arXiv:2504.21034v1 [cs.CR] 27 Apr 2025

SAGA: A Security Architecture for Governing AI Agentic Systems

Georgios Syros*, Anshuman Suri*, Cristina Nita-Rotaru, Alina Oprea
{syros.g, ans.suri, c.nitarotaru, a.oprea}@northeastern.edu
Northeastern University

***Abstract*—Large Language Model (LLM)-based agents increasingly interact, collaborate, and delegate tasks to one another autonomously with minimal human interaction. Industry guidelines for agentic system governance emphasize the need for users to maintain comprehensive control over their agents, mitigating potential damage from malicious agents. Several proposed agentic system designs address agent identity, authorization, and delegation, but remain purely theoretical, without concrete implementation and evaluation. Most importantly, they do not provide user-controlled agent management.**

To address this gap, we propose SAGA, a Security Architecture for Governing Agentic systems, that offers user oversight over their agents' lifecycle. In our design, users register their agents with a central entity, the `Provider`, that maintains agents contact information, user-defined access control policies, and helps agents enforce these policies on inter-agent communication. We introduce a cryptographic mechanism for deriving access control tokens, that offers fine-grained control over an agent's interaction with other agents, balancing security and performance consideration. We evaluate SAGA on several agentic tasks, using agents in different geolocations, and multiple on-device and cloud LLMs, demonstrating minimal performance overhead with no impact on underlying task utility in a wide range of conditions. Our architecture enables secure and trustworthy deployment of autonomous agents, accelerating the responsible adoption of this technology in sensitive environments.

I. INTRODUCTION

AI agents with increased levels of autonomy are being deployed in safety-critical applications, such as healthcare [1, 2], finance [3, 4, 5], and cybersecurity [6, 7, 8]. These agents, built on top of frontier AI models, particularly Large Language Models (LLMs), excel at automating complex tasks traditionally performed by humans. Agents powered by LLMs have sophisticated reasoning capabilities and the ability to understand and generate natural language. They also leverage access to tools installed on user devices, external resources, and the ability to interact with other AI agents autonomously.

The increasing autonomy and functionality of AI agents expand the attack surface of agentic systems, introducing numerous security risks. As AI agents become more integrated into critical applications, securing these systems presents several challenges, as highlighted in a recent OpenAI white paper [9]. Several key requirements include defining unique identities for AI agents, authenticating these agents, and providing secure mechanisms for agent discovery and communication. These requirements must remain effective even under adversarial conditions, as malicious actors may attempt to impersonate agents, intercept communications, or manipulate agent behavior to extract sensitive information or subvert intended functionality. Significantly, the OpenAI white paper emphasizes the necessity of maintaining user control and supervision throughout all operational phases and throughout the entire lifecycle of LLM agents to safeguard against potential harm from malicious agents.

Although various AI agentic system designs have been proposed that incorporate agent identities [10], attribution [11], authorization mechanisms and delegation capabilities [12], these designs largely remain theoretical without implementation or evaluation. Most critically, they fail to adequately address the essential component of user-controlled agent management. Recently, Google's A2A protocol [13] introduces a decentralized identity framework where agents advertise public metadata and initiate direct, encrypted communication using web-based authentication. While A2A promotes interoperability and supports verifiable identifiers, it lacks policy enforcement mechanisms and runtime mediation of agent interactions, and does not provide mitigation against adversarial agents.

In this paper, we propose SAGA, a framework for governing LLM agent deployment, designed to enhance security while offering user oversight on their agents' lifecycle (see Figure 1 for an overview). In SAGA users register themselves and their agents with a `Provider` service that maintains user and agent metadata and facilitates controlled communication establishment between agents. SAGA enables users to control access to their agents through an Access Contact Policy that users define for their agents. The enforcement of the policy is done through cryptographic access control tokens, derived from agent one-time keys stored at the `Provider`. Inter-agent communication over secure TLS channels does not involve the `Provider` for scalability, while utilizing access control tokens to ensure compliance with user-defined policies. The granularity of access control tokens provides a tradeoff between potential windows of vulnerability and performance overhead. The architecture aligns with best practices for securing agentic AI systems and meets key requirements outlined in emerging AI governance frameworks.

Contributions. We highlight our main contributions:

---

* Equal Contribution

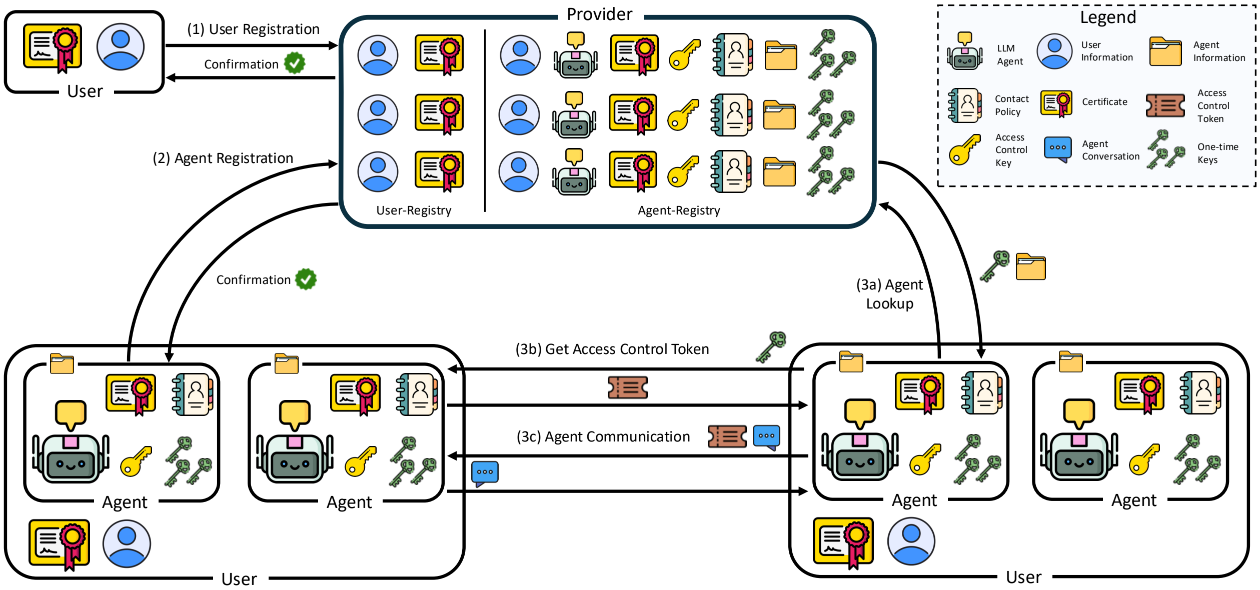

Fig. 1: System overview of SAGA. (1) A user registers with the `Provider`. (2) A registered user registers their agent and receives a confirmation from the `Provider`. (3a) An initiating agent requests a One-time Key (OTK) and the receiving agent's information from the `Provider`. (3b) The initiating agent contacts the receiving agent with the OTK, after which the agents derive a shared key and the receiving agent encrypts an access control token under the shared key. (3c) The initiating agent attaches this token to subsequent communication over TLS. When token expires or reaches limit, a new OTK is retrieved from the `Provider` and a new token is obtained from the receiving agent.

- We present SAGA, a **S**ecurity **A**rchitecture for **G**overning **A**gentic systems, that enables users control and oversight of their LLM-enabled agents.
- We propose a cryptographic mechanism based on access control tokens encrypted under shared agent keys that allows fine-grained control over inter-agent communication to balance security and performance considerations.
- We perform a security analysis of SAGA and demonstrate that the protocol provides protection against a wide range of adversarial capabilities.
- We evaluate our system across several agentic tasks (scheduling meetings, submitting expense reports, and collaborative writing), multiple on-device and cloud LLM models, and agents placed in diverse geolocations to demonstrate the scalability of our design.

II. BACKGROUND AND PROBLEM STATEMENT

In this section we provide background on agentic systems, their security requirements, and define our problem statement.

A. Agentic AI Systems

Agentic AI systems represent an advancement in autonomous systems, enabled by generative AI. These agents independently devise execution plans for user-assigned tasks, leveraging their underlying models for strategic planning and reasoning. As part of the planned steps for completing the assigned tasks, agents might need to leverage additional tools installed on user devices, interact with cloud services, or communicate with external agents running on other devices. LLM agents can automate both professional tasks like scheduling and personal activities such as trip planning. For example, Alice's calendar agent might contact Bob's calendar agent to arrange a meeting, then use an email tool to send confirmations to both participants.

As agentic systems are currently under development without clear boundaries of the interactions between the different participants in the ecosystem, in this paper, we formulate the following definitions and terminology.

Definition 1. *A* `User` *is the owner of one or more agents and is responsible for assigning them tasks. A user will have only one instance of an agent running on each device (e.g., a user will have only one Calendar agent running on a particular device).*

Definition 2. *An* `Agent` *is an autonomous software entity with an LLM as its core decision-making component. Upon receiving a user query, agents use LLMs for planning, store information in memory, and execute plans to complete assigned tasks. Agents can access tools installed on user machines as needed. Each agent operates under a specific user's authority and may interact with other agents to fulfill complex objectives.*

Definition 3. *A task is defined as a sequence of actions that an* `Agent` *must plan and execute to achieve a* `User`*-defined goal. Such actions might involve communicating with an LLM, invoking other tools, or communicating with other agents.*

B. Requirements for Secure Agentic Systems

Agentic systems create a complex ecosystem where multiple requirements must be addressed for them to successfully complete tasks securely. An OpenAI white paper [9] outlines several guidelines and open problems in designing secure agentic systems. Key requirements include defining unique identities for AI agents with verification capabilities, providing secure mechanisms for agents to discover and communicate with each other, and enabling agents to make informed decisions about resource access and collaboration with other agents. Agentic systems must also implement protections against adversarial or rogue agents to limit their influence and prevent harm to benign agents. At the same, developing effective agent shutdown protocols is especially important as autonomous agent self-replication has been demonstrated feasible [14]. Crucially, [9] advocates for users' control and oversight throughout the operation and lifecycle of LLM agents.

C. Problem Statement

While there are many possible deployments of AI agentic systems, in this paper we focus on LLM agents running on users' personal devices. There is a growing trend toward running these agents directly on personal devices, with industry investing heavily to enable on-device AI. For instance, Apple recently introduced on-device Apple Intelligence [15], Google developed AI capabilities for Pixel devices [16], and Qualcomm designed specialized chips for on-device LLMs [17]. On-device agents can access local tools installed on the device, user data stored on the device, and interact with on-device or cloud LLMs. In this context, we aim to design a user-controlled secure agentic framework that addresses several essential core requirements for governing agentic systems.

We will first address the fundamental requirement of creating and managing agent identities while providing effective mechanisms for agent discovery within the ecosystem. We will design secure inter-agent communication protocols, which represents a critical functionality of any agentic system, and address the challenge of how to limit the influence of adversarial agents on benign ones. We also aim to enable user control on an agent's entire lifecycle, including agent registration in the system, managing agent's contacts, and agent's deactivation at any time, according to the user's preferences and policies.

With the rapid evolution of agentic systems and emerging regulations in this space, we expect additional requirements that agentic systems must enforce in the future. To account for these developments, we intend to design a modular framework that is extensible and can support new security properties with minimal changes to its core components.

III. SYSTEM ARCHITECTURE

We begin by outlining the desirable goals for a secure agentic system (Section III-A), followed by an overview of our proposed architecture SAGA (Section III-B). We also discuss the system assumptions we make (Section III-C) and the threat models that our proposed architecture can defend against (Section III-D).

A. System Goals

Although the requirements for a secure agentic framework are still evolving, we outline several desirable goals for our system design:

Agent lifecycle managed by users. An agent's lifecycle should be established by a user, who installs it on their device, registers it within the system, and has the authority to terminate the agent at any time by uninstalling it from the device or deactivating it and preventing other agents from locating it within the ecosystem.

Agent access controlled by user policies. While LLM agents can autonomously decide to contact other agents to perform user-assigned tasks, communication between agents should be controlled by user-defined access control policies to limit harm by adversarial agents.

Limited trust. Agents should have limited trust in other third-parties in the system, including any centralized service provider or other agents.

Scalability. Design the system to efficiently manage agents with minimal performance overhead.

Limited influence on other agents. An agent can control only its own participation in the system and cannot arbitrarily remove other agents from the ecosystem.

Limited vulnerability window. If and when an agent is compromised, it is crucial to limit the vulnerability window while the agent is still active in the system, as it could potentially cause harm. The length of this vulnerability window can be controlled by cryptographic mechanisms and depends on various factors, such as the agent's communication pattern, the nature of the task, and the sensitivity of the exchanged data.

