SAGA: A Security Architecture for Governing AI Agentic Systems

Georgios Syros
Northeastern University
syros.g@northeastern.edu

Anshuman Suri
Northeastern University
ans.suri@northeastern.edu

Jacob Ginesin
Northeastern University
ginesin.j@northeastern.edu

Cristina Nita-Rotaru
Northeastern University
c.nitarotaru@northeastern.edu

Alina Oprea
Northeastern University
a.oprea@northeastern.edu

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 scalable 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, providing formal security guarantees. 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]–[5], and cybersecurity [6]–[8]. These agents, built on top of 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.

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.230869
www.ndss-symposium.org

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

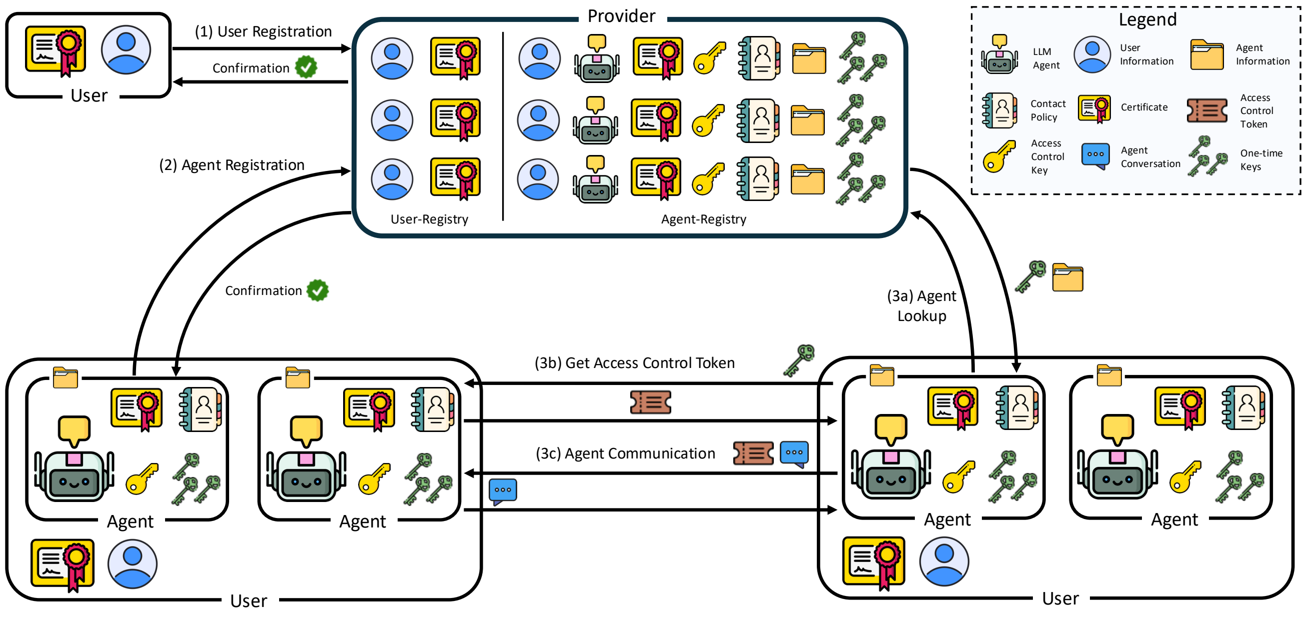


Fig. 1: System overview of SAGA. (1) A user registers with the `Provider`. (2) A registered user registers their agent and receives a confirmation from the `Provider`. (3a) An initiating agent requests a One-time Key (OTK) and the receiving agent's information from the `Provider`. (3b) The initiating 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.

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:

- 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 use PROVERIF to formalize the secrecy of the SAGA token, authentication of communication between agents and the `Provider`, and authentication of communication between any two agents. We automatically prove these properties with respect to an attacker that can observe, intercept, modify, replay, reorder, and synthesize arbitrary messages on the network.
- 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.
- We discuss and evaluate the fault-tolerance and scalability of the `Provider`. Our results demonstrate that making the `Provider` fault-tolerant using RAFT introduces negligible throughput degradation across key operations. They also demonstrate that scaling the `Provider` using sharding results in linear throughput increase.

The code of our protocol and the formal verification is available at https://github.com/gsiros/saga and the full paper, including protocol sequence diagrams and additional security analysis, at https://arxiv.org/abs/2504.21034.

II. Background and Problem Statement

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 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. We formulate the following definitions and terminology that we use in the rest of the paper.

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

Definition 2. *An `Agent` is an autonomous software entity using LLMs for decision making. 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. Each agent operates under a specific user's authority and may interact with other agents to fulfill complex objectives. Agents can be "initiating" (starting communication) or "receiving" (accepting communication) with other agents.*

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

B. Requirements and Challenges for Secure Agentic Systems

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. Crucially, [9] advocates for users' control and oversight throughout the lifecycle of LLM agents.

Challenges: An agent management system must address three key challenges: (1) *discovery*—how agents discover each other; (2) *secure communication*—how agents communicate with each other; and (3) *remote access control*—which agents are allowed to interact with a specific agent, for what tasks, and for how long. While inter-agent communication should be performed automatically by agents, access control and agent discovery should remain under the user's control for better security. Existing solutions address only one of these challenges and are not specifically designed for agents. For example, secure messaging protocols like Signal or Matrix provide peer discovery and secure communication, but lack fine-grained access control; protocols like TLS or QUIC enable secure communication, but offer no agent discovery or fine-grained access control; and systems like Kerberos provide high-granularity access control, but lack secure communication capabilities. The recent A2A protocol [13] is the only agent-specific work we are aware of that ensures communication between agents, but provides no security guarantees or access control functionality.

C. Problem Statement

Our goal is to design a system that addresses all the above challenges by providing agent discovery, secure agent communication, and fine-grained access control to the services provided by an agent, while allowing users to retain control over the management of their agents.

We first address the fundamental requirement of creating and managing agent identities while providing effective mechanisms for agent discovery within the ecosystem. We 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, our goal is to design a modular, extensible framework that can support new security properties with minimal changes to 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 (Section III-C) and the considered threat model (Section III-D).

A. System Goals

Although the requirements for secure agentic systems are still evolving, we outline several desirable goals:

Agent lifecycle managed by users. An agent's lifecycle should be established by a user. The user installs the agent on their device, registers it in the system, and retains the authority to terminate the agent at any time. Termination is performed by deactivating the agent and preventing other agents from locating it in 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.

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 a large number of 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. It is crucial to limit the vulnerability window when compromised agents are still active in the system. The length of this window can be controlled by cryptographic mechanisms and depends on factors such as the agent's communication patterns, 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 oversight over an agent's lifecycle, we delegate the agent registration process to users themselves, with the `Provider` maintaining the agent information. Users must be registered in the system and authenticate before registering any agents and retain the ability to 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 it is installed on. The `Provider` thus maintains the `User Registry` ($\mathcal{D}_U$) holding user identity records, and the `Agent Registry` ($\mathcal{D}_A$) storing 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 agents are permitted to initiate contact. To enable fine-grained access control and limit the vulnerability window, access to agents needs to be cryptographically bounded. 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.

An important design consideration is how to manage agent access control 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 [14]. 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.

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 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 `OTKs` to initiating agents.

C. System Assumptions

The correctness and security guarantees of SAGA rely on a set of clearly defined system assumptions, outlned below.

