systematic analysis of DID security risks. To our knowledge, this is the first effort in Alipay Super App which utilized end-to-end formal verification to the design and implementation of a practical security protocol serving millions of end users. Specifically, we make the following contributions:

1) **Symbolic Analysis of High-level Design.** We develop a symbolic model of IIFAA DID [27] in Alipay, capturing key super app features under a tailored threat model. By

[1]The style of this figure is motivated by [40]

defining key security goals, we conduct symbolic analysis using TAMARIN prover to evaluate its security.

2) **Verification of Low-level Implementation.** We validate the conformance of the implementation to the symbolic design using a top-down verification approach [5]. Specifically, we derive a set of I/O-level specifications from the symbolic model and verify the concrete implementation using VERIFAST, uncovering several inconsistencies between design and code.
3) **Security Risks, Recommendations, and Insights.** Our findings provide recommendations, key lessons, and insights into strengthening DID security in real-world deployments and how end-to-end formal verification can be integrated into the development lifecycle of a practical security protocol.

II. Background

We provide an overview of DID systems, then discuss super apps and their integration with DID technologies.

A. Decentralized Identities

A *DID Document* is a key component of decentralized identity, containing *claims* (attributes), cryptographic keys, authentication protocols, and service endpoints. These elements enable identity verification and secure communication.

A DID system facilitates credential issuance, authentication, and verification among three roles: *holder*, *issuer*, and *verifier*. The issuer issues a *verifiable credential* (VC) to the holder, who later generates a *verifiable presentation* (VP) for the verifier. A key feature, *selective disclosure*, allows the holder to share only necessary claims while protecting other personal information.

A VC is a cryptographically secured digital certificate linking a holder to verified attributes. It contains three components: (1) Meta (metadata like validity dates), (2) Claims (verifiable information), and (3) $\texttt{proof}_{\texttt{VC}}$ (a cryptographic proof ensuring integrity and issuer authenticity).

A VP is a cryptographically signed document compiled by the holder, containing one or more VCs. It proves credential possession while enabling selective disclosure. Like a VC, it includes $\texttt{proof}_{\texttt{VP}}$ to ensure integrity and authenticity, allowing verifiers to trust the claims without contacting the issuer.

B. Super Apps

A *super app* like Alipay hosts multiple mini-programs, offering diverse services through a unified interface. It consists of a frontend client on the phone and a backend server operated by the super app developer, while each mini-program has its own frontend client and remote backend server. In a DID protocol within a super app, the holder, issuer, and verifier roles map to the super app client (securely managing private keys and credentials in trusted environments like TEE), the mini-program client, and the mini-program server, respectively.

Consider a mini-program on Alipay. The four entities interact as follows:

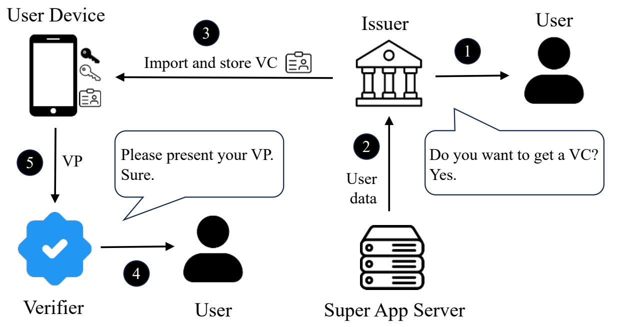


Fig. 2: An user case study

1) **Alipay Client** manages mini-program lifecycle, integrity, authentication, and permissions, providing APIs for interaction with Alipay's lower-layer services [43].
2) **Alipay Server** handles user management, API interactions, and service implementation for the Alipay client.
3) **Mini-Program Client**, developed using web technologies, interacts with the Alipay client via Javascript API (JSAPI [4]) and communicates with its server over TLS.
4) **Mini-Program Server**, maintained by its developer, processes business logic and data while adhering to Alipay's security standards. It communicates with the Alipay server via OpenAPI [3], enforcing mutual message signing for authentication and data integrity.

Mini-program integrity is ensured through signature verification. Upon release, the platform signs the mini-program code and configuration file (generated by the Alipay Server). The Alipay client verifies this signature before execution.

JSAPI access control rules are defined in the mini-program's configuration file. DID-related JSAPIs or plugins require explicit permission, granted only to enterprise mini-programs after a business contract. As a result, multiple mini-programs can access DID services on a holder's device, but the JSAPI link lacks additional encryption and integration protection.

III. Design and Implementation

This section details the specific workflow of the DID protocol within Alipay, elucidating its operational mechanics and its application by users of the super app through a real world use case. Subsequently, we present the code-level implementation of the DID protocol, outlining the requisite software modules for the Issuer, Verifier, and Holder roles.

A. Protocol Flow

To help readers better understand how the DID protocol operates within a real-world super app (i.e., Alipay), we illustrate a typical workflow using a mini-program for identity authentication as an example.

As shown in Figure 2, the process begins with the user initiating a request to obtain a VC by interacting with the Issuer, such as a government agency or financial institution. The super app server, through an OAuth process, transmits basic user data to the Issuer for verification. Once verified, the Issuer generates a VC and delivers it to the user device, where it is securely stored using trusted hardware components such as a secure enclave or TEE. Subsequently, when the user accesses a service that requires identity proof, the verifier sends a request through a mini-program interface. In response, the user device generates a VP, selectively disclosing the necessary identity attributes, and sends it to the Verifier. The detailed protocol workflow is shown in Figure 3.

Mini-Program Initialization. The protocol begins with the initialization of a mini-program within the user's Alipay app. Each mini-program is uniquely registered on the Alipay platform and assigned an `AppID` [4], which serves as its global identifier within the ecosystem. When a user uses a mini-program, Alipay assigns a user-specific identifier, `openid` [4], which is scoped to that mini-program. The mini-program developer can only see this `openid`. On the Alipay platform backend, the `openid` has a trusted mapping relationship with the Alipay user's real identity. This establishes a trusted mapping between the mini-program instance and the user, forming the foundation for subsequent DID operations such as DID key generation, credential issuance, and verification.

DID Creation. When a user first uses DID, the user must create DID. A pair of DID public and private keys are generated in the secure area (e.g., TEE) on the user's device. The private key is stored in the secure area, and the public key is returned to the client. The client sends the DID public key to the IIFAA server, which generates a DID document. The information of this DID document is then placed on the blockchain, completing the DID registration.