Maintain system utility. We aim to preserve the utility of the system, as measured by task completion, without compromising system security. Additionally, the framework should be agnostic to various agent implementations and LLM models that serve as the backbone.

B. Architecture Overview

We provide an overview of our SAGA system architecture in Figure 1. A critical consideration in designing a secure agentic framework is the mechanism for agent discovery. The implementation of an agent registry facilitates this discovery process, with architectural options ranging from centralized to distributed registry models. In our design, we choose to use a centralized registry, maintained by an entity called `Provider`. The `Provider` manages agent and user identities, ensures authentication, and facilitates controlled communication establishment between agents.

To ensure user control and oversight over an agent's lifecycle, we delegate the agent registration process to users themselves, with the `Provider` maintaining the agent registration information. Users must be registered in the system and authenticate before registering any agents and retain the ability to unregister or deactivate their agents, for example as a protective measure when abuse is detected. Each agent is assigned a unique identifier, linked to the user and the device on which it is installed. The `Provider` thus maintains the `User Registry` ($\mathcal{D}_U$) which holds user identity records, and the `Agent Registry` ($\mathcal{D}_A$) which stores agent metadata, cryptographic credentials, endpoint information, and agent communication policies.

While communication between agents can be implemented using standard secure protocols such as TLS, we would like the ability to enforce limits on agent access, controlled by users. Governed by the same principle of users managing access to their agents, each user can specify an *Access Contact Policy* for each agent defining which users and agents are permitted to initiate connections. To enable fine-grained access control and limit the window of vulnerability, access to agents needs to be cryptographically bounded and respect the user-defined policy. For scalability and reduced reliance on the `Provider`, once an agent obtains the necessary connection details for another agent, all subsequent communication should proceed directly, without `Provider` involvement.

One important design consideration is how to manage the access control to an agent in a fine-grained manner while reducing the burden on the `Provider`. Towards this goal, each agent registers a number of public one-time keys (OTKs) with different quotas for each initiating agent (and locally stores the corresponding secret information linked to these `OTK`s). In principle, an `OTK` could be used only once for an inter-agent interaction, but that would require generating and storing a large number of `OTK`s, presenting significant operational overhead. We extend the lifetime of these one-time keys by the receiving agent generating an *Access Control Token* that is encrypted using a dynamically derived shared key between the agents. The shared key must be unique to each initiating–receiving agent pair and derivable only by those two agents. To support the shared key computation, each agent maintains a long-term *Access Control Key*, which serves as the basis for deriving the shared key and binding it to the specific agent pair. The public component of the access control key is stored in the Agent Registry, while each agent maintains the secret part. We then use the Diffie-Hellman key exchange protocol, in which the initiating agent contributes the access control key, and the receiving agent the `OTK`s as public keys. The receiving agent encrypts the *Access Control Token* under the derived shared key. The token includes an expiration timestamp and a limit on the number of permitted requests. The token is reused for inter-agent communication, without involving the `Provider`. When the token expires, a new one is created after the initiating agent obtains another `OTK` from the `Provider`. This design balances the security and performance overhead in the system, by allowing users to tune access to their agents through the number of keys they register and the lifetime of *Access Control Tokens*. Note that the protocol is asymmetric, as access control is enforced for receiving agents, according to user-specified policies for initiating agents.

Below, we outline the main components of our system; protocol details are provided in Section IV.

User Registration (Section IV-B). Users must register with the `Provider` using a persistent identity mechanism, such as OpenID Connect [18]. Upon successful registration, users authenticate to the `Provider` and provision agents for participation in the SAGA ecosystem.

Agent Registration (Section IV-C). A user registers its agents with the `Provider`. During registration, the user generates cryptographic keys for its agents, including TLS certificates and access control keys. The user also signs the agent metadata, such as hostname, port, and device identifiers, to bind it to its identity and specific device. Additionally, the `Provider` signs the agent's metadata, serving as cryptographic proof that the agent is registered.

Agent Management (Section IV-D). The `Provider` enables users to define and update an *Agent Contact Policy* for each of their agents, which governs the policy for permissible incoming communication. This policy allows users to restrict which other agents can initiate contact with their agent, and to impose access control constraints. The `Provider` enforces these policies during initial contact requests, ensuring that agent interactions are governed by user-defined rules. Users can deactivate their own agents at any time, but cannot deactivate agents owned by others.

Agent Communication (Section IV-E). To initiate contact with another agent, the initiating agent queries the `Provider` with the receiving agent's identifier. The `Provider` responds with metadata, including the receiving agent's device, IP address, and a one-time key (`OTK`) for access control. Subsequently, the initiating agent establishes a shared key with the receiving agent using the Diffie-Hellman key exchange protocol. The receiving agent generates an access control token, encrypted under the shared key, which is included in any communication by the initiating agent. When a token expires or reaches its request limit, the initiating agent obtains a new `OTK` from the `Provider`. Note that the `Provider` does not mediate inter-agent communication, but it is critical in enforcing each receiving agent's Access Control Policy by distributing `OTK`s to initiating agents.

C. System Assumptions

The correctness and security guarantees of SAGA rely on a set of clearly defined system assumptions. We outline the assumptions required for the protocol to uphold its security model.

Secure User Authentication and Human Verification. We assume that the `Provider` implements a robust user authentication mechanism (e.g., OpenID Connect) and that user credentials are not compromised. Crucially, we assume that agent registration is restricted to authenticated human users, enforced through human verification during user account creation. This verification process is delegated to a trusted external identity service, which certifies the user's human status on behalf of the `Provider`. This assumption is crucial, as the integrity of the protocol depends on preventing autonomous agents from registering other agents without human oversight.

Agent Identity Control. We assume that attackers can create and register agents under their own identities but cannot impersonate other users. That is, while adversaries may instantiate and operate malicious agents, they cannot register agents under the identity of a benign user.

Public IP Addressing. All agents and providers are assumed to operate under globally routable, public IP addresses. This design avoids NAT traversal and local discovery, relying instead on the assumption that agents are directly reachable at their registered endpoints.

Cryptographic Soundness. We assume that all cryptographic primitives used by the system—signature schemes, key exchange protocols, encryption schemes, and key derivation functions—are secure. Specifically, we assume the unforgeability of digital signatures, the confidentiality of keys, and the security of cryptographic protocols such as Diffie-Hellman. Secret keys are assumed to remain confidential and outside adversarial control.

TLS and Secure Channels. All communication, both between agents and between agents and the `Provider`, is protected by TLS. We assume that TLS provides confidentiality, integrity, and authenticity against network-level adversaries. Standard network-layer attacks such as message tampering, eavesdropping, and replay are considered mitigated by the security guarantees provided by the underlying TLS session.

Network Protections. We assume that the network infrastructure enforces basic protections against denial-of-service attacks and packet flooding.

Together, these assumptions establish the boundary within which the protocol's correctness and security guarantees are valid.

D. Threat Model

The `Provider` is expected to adhere to the SAGA protocol logic, including enforcing contact policies, issuing keys, and performing registry operations. However, it may be *honest-but-curious*: capable of observing agent metadata and traffic patterns without actively attempting to subvert the protocol. The user and agent registries are assumed to be securely stored and not vulnerable to adversarial control or tampering.

We consider several adversarial capabilities which we discuss below. Our security analysis in Section IV-F explains how SAGA mitigates these attacks.

C1: Adversaries might create agents and register them with the `Provider`. These adversarial agents could deviate from the protocol when communicating with other agents. They could also add themselves to the contact policy of benign agents by performing social engineering on users.

C2: A legitimate agent registered with the `Provider` could be compromised by an adversary. This attack could occur when agents interact with external resources, such as websites, or tools installed on user devices, which might trigger a compromise.

C3: Adversaries may instruct an agent to self-replicate on the same device or on another user's device without registering the child agent with the `Provider`. Prior work has demonstrated such self-replication of agents [19]. The parent agent can share TLS keys, access control keys, and existing access control tokens with the child agent.

C4: An adversarial agent may share its TLS public keys, access control keys, and access control tokens with another adversary-controlled agent, enabling communication with a benign victim agent.

C5: An adversary could attempt to mount a Sybil attack, by creating agents with multiple identities.

IV. SAGA PROTOCOL SPECIFICATION

In this section, we begin with a description of the cryptographic primitives involved in our protocol, followed by a description of the key protocols involved in SAGA: user registration (Section IV-B), agent registration (Section IV-C), agent management (Section IV-D), and inter-agent communication (Section IV-E).

A. Cryptographic Primitives and Notation

We leverage the following cryptographic primitives:

Signature schemes. A signature scheme consists of three algorithms: KeyGen() – a key generation function that outputs a (public, private) signing key pair $(\mathsf{pk}, \mathsf{sk})$, $\text{Sign}_{\texttt{SK}}(m)$ – a signing algorithm that takes as input the private key `SK` and a message m, and outputs a signature σ, and $\text{Verify}_{\texttt{PK}}(m, \sigma)$ – a verification algorithm that uses the public key pk, and verifies that the signature on a message m is correct. We assume that the signature scheme is Existential Unforgeable under Chosen Message Attack [20], such as ECDSA [21] and Ed25519 [22]. A certificate generation function $\text{GenCert}_X(m)$ involves entity X creating a certificate for content m as: $\text{GenCert}_X(m) = \langle m, \text{Sign}_X(m)\rangle$

Hash function. We use a collision-resistant hash function $\mathtt{H}(\cdot)$, such as SHA-256 or SHA-3 [23].

Diffie-Hellman Key Exchange. The Diffie-Hellman Key Exchange protocol [20] is a cryptographic method that allows two parties to establish a shared secret key. Each party generates a (secret, public) key pair, and exchanges with the other party the public component. For instance, if Alice has secret x and public key $g^x \mod p$ (for p a large prime number and $g \in Z_p$), and Bob has secret y and public key $g^y \mod p$, the shared secret key is obtained by Alice as $(g^y)^x = g^{xy} \mod p$, and by Bob as $(g^x)^y = g^{xy} \mod p$. We denote by DH the function that takes as input the secret

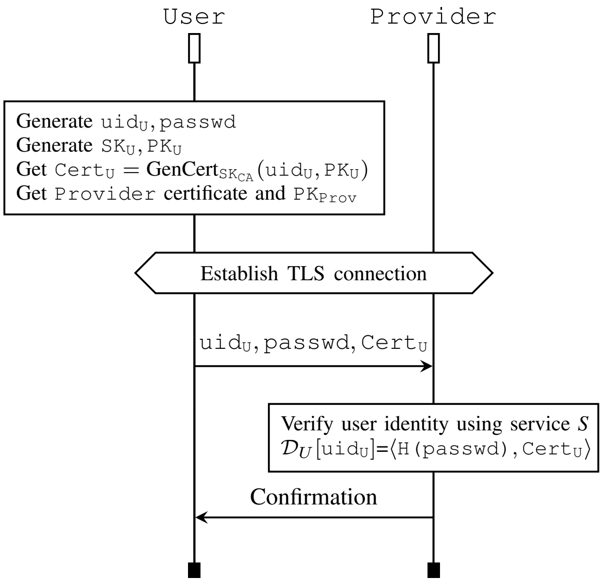
User
Provider
Generate $\texttt{uid}_{\texttt{U}}, \texttt{passwd}$
Generate $\texttt{SK}_{\texttt{U}}, \texttt{PK}_{\texttt{U}}$
Get $\texttt{Cert}_{\texttt{U}} = \text{GenCert}_{\texttt{SK}_{\texttt{CA}}}(\texttt{uid}_{\texttt{U}}, \texttt{PK}_{\texttt{U}})$
Get `Provider` certificate and $\texttt{PK}_{\texttt{Prov}}$
Establish TLS connection
$\texttt{uid}_{\texttt{U}}, \texttt{passwd}, \texttt{Cert}_{\texttt{U}}$
Verify user identity using service S
$\mathcal{D}_U[\texttt{uid}_{\texttt{U}}] = \langle \texttt{H(passwd)}, \texttt{Cert}_{\texttt{U}} \rangle$
Confirmation

Fig. 2: **User Registration** protocol for user `U`.

key of one party and public key of the other party and computes the shared secret key $DH(x, g^y) = DH(y, g^x) = g^{xy} \bmod p$. The security of Diffie-Hellman is based on the Computational Diffie-Hellman (CDH) assumption.