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`.

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, assuming that agents are 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. Secret keys are assumed to remain confidential and outside adversarial control.

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 performing message tampering, eavesdropping, and replay.

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

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:

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 [15]. 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.

C6: An adversary may overhear, intercept, and synthesize any message, and is only limited by the computational hardness constraints of the employed cryptographic primitives.

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 $(\texttt{PK}, \texttt{SK})$,

$\text{Sign}_{\texttt{SK}}(m)$ – a signing algorithm that outputs a signature σ on message m using $\texttt{SK}$, and $\text{Verify}_{\texttt{PK}}(m, \sigma)$ – an algorithm that verifies if the signature σ on message m is correct. We assume that the signature scheme is Existential Unforgeable under Chosen Message Attack [16], such as ECDSA [17] and Ed25519 [18]. A certificate generation function $\text{GenCert}_X(m)$ involves entity X creating a certificate for content m as: $\text{GenCert}_X(m) = \langle m, \text{Sign}_{\texttt{SK}_\texttt{X}}(m) \rangle$

Hash function. We use a collision-resistant hash function $\texttt{H}(\cdot)$, such as SHA-256 or SHA-3 [19].

Diffie-Hellman Key Exchange. The Diffie-Hellman Key Exchange protocol [16] 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. We denote by DH the function that takes as input the secret key of one party and public key of the other party and computes the shared secret key $DH(x, g^y) = DH(y, g^x) = g^{xy} \mod p$. 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) [20] with SHA-256 as the underlying hash function.

Notation. We introduce formal notation throughout the protocol specification. For convenience, Table I provides a summary of all symbols used.

Symbol	Description
$\mathtt{PK}, \mathtt{SK}$	Public/private signing keys
$\mathtt{Cert}$	Certificate issued by CA
$\mathtt{uid}_{\mathtt{U}}$	User identifier
$\mathtt{aid}_{\mathtt{A}}$	Agent $\mathtt{A}$ identifier
$\mathtt{ED}_{\mathtt{A}}$	Agent $\mathtt{A}$ endpoint descriptor
$(\mathtt{PK}_{\mathtt{A}}, \mathtt{SK}_{\mathtt{A}})$	Agent $\mathtt{A}$ public/private TLS credentials
$(\mathtt{PAC}_{\mathtt{A}}, \mathtt{SAC}_{\mathtt{A}})$	Agent $\mathtt{A}$ public/private access control keys
$(\mathtt{OTK}^{\mathtt{i}}_{\mathtt{A}}, \mathtt{SOTK}^{\mathtt{i}}_{\mathtt{A}})$	Agent $\mathtt{A}$ one-time public/private keys
$\mathtt{CP}_{\mathtt{A}}$	Contact policy of agent $\mathtt{A}$
$\mathtt{token}$	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`. 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:

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

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`.

1) **Generating agent information**. The user selects an identifier $\mathtt{name}_{\mathtt{A}}$ for the agent, forming a unique agent ID in combination with their username: $\mathtt{aid}_{\mathtt{A}} = \mathtt{uid}_{\mathtt{U}}\mathtt{:name}_{\mathtt{A}}$. The user specifies the agent's device name $\mathtt{device}_{\mathtt{A}}$ and networking details $\mathtt{IP}_{\mathtt{A}}$ and $\mathtt{port}_{\mathtt{A}}$. These comprise the agent's "endpoint descriptor":
$$\mathtt{ED}_{\mathtt{A}} = \langle \mathtt{device}_{\mathtt{A}}, \mathtt{IP}_{\mathtt{A}}, \mathtt{port}_{\mathtt{A}}\rangle$$
2) **Generating cryptographic keys**. The user generates the following keys for the agent:
   - TLS credentials $(\mathtt{PK}_{\mathtt{A}}, \mathtt{SK}_{\mathtt{A}})$ to establish secure communication channels with other agents and a signed certificate by the `CA`:
$$\mathtt{Cert}_{\mathtt{A}} = \text{GenCert}_{\mathtt{SK}_{\mathtt{CA}}}(\langle \mathtt{aid}_{\mathtt{A}}, \mathtt{PK}_{\mathtt{A}}\rangle)$$
   - A public-private key pair: $(\mathtt{PAC}_{\mathtt{A}}, \mathtt{SAC}_{\mathtt{A}})$ 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 $(\mathtt{OTK}^{1}_{\mathtt{A}}, \mathtt{SOTK}^{1}_{\mathtt{A}}), \ldots, (\mathtt{OTK}^{\mathtt{N}}_{\mathtt{A}}, \mathtt{SOTK}^{\mathtt{N}}_{\mathtt{A}})$, used for deriving tokens for controlling access to the agent. Each public one-time key is signed by the user:
$$\sigma^{\mathtt{U}}_{\mathtt{OTK}^{\mathtt{i}}} = \text{Sign}_{\mathtt{SK}_{\mathtt{U}}}\left(\langle \mathtt{aid}_{\mathtt{A}}, \mathtt{OTK}^{\mathtt{i}}_{\mathtt{A}}\rangle\right)$$

   The user also signs the agent's device and networking information, along with its long-term keys:
$$\sigma^{\mathtt{U}}_{\mathtt{A}} = \text{Sign}_{\mathtt{SK}_{\mathtt{U}}}\left(\langle \mathtt{aid}_{\mathtt{A}}, \mathtt{ED}_{\mathtt{A}}, \mathtt{PK}_{\mathtt{A}}, \mathtt{PAC}_{\mathtt{A}}, \mathtt{PK}_{\mathtt{Prov}}\rangle\right)$$
   The inclusion of $\mathtt{PK}_{\mathtt{Prov}}$ 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 $\mathtt{CP}_{\mathtt{A}}$ (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 $\langle \mathtt{uid}_{\mathtt{U}}, \mathtt{passwd}\rangle$. The `Provider` verifies the credentials and proceeds if successful.

5) **Registration submission**. The user submits to the $\texttt{Provider}$: the agent's information $(\texttt{aid}_\texttt{A}, \texttt{ED}_\texttt{A}, \texttt{CP}_\texttt{A})$, the TLS certificate $\texttt{Cert}_\texttt{A}$, the public access control and one-time keys $\texttt{PAC}_\texttt{A}, \{\texttt{OTK}^1_\texttt{A}, \ldots, \texttt{OTK}^\texttt{N}_\texttt{A}\}$, and signatures $\sigma^\texttt{U}_\texttt{A}$, $\sigma^\texttt{U}_{\texttt{OTK}^\texttt{i}}$ for $i \in [1, N]$. The agent stores locally all the private keys corresponding to the public keys submitted to the provider: $(\texttt{SK}_\texttt{A}, \texttt{SAC}_\texttt{A}, \{\texttt{SOTK}^1_\texttt{A}, \ldots, \texttt{SOTK}^\texttt{N}_\texttt{A}\})$.
6) **Provider verification**. The $\texttt{Provider}$ processes the registration request by checking that $\texttt{aid}_\texttt{A}$ and $\texttt{ED}_\texttt{A}$ are globally unique, and verifying $\texttt{Cert}_\texttt{A}$ and signatures:

$$\text{Verify}_{\texttt{PK}_\texttt{U}}(\langle \texttt{aid}_\texttt{A}, \texttt{ED}_\texttt{A}, \texttt{PK}_\texttt{A}, \texttt{PAC}_\texttt{A}, \texttt{PK}_\texttt{Prov}\rangle, \sigma^\texttt{U}_\texttt{A})$$

$$\text{Verify}_{\texttt{PK}_\texttt{U}}(\langle \texttt{aid}_\texttt{A}, \texttt{OTK}^\texttt{i}_\texttt{A}\rangle, \sigma^\texttt{U}_{\texttt{OTK}^\texttt{i}})$$