VC Issuance. As shown in Figure 3a, the Alipay user requests a VC via the mini-program frontend and then saves the issued VC on the client side. In this step, the Issuer needs to implement: (1) verifying the holder's claims, and (2) constructing the VC. User information is stored on the Alipay server. The Issuer, through an OAuth 2.0 [25] authorization flow and with user consent, obtains trusted user data (e.g., name and mobile number). This saves the user from the trouble of logging into the Issuer's system and manually entering personal information. The Issuer generates claims about the user based on the user information, then constructs a VC according to the W3C standard [18]. We model SD-JWT [22], which enables selective disclosure in a JSON-encoded structure within a JWS payload, aligning with standard [27].

As shown in Figure 3a, the "VC Construction" process works as follows: $\mathtt{C} = \{c_1, \ldots, c_n\}$ denotes all claims available, `digest` represents the concatenation of their hashes, and `sigJWT` is the concatenation of `digest` and its issuer-signed value using the private key ($\mathtt{sk_I}$). For secure transmission to the Alipay client, the VC is encrypted with an envelope key. Detailed message formatting is provided in our formal model [2].

SD-JWT Data Format. SD-JWT [22] is a composite JSON-encoded JWS structure enabling selective disclosure. By separating claims during VC issuance (as shown in the "VC Construction" process in Figure 3a), its signature proof remains valid even when some claims are selectively removed.

VP Verification. As shown in Figure 3b, a VP verification request originates from the mini-program frontend. The Alipay client, as the holder, constructs VP using a single VC stored in TEE on the device. During "VP Construction", the holder selects claims $\mathtt{D} = \{c_{i_1}, ..., c_{i_k}\}$ from $\mathtt{C} = \{c_1, ..., c_n\}$, excluding private claims ($\mathtt{C} \setminus \mathtt{P}$), to form VC'. The holder signs VC' with their private key ($\mathtt{sk_H}$), generating `sigH` as proof of origin. The verifier checks `sigH` with the holder's public key ($\mathtt{pk_H}$), verifies `sigJWT` using the issuer's public key ($\mathtt{pk_I}$) for authenticity, and confirms D matches its hash in `digest`.

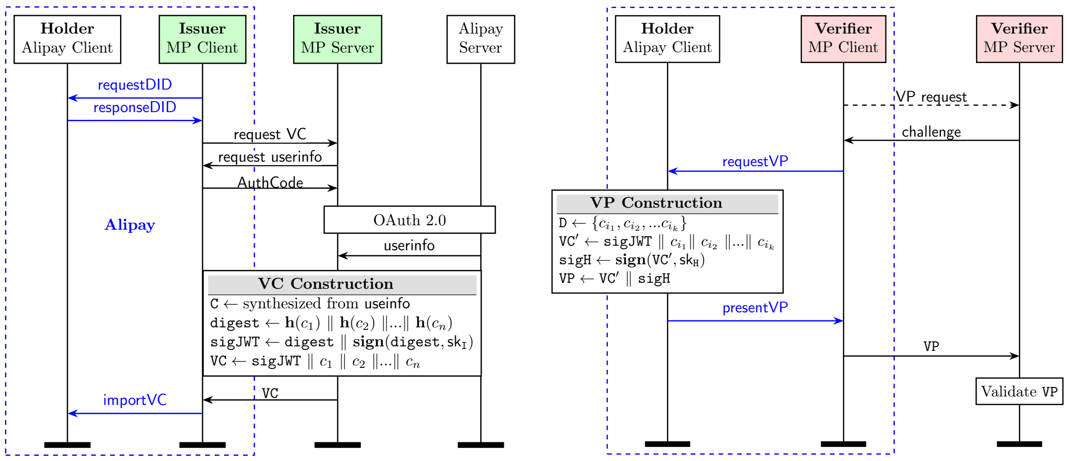
B. Code Implementation of the Protocol

The implementation of DID applications encompasses the functionalities of three distinct roles: Issuer, Holder, and Verifier. Both the Issuer and the Verifier consist of a server-side and a client-side component. The server side, implemented in Java, handles the DID protocol flow. The client-side, specifically a mini-program, uses JavaScript to build the front-end user interface. To facilitate rapid application development for Issuers and Verifiers on the IIFAA DID platform, Alipay provides a DID mini-program plugin. This plugin encapsulates the interfaces for fundamental DID capabilities, which can then be invoked by the Issuer and Verifier mini-programs.

The software architecture for the Holder role is multi-layered: The uppermost layer is the mini-program functional layer, comprising mini-programs and mini-program plugins implemented in JavaScript. Below this is the native app layer, with distinct implementations for various operating systems such as Android, iOS, and HarmonyOS. Taking the Android system as an example, this layer implements the DID protocol flow using Java. Some basic cryptographic libraries are implemented in C code and encapsulated within the IIFAA SoftTEE SDK. The next layer down is the operating system layer, which provides implementations for network library functions (e.g., HTTPS). Below the operating system, there are TrustZone (TEE) and Secure Element (SE) modules, whose firmware is implemented by mobile phone manufacturers and chip vendors. In TEE there is an IIFAA SDK module, a result of many years of collaborative effort by the IIFAA Alliance with over 300 member organizations. As of now, the IIFAA SDK has been pre-installed in the TEE of 1.6 billion mobile phones during manufacture. The IIFAA SDK also contains basic cryptographic library functions, providing a secure environment and a unified interface for DID-related key generation, storage, and operations. This interface is then exposed to the mini-program client after multilayer encapsulation through the operating system, the native app layer, and the DID mini-program plugin.

For all IIFAA applications, cryptographic primitives from SE, TEE, or SoftTEE can be selected and controlled via parameters such as *security level*.

For the code-level verification in this paper, our focus is on verifying the Java code implementations for all three roles.



(a) VC Issuance

(b) VP Presentation

Fig. 3: IIFAA DID: VC issuance and VP verification phases. MP Client and MP Server stand for Mini-Program Client and Mini-Program Server, respectively. JSAPI calls and TLS messages are represented by blue and black arrows.

IV. Symbolic Analysis of Protocol Design

We outline the DID protocol [27] and its abstraction, under the threat model of mini-programs, in TAMARIN [31]. The full TAMARIN model is available at [2]. Based on the TAMARIN model, the security goals and properties are analyzed.

