Attacks and Mitigations for Distributed Governance of Agentic AI under Byzantine Adversaries

Matthew D. Laws
Northeastern University
laws.ma@northeastern.edu

Alina Oprea
Northeastern University
a.oprea@northeastern.edu

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

***Abstract*—Agentic AI governance is a critical component of agentic AI infrastructure ensuring that agents follow their owner's communication and interaction policies, and providing protection against attacks from malicious agents. The state-of-the-art solution, SAGA, assumes a logically centralized point of trust, the `Provider`, which serves as a repository for user and agent information and actively enforces policies. While SAGA provides protection against malicious agents, it remains vulnerable to a malicious `Provider` that deviates from the protocol, undermining the security of the identity and access control infrastructure. Deployment on both private and public clouds, each susceptible to insider threats, further increases the risk of `Provider` compromise.**

In this work, we analyze the attacks that can be mounted from a compromised `Provider`, taking into account the different system components and realistic deployments. We identify and execute several concrete attacks with devastating effects: undermining agent attributability, extracting private data, or bypassing access control. We then present three types of solutions for securing the `Provider` that offer different trade-offs between security and performance. We first present SAGA-BFT, a fully byzantine-resilient architecture that provides the strongest protection, but incurs significant performance degradation, due to the high-cost of byzantine resilient protocols. We then propose SAGA-MON and SAGA-AUD, two novel solutions that leverage lightweight server-side monitoring or client-side auditing to provide protection against most classes of attacks with minimal overhead. Finally, we propose SAGA-HYB, a hybrid architecture that combines byzantine-resilience with monitoring and auditing to trade-off security for performance. We evaluate all the architectures and compare them with SAGA. We discuss which solution is best and under what conditions.

I. Introduction

AI agents – systems capable of adaptively pursuing complex goals in dynamic, real-world environments with minimal direct supervision [1] – are rapidly moving from research prototypes to production deployments. Several frameworks (AutoGen [2], MetaGPT [3]) were introduced that enable developers to build autonomous agents atop large language models (LLMs) and a growing number of enterprises are integrating these agents into operational workflows. In addition, protocols like A2A [4] and MCP [5] allow agents to communicate with each other or to call external tools. None of these come with in-built security and they primarily focus on interoperability and deployment.

The rapid deployment of agentic AI without appropriate secure infrastructure has left the current security stack vulnerable to attacks from untrusted or compromised agents. Recent incidents underscore the severity of this gap. In June 2025, Microsoft discovered CVE-2025-32711 (EchoLeak) [6], a zero-click indirect prompt injection in which a malicious email exfiltrates sensitive data from Copilot's context window without any user interaction. In July 2025, Replit's AI agent deleted an entire production database belonging to SaaStr despite explicit user instructions to the contrary [7]. And in November 2025, Anthropic disclosed the first documented AI-agent-orchestrated cyberattack [8]. These examples illustrate how deeply agents can be embedded in critical workflows and the risks they introduce. It is therefore essential to design secure governance architectures that uniquely identify agents, authenticate their interactions, ensure they follow their owner's policies, attribute their actions to the users that own them, and revoke malicious agents from the system [9].

Several solutions have been proposed, including requirements for agent identities [10], capability taxonomies [11], attribution mechanisms [12], authorization with delegation [13], indexing [14] and decentralized identifiers [15], but these remain largely theoretical, lacking implementation, empirical evaluation, or provable guarantees. More concrete proposals exist but each carries significant limitations: constitution-based defenses [16] and just-in-time policy determination [17] rely on the LLM itself for enforcement, while protocol-hardening approaches [18], [19], [20] lack empirical evaluation.

One exception, is SAGA [21], a publicly available [22] governance architecture for agentic AI systems that enables user-controlled agent management, enforces accountability through verifiable governance, and incorporates cryptographic mechanisms to provide forward secrecy and secure inter-agent communication. SAGA relies on a trusted centralized `Provider`, that serves as a registry for user and agent information, and plays an active role in policy enforcement.

While SAGA provides protection against malicious agents, it remains vulnerable to a malicious `Provider`. Deployment on both private and public clouds, each susceptible to insider threats, further increases the risk of `Provider` compromise. In cloud and multi-tenant infrastructure, the shared nature of underlying hardware exposes containers to side-channel attacks that can compromise co-located components [23], [24] and privilege escalation (e.g., CVE-2024-21626, CVE-2019-5736, CVE-2023-1260). A compromised `Provider` could selectively not follow the protocols and undermine the safety guarantees that SAGA is designed to provide. Such behavior is typically referred to as byzantine behavior, and systems

designed to withstand it are referred to as byzantine-resilient.