7) **Completion.** Upon successful verification, the $\texttt{Provider}$ stores in the agent registry the agent's metadata $\texttt{M}_\texttt{A}$, the contact policy $\texttt{CP}_\texttt{A}$, the agent's signatures $\sigma^\texttt{U}_\texttt{A}$ and $\sigma^\texttt{U}_{\texttt{OTK}^\texttt{i}}$, along with $\texttt{uid}_\texttt{U}$ to associate the agent $\texttt{A}$ with user $\texttt{U}$:

$$\texttt{M}_\texttt{A} = \{\texttt{ED}_\texttt{A}, \texttt{Cert}_\texttt{A}, \texttt{PAC}_\texttt{A}, \texttt{OTK}^\texttt{i}_\texttt{A}, i \in [1, N]\}$$
$$\mathcal{D}_A[\texttt{aid}_\texttt{A}] = \langle \texttt{uid}_\texttt{U}, \texttt{M}_\texttt{A}, \texttt{CP}_\texttt{A}, \sigma^\texttt{U}_\texttt{A}, \sigma^\texttt{U}_{\texttt{OTK}^\texttt{i}}\rangle$$

The $\texttt{Provider}$ then signs the agent $\texttt{A}$'s information

$$\sigma^{Prov}_\texttt{A} = \text{Sign}_{\texttt{SK}_\texttt{Prov}}\left(\langle \texttt{aid}_\texttt{A}, \texttt{Cert}_\texttt{A}, \texttt{ED}_\texttt{A}, \texttt{PAC}_\texttt{A}, \sigma^\texttt{U}_A\rangle\right)$$

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 $\texttt{A}$ is now officially registered and can securely communicate within the ecosystem.

D. Agent Management

Agent management in SAGA involves managing access control polices, policy updates, 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 OTKs 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 OTKs.