A. *The* TAMARIN *Prover*

The TAMARIN Prover is an automated tool designed for symbolic protocol analysis. It offers automated and interactive proof strategies, enabling users to visually inspect proof states and examine counterexamples through dependency graphs. State-of-the-art features, such as subterm-based proofs [17], make TAMARIN well-suited for analyzing modern protocols.

In symbolic analysis, messages are abstracted as terms. Cryptographic primitives are modeled as a set of *perfect functions* [15]. The semantics of these function symbols are defined by equational theories. For instance, the equation $\mathsf{sdec}(\mathsf{senc}(n, k), k) = n$ illustrates the decryption of a symmetrically encrypted message. Here, senc denotes the encryption of a message n with key k, and $\mathsf{sdec}(c, k)$ represents the decryption of ciphertext c with key k.

In TAMARIN, the symbolic model of a protocol is encoded as a set of *multiset rewriting rules (MSRs)*, typically represented in the form [l] − [a] → [r]. In this notation, l, a, and r denote multisets of *facts* that capture the protocol's state during execution. Intuitively, applying an MSR involves consuming the facts in l and producing the facts in r, while the facts in a annotate execution traces without affecting the state transition itself. Facts in TAMARIN are classified into two types: *linear* facts and *persistent* facts (denoted by the prefix !). Linear facts can represent limited resources, which are consumed upon use, whereas persistent facts model enduring entities (e.g., a long-term private key) that remain available throughout the protocol's execution. To model the installation of a super app on a phone, we can use the rule below:

```
rule InstallSuperAppClient:
   [ Fr(~userid), !Phone($Name, $Number) ]
   --[ InstallApp($Name, ~userid)]->
   [ !SuperApp($Name, $Number, ~userid) ]
```

Intuitively, the rule states that an client !SuperApp is installed on a end device !Phone, where the unique user ID is a fresh nonce ~userid. TAMARIN encodes desired properties intended to verify by *first-order logic (FOL)* formulas over action facts and *timepoints* as lemmas. Consider a simple secrecy property secrecyOfID related to the rule:

```
lemma secrecyOfID:
  "All name userid #i. InstallApp(name, userid)@i
   ==> not (Ex #j. K(userid)@j)"
```

Intuitively, the FOL states that For any installation of SuperApp at timepoint $\#i$, there does not exist a timepoint #j that the adversary can know the term userid. Here the symbol $\Longrightarrow$ denotes implication, while $\#i$ and $\#j$ are temporal variables. The fact $\mathtt{K}(t)$ is a built-in predicate in TAMARIN, indicating that the adversary knows the term t. By default, TAMARIN assumes a Dolev-Yao adversary model [21], allowing the adversary to inject, modify, and intercept messages on the network. If we replace the keywords lemma with restriction, the state space of the formal will be pruned, only preserving the traces that are satisfied with the restriction FOL. We refer the reader to manual [31] and textbook [7] for more details about TAMARIN.

B. *Threat Model*

Our threat model, tailored for super apps, extends the Dolev-Yao model [21], where an adversary can intercept, modify, and inject messages over public channels. Additionally, we consider:

(1) **Hooks in Mobile Apps.** Clients are inherently insecure and may be compromised, exposing JSAPI calls between mini-programs and the super app.

(2) **Private Key Leakage.** Many mini-programs hardcode their private keys (or master keys in WeChat [44]) in their frontends, risking exposure. Similarly, DID secret keys may also be leaked.

IIFAA DID [27] is a high-level protocol relying on TLS and OAuth [25]. Following [30], we assume such low-level protocols are secure, and adversaries are constrained by cryptographic primitives. In addition, TLS sessions and the super app server remain uncompromised. To manage state space explosion while preserving core DID aspects, we assume: (**A1**) A mini-program acts as either an issuer or a verifier. (**A2**) Each set of claims results in a single `VC` per holder, as we focus on the common case where each `VP` derives from a single `VC`, simplifying analysis. (**A3**) Only legitimate holders can request `VC`s, preventing unauthorized issuance. Note that our model, however, allows an issued `VC` to generate all possible `VP` combinations for a given verifier.

C. Protocol Modeling

Like other DID standards [1], [12], [32], IIFAA DID [27] is informal and written in a natural language. Figure 3 outlines its implementation in Alipay for a specific mini-program on the platform. We distilled the protocol into two sub-protocols—`VC` *Issuance* and `VP` *Verification*—by manually reviewing the IIFAA standard [27] and Alipay implementation documents.

As described in Section II-B, the Alipay client acts as the holder, managing private keys and credentials in a trusted environment (e.g. TEE). The mini-program's client (frontend) and server (backend server) function as the Issuer for `VC` issuance and the Verifier for `VP` verification. Message exchanges primarily rely on JSAPI calls (blue arrows in Figure 3) and TLS (black arrows).

The formal model consists of 33 rules covering the following aspects of IIFAA: (1) initialization (3 rules), (2) `VC` Issuance (9 rules), (3) `VP` Verification (3 rules), (4) SD-JWT Data Format (2 rules), (5) TLS and JSAPI Calls (14 rules), and (6) Adversary Capabilities (10 rules). Some rules overlap across aspects, leading to a total exceeding 33, and rule count does not directly indicate modeling complexity.

We model a term structure in TAMARIN using the tuple operator $\langle\rangle$ and multiset union $+$. The associative and commutative properties of $+$ enable selective disclosure modeling via the subterm relation $\sqsubset$ [17].

D. Security Goals

Security requirements in DID systems have been informally discussed across various ecosystems [1], [12], [27], [32]. TMCG [42] identifies risks through high-level *evil stories*, such as a verifier over-collecting data from holders or an issuer tracking holders. In a recent work, Braun et al. [11] formalize authentication lemmas for requirements like *the holder presented a credential to the intended verifier* but lack clear mappings to security properties like secrecy or integrity.

To address this, we clarify the security requirements, Section 5.2 Data Security in IIFAA standard [28], with FOL formulas in TAMARIN [31] for 1 executability goal and 5 security goals, supporting the formal analysis of IIFAA in Alipay. The eight supporting lemmas are detailed in our open-source artifact [2].

Executability. We validate our model (Section IV) by defining a lemma that ensures legal executions without an adversary, preventing modeling errors.