Key Derivation Function. A Key Derivation Function (KDF) is a cryptographic algorithm that derives one or more secret keys from a master secret. We use the HMAC-based Extract-and-Expand Key Derivation Function (HKDF) [24] with SHA-256 as the underlying hash function.

Notation. We introduce formal notation incrementally throughout the protocol specification, defining each symbol in context. For convenience, Table I provides a summary of all symbols used.

Symbol	Description
`PK,SK`	Public/private signing keys
`Cert`	Certificate issued by CA
$\texttt{uid}_{\texttt{U}}$	User identifier
$\texttt{aid}_{\texttt{A}}$	Agent `A` identifier
$\texttt{ED}_{\texttt{A}}$	Agent `A` endpoint descriptor
$(\texttt{PK}_{\texttt{A}}, \texttt{SK}_{\texttt{A}})$	Agent `A` public/private TLS credentials
$(\texttt{PAC}_{\texttt{A}}, \texttt{SAC}_{\texttt{A}})$	Agent `A` public/private access control keys
$(\texttt{OTK}^{\texttt{i}}_{\texttt{A}}, \texttt{SOTK}^{\texttt{i}}_{\texttt{A}})$	Agent `A` one-time public/private keys
$\texttt{CP}_{\texttt{A}}$	Contact policy of agent `A`
`token`	Access control token
σ^Y_X	Entity Y-issued signature regarding entity X.
$\mathcal{D}_U$	Provider's user registry
$\mathcal{D}_A$	Provider's agent registry

TABLE I: Notation used in the SAGA protocol.

B. User Registration

The first step for any user seeking to deploy agents is to register with the `Provider`. This process establishes the user's identity and enables them to link future agents to their identity and manage them securely. The user obtains a certificate signed by an external certificate authority `CA` on their public key, sent to the `Provider` during registration. We assume that the user can establish a TLS connection with the `Provider` and that the provider can verify the user's identity with the help of an external service such as OpenIDConnect. The protocol follows these steps as shown in Figure 2.

1) **User Account Setup.** The user selects a public identifier $\texttt{uid}_{\texttt{U}}$ corresponding to their email address, e.g., $\texttt{uid}_{\texttt{U}}$ = `'alice@domain.com'`, and a secret passphrase `passwd` to authenticate to the `Provider`.
2) **User signature key generation:** The user generates a signature key pair $(\texttt{PK}_{\texttt{U}}, \texttt{SK}_{\texttt{U}})$ for signing information about user agents.
   The user contacts external Certificate Authority `CA` to generate its certificate $\texttt{Cert}_{\texttt{U}} = \text{GenCert}_{\texttt{SK}_{\texttt{CA}}}(\langle \texttt{uid}_{\texttt{U}}, \texttt{PK}_{\texttt{U}} \rangle)$, and shares it with the `Provider`.
3) **Connection establishment.** The user obtains the `Provider`'s certificate and public key $\texttt{PK}_{\texttt{Prov}}$ from the `CA` and verifies them. A TLS session is established between the user and the `Provider`.
4) **Sending user information.** The user submits $(\texttt{uid}_{\texttt{U}}, \texttt{passwd})$ and $\texttt{Cert}_{\texttt{U}}$ to the `Provider`.
5) **User identity verification.** The `Provider` verifies the user's identity using an external partner service S (see Section III-C). If the verification is successful and the account does not exist, the `Provider` finalizes the user registration.
6) **User account storage and confirmation:** The `Provider` updates the user registry: $\mathcal{D}_U[\texttt{uid}_{\texttt{U}}] = \langle \texttt{H(passwd)}, \texttt{Cert}_{\texttt{U}} \rangle$ and sends a confirmation to the user.

After the user registration is completed successfully, the user can proceed to register its agents with the `Provider`, described in the next stage.

C. Agent Registration by User

The agent registration process ensures that each agent is cryptographically bound to its user and a specific user's device. The cryptographic information generated by a user for its agents is stored by the `Provider` and subsequently used to establish secure agent communication mediated by the `Provider`. The following protocol is executed by user `U` to register their agent `A` (shown in Figure 3).

1) **Generating agent information**. The user selects an identifier $\texttt{name}_{\texttt{A}}$ for the agent, forming a unique agent ID in combination with their username: $\texttt{aid}_{\texttt{A}} = \texttt{uid}_{\texttt{U}}\texttt{:name}_{\texttt{A}}$. The user specifies the agent's device name $\texttt{device}_{\texttt{A}}$ and networking details $\texttt{IP}_{\texttt{A}}$ and $\texttt{port}_{\texttt{A}}$. These comprise the agent's "endpoint descriptor":

$$\texttt{ED}_{\texttt{A}} = \langle \texttt{device}_{\texttt{A}}, \texttt{IP}_{\texttt{A}}, \texttt{port}_{\texttt{A}} \rangle$$

2) **Generating cryptographic keys**. The user generates the following keys for the agent:

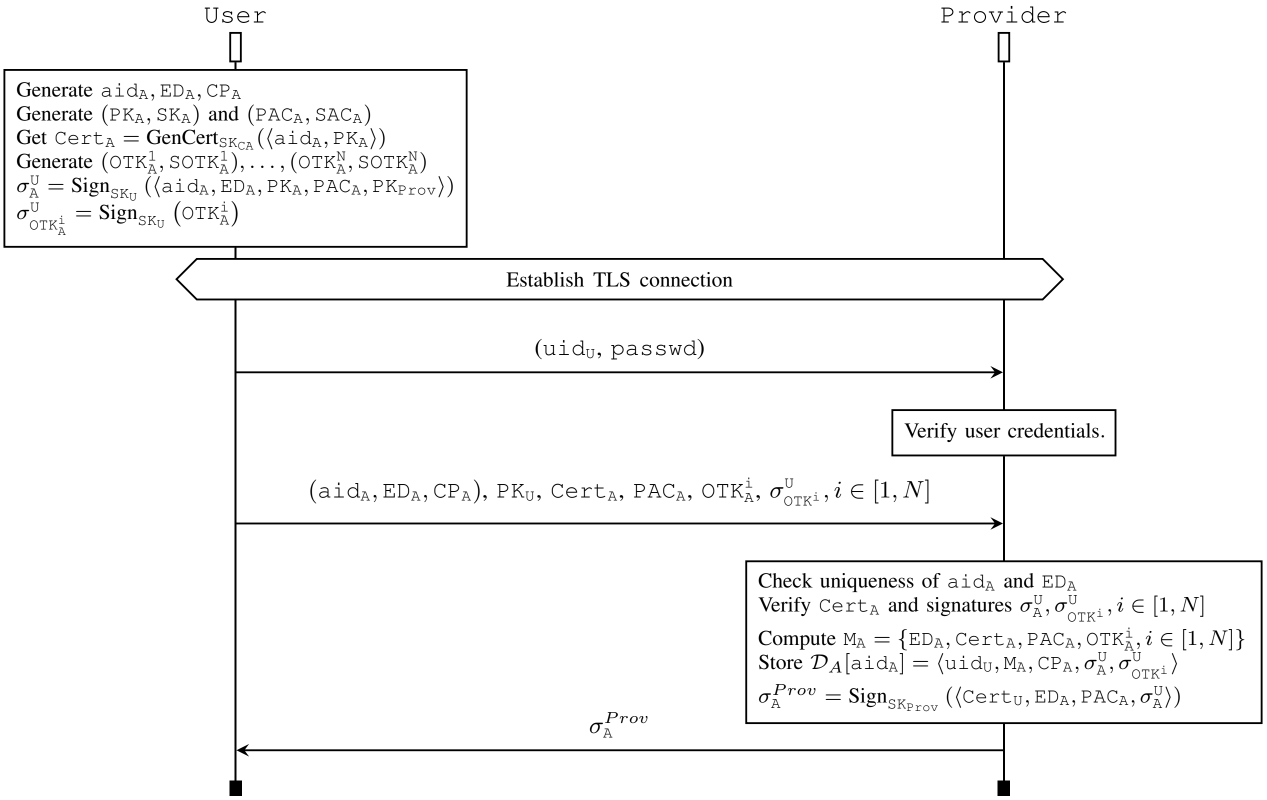
User
Provider
Generate $\mathtt{aid_A}, \mathtt{ED_A}, \mathtt{CP_A}$
Generate $(\mathtt{PK_A}, \mathtt{SK_A})$ and $(\mathtt{PAC_A}, \mathtt{SAC_A})$
Get $\mathtt{Cert_A} = \text{GenCert}_{\mathtt{SK_{CA}}}(\langle \mathtt{aid_A}, \mathtt{PK_A} \rangle)$
Generate $(\mathtt{OTK_A^1}, \mathtt{SOTK_A^1}), \ldots, (\mathtt{OTK_A^N}, \mathtt{SOTK_A^N})$
$\sigma_{\mathtt{A}}^{\mathtt{U}} = \text{Sign}_{\mathtt{SK_U}}(\langle \mathtt{aid_A}, \mathtt{ED_A}, \mathtt{PK_A}, \mathtt{PAC_A}, \mathtt{PK_{Prov}} \rangle)$
$\sigma_{\mathtt{OTK_A^i}}^{\mathtt{U}} = \text{Sign}_{\mathtt{SK_U}}(\mathtt{OTK_A^i})$
Establish TLS connection
$(\mathtt{uid_U}, \mathtt{passwd})$
Verify user credentials.
$(\mathtt{aid_A}, \mathtt{ED_A}, \mathtt{CP_A})$, $\mathtt{PK_U}$, $\mathtt{Cert_A}$, $\mathtt{PAC_A}$, $\mathtt{OTK_A^i}$, $\sigma_{\mathtt{OTK^i}}^{\mathtt{U}}, i \in [1, N]$
Check uniqueness of $\mathtt{aid_A}$ and $\mathtt{ED_A}$
Verify $\mathtt{Cert_A}$ and signatures $\sigma_{\mathtt{A}}^{\mathtt{U}}, \sigma_{\mathtt{OTK^i}}^{\mathtt{U}}, i \in [1, N]$
Compute $\mathtt{M_A} = \{\mathtt{ED_A}, \mathtt{Cert_A}, \mathtt{PAC_A}, \mathtt{OTK_A^i}, i \in [1, N]\}$
Store $\mathcal{D}_A[\mathtt{aid_A}] = \langle \mathtt{uid_U}, \mathtt{M_A}, \mathtt{CP_A}, \sigma_{\mathtt{A}}^{\mathtt{U}}, \sigma_{\mathtt{OTK^i}}^{\mathtt{U}} \rangle$
$\sigma_{\mathtt{A}}^{Prov} = \text{Sign}_{\mathtt{SK_{Prov}}}(\langle \mathtt{Cert_U}, \mathtt{ED_A}, \mathtt{PAC_A}, \sigma_{\mathtt{A}}^{\mathtt{U}} \rangle)$
$\sigma_{\mathtt{A}}^{Prov}$

Fig. 3: **Agent Registration** protocol for agent A by user U.

- TLS credentials $(\mathtt{PK_A}, \mathtt{SK_A})$ to establish secure communication channels with other agents and a signed certificate by the $\mathtt{CA}$:

$$\mathtt{Cert_A} = \text{GenCert}_{\mathtt{SK_{CA}}}(\langle \mathtt{aid_A}, \mathtt{PK_A} \rangle)$$

- A public-private key pair: $(\mathtt{PAC_A}, \mathtt{SAC_A})$ for access control. These long term keys are used for the receiving agent to derive a shared key with the initiating agent for encrypting the access control token in inter-agent communication (Section IV-E).
- A batch of N one-time public and secret key pairs $(\mathtt{OTK_A^1}, \mathtt{SOTK_A^1}), \ldots, (\mathtt{OTK_A^N}, \mathtt{SOTK_A^N})$, used for deriving tokens for controlling access to the agent. Each public one-time key is signed by the user:

$$\sigma_{\mathtt{OTK^i}}^{\mathtt{U}} = \text{Sign}_{\mathtt{SK_U}}(\langle \mathtt{aid_A}, \mathtt{OTK_A^i} \rangle)$$

The user also signs the agent's device and networking information, along with its long-term keys:

$$\sigma_{\mathtt{A}}^{\mathtt{U}} = \text{Sign}_{\mathtt{SK_U}}(\langle \mathtt{aid_A}, \mathtt{ED_A}, \mathtt{PK_A}, \mathtt{PAC_A}, \mathtt{PK_{Prov}} \rangle)$$

The inclusion of $\mathtt{PK_{Prov}}$ in the signature indicates that the agent is registered with the specified `Provider`.