If multiple rules match, the one with the *highest specificity* is selected. For instance, in Listing 1, agent `alice@company.com:calendar_agent` attempts to contact another 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 OTKs.

```
// Agent C's Contact Policy
[
  {
    "agents": "alice@company.com:calendar_agent",
    "budget": 15
  },
  {
    "agents": "*@company.com:calendar_agent",
    "budget": 10
  },
  {
    "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 $\texttt{A}$ with contact policy $\texttt{CP}_\texttt{A}$ and an initiating agent $\texttt{B}$ with identifier $\texttt{aid}_\texttt{B}$, the number of OTKs issued is defined as:

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

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

2) One-Time Key Generation: To control communication to registered agents, the Provider facilitates the distribution of OTKs: one-time keys generated and uploaded by each user for their agents. Each OTK is consumed when an initiating agent contacts the Provider to obtain information for a receiving agent. OTKs are defined for receiving agents, as the protocol is asymmetric. Additionally, the Provider maintains the number of OTKs remaining for a particular agent communicating with another agent. While it might be possible for the initiating agent to receive multiple OTKs from the Provider, thereby reducing the number of times the agent must contact the Provider, we chose a more conservative design. In our approach, the agent receives only one OTK per request to limit the vulnerability window in case the 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 $(\texttt{CP}_\texttt{A})$, as detailed above. If this is the first time B is contacting A, the Provider creates a counter $\text{Counter}_\texttt{OTK}[\texttt{aid}_\texttt{A}][\texttt{aid}_\texttt{B}]$ to keep track of the number of remaining OTKs, and initializes it with $\text{Budget}_\texttt{OTK}(\texttt{aid}_\texttt{A}, \texttt{aid}_\texttt{B})$.

If the policy check succeeds and a valid OTK is available (indicated by a positive value for $\text{Counter}_\texttt{OTK}[\texttt{aid}_\texttt{A}][\texttt{aid}_\texttt{B}]$), 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 $\mathtt{OTK}$ at this stage may fail due to: (a) exhaustion of $\mathtt{aid}_\mathtt{A}$'s overall $\mathtt{OTK}$ pool, or (b) depletion of $\mathtt{aid}_\mathtt{B}$'s $\mathtt{OTK}$ quota as defined by $\mathtt{CP}_\mathtt{A}$. The user of agent $\mathtt{A}$ can update the contact policy and refresh the $\mathtt{OTK}$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 design goal of giving the user full control over their agent's 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 [21]. 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 [22].

E. Inter-Agent Communication

We describe 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`. 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. This protocol runs either the first time `B` contacts `A`, or when 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 $(\mathtt{aid}_\mathtt{A}, \mathrm{ED}_\mathtt{A})$, 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:
$$\text{Verify}_{\mathtt{PK}_{\mathtt{U1}}}(\langle \mathtt{aid}_\mathtt{A}, \mathrm{ED}_\mathtt{A}, \mathtt{PK}_\mathtt{A}, \mathtt{PAC}_\mathtt{A}, \mathtt{PK}_{\mathtt{Prov}}\rangle, \sigma^{\mathtt{U}}_{\mathtt{A}})$$
$$\text{Verify}_{\mathtt{PK}_{\mathtt{U1}}}(\langle \mathtt{aid}_\mathtt{A}, \mathtt{OTK}^{\mathtt{i}}_{\mathtt{A}}, \rangle, \sigma^{\mathtt{U}}_{\mathtt{OTK}^{\mathtt{i}}})$$
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 $\sigma^{Prov}_{\mathtt{B}}$:
$$\text{Verify}_{\mathtt{PK}_{\mathtt{Prov}}}(\langle \mathtt{aid}_\mathtt{B}, \mathtt{Cert}_\mathtt{B}, \mathrm{ED}_\mathtt{B}, \mathtt{PAC}_\mathtt{B}, \sigma^{\mathtt{U2}}_{\mathtt{B}}\rangle, \sigma^{Prov}_{\mathtt{B}})$$
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 $\texttt{Q}_{\texttt{max}}$ reduces the overhead of contacting the `Provider` to obtain `OTK`s, but increases the potential exposure in case an agent is compromised.

F. Formal Protocol Analysis

We formalized SAGA using the state-of-the-art symbolic cryptographic analysis tool, ProVerif, to reason about cryptographic attackers [23]. Our ProVerif model precisely captures the SAGA protocol description, using standard ProVerif protocol modeling techniques and assumptions. Within our ProVerif model of SAGA, we encoded formal properties specifying the secrecy of the SAGA token, authentication of communication between agents and the provider, and authentication of communication between any two agents. Using ProVerif's automated reasoning capabilities, we automatically proved each of the afformentioned properties with respect to an attacker that can observe, intercept, modify, replay, reorder, and synthesize arbitrary messages on the network. This directly corresponds to the Dolev-Yao symbolic cryptographic protocol model [23]. For details on formal verification, see Appendix C. We also empirically validate the protocol by implementing eight distinct attacker behaviors, i.e., malicious agents that deviate from the protocol within the capabilities defined in the threat model (Section III-D). Our evaluation demonstrates that SAGA reliably enforces its security guarantees under all tested scenarios, confirming the correctness of the implementation and reinforcing the conclusions of our formal analysis. A detailed description of the empirical evaluation is provided in Appendix E of the full version of the paper[1].

V. Extensions to the SAGA architecture

We discuss several extensions to SAGA to enable fault tolerance, scalability and resilience to server compromise in Section V-A, and describe the integration with the A2A protocol in Section V-B.

A. Provider Architecture Design

In our proposed architecture, the `Provider` is trusted, a model similar to that in existing systems as *Active Directory* [24] (based on Kerberos). We discuss below how the design can be augmented to enable fault-tolerance, resilience to attacks and server compromises, and scalability.

Fault-tolerance. The centralized `Provider` design we propose can be made fault-tolerant by using standard distributed systems techniques. Specifically, the `Provider` functionality can be implemented as a RAFT [25] or Paxos [26] cluster, where the agent registry is replicated across the RAFT nodes. Operations such as agent registration, policy update, and `OTK` retrieval are submitted to the leader of the RAFT cluster. Such systems are provisioned to tolerate a certain number of failures, for example typical deployments consist of 5 servers tolerating 2 faulty servers. We present fault tolerance results in Section VI-D.

[1]https://arxiv.org/abs/2504.21034

Protection against denial-of-service (DoS). Additionally, protection against DoS can be achieved by using rate limiters that control the volume and frequency of incoming requests, preventing attackers from overwhelming server resources with excessive traffic. Advanced rate limiters can detect abnormal traffic patterns, adapt their restrictions based on current server load, and implement tiered responses like introducing delays rather than complete blocking. While particularly effective against single-source attacks, rate limiters serve as a crucial first line of defense that maintains system availability during attacks, though they work best when combined with other protective measures like firewalls and DDoS protection services for comprehensive security against sophisticated distributed attacks.

Resilience to compromised servers. A compromised `Provider` might refuse agent registration, refuse to forward agent metadata, register malicious agents, or not follow agent policies. The `Provider` can be made resilient to these types of attacks by implementing it as a Byzantine-resilient service using existing protocols such as PBFT [27]. In this design, a quorum of servers is required to participate in each operation ensuring that a majority of honest servers make the decision. Such a system can tolerate f compromised servers out of $3f + 1$ participating servers. An alternative, more lightweight design is using fault-tolerant algorithms (RAFT or Paxos), augmented with proactive auditing mechanisms to detect server compromises.

Scalability. Finally, the scalability of the system can be elastically increased by using a standard technique in databases [28]–[30] called *sharding* [31], [32]. In this approach the agent registry is partitioned across several entities called sharders, and we can partition based on the space of agent ID. Each sharder can be made fault-tolerant with a RAFT cluster, and a set of load-balancers forwarding the requests for different agent entries to the corresponding sharder. We present scalability results in Section VI-D.

Federation. The `Provider` service can be further decentralized by using federation, where several organizations each managing their own provider, participate in the protocol. Similar to cross-realm Kerberos [33], multiple providers establish trust relationships via shared cryptographic keys or certificates, enabling cross-domain agent authentication where agents from one provider can securely communicate with agents from other providers. This model allows organizations to maintain control over their agent registries and policies while enabling inter-organizational communication, with each provider validating its local agents during cross-boundary interactions.

B. Integration with the A2A Protocol

We discuss the integration of SAGA with Google's Agent2Agent (A2A) protocol [13]. A2A defines an agentic framework where AI agents advertise public metadata and establish secure communication with other agents. While A2A defines a unified interface for structured task exchange via "agent cards", it lacks support for authentication, access control, and agent governance. Our integration bridges this gap by protecting agent cards with SAGA access control policies and by encapsulating A2A messages within SAGA's secure communication layer. We highlight below the changes that enable integration with SAGA.

The `Provider` stores agent cards in the agent registry as part of each agent's metadata entry as per Step 7 of Agent Registration in Section IV-C.

$$\mathcal{D}_A[\mathtt{aid_A}] = \langle \mathtt{uid_U}, \mathtt{M_A}, \mathtt{CP_A}, \mathbf{\mathtt{A2ACard}}_\mathtt{A}, \sigma^{\mathtt{U}}_{\mathtt{A}}, \sigma^{\mathtt{U}}_{\mathtt{OTK^i}} \rangle$$

Unlike A2A's recommended deployment via public URLs, SAGA protects these cards under user-specified access control policies. This ensures that only authorized agents may retrieve the agent cards. Additionally, the integrity of the agent cards are protected with user signatures (Step 2, Section IV-C).