Goal 1. Secrecy. *Undisclosed data and issued credentials must remain private.* We define secrecy as ensuring that private claims are not revealed, even to honest verifiers or third parties. Our formalization guarantees that undisclosed data and credentials are inaccessible to the Dolev-Yao adversary [21]. This property is formally defined below, styled after the approach in [9]:

$$\textbf{For any protocol's trace } \alpha: \quad \forall\, uid,\ \texttt{VC},\ t_1.\ \mathsf{StoreVC}(uid, \texttt{VC}) \in \alpha_{t_1} \implies \neg(\exists\, t_2, uid.\ \mathsf{K}(\texttt{VC}) \in \alpha_{t_2})),$$

where α_t denotes the set of action events that occur at time t in trace α. Additionally, to preserve the maximal privacy of the user, IIFAA DID supports Zero-Knowledge Proof (ZKP) techniques, to construct `VC` and `VP`.

Goal 2. Integrity. *A credential must originate from the intended issuer and remain unmodified.* This ensures that an issued credential cannot be forged or altered. Formally, we model integrity as non-injective authentication.

$$\forall\ \texttt{VC}, v, id, t_1.\ \mathsf{StoreVC}(uid, \texttt{VC}) \in \alpha_{t_1} \implies (\exists\, t_2, aid.\ \mathsf{IssueVC}(aid, \texttt{VC}) \in \alpha_{t_2} \land t_2 < t_1))$$

Goal 3. Injective Agreement. *Each verifier-approved* `VP` *should correspond to a unique* `VP` *request.* This prevents replay attacks and ensures that only a legitimate holder can generate a valid `VP`.

$$\begin{aligned}\forall\ \texttt{VP}, v, id, t_1.\ &\mathsf{PassVerify}(aid, \texttt{VP}) \in \alpha_{t_1}\\ &\implies ((\exists\, t_2, uid.\ \mathsf{HolderPresent}(uid, \texttt{VP}) \in \alpha_{t_2} \land t_2 < t_1)\\ &\land \neg((\exists\, t_3, aid'.\ \mathsf{PassVerify}(aid', \texttt{VP}) \in \alpha_{t_3}) \land (t_1 \neq t_3))\\ &\land (aid \neq aid'))\end{aligned}$$

Goal 4. Legal Presentation. *The "wallet" presenting a* `VC` *must be the one to which it was issued.* This requirement strengthens authentication by ensuring a one-to-one mapping between a `VC` and its designated holder, preventing unauthorized use of issued credentials.

$$\begin{aligned}\forall\ user, t_1.\ &\mathsf{GetRights}(user) \in \alpha_{t_1}\\ &\implies (\exists\, t_2, uid.\ \mathsf{CredentialPresent}(user, \texttt{VP}) \in \alpha_{t_2}\\ &\land t_2 < t_1))\end{aligned}$$

Goal 5. Selective Disclosure. *A verifier can access only a subset of the claims in a* `VC`. This enforces *data minimization*, ensuring that only necessary claims are disclosed while preserving the holder's privacy.

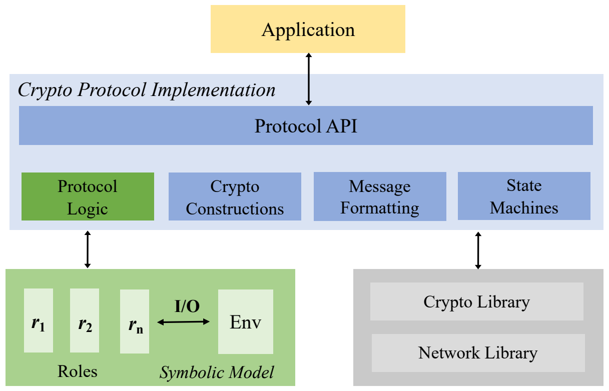


Fig. 4: Our focus in this work (green boxes). Crypto Protocol Implementation figure adapted from [10] by Bhargavan et al.

$$\begin{aligned}
&\forall\ \texttt{VP}, v, id, t_1.\ \mathsf{VPGetPass}(v, id, \texttt{VC}', \texttt{VP}) \in \alpha_{t_1} \\
&\quad\implies (\exists\ t_2, uid, \texttt{VC}.\ \mathsf{importVC}(uid, \texttt{VC}) \in \alpha_{t_2} \wedge t_2 < t_1 \\
&\qquad \wedge (\exists\ \texttt{sigJWT}, \texttt{d}, \texttt{C}.\ \texttt{VC} = \langle \texttt{sigJWT}, \texttt{C} \rangle \\
&\qquad \wedge\ \texttt{VC}' = \langle \texttt{sigJWT}, \texttt{D} \rangle \wedge \texttt{D} \sqsubset \texttt{C}))
\end{aligned}$$

Here, VC and C come from the "VC Construction" in Figure 3a, while VP and D originate from the "VP Construction" in Figure 3b. The mini-program verifier v with identity id first validates VP at timestamp t_1 (Line 1) and then verifies that a holder (uid) possesses a VC from an earlier timestamp t_2 (line 2), from which VP is constructed (line 3). Note that the symbol $\sqsubset$ denotes the subterm relation [17], representing the subset relationship between VP and VC.

V. Leveraging for Code-Level Verification

Although we have conducted symbolic analysis, concerns remain regarding how to ensure that consistency is preserved between the verified abstract design and the concrete low-level implementation, as the verified protocol represents a highly abstracted version of the actual protocol.

To address this challenge, we adopt a lightweight and automated verification approach proposed by Arquint et al. [5]. As illustrated in Figure 4, at the core of their method lies the observation that a symbolic protocol model is composed of three core elements: (1) interactions between protocol roles, (2) interactions between a role and its environment, and (3) internal state transitions that track a role's progress. Notably, inter-role communication can be interpreted as a sequence of role-environment interactions. For example, if role A sends a message m to role B, this is modeled as A first outputting m to the environment, followed by B inputting m from the environment.

All such role-environment interactions are categorized as I/O operations—namely, observable operations such as message transmission, message reception, and randomness generation. Meanwhile, internal state transitions, which reflect how a role's state evolves after executing a protocol rule, are modeled as internal (non-observable) I/O operations. These state changes are critical to preserving the semantic integrity of the protocol.

Leveraging the toolchain proposed by [5], we automatically extract these observable and internal I/O operations from the Tamarin protocol model and translate them into formal I/O specifications. These specifications are then embedded into the Java implementation as verification annotations. We subsequently apply the modular static verifier VeriFast [39] to check that the annotated implementation behaves consistently with the symbolic model across all reachable execution paths.