3) **Specifying the contact policy**: The user specifies the agent's contact policy $\mathtt{CP_A}$ (discussed in Section IV-D).
4) **User authentication to `Provider`**. The user establishes a secure TLS connection with the `Provider`, and authenticates by providing credentials $\langle \mathtt{uid_U}, \mathtt{passwd} \rangle$. The `Provider` verifies the credentials and proceeds if successful.
5) **Registration submission**. The user submits to the `Provider`: the agent's information $(\mathtt{aid_A}, \mathtt{ED_A}, \mathtt{CP_A})$, the TLS certificate $\mathtt{Cert_A}$, the public access control and one-time keys $\mathtt{PAC_A}, \{\mathtt{OTK_A^1}, \ldots, \mathtt{OTK_A^N}\}$, and signatures $\sigma_{\mathtt{A}}^{\mathtt{U}}$, $\sigma_{\mathtt{OTK^i}}^{\mathtt{U}}$ for $i \in [1, N]$.
The agent stores locally all the private keys corresponding to the public keys submitted to the provider: $(\mathtt{SK_A}$, $\mathtt{SAC_A}$, $\{\mathtt{SOTK_A^1}, \ldots, \mathtt{SOTK_A^N}\})$.
6) **Provider verification**. The `Provider` processes the registration request by checking that $\mathtt{aid_A}$ and $\mathtt{ED_A}$ are globally unique within the ecosystem, and verifying $\mathtt{Cert_A}$ and signatures:

$$\text{Verify}_{\mathtt{PK_U}}(\langle \mathtt{aid_A}, \mathtt{ED_A}, \mathtt{PK_A}, \mathtt{PAC_A}, \mathtt{PK_{Prov}} \rangle, \sigma_{\mathtt{A}}^{\mathtt{U}})$$

$$\text{Verify}_{\mathtt{PK_U}}(\langle \mathtt{aid_A}, \mathtt{OTK_A^i} \rangle, \sigma_{\mathtt{OTK^i}}^{\mathtt{U}})$$

7) **Completion.** Upon successful verification, the `Provider` stores in the agent registry the agent's metadata $\mathtt{M_A}$, the contact policy $\mathtt{CP_A}$, the agent's signatures $\sigma_{\mathtt{A}}^{\mathtt{U}}$ and $\sigma_{\mathtt{OTK^i}}^{\mathtt{U}}$, along with $\mathtt{uid_U}$ to associate the agent A with user U:

$$\mathtt{M_A} = \{\mathtt{ED_A}, \mathtt{Cert_A}, \mathtt{PAC_A}, \mathtt{OTK_A^i}, i \in [1, N]\}$$
$$\mathcal{D}_A[\mathtt{aid_A}] = \langle \mathtt{uid_U}, \mathtt{M_A}, \mathtt{CP_A}, \sigma_{\mathtt{A}}^{\mathtt{U}}, \sigma_{\mathtt{OTK^i}}^{\mathtt{U}} \rangle$$

The `Provider` then signs the agent `A`'s information

$$\sigma_{\mathtt{A}}^{Prov} = \mathsf{Sign}_{\mathtt{SK}_{\mathtt{Prov}}}\left(\langle \mathtt{aid}_{\mathtt{A}}, \mathtt{Cert}_{\mathtt{A}}, \mathtt{ED}_{\mathtt{A}}, \mathtt{PAC}_{\mathtt{A}}, \sigma_{A}^{\mathtt{U}}\rangle\right)$$

and returns it as confirmation to the user. The user stores this signature, which it uses when initiating agent communication (Section IV-E).

The agent A is now officially registered and can securely communicate within the ecosystem.

D. Agent Management

Agent management in SAGA involves managing access control polices and keys, enabling updates to policies, and cryptographic key management.

Access control management. There are two fundamental capabilities in SAGA for managing access control: (1) specification and enforcement of the *Agent Contact Policy* (`CP`), and (2) provisioning of fresh *One-Time Keys* (`OTK`) to facilitate user-controlled communication.

1) Agent Contact Policy: In SAGA, each agent is associated with a contact policy `CP` that specifies which initiating agents are authorized to establish contact. This policy is defined by the user when registering their agent, and is enforced by the `Provider` during contact resolution. `CP` consists of a set of declarative rules, along with the number of `OTK`s that should be allotted to an agent that matches that rule. These rules may use pattern matching over agent identifiers to allow flexible yet controlled specification of acceptable contacts. For instance, the rule `("*@company.com:email_agent", 10)` permits any email-handling agent from a specified domain to initiate contact and be allotted at most 10 `OTK`s.

If multiple rules match, the one with the *highest specificity* is selected. For instance in Listing 1, an agent `alice@company.com:calendar_agent` attempts to contact the receiving agent. The agent's identifier matches the top three patterns, but the first rule is the most specific and therefore determines the `OTK` budget, resulting in 15 `OTK`s.

```
// Agent C's Contact Policy
[
  {
    "agents": "alice@company.com:calendar_agent",
    "budget": 15
  },
  {
    "agents": "*@company.com:calendar_agent",
    "budget": 10
  },
  {
    "agents": "*@company.com:*",
    "budget": 25
  },
  {
    "agents": "bob@mail.com:*",
    "budget": 100
  }
]
```

Listing 1: Example contact policy for an agent. More specific patterns take precedence over general ones.

Formally, for a receiving agent `A` with contact policy $\mathtt{CP}_{\mathtt{A}}$ and an initiating agent `B` with identifier $\mathtt{aid}_{\mathtt{B}}$, the number of `OTK`s issued is defined as:

$$\text{Budget}_{\mathtt{OTK}}(\mathtt{aid}_{\mathtt{A}}, \mathtt{aid}_{\mathtt{B}}) = \begin{cases} -1 & \text{if } \mathcal{R} = \emptyset \\ \mathtt{B}(r^*) & \text{if } \mathcal{R} \neq \emptyset \end{cases},$$

where r^* is the most specific rule among all rules $\mathcal{R}$ in $\mathtt{CP}_{\mathtt{A}}$ that match $\mathtt{aid}_{\mathtt{B}}$, and $\mathtt{B}(r^*)$ corresponds to the budget for rule r^* indicated in $\mathtt{CP}_{\mathtt{A}}$. The distinction for $\mathcal{R} = \emptyset$ helps the user differentiate between no match and an expired `OTK` budget.

2) One-Time Key Generation: To control communication to registered agents, the `Provider` facilitates the distribution of `OTK`s: one-time keys generated and uploaded by each user for their agents. Each `OTK` is consumed when an initiating agent contacts the `Provider` to initiate contact with a receiving agent. `OTK`s are defined for receiving agents, as the protocol is asymmetric. Additionally, the `Provider` maintains the number of `OTK`s remaining for a particular agent communicating with another agent. While it might be possible for the initiating agent to receive multiple `OTK`s from the `Provider` reducing the number of times an agent must contact the `Provider`, we chose the design where the agent receives only one `OTK` to limit the vulnerability window, in case an agent is compromised.

When an initiating agent `B` queries the `Provider` to contact a recipient agent `A`, the `Provider` first verifies that the initiating agent satisfies `A`'s Agent Contact Policy ($\mathtt{CP}_{\mathtt{A}}$), as detailed above. If this is the first time `B` is contacting `A`, the `Provider` creates a counter $\text{Counter}_{\mathtt{OTK}}[\mathtt{aid}_{\mathtt{A}}][\mathtt{aid}_{\mathtt{B}}]$ to keep track of the number of remaining `OTK`s, and initializes it with $\text{Budget}_{\mathtt{OTK}}(\mathtt{aid}_{\mathtt{A}}, \mathtt{aid}_{\mathtt{B}})$.

If the policy check succeeds and a valid `OTK` is available (indicated by a positive value for $\text{Counter}_{\mathtt{OTK}}[\mathtt{aid}_{\mathtt{A}}][\mathtt{aid}_{\mathtt{B}}]$), the `Provider` returns an `OTK` to the initiating agent (along with the recipient's metadata and its signature, discussed in Section IV-E) and decreases the counter by one.

Obtaining an `OTK` at this stage may fail due to: (a) exhaustion of $\mathtt{aid}_{\mathtt{A}}$'s overall `OTK` pool, or (b) depletion of $\mathtt{aid}_{\mathtt{B}}$'s `OTK` quota as defined by $\mathtt{CP}_{\mathtt{A}}$. The user of agent `A` can update the contact policy and generate additional `OTK`s at any time.

Policy Updates and Revocation. Users can dynamically update their agents' contact policies via the `Provider` interface. Updates may include adding rules (e.g., to onboard collaborators) or removing them (e.g., to revoke access), allowing policies to adapt as trust relationships shift or threats emerge.

It is important to allow users to block contact from specific agents, as they might detect abusive behavior. To block specific agents, a receiving agent can update its contact policy with a rule that assigns a $\mathtt{B}(\cdot)$ score of -1. The modified `CP` is then pushed to the `Provider`. Alternatively, to completely disable incoming contact, a user can request the `Provider` to deactivate their agent at any time. This functionality is motivated by our principle of user controlling their agents entire lifecycle. Crucially, we do not allow users the ability to deactivate agents registered by other users, as they should control only their own agents.

Cryptographic key management. Cryptographic keys for agents should be managed by users according to best practice principles for key management and key rotation [25]. In particular, users should periodically rotate their agents' TLS key and access control keys. Best practices should also be followed by the `Provider` for user authentication and password management.

E. Inter-Agent Communication

In previous sections, we described how a user can register an agent and how it can manage the access to that agent through the registry information maintained by the `Provider`. Here, we detail how two registered agents can communicate with each other securely, while respecting the Access Contact Policy defined by users for their agents.

When an agent `B` (initiating agent) wants to contact agent `A` (receiving agent), it first queries the `Provider` to verify `A`'s registration. If permitted by `A`'s contact policy, `B` receives `A`'s metadata and a one-time key (`OTK`) from the `Provider`. This `OTK` is used by both agents to derive a shared key, which the receiving agent `A` will use to encrypt an access control token (ACT) for `B`.

Assume agents `B` and `A` are registered under users `U2` and `U1`, respectively. Each agent uses several types of cryptographic keys: TLS credentials, including signing keys, and access control keys for regulating communication. The latter includes long-term access control keys (`PAC`), and one-time keys (`OTK`). The `Provider` stores one-time keys and enables the enforcement of each agent's contact policy. A valid ACT (which we refer to as `token`) is bound to both the initiating and the receiving agents' identities by deriving a key based on the PAC of the initiating agent and `OTK` of the receiving agent. The ACT is scoped to a specific task, but the granularity of tokens can be adjusted further if desired. The token will have limited validity, as well as a limit on the total number of requests.

The following steps outline the agent communication protocol involving initiating agent `B` contacting receiving agent `A` to obtain an access control token (shown in Figure 4). This protocol runs either the first time agent `B` contacts `A`, or when previously issued tokens have expired or exceeded their usage limits.