$$\sigma^{\mathtt{U}}_{\mathtt{A}} = \mathrm{Sign}_{\mathtt{SK_U}}\left(\langle \mathtt{aid_A}, \mathtt{ED_A}, \mathtt{PK_A}, \mathtt{PAC_A}, \mathtt{PK_{Prov}}, \mathbf{\mathtt{A2ACard}}_\mathtt{A} \rangle\right)$$

For inter-agent communication, the standard message in the *msg* field is wrapped into an `A2A Request` during Step 8 of Section IV-E.

$$\langle \mathtt{token}, \mathtt{msg} \rangle \rightarrow \langle \mathtt{token}, \mathbf{\mathtt{A2ARequest}}\,(\mathtt{msg}) \rangle$$

In addition to the message content, `A2A Requests` include internal metadata such as task and message IDs, and the content type (e.g., text, image). Before forwarding the request to the agent's A2A stack, the SAGA stack verifies the token for authenticity, freshness, and contact authorization as per Step 8 of Section IV-E. If any check fails, the message is discarded and never reaches the A2A layer, effectively preventing unauthorized task execution.

SAGA integrates with A2A through minimal changes that preserve the protocol's functionality, while augmenting its security. The integration remains agnostic to agent design and task semantics, enabling secure, interoperable agent communication controlled by user policies.

VI. Evaluation

To evaluate SAGA, we implement the full protocol (VI-A), measure its overhead in VI-B, and evaluate it on three agentic tasks in VI-C. We measure the cost of fault tolerance and protocol scalability in VI-D.

A. Implementation

The `Provider` is implemented as an HTTPS service. 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. We also implemented the integration of SAGA with A2A, described in V-B.

All cryptographic operations in the protocol are built on Curve25519 [34]. Both long-term and ephemeral keys are generated using the `X25519` elliptic-curve Diffie-Hellman (ECDH) [35] scheme, using 256-bit shared secrets. Certificates adhere to the X.509 PKI standard [36] 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 [19].

For LLM agents, we experimented with a local Qwen-2.5 [37] 72B model running on NVIDIA H100, as well as two OpenAI models hosted in the cloud and accessed via API.

B. Overhead Evaluation

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 shown in Table III (Appendix A) most operations are lightweight, on the order of several ms.

Key Management Overhead. SAGA relies on three classes of cryptographic keys, each with distinct lifetimes and rotation patterns: short-term one-time keys (`OTK`), medium-term access control keys (`PAC, SAC`), and long-term identity keys (`PK, SK`). `OTK`s are ephemeral and rotated frequently once consumed for the derivation of access control tokens. On the other hand, access control keys are medium-term, typically rotated on a weekly or biweekly basis to balance security with operational stability. Long-term identity keys are rotated infrequently, commonly every 30 to 90 days, following established key management guidelines [21].

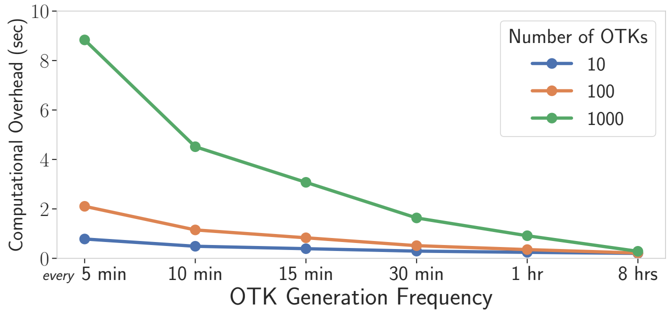


Fig. 2: Computational overhead of `OTK` generation for the user, as a function of frequency and key-chain length. Even with frequent generation (every 5 minutes) and long key-chains (1000 `OTK`s), the total cost remains low.

Figure 2 measures the overhead for OTK generation at the user, showing that even under frequent generation (1000 `OTK`s every 5 minutes over an 8-hour period) the total computational cost remains under 10 seconds for a single user. The computational cost at the `Provider` (validation and storage of the `OTK`s) does not exceed 0.5 seconds under the same conditions (1000 `OTK`s/5min). In contrast, both access control and identity key pairs require only 0.11 milliseconds to generate on commodity hardware, and due to their infrequent rotation, their cost is effectively amortized.

Figure 3 reports the total cost of deriving access control tokens with different configurable lifetimes (L) from `OTK`s for a single initiating agent with 1, 10 and 100 receiving agents. The process includes a Diffie-Hellman handshake, the encryption and decryption (validation) of the token as described in Section IV-E. Even for very short lifetimes (1 minute), the total cost of derivation of 144K tokens is just below 400 seconds over the span of 1 day for a single agent.

We conclude that a longer lifetime reduces reliance on the Provider but increases the window for compromised agents to operate without interruption. Shorter lifetimes mitigate security risks by requiring more frequent cryptographic validation, at the expense of additional overhead.

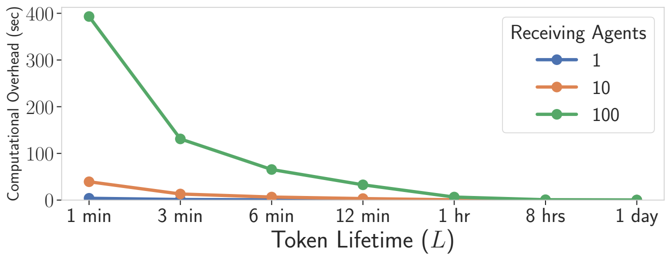


Fig. 3: Computational overhead of access control token derivation between one initiating agent and 1, 10 and 100 receiving agents, varying by token lifetime (L). Even for short lifetimes (1 minute), the total cost remains low.

Importantly, all aforementioned overheads are very small compared to the total task execution times (Section VI-C), which typically last at least a few minutes, depending on task complexity and LLM backend latency.

Protocol Overhead. We measure the overhead introduced by SAGA's access control and provider coordination mechanisms (Section VI-B). 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:

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

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.

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 [38] and AWS [39], and use these to approximate protocol overhead. Figure 4 shows the amortized protocol setup overhead:

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

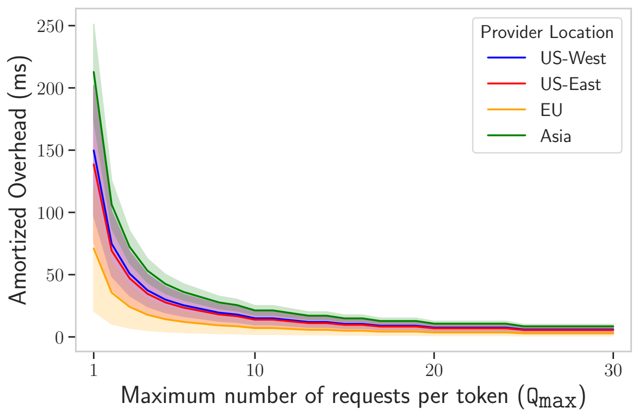


Fig. 4: Amortized protocol overhead per request $\bar{c}_{\text{proto}}(m)$ as a function of maximum number of requests token is reused. We measure the overhead for several geographic locations for the Provider. The shaded region reflects variability for agents position worldwide.

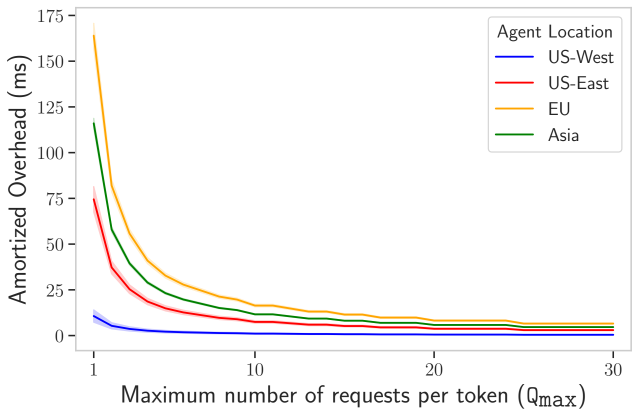


Fig. 5: Amortized protocol overhead per request $\bar{c}_{\text{proto}}(m)$ as a function of maximum number of requests token is reused. We measure the overhead for several geographic locations for the initiating agent, where the Provider is fixed in US-West. The shaded region captures variability under sampled network conditions.

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 shown, 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 is very low—under 25ms when agents interact for at least 4–5 requests (Figure 5). This effect is even more pronounced when the agent and Provider are geographically close.

C. 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 B for more details.

Task	**LLM Backend**	**Standard Cost**		**SAGA**
		LLM	Networking	**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 II: Task execution time (in seconds). `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.

We measure the standard task completion cost i.e., the time taken by the LLM to generate responses, and the network latency (Table II, 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 [40], and tool calls [41], 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.

D. Fault Tolerance and Scalability

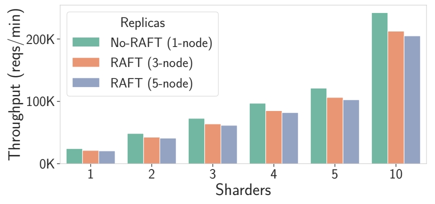
We evaluate SAGA's scalability and fault tolerance under varying deployment configurations. Specifically, we measure the throughput of the `Provider` for core operations; agent registration, `OTK` request (i.e., issuance of one-time keys to agents), and `OTK` refresh (i.e., generation of new OTKs by the user). We vary system parameters such as replication factor (RAFT nodes), number of sharding workers N_S, `OTK` keychain length, and access control `token` lifetime. Our findings show that SAGA maintains high throughput under replication, scales linearly with added compute, and supports large agent populations through configurable token lifetimes.

Setup. We deploy a `Provider` backed by a replicated RethinkDB cluster. RethinkDB [42] is a distributed, open-source database using the RAFT consensus algorithm for strong consistency and fault tolerance. For replication cost evaluation, we consider typical cluster configurations with 3 and 5 RAFT nodes (supporting 1 and 2 crash faults), and as baseline a configuration with only 1 node (no fault tolerance). For scalability evaluation, we vary the number of *sharders* and observe throughput assuming that each sharder (consisting of a RAFT cluster) runs on a separate machine and requests are routed by a proxy to the right sharder. To analyze the system's sensitivity to storage-related operations, we vary the number of submitted `OTK`s between 10, 100, and 1000 keys. Experiments are conducted on a workstation with a 16-core AMD Threadripper PRO 5955WX CPU and Samsung's MZ1L21T9HCLS-00A07 SSD.

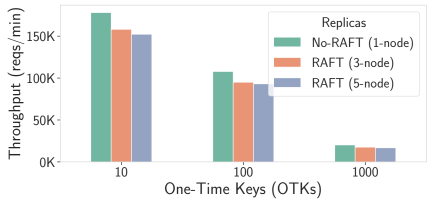
Fault Tolerance. Figures 6a, 6b, and 7 demonstrate that making the `Provider` fault-tolerant (3, 5 RAFT nodes) introduces negligible throughput degradation across key operations. For example, for `OTK` Requests (Figure 6a), the No-RAFT (1-node) setup achieves 242K requests per minute, compared to 212K for 3-node and 204K for 5-node RAFT configurations, which translates to a throughput decrease of 12-15%. In Figure 6b, the cost of replication for `OTK` refresh is lower, dropping from 173K (No-RAFT) to 153K (3-node RAFT), a difference of just 20K requests per minute ($\sim$11%). For more I/O-intensive operations such as refreshing large keychains (1000 `OTK`s), the tradeoff shrinks further to only 2K requests per minute. Agent registration performance remains largely stable even under 5-node replication, regardless of the number of `OTK`s provisioned per agent achieving 511K requests per minute. Similarly, `OTK` request and refresh incur minimal overhead.

Scalability. As expected, throughput scales linearly with the number of sharders. As shown in Figures 6a, 7 and 8 adding just 10 sharders increases `OTK` throughput nearly tenfold, indicating no early saturation and achieving 178K to 511K requests per minute depending on the operation. We omit the latency of forwarding the request to the right sharder, as within the same datacenter this would be very small. Experiments on AWS including the cloud latency show a similar trend (see Appendix E). We expect that in production, `Provider` can scale to hundreds of sharders.

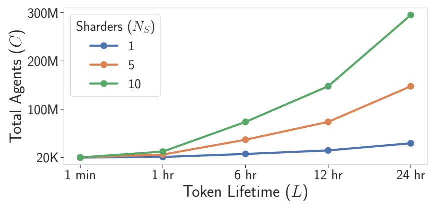
Agent Capacity. We seek to answer the question: *How many active agents can the system support at any given time?* We define this as the number of agents a `Provider` can continuously serve with `OTK`s. Let $L = \mathtt{T}_{\mathtt{expire}} - \mathtt{T}_{\mathtt{issue}}$ denote the `token` lifetime. Each agent needs to contact the `Provider` only once per L to obtain an `OTK`, which it then uses to generate a token and communicate with another agent throughout the interval. Hence, the provider's total supported population is: $C = \mathcal{T}(N_S) \cdot L$ where $\mathcal{T}(N_S)$ is the `Provider` 's `OTK` issuance throughput with N_S sharder nodes. As shown in Figure 6c, with 10 sharders and 24-hour tokens, the system can support up to 300 million agents. These results demonstrate SAGA's ability to operate under realistic, large-scale deployment scenarios.