While the approach of Arquint et al. assumes a static Dolev-Yao adversary, our verification considers a more realistic and complex attacker model. Additionally, due to the heterogeneous nature of our implementation—which spans multiple languages including JavaScript, Java, and C—it is hard to verify all I/O operations across all roles uniformly. As a result, we cannot soundly guarantee that the security property proven in the symbolic model is preserved in the implementation. Nevertheless, this method still offers a practical and effective means to check the consistency between the verified symbolic model and the real-world system implementation.

Specifically, as shown in Figure 4, the focus of our verification at the implementation level is on the protocol logic, i.e., verifying the consistency between the abstract protocol model and the actual low-level implementation. Using I/O specifications, we verify the behavior of key functions across all reachable paths by adding preconditions, postconditions, and assertions. This ensures that the implementation aligns semantically with the abstract design.

We assume that the underlying cryptographic implementations (both the C code in SoftTEE and the TEE SDK code) are implemented correctly. The cryptographic functionalities provided by these low-level implementations are abstracted into Java's standard cryptographic primitive interfaces (e.g., Cipher, Signature), which define clear functional contracts and expected security specifications. This is consistent with the abstraction of cryptographic primitives in symbolic analysis.

A. *VeriFast Verification Tool*

VeriFast [39] is an open source, modular and annotation-based verification tool to verify the safety and correctness properties of single-threaded and multi-threaded C, Rust and Java programs. It verifies that a function complies with its specification—consisting of preconditions, postconditions, and heap-based invariants—using separation logic.

In our work, VeriFast serves as the key backend to ensure that the Java implementation conforms to the I/O specifications extracted from the symbolic protocol model. Developers annotate each function with logical contracts, such as `@requires` and `@ensures`, to declare its expected behavior. VeriFast statically checks whether the implementation satisfies these contracts along all execution paths. VeriFast's modular verification architecture allows reasoning about each function in isolation, which is critical for scaling verification to large codebases. It also supports user-defined predicates, inductive definitions, and lemma functions to aid in modeling complex behaviors.

B. I/O Specification

Separation Logic enables reliable and modular formal verification of heap-manipulating programs by associating each allocated heap location with a permission. In this framework, Penninckx et al. [36] introduced the concept of binding I/O operations (such as receiving and sending messages) to permissions, thereby defining I/O specifications. The permission must be held when calling the I/O operation, and it is consumed after the operation is completed. By combining Separation Logic with the permission mechanism, I/O specifications ensure the legitimacy and consistency of the program's execution during I/O operations. Here is a simplified example of the I/O specification for the `send` operation:

```
boolean send(String mess);
 //@requires token(p1) && out(p1, mess, p2);
 //@ensures token(p2);
```

In this example, `token(p1)` indicates that the protocol is currently at position `p1`, and `out(p1, mess, p2)` grants permission to send the message `mess` and transition to position `p2`. To perform the send operation, both `token(p1)` and `out(p1, mess, p2)` must be held simultaneously. Upon successful execution, the protocol state is updated to `token(p2)`.

C. I/O Behavior Verification

To verify the consistency of the implementation's I/O behavior with the symbolic model, we use Arquint et al.'s tool to directly extract the I/O specifications from the TAMARIN model and embed them into the implementation code as annotations. These embedded specifications are then statically verified using VERIFAST, ensuring that the sequence and semantics of all relevant I/O operations are faithfully aligned with those of the symbolic model.

Furthermore, to address the semantic gap between symbolic terms (e.g., *term* in Tamarin) and their concrete byte representations in the implementation, we use a mapping function $\gamma : M \rightarrow S$ following [5], where M represents the set of symbolic terms and S represents the set of implementation-level strings. This mapping ensures that each symbolic message term has a clear correspondence in the implementation, enabling cross-level reasoning about I/O correctness.

```
1  rule ConstructVC:
2  let eVC = aenc(< claims, sign(claims, skI) >, pkH) in
3    [ MP1(appid, claims, skI, pkH)) ] →
4    [ Out(eVC), MP2(appid, claims, skI, pkH, eVC)]
5
6  String sendConstructVC(Claims C){
7  //@requires token(p1) && out(p1, eVC, p2);
8  //@ensures token(pp) &*& result == gamma(eVC);
9    s = signature(C,sk);
10   cipherVC = cipher(C + s, pkH);
11   //@assert gamma(eVC) == cipherVC;
12   return cipherVC;
13 }
```

TABLE I: Proof results.

Goal	Property	Verified	Time (secs)
-	Executability	✓	16.76
1	Secrecy	✗	95.92
2	Integrity	✓	25.95
3	Injective Agreement	✗	49.52
4	Legal Presentation	✗	33.40
5	Selective Disclosure	✓	12.47

We present above a simplified MSR rule of the VC construction process together with its corresponding implementation. This example demonstrates how to verify the consistency between the symbolic model (Lines 1–4) and the implementation using I/O specifications (Lines 6–13). In rule `ConstructVC`, there is a primary I/O operation, `Out(eVC)`. In the implementation, executing the corresponding I/O operation requires two conditions: first, the protocol must be at position $p1$ and hold the corresponding permission `out(p1, eVC, p2)`. Second, the content sent, `cipherVC`, must match the symbolic term `eVC`, i.e., `@assert gamma(eVC) == cipherVC`. After executing this I/O operation, the protocol state will be updated to `p2`.

VI. VERIFICATION RESULTS DISCUSSION

A. Design-level Verification Results

Proof Effort. Formal analysis of real-world protocols is often highly time-consuming and requires substantial engineering effort. In our case, the complete verification process—including protocol extraction, modeling in TAMARIN, and the development of customized proof strategies framed as TAMARIN tactics [2], took approximately three person-months. The final model consists of 33 rules, amounting to around 731 lines of code (LoC).

TAMARIN [31] requires manual intervention for protocols with large state spaces. We use interactive mode to refine proof strategies, record solving steps, and apply regex-based heuristics for automation. Auxiliary lemmas further aid verification.

Following [24], [33], we formalize each security goal as $\tau \implies C \vee P_1 \vee \cdots \vee P_n$, where τ is the trigger event [24], C is the intended security property, and P_i represents the i-th attack vector violating C. This rule states: *if τ holds, then either C is satisfied, or an attack vector P_i exists*. To identify all the attack vectors, we start with $\tau \implies C$. When TAMARIN finds the first attack trace P_1, it is added to the formula. This process repeats iteratively until $\tau \implies C \vee P_1 \vee \cdots \vee P_n$ holds, ensuring no further attack vectors exist. Figure 5 illustrates how this method can be applied to a specific goal (i.e., secrecy of VC here) in our work.