1) **Establishing a TLS connection with the `Provider`:** This step follows a standard TLS session establishment between `B` and the `Provider`.
2) **Receiving agent information retrieval:** `B` requests permission to contact `A` by specifying their identity ($\mathtt{aid}_\mathtt{B}$) and the identity of the receiving agent ($\mathtt{aid}_\mathtt{A}$). The `Provider` verifies that `B` is in `A`'s contact policy and has sufficient `OTK`s allotted to it (by making sure $\text{Counter}_\mathtt{OTK}[\mathtt{aid}_\mathtt{A}][\mathtt{aid}_\mathtt{B}] > 0$). It returns `A`'s access information: user's certificate $\mathtt{Cert}_\mathtt{U1}$, agent's device and network information ($\mathtt{aid}_\mathtt{A}, \mathtt{ED}_\mathtt{A}$), agent's TLS and access control keys ($\mathtt{Cert}_\mathtt{A}, \mathtt{PAC}_\mathtt{A}$) and a signed one-time key $\mathtt{OTK}^\mathtt{i}_\mathtt{A}, \sigma^{\mathtt{U1}}_{\mathtt{OTK}^\mathtt{i}}$.
Subsequently, the `Provider` decrements the counter $\text{Counter}_\mathtt{OTK}[\mathtt{aid}_\mathtt{A}][\mathtt{aid}_\mathtt{B}]$ by one.
3) **Receiving agent information verification:** `B` first verifies `A`'s user's certificate $\mathtt{Cert}_\mathtt{U1}$ including the user's public key $\mathtt{PK}_\mathtt{U1}$. `B` also verifies the signatures on agent's `A` information and the received `OTK` as follows:
$$\text{Verify}_{\mathtt{PK}_\mathtt{U1}}(\langle \mathtt{aid}_\mathtt{A}, \mathtt{ED}_\mathtt{A}, \mathtt{PK}_\mathtt{A}, \mathtt{PAC}_\mathtt{A}, \mathtt{PK}_\mathtt{Prov}\rangle, \sigma^\mathtt{U}_\mathtt{A})$$
$$\text{Verify}_{\mathtt{PK}_\mathtt{U1}}(\langle \mathtt{aid}_\mathtt{A}, \mathtt{OTK}^\mathtt{i}_\mathtt{A}, \rangle, \sigma^\mathtt{U}_{\mathtt{OTK}^\mathtt{i}})$$
4) **Establishing a TLS connection between agents:** `B` initiates a TLS connection with `A`, and both agents verify each other's certificates ($\mathtt{Cert}_\mathtt{A}, \mathtt{Cert}_\mathtt{B}$).
5) **Token request sent:** `B` sends `A` its information and a signature $\sigma^{Prov}_\mathtt{B}$ from the `Provider` (generated during agent registration: step 7), along with one-time key $\mathtt{OTK}^\mathtt{i}_\mathtt{A}$, requesting access.
6) **Token request received:** `A` verifies `U2`'s certificate, as well as $\sigma^{Prov}_\mathtt{B}$:
$$\text{Verify}_{\mathtt{PK}_\mathtt{Prov}}(\langle \mathtt{aid}_\mathtt{B}, \mathtt{Cert}_\mathtt{B}, \mathtt{ED}_\mathtt{B}, \mathtt{PAC}_\mathtt{B}, \sigma^{\mathtt{U2}}_\mathtt{B}\rangle, \sigma^{Prov}_\mathtt{B})$$
If $\mathtt{OTK}^\mathtt{i}_\mathtt{A}$ is valid, both agents perform a Diffie-Hellman (DH) key exchange protocol to derive a shared key:
$$\mathtt{DH}_\mathtt{A} = DH(\mathtt{SOTK}^\mathtt{i}_\mathtt{A}, \mathtt{PAC}_\mathtt{B}), \quad \mathtt{DH}_\mathtt{B} = DH(\mathtt{SAC}_\mathtt{B}, \mathtt{OTK}^\mathtt{i}_\mathtt{A})$$
$$\mathtt{SDHK} = KDF(\mathtt{DH}_\mathtt{A}) = KDF(\mathtt{DH}_\mathtt{B})$$
7) **Token generation:** `A` creates the access token with a randomly generated nonce ($\mathtt{N} \xleftarrow{\$} \mathcal{R}$), issue ($\mathtt{T}_\mathtt{issued}$) and expiration ($\mathtt{T}_\mathtt{expire}$) timestamps, the maximum number of requests linked to this token ($\mathtt{Q}_\mathtt{max}$), and `B`'s access control key ($\mathtt{PAC}_\mathtt{B}$):
$$\mathtt{token} = \text{Enc}_\mathtt{SDHK}(\langle \mathtt{N}, \mathtt{T}_\mathtt{issued}, \mathtt{T}_\mathtt{expire}, \mathtt{Q}_\mathtt{max}, \mathtt{PAC}_\mathtt{B}\rangle).$$
`A` stores the token and sends it to `B`.
8) **Inter-Agent communication:** `B` receives the token and initiates the conversation to complete its task. For each subsequent request to `A`, `B` attaches the token. Upon receiving a request, `A` verifies that the token was issued for `B` (not for another agent), has not expired, and has not exceeded its usage quota. Once a task is deemed completed, the token is discarded by both parties.

Token reuse. Once an `Agent` obtains a token, it can reuse it as long as it remains valid and hasn't exceeded its request limit. If `B` holds a valid token for `A`, it can skip ahead to step 8 to initiate secure communication. If the agents' existing TLS session is reset, the agents will establish a new TLS session (step 1) and proceed directly to agent communication (step 8). The expiration time and request limit in the token offer a balance between security and performance considerations. A larger number of requests $\mathtt{Q}_\mathtt{max}$ reduces the overhead of contacting the `Provider` to obtain `OTK`s, but increases the potential exposure in case an agent is compromised.

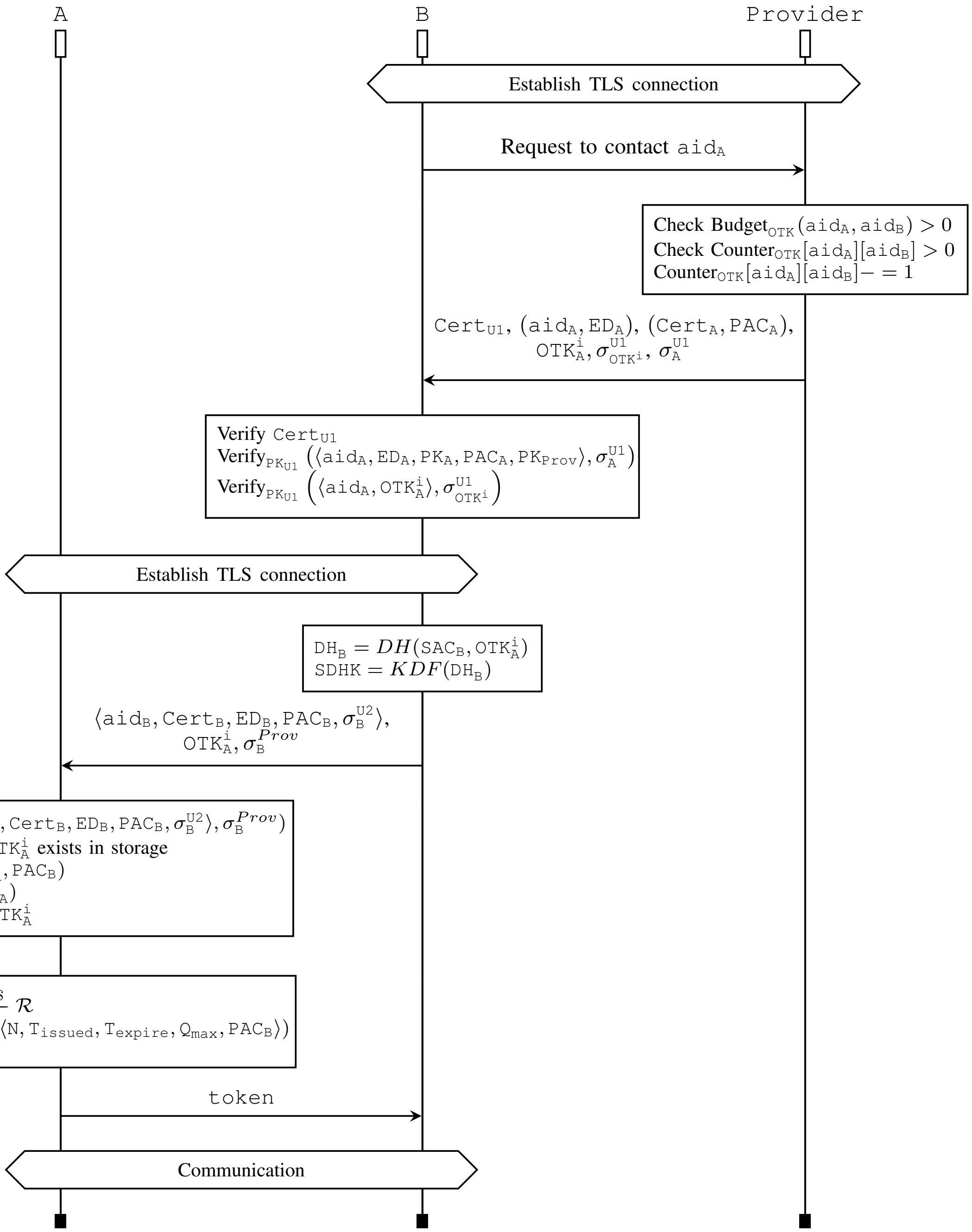
A
B
Provider
Establish TLS connection
Request to contact $\texttt{aid}_\texttt{A}$
Check Budget$_\texttt{OTK}(\texttt{aid}_\texttt{A}, \texttt{aid}_\texttt{B}) > 0$
Check Counter$_\texttt{OTK}[\texttt{aid}_\texttt{A}][\texttt{aid}_\texttt{B}] > 0$
Counter$_\texttt{OTK}[\texttt{aid}_\texttt{A}][\texttt{aid}_\texttt{B}] -= 1$
$\texttt{Cert}_\texttt{U1}$, $(\texttt{aid}_\texttt{A}, \texttt{ED}_\texttt{A})$, $(\texttt{Cert}_\texttt{A}, \texttt{PAC}_\texttt{A})$,
$\texttt{OTK}^\texttt{i}_\texttt{A}, \sigma^\texttt{U1}_{\texttt{OTK}^\texttt{i}}$, $\sigma^\texttt{U1}_\texttt{A}$
Verify $\texttt{Cert}_\texttt{U1}$
Verify$_{\texttt{PK}_\texttt{U1}}\left(\langle \texttt{aid}_\texttt{A}, \texttt{ED}_\texttt{A}, \texttt{PK}_\texttt{A}, \texttt{PAC}_\texttt{A}, \texttt{PK}_\texttt{Prov}\rangle, \sigma^\texttt{U1}_\texttt{A}\right)$
Verify$_{\texttt{PK}_\texttt{U1}}\left(\langle \texttt{aid}_\texttt{A}, \texttt{OTK}^\texttt{i}_\texttt{A}\rangle, \sigma^\texttt{U1}_{\texttt{OTK}^\texttt{i}}\right)$
Establish TLS connection
$\texttt{DH}_\texttt{B} = DH(\texttt{SAC}_\texttt{B}, \texttt{OTK}^\texttt{i}_\texttt{A})$
$\texttt{SDHK} = KDF(\texttt{DH}_\texttt{B})$
$\langle \texttt{aid}_\texttt{B}, \texttt{Cert}_\texttt{B}, \texttt{ED}_\texttt{B}, \texttt{PAC}_\texttt{B}, \sigma^\texttt{U2}_\texttt{B}\rangle$,
$\texttt{OTK}^\texttt{i}_\texttt{A}, \sigma^{Prov}_\texttt{B}$
Verify$_{\texttt{PK}_\texttt{Prov}}(\langle \texttt{aid}_\texttt{B}, \texttt{Cert}_\texttt{B}, \texttt{ED}_\texttt{B}, \texttt{PAC}_\texttt{B}, \sigma^\texttt{U2}_\texttt{B}\rangle, \sigma^{Prov}_\texttt{B})$
Verify $\texttt{OTK}^\texttt{i}_\texttt{A} \rightarrow \texttt{SOTK}^\texttt{i}_\texttt{A}$ exists in storage
$\texttt{DH}_\texttt{A} = DH(\texttt{SOTK}^\texttt{i}_\texttt{A}, \texttt{PAC}_\texttt{B})$
$\texttt{SDHK} = KDF(\texttt{DH}_\texttt{A})$
Delete $\texttt{OTK}^\texttt{i}_\texttt{A} \rightarrow \texttt{SOTK}^\texttt{i}_\texttt{A}$
Generate nonce $\texttt{N} \xleftarrow{\$} \mathcal{R}$
$\texttt{token} = \text{Enc}_\texttt{SDHK}(\langle \texttt{N}, \texttt{T}_\texttt{issued}, \texttt{T}_\texttt{expire}, \texttt{Q}_\texttt{max}, \texttt{PAC}_\texttt{B}\rangle)$
Store $\texttt{token}$
token
Communication

Fig. 4: **Agent Communication** protocol between initiating agent `B` and receiving agent `A` to obtain an Access Control Token with the `Provider` mediation if first communication between agents or the previous token expired.

F. Security analysis of SAGA

In this section, we discuss how SAGA protects against the adversarial capabilities detailed in Section III-D.

C1: Adversarial agents created by attackers

C1 Protections: SAGA has several protections to prevent abuse by adversarial agents. First, agents can only be registered by users authenticated by the `Provider`. Second, each agent has an Agent Contact Policy created by its user when registering the agent. Third, even if an adversarial agent is added to a victim agent's contact policy, and initiates communication with the victim, the access control token ensures a limited window of vulnerability in which the two agents can communicate. If abuse is detected, the user owning the victim agent can block the adversarial agent by updating its contact policy.

C2: Compromised legitimate agents

C2 Protections: In this scenario, the compromised agent assumes the identity of legitimate agent A and can potentially reach out to any agents from A's contact list. Furthermore, the compromised agent can utilize any access control tokens previously obtained by A prior to the compromise. However, we restrict the attack surface by allowing the agent to receive only one OTK at a time, and limiting the validity of the tokens to a specific time interval or a maximum number of requests.