(a) `OTK` Request Throughput



(b) `OTK` Refresh Throughput



(c) Total Capacity C of the system.

Fig. 6: SAGA's Throughput and Capacity for different protocol operations.

Sensitivity to the `OTK` Key-Chain Length. We analyze the effect of increasing the `OTK` key-chain length (number of `OTK`s submitted to the `Provider`) on the throughput for agent registration and `OTK` refresh. We omit `OTK` request, as each request consumes exactly one `OTK` by design. As expected, Figures 7 and 6b show a gradual throughput decrease as the number of submitted `OTK`s increases. This is due to the increased payload size and the cost of writing more state to the replicated agent registry.

While longer `OTK` key-chains incur higher cost, SAGA maintains high throughput across practical configurations. For example, agent registration with 1000 `OTK`s still exceeds 40K requests per minute with 10 sharders (Figure 7c), and `OTK` refresh remains comparably efficient across RAFT configurations (Figure 6b). Despite processing the same number of `OTK`s, registration outperforms `OTK` refresh due to differences in storage operations: registration involves appending to the registry whereas `OTK` refresh updates existing entries. These results confirm that users can pre-provision large key batches with small performance penalties, enhancing resilience under intermittent connectivity.

VII. 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. 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. Shavit et al. [9] discuss governance practices for agentic systems, highlighting unique agent identification and interruptibility as critical features. None of these designs are actually implemented and evaluated in a real system.

LLM Agent Development Frameworks. LangChain [43], AutoGen [44], and smolagents [45] are development libraries that help build LLM agents, but do not address governance—how users control their agents or how agents securely discover and communicate with each other. We used smolagents in our experiments, but SAGA is compatible with any other agent development framework. SAGA provides user-level control of agent interaction with other agents and secure agent communication.

Inter-Agent Protocol Implementations. Recent surveys [46] reveal that implementations typically assume pre-established connections between agents with static communication patterns. Smyth OS[2] offers a portal for agent creation and integration limited within an particular organization. Current software implementations for agent interactions generally fall short of enabling secure cross-organizational communication. While frameworks like LangChain's Agent Protocol[3] provide specifications for interactions, they do not address critical aspects like authentication or access control mechanisms. Commercial platforms like Amazon Bedrock [47] enable agent orchestration but assume all agents belong to the same restricted environment. Packages like AutoGen [44] support multi-agent interactions, but only when agents are all running locally. AACP [48] introduces a cryptographic foundation using verifiable credentials but adopts a fully peer-to-peer architecture without mediation layers to govern agent interactions.

Key Pre-Distribution and Trust Models. Users of Signal and Matrix provide a server with signed one-time keys and ephemeral keys in-advance to allow offline shared key establishment; users either choose to trust the central server, or verify identity keys off-band in the case server compromise is a concern. [49], [50]. SAGA employs a similar technique, allowing users to register their agents and provide one-time keys with a `Provider` to enable fine access control to agents, in the spirit of Kerberos [33]. Similar to Signal, users who do not trust the identity service (third party in SAGA) must verify identity keys off-band. Additionally, the `Provider` can be made resilient to compromise, as discussed in Section V-A.

Attacks on Multi-Agent Systems. Several works examine adversarial propagation in multi-agent communication [51]–[55], where rogue agents can propagate malicious outputs via interactions with other agents. Other works consider fixed

[2]https://smythos.com/
[3]https://github.com/langchain-ai/agent-protocol

communication patterns with slightly different goals, such as multi-agent debate [53] and question-answer collaboration [56]. In orchestrated multi-agent systems, Triedman et al. [57] describe attacks against orchestrator agents using adversarial content via metadata. Khan et al. [58] introduce an attack approach for pragmatic multi-agent LLM systems operating under real-world constraints like token bandwidth limits and message latency. 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."

VIII. Conclusion

SAGA establishes a scalable framework for secure inter-agent communication that balances security, autonomy, and governance through a `Provider`-mediated architecture enforcing user policies. Unlike prior works that only offer conceptual designs or high-level architectures for agent governance [9]–[12], SAGA provides the first concrete protocol specification with strong formal security guarantees and a reference implementation. SAGA is compatible with existing agent protocols, such as A2A and Model Context Protocol [59] for standardized tool-use and it can be integrated with defenses against prompt-injection attacks [60], and privacy-preserving data minimization techniques [61].

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] 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.

[15] 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.

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

[17] 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.

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

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

[20] 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.

[21] 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

[22] P. A. Grassi, M. E. Garcia, and J. L. Fenton, "Digital identity guidelines," *NIST special publication*, vol. 800, pp. 63–3, 2017.

[23] B. Blanchet, B. Smyth, V. Cheval, and M. Sylvestre, "Proverif 2.00: automatic cryptographic protocol verifier, user manual and tutorial," *Version from*, vol. 16, pp. 05–16, 2018.

[24] B. Desmond, J. Richards, R. Allen, and A. G. Lowe-Norris, *Active Directory: Designing, Deploying, and Running Active Directory.* " O'Reilly Media, Inc.", 2008.

[25] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 305–319. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[26] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, p. 133–169, May 1998. [Online]. Available: https://doi.org/10.1145/279227.279229

[27] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *3rd Symposium on Operating Systems Design and Implementation (OSDI 99)*. New Orleans, LA: USENIX Association, Feb. 1999. [Online]. Available: https://www.usenix.org/conference/osdi-99/practical-byzantine-fault-tolerance

[28] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, Jun. 2008. [Online]. Available: https://doi.org/10.1145/1365815.1365816

[29] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's Globally-Distributed database," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX Association, Oct. 2012, pp. 261–264. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett

[30] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *Proceedings of the Conference on Innovative Data system Research (CIDR)*, 2011, pp. 223–234. [Online]. Available: http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf

[31] A. Silberschatz, H. F. Korth, and S. Sudarshan, "Database system concepts," 2011.

[32] S. Solat, "Sharding distributed databases: A critical review," *arXiv preprint arXiv:2404.04384*, 2024.

[33] C. Neuman, S. Hartman, K. Raeburn, and T. Yu, "The kerberos network authentication service (v5)," RFC 4120, 2005. [Online]. Available: https://www.rfc-editor.org/info/rfc4120

[34] 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.

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

[36] 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

[37] 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.

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

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

[40] 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.

[41] 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.

[42] L. Walsh, V. Akhmechet, and M. Glukhovsky, "Rethinkdb-rethinking database storage," *Hexagram 49, Inc*, p. 85, 2009.

[43] "Agent Protocol — langchain-ai.github.io," https://langchain-ai.github.io/agent-protocol/api.html, 2024, [Accessed 26-03-2025].

[44] 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.

[45] 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.

[46] 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.

[47] 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].

[48] 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/

[49] M. R. Albrecht, B. Dowling, and D. Jones, "Device-oriented group messaging: a formal cryptographic analysis of matrix'core," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 2666–1685.

[50] E. Kret and R. Schmidt, "The pqxdh key agreement protocol," 2024. [Online]. Available: https://signal.org/docs/specifications/pqxdh

[51] 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.

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

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] H. Triedman, R. Jha, and V. Shmatikov, "Multi-agent systems execute arbitrary malicious code," *arXiv preprint arXiv:2503.12188*, 2025.

[58] 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.

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

[60] 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.

[61] 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.

[62] 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. *More Cryptographic Overhead Results*

We provide details about the cryptographic cost of key operations in the SAGA protocol in Table III.

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.

B. *Agent-Interaction Task Details*

We implement our LLM-agent using the smolagents library [45], 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. [62], 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.

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.

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 [37] as the LLM backbone. Due to the length of messages involved (often exceeding 2000 words), we omit the full example here.