Our proofs were conducted on a server with dual Intel Xeon Gold 6230R CPUs (2.10GHz, up to 4.00GHz) and 16GB RAM using TAMARIN v1.8.0. We verified six properties (Goal 0–Goal 5), with proofs available in [2]. Proof times, listed in Table I, range from 12.47 to 96.92 seconds but may vary across computing environments.

Findings and Recommendations. As shown in Table I, our analysis validates three goals (Executability, Integrity, and Selective Disclosure) and finds risks in three others (Secrecy,

$$
\begin{aligned}
&\forall\, userid, \texttt{VC}, \#i.\ \mathsf{StoreVC}(userid, \texttt{VC})@i\\
&\quad\Longrightarrow\\
&\quad\neg\Big(\exists \#j.\ \mathsf{K}(\texttt{VC})@j\Big) \quad \text{// Desired property C}\\
&\quad\vee \Big(\exists r, \#x.\ \mathsf{CompromisedDomain}(r)@x\Big)\\
&\quad\vee \Big(\exists \#j, \#x, app.\ \mathsf{MalProg}(\text{'importVC'}, app)@j\\
&\qquad\wedge \mathsf{CompromisedTEE}(userid)@x\Big) \quad \text{// } P_2\\
&\quad\vee \Big(\exists \#j, \#x, app_1, app_2.\ \mathsf{MalProg}(\text{'PresentVP'}, app_1)@j\\
&\qquad\wedge \mathsf{MalProg}(\text{'PresentVP'}, app_2)@x\\
&\qquad\wedge \neg(app_1 = app_2)\Big) \quad \text{// } P_3
\end{aligned}
$$

Fig. 5: The complete formula for secrecy property.

Injective Agreement, and Legal Presentation). We reported these to developers, who confirmed ongoing mitigation efforts.

a) VC *Secrecy is More Vulnerable Than Expected:* The falsification of Goal 1 (Secrecy) reveals that an adversary controlling multiple mini-program verifiers can reconstruct a VC from its VPs by observing inputs, outputs, and using compromised private keys for decryption.

The attack proceeds as follows: (1) Malicious verifiers V send VP requests to the same holder. (2) Each verifier $v_i \in V$ receives $\texttt{VP}_i$, encrypted with its public key, decrypts it using sk_{v_i}, and collects the disclosed claims. (3) Since different VPs reveal different parts of a VC, the adversary aggregates claims c_i from multiple $\texttt{VP}_i$ to reconstruct the original VC as $\texttt{VC} \leftarrow \texttt{sigJWT} \parallel c_1 \parallel \cdots \parallel c_n$.

In practice, enterprise mini-program verifiers undergo strict registration and audits, making it difficult but not impossible for an attacker to control multiple malicious mini-programs to execute this attack. This attack underscores the need for stricter security in enterprise mini-program verifiers, including rigorous registration, auditing, and compliance checks.

Another countermeasure is enabling ZKP, particularly for sensitive data, ensuring undisclosed claims remain protected.

b) Lack of Challenge Code Enables VP *Replay:* The falsification of Goal 3 (Injective Agreement) confirms a replay attack in VP verification, as noted in [11]. Like W3C standards [18], IIFAA DID [27] treats challenge codes as optional. While we recommend using a challenge-response mechanism to prevent replay attacks, developers argue that some applications omit it for efficiency, especially when replay attacks have inconsequential consequences.

We recommend to set challenge codes enabling as default config, but optional. Mini-program developer may disable it, according to the application's security needs.

c) Decoupling VP *from Presenter Identity:* Verifying Goal 4 (Legal Presentation) reveals that VPs are not directly bound to the presenter. A VP remains valid even if forwarded to another holder, creating a privacy risk where verification does not confirm the original presenter's identity.

On mobile platforms (e.g., Alipay client), we recommend implementing VP verification as an "atomic operation" (e.g., one-click actions like "click-to-jump") to prevent interception and misuse. Optionally, biometric authentication (e.g., facial or fingerprint recognition [37]) can be integrated based on risk levels, further binding a VP to its legitimate user.

TABLE II: Summary of proof efforts

Category	LoC
Symbolic Model	731
Code implementation	415
Total specification and annotation code	5,517
– I/O specifications	4,137
– Manually written library function specifications	851
– I/O specification verification code	529

B. Code-level Verification Results

Verification Efforts. We extracted I/O specifications from the previously constructed Tamarin models and embedded them into the Java implementation of the protocol roles. Leveraging the I/O verification framework described earlier, we applied this process to the key components of three core roles: *Issuer* and *Verifier*, which are developed in a demo mini-program; *Holder*, which is part of the actual production-level Alipay client source codes.

The verification scope covers critical operations such as VC issuance, secure storage, presentation, and VP verification, implemented in a total of 415 lines of Java code. To enable static verification with VeriFast, we introduced approximately 5,517 lines of specification and annotation code. Among them, 4,137 lines were automatically generated by the tool proposed by Arquint et al., encoding the I/O specifications and their supporting definitions. Additionally, 851 lines were manually written to specify behaviors of library functions used by the protocol roles. Finally, 529 lines were dedicated to verifying the extracted I/O specifications within the implementation.

Inconsistencies. During the verification process, we identified several discrepancies between the actual implementation and the protocol model.

a) Missing Signature Verification in the Holder Module: The *Holder* module does not perform signature verification on received VCs, which may allow a tampered VC to be stored locally—violating the integrity property at this stage. According to feedback from the development team, this limitation stems from the absence of a trusted interface on the Holder device for retrieving the Issuer's DID, rendering signature verification currently infeasible. However, they confirmed that this verification step will be addressed in future updates. Notably, although a malformed VC could be stored by the Holder, it would ultimately be rejected during VP presentation when verified by the *Verifier*. In addition, since any VC import must originate from a certified institution, and the likelihood of a legitimate institution importing a forged VC is negligible, the associated security risk remains low.

b) Encrypted Storage of VCs on the Holder Device: The *Holder* stores VCs in their envelope-encrypted form, rather than as decrypted plaintext. According to the development team, this design choice is intentional: in certain scenarios, the envelope key is bound to device-specific biometric or local authentication mechanisms (e.g., fingerprint or PIN). This ensures that if the same device is shared among multiple users, VCs encrypted with a user-specific PIN cannot be accessed by others. From a security standpoint, this approach strengthens protection against unauthorized access to sensitive credentials under device-sharing conditions.