C3: Agent self-replication

C3 Protections: While self-replication of agents has been demonstrated in prior work [19], SAGA prevents the registration of the child agent with the Provider since only users are permitted to register agents. Consequently, the child agent inherits the same identity, TLS, and access control keys as the parent agent. In effect, the replicated agent is constrained to operate within the parent agent's capabilities.

C4: Sharing keys from an adversarial agent A to agent B

C4 Protections: While this information allows B to contact any victim agent in A's contact list, it only grants B the same capabilities as A. For example, the access control token remains usable only until it expires, hits its request limit, or A is deactivated or removed from the victim's contact list.

C5: Creating agents with multiple identities (Sybil attacks)

C5 Protections: Sybil attacks are prevented in our system as each agent is required to be uniquely identified and registered with the Provider.

V. EVALUATION

To evaluate SAGA, we implement its full stack (Section V-A) and test it under both adversarial (Section V-B) and benign scenarios, where we measure protocol overhead (Section V-C) and demonstrate that the protocol does not interfere with the task completion of underlying LLM-agents (Section V-D).

A. *Implementation*

The Provider is implemented as an HTTPS service exposing dedicated endpoints for each user-provider and agent-provider interaction defined in the protocol. Inter-agent communication is conducted over TLS configured with mutual authentication, with protocol-level authentication and encryption enforced via ephemeral session keys. Our framework is agnostic to the underlying LLM-agent implementation. This design enables seamless integration with arbitrary agent implementations or libraries.

All cryptographic operations in the protocol are built on Curve25519 [26]. Both long-term and ephemeral keys are generated using the X25519 elliptic-curve Diffie-Hellman (ECDH) [27] scheme, which produces 256-bit shared secrets. Certificates adhere to the X.509 PKI standard [28] and are issued by an internal certificate authority (CA) deployed as part of the provider. All digital signatures and key derivation steps utilize the SHA256 hash function [23].

For LLM agents, we experimented with a local Qwen-2.5 [29] 72B model running on an NVIDIA H100 GPU, as well as two OpenAI models hosted in the cloud and accessed via API.

B. *Attacker Evaluation*

We evaluate SAGA's resilience against adversarial behavior as defined in the threat model in Section III-D. We consider 8 adversarial agents that attempt to bypass authentication or access control, including unauthorized connections, token misuse, and contact policy violations. As noted in Section III-C, we assume the adversary cannot compromise the TLS layer, break standard cryptographic primitives (e.g., signature forgery or DH key extraction), or bypass human verification during user registration. To verify that SAGA does indeed work in defending against such threats, we implemented these attacker models in our evaluation.

We denote the adversarial agent as M and the victim agent as A. Table II summarizes the attack models, the required adversarial capabilities, and the specific protocol step at which each attack is detected. The following attack models represent distinct ways in which the protocol may be misused or subverted by an adversarial agent:

A1: M attempts to contact A without presenting valid TLS credentials.

In this case, the TLS handshake fails since M cannot authenticate to A. The connection is rejected before any application-layer interaction takes place.

A2: M contacts A without a one-time access key or token.

Here, M completes the TLS handshake and is authenticated, but fails to provide either a valid $\mathrm{OTK}_{\mathrm{A}}^{i}$ or a previously issued access token. A terminates the session without issuing a token.

A3: M contacts A with an invalid token.

The connection is established and the token is received by A, but the token is invalid, either because its timestamp has passed or it has exceeded its limit. Upon detecting this token replay, A immediately closes the connection.

A4: M attempts to impersonate benign agent A by providing its Provider-issued signature and public information upon contact with other agents.

M uses A's public information and the provider-issued signature ($\sigma_{\mathrm{M}}^{Prov} \leftarrow \sigma_{\mathrm{A}}^{Prov}$) when contacting other agents. Receiving agents verify the signature using M's information (Step 6, Section IV-E), and the verification fails.

A5: M contacts A and reuses a valid token issued by A to a different agent.

Attack Model	Capabilities	Protocol Step Detected	Details
A1	C1, C2	IV-E, Step 4	The receiving agent rejects the TLS connection attempt, as a valid certificate is required to establish communication.
A2	C1, C2	IV-E, Step 6	The receiving agent terminates the session due to the absence of a valid access control credential (either an OTK or token).
A3	C1, C2	IV-E, Step 8	The receiving agent detects that the token presented is either expired or has exceeded its usage quota, and aborts the connection.
A4	C1, C2	IV-E, Step 6	The provider-issued signature on the initiating agent's metadata fails verification, as the metadata does not match the registered information.
A5	C1, C2, C3, C4	IV-E, Step 8	The token presented by the initiating agent was issued to a different agent, and the mismatch is detected by the receiving agent during token validation, leading to connection termination.
A6	C1, C2, C3, C4	IV-E, Step 2	The Provider denies issuance of an OTK since the initiating agent is not permitted under the receiving agent's contact policy.
A7	C3	IV-B, Step 5	User registration requires human verification. As only users can register agents in SAGA, the adversarial agent fails to satisfy the verification requirements and is denied registration.
A8	C1, C2, C3, C4	IV-E, Token reuse	The malicious initiating agent proceeds with establishing a secure connection with the receiving agent. However, there's a limited amount of communication between the two parties dictated by the token quota or the expiry date.

TABLE II: Different attack models with corresponding required adversarial capabilities, and the specific SAGA protocol step at which each attack is detected.

In this scenario, A has issued a valid token to some other agent, which is later intercepted or exfiltrated by M. When M attempts to use it, *A* detects the mismatch between the token's intended recipient and the initiating party's identity (PAC$_{M}$). The connection is immediately aborted.

> **A6**: M attempts to contact A despite not being authorized under A's contact policy.

The Provider enforces contact policies on behalf of receiving agents. When M attempts to query or initiate a connection with A, the Provider denies the request based on A's contact policy.

> **A7**: M attempts to register copies of itself or other agents with the Provider.

M attempts to register new identities, either by self-replication or creating agent variants. However, the Provider requires human-in-the-loop verification during user registration (see Section IV-B), enforced through integration with trusted identity services.

> **A8**: M contacts A with a valid token.

This represents a short-lived window of vulnerability. The protocol accepts the connection, but the token's expiration and communication quota strictly bound the scope of abuse. Once either limit is reached, the token becomes invalid and further requests are rejected. If malicious activity is detected, A's user may blocklist M, thereby preventing it from obtaining new OTKs.

C. Overhead Evaluation

In this subsection we evaluate the cost of enforcing SAGA's security guarantees along the agent communication path. This includes both fixed algorithmic costs of protocol operations and the dynamic impact of access control token quotas, cryptographic validation, and Provider-mediated authorization. Our results show that the overhead is low and decreases rapidly as the frequency of agent interactions increases.

Cryptographic Overhead. We begin with measuring the cryptographic overhead of core protocol operations at the user, Provider, and agents. These costs cover cryptographic primitives such as hashing, key generation, signing, verification, and Diffie-Hellman key exchange. As visible in Table III most operations are lightweight, on the order of several ms overhead. The highest cost components are user and agent registration at the provider (194 ms and 213 ms), due to certificate issuance and registry updates. As these are one-time operations (per user and agent), they do do not impact inter-agent communication.

Protocol Component	Overhead (ms)
User Registration	
User Registration (User)	2.34
User Registration (Provider)	194.09
Agent Registration	
Agent Registration (User)	15.09
Agent Registration (Provider)	212.85
Agent Communication	
Contact Resolution (Provider)	1.46
Setup Phase (Initiator)	2.14
Setup Phase (Receiver)	1.83
Access Control Tokens	
Token Generation (Initiator)	1.03
Token Decryption (Receiver)	1.20
Token Validation (Initiator)	0.24
Token Validation (Receiver)	0.26

TABLE III: Computational overhead of key operations in the SAGA protocol.

Protocol Overhead. We measure the overhead introduced by SAGA's access control and provider coordination mechanisms (Section V-C). Specifically, we analyze the overhead incurred by an initiating agent `B` issuing m requests to a receiving agent `A`. This includes a network component for establishing secure communication, and a cryptographic component t_{crypto} for certificate validation, signature verification, Diffie-Hellman key exchange, key derivation, token encoding, and symmetric encryption. The total protocol overhead is modeled as:

$$c_{\text{proto}}(m) = (\text{RTT}_{B,P} + t_{\text{crypto}}) \cdot \left\lceil \frac{m}{\mathtt{Q}_{\mathtt{max}}} \right\rceil, \qquad (1)$$

where P is the `Provider`, and $\text{RTT}_{B,P}$ is the round-trip time for agent `B` contacting the `Provider` and receiving a response. Each authorization cycle involves agent `B` retrieving metadata and a one-time key for agent `A` from the `Provider`. This round-trip, along with local cryptographic operations, must be performed once every $\mathtt{Q}_{\mathtt{max}}$ requests, as the token quota is exhausted.

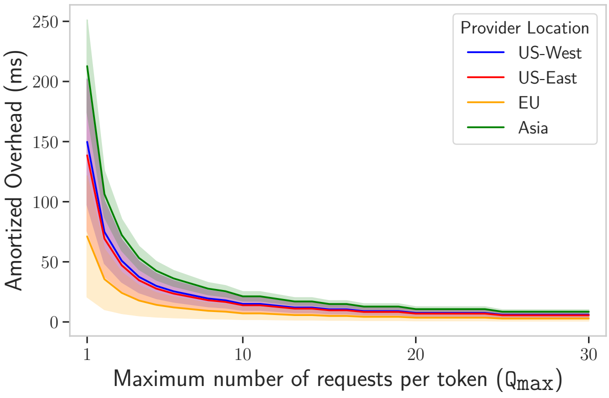

Fig. 5: Amortized protocol overhead per request $\bar{c}_{\text{proto}}(m)$ as a function of maximum number of requests token is reused. We measure the overhead for several geographic locations for the `Provider`. The shaded region reflects variability for agents position worldwide.

We sample round-trip times ($\text{RTT}_{B,P}$) from empirical measurement distributions using monitors in US-East, US-West, Europe and Asia, made available by CAIDA [30] and AWS [31], and use these to approximate protocol overhead. Figure 5 shows the amortized protocol setup overhead:

$$\bar{c}_{\text{proto}}(m) = \frac{c_{\text{proto}}(m)}{m}$$

as a function of token quota $\mathtt{Q}_{\mathtt{max}}$, using $m = 100$ requests, where the measured cryptographic overhead $t_{\text{crypto}} = 7\,\text{ms}$. As visible, the protocol overhead decreases sharply with increasing token quota $\mathtt{Q}_{\mathtt{max}}$, demonstrating how it can be effectively amortized across inter-agent interactions. Moreover, the overhead is low for all `Provider` geolocations, with slightly higher overhead for Europe and Asia.

We also evaluate the effect of geolocation on protocol overhead by varying the placement of the initiating agent under a fixed `Provider` location (US-West). The overhead becomes very low——typically under 25ms when agents interact for at least 4–5 requests (Figure 6). This effect is even more pronounced when the agent and provider are geographically close.

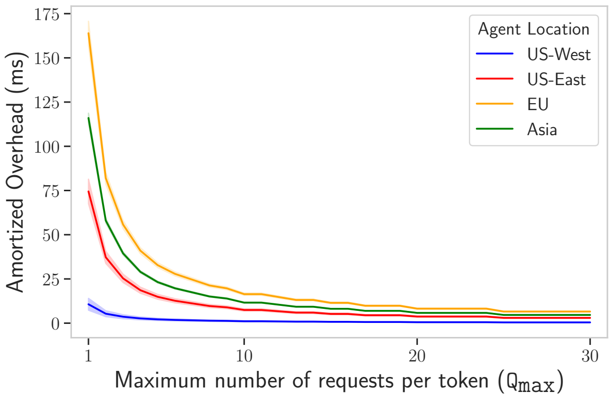

Fig. 6: Amortized protocol overhead per request $\bar{c}_{\text{proto}}(m)$ as a function of maximum number of requests token is reused. We measure the overhead for several geographic locations for the initiating agent, where the `Provider` is fixed in US-West. The shaded region captures variability under sampled network conditions.

Token Quota. The token quota $\mathtt{Q}_{\mathtt{max}}$ defines the number of requests allowed before contacting the `Provider`. A larger quota reduces latency and reliance on the provider but increases the window for compromised agents to operate without interruption. Smaller quotas mitigate security risks by requiring more frequent cryptographic validation, at the expense of additional overhead.