```
// required types
type key [data].
type skey [data].
type pkey [data].
fun pk(skey): pkey.

// DH formalized as a symmetric equation
fun dh(pkey, skey): key.
equation forall a: skey, b: skey;
    dh(pk(a), b) = dh(pk(b), a).

// key signing
fun sign(skey, bitstring): bitstring.
reduc forall m: bitstring, sk: skey;
    checksign(pk(sk), m, sign(sk, m)) = m.

// symmetric encryption
fun senc(key, bitstring): bitstring.
reduc forall m: bitstring, k: key;
    sdec(k, senc(k,m)) = m.

// hashing/key derivation
// (recall, proverif functions are uni-directional
// unless [data] is specified
fun hash(bitstring): bitstring
fun kdf(bitstring): key
```

Listing 2: Cryptographic primitives for SAGA modeled in ProVerif.

C. Verification Details

To formally verify the SAGA protocol, we employ ProVerif, an automated cryptographic protocol verifier in the symbolic Dolev-Yao model. In the Dolev-Yao model, attackers are assumed to be capable of arbitrarily observing, intercepting, replaying, and synthesizing on-network messages. We construct two models: one model for reasoning about the authentication of agent registration, and another for reasoning about both the authentication and secrecy of agent communication.

We model the required cryptographic primitives for SAGA in ProVerif, including Diffie-Hellman, key signing, symmetric encryption, hashing, and key derivation, as shown in Listing 2.

To formally model SAGA, we precisely replicated the cryptographic operations of the agent registration and agent communication handshakes as described in Section IV-C and Section IV-E respectively. Within ProVerif, we specified secrecy, authentication, and reachability queries for SAGA.

Secrecy. To specify secrecy, we simply employ ProVerif's `attacker` primitive, which soundly determines whether a Dolev-Yao attacker can obtain any given term. In this case, we specify the `attacker` to check whether it can obtain the `token` term.

Authentication and Reachability. To specify authentication and reachability, we employ ProVerif's `event` functionality. We construct six events in our model:

- `EndAuthA` and `EndAuthB`, which trigger once agents A and B complete authentication with the provider.
- `EndProviderAuthA` and `EndProviderAuthB`, which trigger once the provider completes authenticating A and B respectively.

- `EndAgentAuthA` and `EndAgentAuthB`, which trigger once agents A and B complete communication with each other.

To specify authentication queries using these events, we employ the general pattern of proving that *received* content must have been *sent* in exactly the same form. That is, Peer A *receives* a token X, Peer B must have *sent* exactly the token X. To formulate this in PROVERIF, we employ the implies primitive as such. For example, agent A as described in Section IV-C is authenticated against the provider with the following PROVERIF query:

```
query x: bitstring, k:pkey; inj-event(EndAuthA(x, k))
    ==> inj-event(EndProviderAuthA(x, k)).
```

And, to specify reachability, we simply query PROVERIF to check if each specified event occurs.

Finally, our PROVERIF models are open source, and our testing environment is fully declared and reproducible using a nix flake. Using a 6th-generation i7 laptop with 16 GB of RAM, PROVERIF terminates in approximately 10 minutes.

D. Additional Fault Tolerance & Scalability Results

Figure 7 and Figure 8 provide extended throughput results for two core SAGA operations: agent registration and `OTK` refresh, across varying key-chain lengths and fault-tolerant configurations.

Figure 7 shows that agent registration throughput remains high and scales linearly with the number of sharders, even as the number of provisioned `OTK`s increases from 10 to 1000. Although higher key-chain lengths reduce throughput due to larger payload sizes and increased registry writes, the system still achieves over 40K registrations per minute with 1000 `OTK`s and 10 sharders. Additionally, the impact of RAFT-based replication remains marginal across all configurations. This confirms that the append-only nature of agent registration enables efficient write performance.

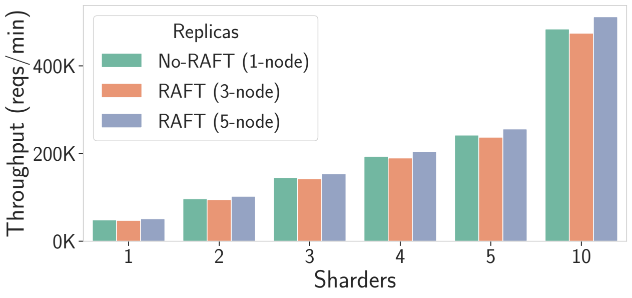
Figure 8 reports `OTK` refresh throughput under the same settings. As expected, throughput decreases with larger key-chains due to the more I/O-intensive update operations. However, even with 1000 `OTK`s, the system sustains nearly 20K refreshes per minute at 10 sharders, even with 5 RAFT replicas. This demonstrates that agents can efficiently pre-provision large key bundles under realistic deployment conditions.

E. Experiments on Amazon Web Services

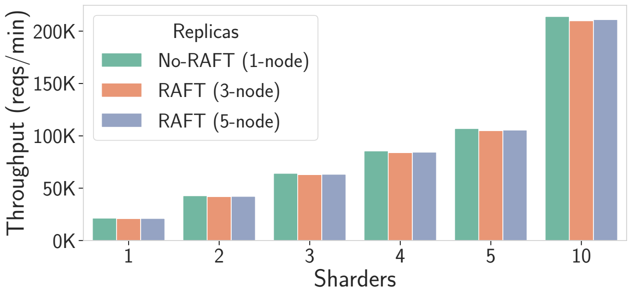
This section presents additional experiments evaluating the system's *Agent Capacity* (C) under realistic cloud deployment conditions. Recall that *Agent Capacity* is defined as the total number of active agents that the system can support concurrently.

We deploy the `Provider` registry service on *Amazon Web Services* (AWS)[4]. The deployment consists of a single

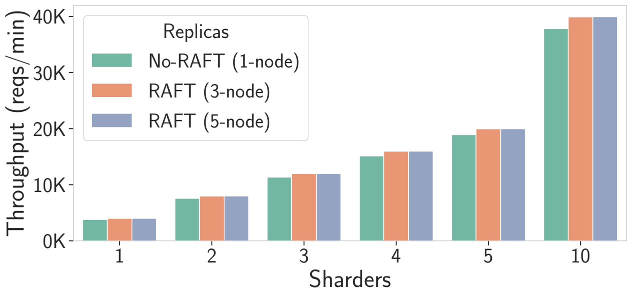
[4]https://aws.amazon.com/



(a) 10 `OTK`s



(b) 100 `OTK`s

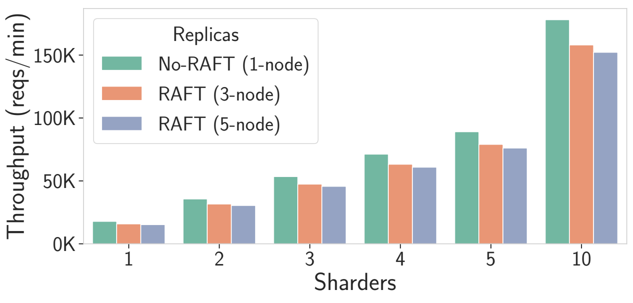


(c) 1000 `OTK`s

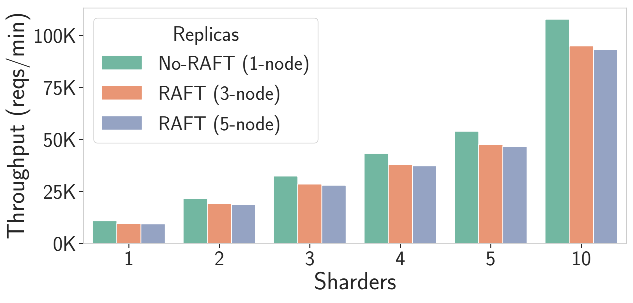
Fig. 7: Agent Registration Throughput for varying `OTK` key-chain sizes.

proxy node and up to seven *sharder* nodes, each hosted on a dedicated *EC2* instance in the US-East region. The proxy node routes incoming requests to the appropriate sharder to ensure balanced workload distribution across the cluster. Sharders are configured to run RAFT with 5-node replication for fault tolerance.

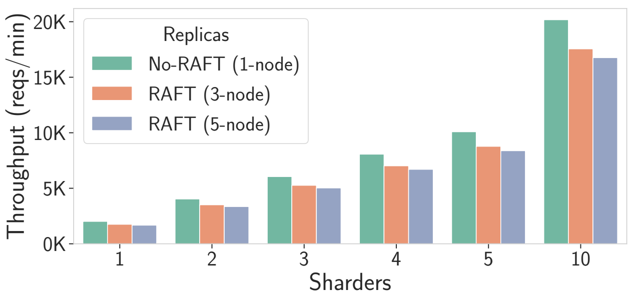
The proxy instance uses a virtual machine of type `c7i.xlarge` (4 vCPUs) optimized for compute, while the sharder instances use `i4i.xlarge` (4 vCPUs) for storage-intensive workloads. Due to AWS account limits, we were constrained to 8 total EC2 instances, 32 vCPUs, and approximately 100K IOPS in total (4 vCPUs/12GB of attached storage per instance). Within these limits, we selected the best available configuration for high-throughput, fault-tolerant deployment. The proxy is configured with 128 threads to maximize request concurrency and dispatches 80,000 requests per



(a) 10 OTKs



(b) 100 OTKs



(c) 1000 OTKs

Fig. 8: OTK Refresh Throughput for varying OTK key-chain sizes.

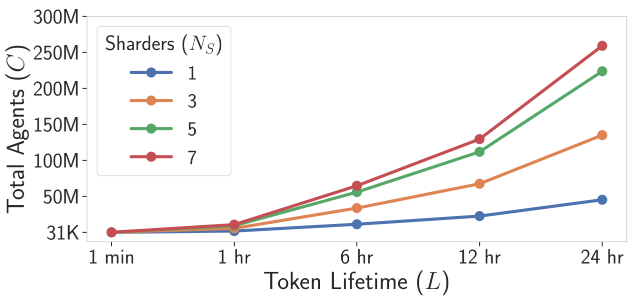


Fig. 9: Total Capacity C of the system for various numbers of sharders on AWS. Each sharder is running RAFT with 5 nodes.

experiment. These parameters were tuned to avoid triggering AWS rate limits during evaluation.

Figure 9 reports the system's agent capacity for varying token lifetimes (L) and number of sharders (N_S). Results confirm the scalability trends shown in Figure 6c, reinforcing the validity of our analysis. With 7 sharders and a 24-hour token lifetime, the system supports up to 260 million agents. We note that as expected, the absolute values for C are slightly lower than those reported in Figure 6c, due to practical constraints: our AWS experiments were limited to 7 sharders (versus 10), and despite co-locating all instances in the same geographic region, we cannot guarantee placement within the same physical datacenter, introducing additional network latency. Nevertheless, the observed trends align closely with our previous analysis. Increasing the number of sharders or extending the token lifetime yields a predictable, nearly multiplicative increase in supported agent population. This empirical validation confirms that SAGA remains performant and scalable under realistic cloud-based infrastructure conditions.