c) Hashing Strategy for Claims in the Issuer Module: Another minor discrepancy lies in how the Issuer handles claim hashing during `VC` construction (see the following code example).

```
1 public String ConstructVC(...) {
2   PlainVC plainvc = new PlainVC();
3   plainvc.setCredentialSubject(PlainVCSubject);
4   plainvc.setProof(proof);
5   - //@Term VC = ⟨ctx, ⟨⟨Claims, mClaims⟩, proof⟩⟩;
6   + //@Term VC = ⟨ctx, ⟨⟨Claims, seed⟩, proof⟩⟩;
7   //@assert gamma(VC) == plainvc;
```

In the symbolic model, claims are hashed directly and included as part of the `VC`, relying on the standard assumption that the underlying hash function is collision-resistant. However, the actual implementation adopts a more conservative approach: it adds a random salt to the claims before hashing them, and instead of including the hash value itself in the `VC`, it includes the random salt used for hashing. This design ensures that the hash can still be recomputed and verified externally, while enhancing practical resilience against potential hash collision attacks. Despite this technical difference, the implementation remains semantically aligned with the symbolic model, which assumes no hash collisions and therefore treats both approaches as equivalent from a security standpoint. Here, the first encoding using `mClaims` fails verification (Line 5), while the revised encoding with `seed` succeeds (Line 6).

VII. Insights, Limitations, and Future Work

In our design-level verification, we found that a bottom-up modeling approach significantly improved the efficiency and scalability of DID protocol analysis. Enhancing the efficiency of formal analysis is crucial for its adoption in the real world. In practice, protocols such as DID evolve rapidly from design to deployment; for example, a product based on a new protocol variant might go online in one month. This requires a verification approach distinct from most academic security studies [8], [14], [16], [20]. Existing research often focuses on low-level details (e.g., handshake mechanisms), leading to complex models with limited scalability.

Given DID's reliance on infrastructure like TLS, a bottom-up methodology proved effective in formalizing complex models such as IIFAA DID. By first modeling foundational components like TLS and JSAPI, integrating these into updated or entirely new DID protocol models becomes far more streamlined. This model reuse boosts scalability and reduces both modeling and proof efforts, which is vital as IIFAA DID and similar standards (e.g., protocols like MCP over DID [23]) continue to evolve.

To further boost the efficiency of symbolic analysis, we could explore LLM (Large Language Model) assistance in transforming natural language protocol documentation into Tamarin models. We have made some preliminary attempts in this direction. The main challenge is not in automating Tamarin model verification, but rather in ensuring that the Tamarin model and security specifications accurately reflect the content and intent of the protocol documentation. We will continue to research this in future work.

The mini-program application ecosystem is incredibly rich and diverse, with varied security requirements. Therefore, design-level symbolic analysis should consider each application scenario and its corresponding threat model to determine the security properties of the protocol. For example, preventing replay attacks might be unnecessary in some scenarios, allowing developers the freedom to choose for improved user experience. In the future, we could explore integrating formal methods with behavioral economics theory to introduce probabilistic factors into symbolic models.

Beyond our current static verification pipeline, we plan to incorporate dynamic techniques like conformance testing [35], [40] and runtime monitoring [34] into our end-to-end verification framework. This can serve as a complementary method to enhance the low-level security of DID applications.

VIII. Conclusion

This study conducts an end-to-end formal verification of the IIFAA DID protocol within the Alipay super app, spanning from the protocol design level to the code implementation level. Our Tamarin symbolic model precisely formalizes 5 key security properties: secrecy, integrity, injective agreement, legal presentation, and selective disclosure. We clearly define the security assumptions and conditions, under which these security goals can be achieved. For code-level verification, leveraging I/O specifications and the VeriFast tool, we verified the consistency between the model and the implementation, identifying 3 discrepancies. These formal verification efforts complement the limitations of manual auditing. Our work, released as open-source, contributes to future security analysis of DID-related protocols and enlightens how formal verification can be utilized throughout the security protocol development lifecycle to fundamentally enhance its reliability.

Acknowledgements

We thank the anonymous reviewers for their constructive comments on this paper. This work was partially done while Ziyu Mao was a research intern at Ant Group. This research was supported by Ant Group and the Fundamental Research Funds for the Central Universities (2025ZFJH02).

References

[1] The European digital identity regulation. https://www.european-digital-identity-regulation.com/

[2] VerifyDID: A project for verifying iifaa decentralized identifiers (2024), https://github.com/zerrymore11/VerifyDID, accessed: 2024-11-17

[3] Alipay: OpenAPI signature methods, https://opendocs.alipay.com/common/02kf5p

[4] Alipay: Alipay mini program documentation (2024), https://miniprogram.alipay.com/docs/

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

[6] Badertscher, C., Banfi, F., Diaz, J.: What did come out of it? analysis and improvements of didcomm messaging. Cryptology ePrint Archive (2024)

[7] Basin, D., Cremers, C., Dreier, J., Sasse, R.: Modeling and Analyzing Security Protocols with Tamarin: A Comprehensive Guide (2025), https://tamarin-prover.com/book/downloads/Tamarin%20book-Draft%20v0.9.5.pdf, draft v0.9.5, May 14

[8] Basin, D., Dreier, J., Hirschi, L., Radomirovic, S., Sasse, R., Stettler, V.: A formal analysis of 5G authentication. In: Proceedings of the 2018 ACM SIGSAC conference on computer and communications security. pp. 1383–1396 (2018)

[9] Basin, D., Sasse, R., Toro-Pozo, J.: The EMV standard: Break, fix, verify. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 1766–1781. IEEE (2021)

[10] Bhargavan, K., Hansen, L.L., Kiefer, F., Schneider-Bensch, J., Spitters, B.: Formal security and functional verification of cryptographic protocol implementations in rust. Cryptology ePrint Archive (2025)

[11] Braun, C.H.J., Horne, R., Käfer, T., Mauw, S.: SSI, from specifications to protocol? formally verify security! In: Proceedings of the ACM on Web Conference 2024. pp. 1620–1631 (2024)

[12] BSN: BSN DID, https://www.bsnbase.com/

[13] Chadwick, K.N., Vercammen, J.: OpenID for verifiable credentials (2022), https://openid.net/wordpress-content/uploads/2022/05/OIDF-Whitepaper_OpenID-for-Verifiable-Credentials_FINAL_2022-05-12.pdf