D. Task Completion

SAGA operates as a protocol layer above the underlying LLM agents, whose communications are not impacted by the protocol. To illustrate SAGA in realistic agent interactions, we deploy three types of agents: (a) *Calendar* agents that determine a mutually available time and schedule a meeting, (b) *Email* agents that extract relevant expense-related emails and collaborate to submit an expense report, and (c) *Writing* agents with different expertise areas collaborating to write a blogpost. As expected, all agents successfully complete their tasks. See Appendix A for more details.

Task	**LLM Backend**	**Standard Cost** LLM	Networking	**SAGA Overhead**
Calendar	`GPT-4.1-mini`	50.001	0.791	0.165
Email	`GPT-4.1`	26.862	1.319	0.165
Writing	`Qwen-2.5`	363.563	1.319	0.165

TABLE IV: Task execution time breakdown (in seconds). We assume that `A`, `B`, and the `Provider` are located in Asia, Europe, and US-West, respectively, and the token quota is 10. **Standard Cost** is the minimum runtime for two agents communicating directly without SAGA, including LLM cost and network latency.

To contextualize SAGA's overhead, we also measure the standard task completion cost i.e., the time taken by the LLM to generate responses, and the network latency (Table IV, under **Standard Cost**). The LLM response time depends on both model execution speed and task complexity. For example, using a local `Qwen-2.5` model instead of the cloud-based `GPT-4.1` model for the Email task increases the runtime from 26.862 to 43.730 seconds, as the Qwen-2.5 (72B) model is slower than the highly optimized models served by OpenAI. Tasks like blog post writing require substantially more input and output tokens and result in longer runtimes, as observed for the Writing task. Since most of the task completion time is spent by LLM-agents during intermediate planning [32], and tool calls [33], the amortized overhead of our protocol is significantly lower in comparison. For example, even when agents and the `Provider` are geographically distant, the protocol overhead accounts for less than 0.6% of the end-to-end cost of completing the fastest calendar task.

VI. RELATED WORK

This section reviews existing frameworks for secure agent interactions and current implementation landscapes, highlighting the gaps our work addresses. We also explore existing multi-agent attacks, highlighting their limited scope.

High-Level Designs for Inter-Agent Interaction. Several frameworks and protocols have been proposed to govern AI agent interactions securely. South et al. [12] introduce a framework for authenticated delegation using OAuth 2.0 design patterns, where users register their agents with providers and issue delegation tokens. Their system incorporates W3C verifiable credentials for decentralized certification and natural-language interfaces for permission scoping, translating user instructions into structured formats for review. Chan et al. [11] outline agent infrastructure protocols defining interaction standards, focused on three core functions: attribution, interaction, and response, Chan et al. [10] propose a system for agent identification that tracks AI systems along with their context windows and initial users. Their framework enables tracking potential misuse through linked IDs and detecting jailbreak attempts through context inclusion. Shavit et al. [9] discuss governance practices for agentic systems, highlighting unique agent identification and interruptibility as critical features. They specifically note the challenge of preventing identity spoofing and emphasize the importance of graceful shutdown procedures for revoking permissions. It should be noted that none of these designs are actually implemented and evaluated in a real system.

Inter-Agent Protocol Implementations. Recent surveys [34] reveal that while multi-agent systems are extensively studied, implementations typically assume pre-established connections between agents with static communication patterns. Yan et al. [35] specifically note the need for "encryption and authentication protocols tailored for decentralized multi-agent environments." Smyth OS offers a portal for agent creation and integration limited within a particular organization. Current software implementations for agent interactions generally fall short of enabling secure cross-organizational communication. While frameworks like LangChain's Agent Protocol provide specifications for interactions, they do not address critical aspects like authentication or access control mechanisms. Commercial platforms like Amazon Bedrock [36] enable agent orchestration but assume all agents belong to the same restricted environment. Packages like AutoGen [37] support multi-agent interactions, but only when agents are all running locally. Recently, Google proposed the A2A protocol [13], a decentralized identity framework where agents advertise public metadata and initiate direct communication through web-based authentication. While A2A promotes interoperability and supports verifiable identifiers, it does not include a centralized registry, policy enforcement, or runtime mediation of agent contact. Additionally, the A2A specification lacks mechanisms for specifying or enforcing per-agent communication policies. AACP [38] introduces a cryptographic foundation using verifiable credentials but adopts a fully peer-to-peer architecture without mediation layers to govern agent interactions. These approaches highlight a consistent gap in addressing how agents might dynamically discover, authenticate, and establish secure communications with one another in open, cross-organizational environments with appropriate user-controlled governance mechanisms.

Attacks on Multi-Agent Systems. Several works examine adversarial propagation in multi-agent communication [39, 40, 41, 42, 43], where rogue agents can propagate malicious outputs via interactions with other agents. Other works consider fixed communication patterns with slightly different goals, such as multi-agent debate [41] and question-answer collaboration [44]. In orchestrated multi-agent systems, Triedman et al. [45] describe attacks against orchestrator agents using adversarial content via metadata. Khan et al. [46] introduce an attack approach for pragmatic multi-agent LLM systems operating under real-world constraints like token bandwidth limits and message latency. By formulating attack paths as a maximum-flow minimum-cost problem, they demonstrate how these practical limitations can be exploited to bypass distributed safety mechanisms. SAGA provides protection against such attacks through its token-driven system, which explicitly limits the number of interactions between agents. Additionally, benign agents can easily update their contact policies to block malicious agents, preventing any widespread "outbreak." User-specified budgets in contact policies effectively contain potential threats, and problematic agents can be quickly blocked or removed from the protocol to prevent incoming connections.

VII. DISCUSSION AND CONCLUSION

SAGA establishes a protocol for secure inter-agent communication that balances security, autonomy, and governance through a `Provider`-mediated architecture enforc-

---

https://smythos.com/

https://github.com/langchain-ai/agent-protocol ing user policies. Unlike prior works that only offer conceptual designs or high-level architectures for agent governance [9, 10, 11, 12], we provide the first concrete protocol specification and a functional reference implementation, enabling immediate adoption and practical deployment of these security mechanisms.

The computational overhead introduced by our protocol is minimal across various configurations, multiple agent tasks, and LLM models (Section V-C), making SAGA both secure and practical for widespread deployment. By enabling secure cryptographic mechanisms based on fine-grained access control tokens, SAGA creates an ecosystem where AI agentic systems maintain balance between `Provider`-enabled control and decentralized operation—essential properties for responsible deployment as these systems become increasingly autonomous and widespread. Token granularity can be configured to allow a larger number of requests reusing the token, minimizing `Provider` interactions, but that extends the vulnerability window should an agent become compromised.

While SAGA provides a robust foundation for inter-agent communication, it complements related works that address LLM vulnerabilities. SAGA is an extensible framework that provides the core components for secure agent registration, communication, and agent lifecycle management. SAGA can be used in conjunction with defenses such as Model Context Protocol [47] for standardized tool-use, defenses against prompt-injection attacks [48], and privacy-preserving data minimization techniques [49]. SAGA can also be extended to support multiple providers, but protocols for provider inter-operability need to be designed.

In conclusion, SAGA creates an architecture for governing AI agentic systems that enables user control of their agents lifecycle and provides cryptographic mechanisms for secure inter-agent communication.

REFERENCES

[1] N. Mehandru, B. Y. Miao, E. R. Almaraz, M. Sushil, A. J. Butte, and A. Alaa, "Evaluating large language models as agents in the clinic," *NPJ digital medicine*, vol. 7, no. 1, p. 84, 2024.

[2] W. Wang, Z. Ma, Z. Wang, C. Wu, W. Chen, X. Li, and Y. Yuan, "A survey of llm-based agents in medicine: How far are we from baymax?" *arXiv preprint arXiv:2502.11211*, 2025.

[3] S. Wu, O. Irsoy, S. Lu, V. Dabravolski, M. Dredze, S. Gehrmann, P. Kambadur, D. Rosenberg, and G. Mann, "BloombergGPT: A large language model for finance," *arXiv preprint arXiv:2303.17564*, 2023.

[4] Y. Nie, Y. Kong, X. Dong, J. M. Mulvey, H. V. Poor, Q. Wen, and S. Zohren, "A survey of large language models for financial applications: Progress, prospects and challenges," *arXiv preprint arXiv:2406.11903*, 2024.

[5] T. Zhou, P. Wang, Y. Wu, and H. Yang, "Finrobot: Ai agent for equity research and valuation with large language models," *arXiv preprint arXiv:2411.08804*, 2024.

[6] J. Zhang, H. Bu, H. Wen, Y. Liu, H. Fei, R. Xi, L. Li, Y. Yang, H. Zhu, and D. Meng, "When llms meet cybersecurity: A systematic literature review," *Cybersecurity*, vol. 8, no. 1, pp. 1–41, 2025.

[7] M. Rigaki, C. Catania, and S. Garcia, "Hackphyr: A local fine-tuned llm agent for network security environments," *arXiv preprint arXiv:2409.11276*, 2024.

[8] M. Kobayashi, M. Fuchi, A. Zanashir, T. Yoneda, and T. Takagi, "Construction and evaluation of llm-based agents for semi-autonomous penetration testing," *arXiv preprint arXiv:2502.15506*, 2025.

[9] Y. Shavit, S. Agarwal, M. Brundage, S. Adler, C. O'Keefe, R. Campbell, T. Lee, P. Mishkin, T. Eloundou, A. Hickey *et al.*, "Practices for governing agentic AI systems," *Research Paper, OpenAI*, 2023.

[10] A. Chan, N. Kolt, P. Wills, U. Anwar, C. S. de Witt, N. Rajkumar, L. Hammond, D. Krueger, L. Heim, and M. Anderljung, "IDs for AI systems," *arXiv preprint arXiv:2406.12137*, 2024.

[11] A. Chan, K. Wei, S. Huang, N. Rajkumar, E. Perrier, S. Lazar, G. K. Hadfield, and M. Anderljung, "Infrastructure for ai agents," *arXiv preprint arXiv:2501.10114*, 2025.

[12] T. South, S. Marro, T. Hardjono, R. Mahari, C. D. Whitney, D. Greenwood, A. Chan, and A. Pentland, "Authenticated delegation and authorized AI agents," *arXiv preprint arXiv:2501.09674*, 2025.

[13] R. Surapaneni, M. Jha, M. Vakoc, and T. Segal, "Announcing the agent2agent protocol (A2A)," Google Developers Blog, April 2025, accessed: 2025-04-10. [Online]. Available: https://developers.googleblog.com/en/a2a-a-new-era-of-agent-interoperability/

[14] X. Pan, J. Dai, Y. Fan, and M. Yang, "Frontier ai systems have surpassed the self-replicating red line," *arXiv preprint arXiv:2412.12140*, 2024.

[15] P. Release, "Introducing apple intelligence for iphone, ipad, and mac," Introducing Apple Intelligence for iPhone, iPad, and Mac, 2024, [Accessed 21-04-2025].

[16] M. McHugh-Johnson, "14 new things you can do with pixel thanks to ai," https://blog.google/products/pixel/google-pixel-9-new-ai-features/, 2024, [Accessed 21-04-2025].

[17] P. Note, "Qualcomm brings the best of on-device AI to more smartphones with snapdragon 8s gen 3," Qualcomm Brings the Best of On-Device AI to More Smartphones with Snapdragon 8s Gen 3, 2024, [Accessed 21-04-2025].

[18] N. Natarajan, J. Bradley, N. Sakimura, M. B. Jones, and E. Jay, "Openid connect core 1.0 incorporating errata set 1," https://openid.net/specs/openid-connect-core-1_0.html, 2014, openID Foundation.

[19] S. Cohen, R. Bitton, and B. Nassi, "Here comes the AI worm: Unleashing zero-click worms that

target GenAI-powered applications," *arXiv preprint arXiv:2403.02817*, 2024.

[20] J. Katz and Y. Lindell, *Introduction to Modern Cryptography, Second Edition*, 2nd ed. Chapman & Hall/CRC, 2014.

[21] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ecdsa)," *International journal of information security*, vol. 1, pp. 36–63, 2001.

[22] B. Harris and L. Velvindron, "RFC 8709: Ed25519 and ed448 public key algorithms for the secure shell (SSH) protocol," USA, 2020.

[23] M. J. Dworkin *et al.*, "SHA-3 standard: Permutation-based hash and extendable-output functions," 2015.

[24] H. Krawczyk and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)," RFC 5869, May 2010, https://datatracker.ietf.org/doc/html/rfc5869.