[14] Cremers, C., Dax, A., Naska, A.: Formal analysis of SPDM: Security protocol and data model version 1.2. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 6611–6628 (2023)

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

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

[17] Cremers, C., Jacomme, C., Lukert, P.: Subterm-based proof techniques for improving the automation and scope of security protocol analysis. In: 2023 IEEE 36th Computer Security Foundations Symposium (CSF). pp. 200–213. IEEE (2023)

[18] Dave Longley, Manu Sporny: Verifiable Credential Data Integrity 1.0. https://www.w3.org/TR/vc-data-integrity/ (2023)

[19] Decentralized Identity Foundation: DIDComm Messaging v2.x Editor's Draft. https://identity.foundation/didcomm-messaging/spec/ (2024), dIDComm Messaging v2.x Editor's Draft

[20] Delaune, S., Lallemand, J., Patat, G., Roudot, F., Sabt, M.: Formal security analysis of widevine through the W3C EME standard. In: 33rd USENIX Security Symposium (USENIX Security 24). pp. 6399–6415 (2024)

[21] Dolev, D., Yao, A.: On the security of public key protocols. IEEE Transactions on information theory **29**(2), 198–208 (1983)

[22] Fett, D., Yasuda, K., Campbell, B.: Selective Disclosure for JWTs (SD-JWT). Internet-Draft draft-ietf-oauth-selective-disclosure-jwt-13, Internet Engineering Task Force (Oct 2024), https://datatracker.ietf.org/doc/draft-ietf-oauth-selective-disclosure-jwt/13/, work in Progress

[23] Foundation, O.: Mcp over tsp python sdk (2025), https://github.com/openwallet-foundation-labs/mcp-over-tsp-python

[24] Girol, G., Hirschi, L., Sasse, R., Jackson, D., Cremers, C., Basin, D.: A spectral analysis of noise: A comprehensive, automated, formal analysis of Diffie-Hellman protocols. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 1857–1874 (2020)

[25] Hardt, D.: The OAuth 2.0 Authorization Framework. RFC 6749 (Oct 2012). https://doi.org/10.17487/RFC6749, https://www.rfc-editor.org/info/rfc6749

[26] Hauck, F.: OpenID for Verifiable Credentials: Formal Security Analysis Using the Web Infrastructure Model. Master's thesis, University of Stuttgart (2023). https://doi.org/10.18419/opus-13772, http://elib.uni-stuttgart.de/handle/11682/13791, also known as: "OpenID for Verifiable Credentials: Formale Sicherheitsanalyse unter Verwendung des Web Infrastructure Model"

[27] IIFAA: IIFAA decentralized authentication technical specification, part 1: General requirement (in chinese), https://www.iifaa.org.cn/technical#standard

[28] IIFAA: Iifaa decentralized trusted authentication technical specification - part 1: General requirements, https://www.w3.org/community/reports/cndid/CG-FINAL-iifaa-did-20241231/

[29] IIFAA: The IIFAA official web page, https://www.iifaa.org.cn/

[30] Linker, F., Basin, D.: SOAP: A social authentication protocol. In: 33rd USENIX Security Symposium (USENIX Security 24). pp. 3223–3240. USENIX Association, Philadelphia, PA (Aug 2024), https://www.usenix.org/conference/usenixsecurity24/presentation/linker

[31] Meier, S., Schmidt, B., Cremers, C., Basin, D.: The TAMARIN prover for the symbolic analysis of security protocols. In: Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25. pp. 696–701. Springer (2013)

[32] Microsoft: Microsoft entra verified ID, https://www.microsoft.com/en-us/security/business/identity-access/microsoft-entra-verified-id

[33] Morio, K., Esiyok, I., Jackson, D., Künnemann, R.: Automated security analysis of exposure notification systems. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 6593–6610 (2023)

[34] Morio, K., Künnemann, R.: Specmon: Modular black-box runtime monitoring of security protocols. In: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security. pp. 2741–2755 (2024)

[35] Ouyang, L., Sun, X., Tang, R., Huang, Y., Jivrajani, M., Ma, X., Xu, T.: Multi-grained specifications for distributed system model checking and verification. In: Proceedings of the Twentieth European Conference on Computer Systems. pp. 379–395 (2025)

[36] Penninckx, W., Jacobs, B., Piessens, F.: Sound, modular and compositional verification of the input/output behavior of programs. In: European Symposium on Programming Languages and Systems. pp. 158–182. Springer (2015)

[37] Rila, L., Mitchell, C.J.: Security protocols for biometrics-based cardholder authentication in smartcards. In: Applied Cryptography and Network Security: First International Conference, ACNS 2003, Kunming, China, October 16-19, 2003. Proceedings 1. pp. 254–264. Springer (2003)

[38] Shostack, A.: 4 question frame. https://github.com/adamshostack/4QuestionFrame (2024), accessed: 2024-09-14

[39] Smans, J., Jacobs, B., Piessens, F.: Verifast for java: A tutorial. Aliasing in Object-Oriented Programming. Types, Analysis and Verification pp. 407–442 (2013)

[40] Tang, R., Wang, M., Sun, X., Huang, L., Huang, Y., Ma, X.: Converos: Practical Model Checking for Verifying Rust OS Kernel Concurrency. In: Proceedings of the 2025 USENIX Annual Technical Conference (USENIX ATC '25). pp. 1143–1159. USENIX Association (2025)

[41] Terbu, O., Lodderstedt, T., Yasuda, K., Looker, T.: OpenID for verifiable presentations. Git commit: 11157695d140f4c47f742f5d82e25283b90dd952 (Mar 2023), https://openid.net/specs/openid-4-verifiable-presentations-1_0-17.html, visited on 09/11/2023. Cited on pages 3, 4, 18, 24, 25, 32, 50, 51

[42] Threat Modeling Community Group (TMCG).: Decentralized identities threat model. https://github.com/w3c-cg/threat-modeling/blob/main/models/decentralized-identities.md (2025), latest update: 2025-05-29

[43] Wang, Y., Yao, Y., Shi, S., Chen, W., Huang, L.: Towards a better super-app architecture from a browser security perspective. In: Proceedings of the 2023 ACM Workshop on Secure and Trustworthy Superapps. pp. 23–28 (2023)

[44] Zhang, Y., Yang, Y., Lin, Z.: Don't leak your keys: Understanding, measuring, and exploiting the appsecret leaks in mini-programs. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. pp. 2411–2425 (2023)