[25] E. Barker, "Recommendation for Key Management, Part 1: General (Revision 5)," National Institute of Standards and Technology, NIST Special Publication 800-57 Part 1 Rev. 5, May 2020, provides foundational guidance on cryptographic key lifecycle management, including cryptoperiod determination and key rotation practices. [Online]. Available: https://doi.org/10.6028/NIST.SP.800-57pt1r5

[26] D. J. Bernstein, "Curve25519: New diffie-hellman speed records," in *Public Key Cryptography - PKC 2006*, M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 207–228.

[27] A. Langley, M. Hamburg, and S. Turner, "Elliptic Curves for Security," RFC 7748, Jan. 2016. [Online]. Available: https://www.rfc-editor.org/info/rfc7748

[28] S. Boeyen, S. Santesson, T. Polk, R. Housley, S. Farrell, and D. Cooper, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," RFC 5280, May 2008. [Online]. Available: https://www.rfc-editor.org/info/rfc5280

[29] A. Yang, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Li, D. Liu, F. Huang, H. Wei, H. Lin, J. Yang, J. Tu, J. Zhang, J. Yang, J. Yang, J. Zhou, J. Lin, K. Dang, K. Lu, K. Bao, K. Yang, L. Yu, M. Li, M. Xue, P. Zhang, Q. Zhu, R. Men, R. Lin, T. Li, T. Xia, X. Ren, X. Ren, Y. Fan, Y. Su, Y. Zhang, Y. Wan, Y. Liu, Z. Cui, Z. Zhang, and Z. Qiu, "Qwen2.5 technical report," *arXiv preprint arXiv:2412.15115*, 2024.

[30] CAIDA, "The CAIDA archipelago monitor statistics," https://www.caida.org/projects/ark/statistics/, accessed April 2025.

[31] M. Adorjan, "cloudping.co: Aws inter-region latency monitoring," 2025, accessed: 2025-04-18. [Online]. Available: https://github.com/mda590/cloudping.co

[32] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," in *International Conference on Learning Representations*, 2023.

[33] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, and T. Scialom, "Toolformer: Language models can teach themselves to use tools," *Advances in Neural Information Processing Systems*, 2023.

[34] K.-T. Tran, D. Dao, M.-D. Nguyen, Q.-V. Pham, B. O'Sullivan, and H. D. Nguyen, "Multi-agent collaboration mechanisms: A survey of llms," *arXiv preprint arXiv:2501.06322*, 2025.

[35] B. Yan, X. Zhang, L. Zhang, L. Zhang, Z. Zhou, D. Miao, and C. Li, "Beyond self-talk: A communication-centric survey of LLM-based multi-agent systems," *arXiv preprint arXiv:2502.14321*, 2025.

[36] A. Sempf, J. Toth, and S. v. d. Moosdijk, "Creating asynchronous AI agents with Amazon Bedrock — Amazon Web Services — aws.amazon.com," https://aws.amazon.com/blogs/machine-learning/creating-asynchronous-ai-agents-with-amazon-bedrock/, 2025, [Accessed 21-04-2025].

[37] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu *et al.*, "Autogen: Enabling next-gen llm applications via multi-agent conversations," in *Conference on Language Modeling (COLM)*, 2024.

[38] K. Royce, "AI agent-to-agent communications protocol," *kossisoroyce.com*, 2025, accessed: 2025-04-10. [Online]. Available: https://kossisoroyce.com/2025/03/28/ai-agent-to-agent-communications-protocol/

[39] X. Gu, X. Zheng, T. Pang, C. Du, Q. Liu, Y. Wang, J. Jiang, and M. Lin, "Agent smith: A single image can jailbreak one million multimodal llm agents exponentially fast," in *International Conference on Machine Learning*, 2024.

[40] D. Lee and M. Tiwari, "Prompt infection: Llm-to-llm prompt injection within multi-agent systems," *arXiv preprint arXiv:2410.07283*, 2024.

[41] A. Amayuelas, X. Yang, A. Antoniades, W. Hua, L. Pan, and W. Y. Wang, "Multiagent collaboration attack: Investigating adversarial attacks in large language model collaborations via debate," in *Findings of the Association for Computational Linguistics: EMNLP 2024*, 2024, pp. 6929–6948.

[42] P. He, Y. Lin, S. Dong, H. Xu, Y. Xing, and H. Liu, "Red-teaming llm multi-agent systems via communication attacks," *arXiv preprint arXiv:2502.14847*, 2025.

[43] W. Yu, K. Hu, T. Pang, C. Du, M. Lin, and M. Fredrikson, "Infecting llm agents via generalizable adversarial attack," in *Red Teaming GenAI: What Can We Learn from Adversaries?*, 2024.

[44] T. Ju, Y. Wang, X. Ma, P. Cheng, H. Zhao, Y. Wang, L. Liu, J. Xie, Z. Zhang, and G. Liu, "Flooding spread of manipulated knowledge in llm-based multi-agent communities," *arXiv preprint arXiv:2407.07791*, 2024.

[45] H. Triedman, R. Jha, and V. Shmatikov, "Multi-

agent systems execute arbitrary malicious code," *arXiv preprint arXiv:2503.12188*, 2025.

[46] R. M. S. Khan, Z. Tan, S. Yun, C. Flemming, and T. Chen, "Agents Under Siege: Breaking pragmatic multi-agent llm systems with optimized prompt attacks," *arXiv preprint arXiv:2504.00218*, 2025.

[47] "Model context protocol," https://modelcontextprotocol.io/, 2025, [Accessed 26-03-2025].

[48] E. Debenedetti, I. Shumailov, T. Fan, J. Hayes, N. Carlini, D. Fabian, C. Kern, C. Shi, A. Terzis, and F. Tramèr, "Defeating prompt injections by design," *arXiv preprint arXiv:2503.18813*, 2025.

[49] E. Bagdasarian, R. Yi, S. Ghalebikesabi, P. Kairouz, M. Gruteser, S. Oh, B. Balle, and D. Ramage, "Airgapagent: Protecting privacy-conscious conversational agents," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '24. Association for Computing Machinery, 2024, p. 3868–3882.

[50] A. Roucher, A. V. del Moral, T. Wolf, L. von Werra, and E. Kaunismäki, "smolagents: a smol library to build great agentic systems." https://github.com/huggingface/smolagents, 2025.

[51] S. Abdelnabi, A. Gomaa, E. Bagdasarian, P. O. Kristensson, and R. Shokri, "Firewalls to secure dynamic llm agentic networks," *arXiv preprint arXiv:2502.01822*, 2025.

APPENDIX

A. *Agent-Interaction Task Details*

We implement our LLM-agent using the smolagents library [50], specifically leveraging the CodeAgent class. This class enables agents to write and execute Python code during their reasoning process, significantly reducing the number of interactions required with the user—which, in our setup, is another agent. To support autonomous agent-to-agent communication, we modify the system prompts provided to each agent, tailoring the instructions slightly based on whether the agent is in the initiating or receiving role.

To emulate realistic tool usage, we implement database-backed services for email, calendar, and document read/write operations. For the email and calendar tools, we implement end-to-end functionality: sending an email or calendar invite causes the content to actually appear in the recipient's inbox or calendar. This stands in contrast to recent agent-agent interaction work, which typically simulates tool use via another LLM rather than through explicit, stateful updates. We initialize each user profile (and corresponding agent) with synthetic data adapted from Abdelnabi et al. [51], extending it as needed to suit the scope of our tasks.

Each task (such as one agent requesting another to find a common meeting time) spawns a fresh instance of the agent class. While the underlying LLM remains the same across tasks, no prior context or conversation history is shared between them. This ensures a clean isolation layer between tasks, without introducing any additional overhead.

Calendar. In this task, one agent contacts another to schedule a meeting on a specific date. Both agents inspect their respective calendars to determine availability, negotiate a mutually agreeable time, and then create a calendar event. Task success is defined by the following criteria:

1) the calendar invite appears in both agents' calendars for the correct duration, listing both as attendees,
2) there are no scheduling conflicts with pre-existing events, and
3) the meeting is scheduled for a future date and time.

We use GPT-4.1-mini as the LLM backbone for this task. The full interaction trace is shown in Figure 7.

Email. In this task, one agent reaches out to another for help compiling information to submit an expense report. Both agents search their respective inboxes for expenses related to a shared event. The receiving agent extracts relevant entries and sends them to the initiator, who combines the results and submits a final report using a tool-call. We evaluate success based on:

1) whether the total amount submitted matches the expected value (as all data is synthetic and fully known),
2) whether both users are listed as participants in the expense report, and
3) whether no extraneous users are included.

We use GPT-4.1 as the LLM backbone for this task. The complete interaction is provided in Figure 8.

Writing. In this task, two agents collaborate on writing a blog post about the privacy implications of AI. Each agent represents a user with expertise in either law or machine learning. They begin by reading existing blog posts associated with their respective users and then engage in a multi-step writing process to produce a unified article. Agents are encouraged to both internally reflect and externally revise across multiple interaction rounds. Once the final blog post is agreed upon, one of the agents uses a tool-call to save the document under a specified filename. Success is measured by whether the blog post is saved correctly with the expected filename. We use Qwen2.5-72B-Instruct [29] as the LLM backbone. Due to the length of messages involved (often exceeding 2000 words), we omit the full example here.

Agent `B`

Let's find some time to discuss our NDSS submission. Are you available on Tuesday for a 30-minute meeting? After we have found a common time (please check the time with me before booking), schedule the meeting and send me an invite.

Agent `A`

I am available for a 30-minute meeting on Tuesday, April 22, 2025, from 9:00 AM to 9:30 AM. Does this time work for you? Please confirm before I schedule the meeting.

Agent `B`

I am not available at 9:00 AM to 9:30 AM on Tuesday, April 22, 2025. However, I am available from 12:00 PM to 12:30 PM on the same day. Does this time work for you?

Agent `A`

I am not available at 12:00 PM to 12:30 PM on Tuesday, April 22, 2025. However, I am available from 2:00 PM to 2:30 PM on the same day. Does this time work for you? Please confirm before I schedule the meeting.

Agent `B`

I am available from 2:00 PM to 2:30 PM on Tuesday, April 22, 2025. Please confirm if this time works for you and provide your email address so I can schedule the meeting.

Agent `A`

Yes, I am available for the meeting on Tuesday, April 22, 2025, from 2:00 PM to 2:30 PM. Please go ahead and schedule the meeting and send the invite.

Agent `B`

Please provide your email address so I can add you as a participant and send the meeting invite for the NDSS submission discussion on Tuesday, April 22, 2025, from 2:00 PM to 2:30 PM.

Agent `A`

My email address is `<REDACTED>`. Please use it to add me as a participant and send the meeting invite for the NDSS submission discussion on Tuesday, April 22, 2025, from 2:00 PM to 2:30 PM.

Agent `B`

The meeting for the NDSS submission discussion has been scheduled on Tuesday, April 22, 2025, from 2:00 PM to 2:30 PM. You have been added as a participant, and the invite has been sent to `<REDACTED>`.

Agent `A`

`<TASK_FINISHED>`

Fig. 7: Example dialog between agents for the calendar-scheduling task, using GPT-4.1-mini (`gpt-4.1-mini-2025-04-14`) as the backend LLM. Contact information mentioned during interactions is redacted for visualization.

Agent B

Can you please check your emails for any expenses for our recent trip for NerurIPS to New Orleans from 03-01 to 03-03 Tell me what your expenses were (including hotel, travel, food, etc.) and your email ID. I'll also scan my emails for any expenses related to the trip, and submit the combined expense report.

Agent A

Here are my expenses for the NeurIPS trip to New Orleans (03-01 to 03-03):
- Hotel: $500 (Hilton, 2 nights: 03-01 to 03-03)
- Food: $20 (Breakfast at Dumbo Gumbo, 03-01)
- Food: $50 (Dinner at Dumbo Gumbo, 03-01)
- Conference Registration: $1000 (NeurIPS workshop)

My email ID is: `<REDACTED>`

Agent B

The expense report for the NeurIPS trip to New Orleans (03-01 to 03-03) has been submitted successfully. The report includes your provided expenses as well as additional receipts found in my inbox (flight, extra meals, and a WH Smith purchase). The total submitted amount is $2,140, and you (`<REDACTED>`) are listed as involved. If you need a detailed breakdown or a copy of the report, let me know.

Agent A

Thank you for letting me know! I appreciate the update. If I need a detailed breakdown or a copy of the report, I'll reach out.

Agent B

`<TASK_FINISHED>`

Fig. 8: Example dialog between agents for the expense-report task, using GPT-4.1 (`gpt-4.1-2025-04-14`) as the backend LLM. Contact information mentioned during interactions is redacted for visualization.