In this work, we analyze the attacks that can happen from a byzantine `Provider`, taking into account the different components of the system and realistic deployments. We demonstrate that such compromises can have a devastating effect by executing concrete attacks that allow an attacker to undermine agent attributability, extract private data, prevent authorized communication, or allow unauthorized one.

Mitigating byzantine attacks is a challenging task, particularly for systems that have to meet high throughput and low latency requirements such as SAGA's `Provider`, and where the access control nature of the application, requires deterministic finality. On one side byzantine state machine replication with deterministic finality is well understood and many solutions exist based on the pioneering BFT protocol [25]. On the other side, such protocols are notoriously expensive as they require multiple rounds of communication between a significant set of participants (typically two thirds) and difficult to implement and deploy in practice.

We present three types of solutions that offer different trade-offs between security and performance. We first present SAGA-BFT, a fully byzantine-resilient architecture that provides the strongest protection. Specifically, it prevents all the attacks from any compromised component of the `Provider` (e.g. access control engine, database) by replicating these components on a set of replicas, binding their decisions cryptographically, and requiring a quorum of honest replicas for each operation. This architecture incurs significant performance degradation, due to high-cost of byzantine resilient protocols and we provide insights into this cost.

We then propose SAGA-MON and SAGA-AUD, two novel solutions that rely on monitoring and auditing to overcome the cost of byzantine resilient replication. SAGA-MON can be run by the cloud operator or the agentic system operator to detect malicious `Provider` behavior by analyzing network, host, and database logs. SAGA-AUD is intended to be run by the agents owners, where a small set of auditing agents can perform probing requests to check the correctness of the `Provider`'s answers. Both approaches have small overhead, can be performed with different levels of periodicity and intensity and we provide a security analysis that shows how to configure them to ensure high-probability of detection.

Finally, we propose SAGA-HYB, a hybrid architecture that combines byzantine-resilience with monitoring and auditing to trade-off security for performance. SAGA-HYB accommodates different level of security requirements for different type of agents by partitioning the user and agent registries on shards with different levels of security. Agents with higher level of privilege can run on byzantine-resilient `Provider` shards, agents with medium-level of security requirements can ran on fault-tolerant `Provider` shards augmented with auditing and monitoring, and lastly, the agents with small security risk can ran on `Provider` shards that are fault-tolerant.

We evaluate all solutions and compare them against SAGA. SAGA-BFT incurs significant overhead, achieving roughly 1% of SAGA's throughput for the most common class of requests. In contrast, monitoring and auditing achieve roughly 95% and 85% of SAGA's throughput, respectively, on the same workload. Finally, we show that SAGA-HYB amortizes the cost of byzantine resilience, incurring as little as $2\times$ the latency of SAGA while providing byzantine security for a chosen set of agents and users. Our contributions are:

1) We identify and demonstrate concrete attacks against SAGA's distributed `Provider` architecture that compromise agent attributability, data confidentiality, and communication integrity.
2) We design three types of solutions – SAGA-BFT, SAGA-MON and SAGA-AUD, and SAGA-HYB – that defend against these attacks while offering distinct security–performance trade-offs.
3) We evaluate all solutions against SAGA, showing the conditions under which each solution is most effective.

Paper roadmap: Section II provides background on SAGA. Section III describes the attacker model for a malicious `Provider` and the attacks we identify. Section IV, Section V, and Section VII describe our solutions, SAGA-BFT, SAGA-AUD and SAGA-MON, and SAGA-HYB, respectively. We present the security analysis in Section VI and our results in Section VIII. We overview related work in Section IX and conclude our paper in Section X.

II. Background

A. SAGA Overview

SAGA [21] is an architecture for secure governance of AI systems empowering users to have control over the lifecycle of their agents. A `user` is a human with a verifiable identity who owns and controls one or more `agents`, while an `agent` is an autonomous software designed to execute tasks, often relying on LLMs for decision-making. In SAGA, agents are cryptographically linked to the users who own them, and users create and control a contact policy that specifies what agents can interact with their agents and for how long.

SAGA uses a logically centralized entity called `Provider` that plays two roles: (1) serves as a *registry* for users and agents information, and (2) serves as an *access control engine* enforcing the contact policy for each agent as specified by the user who owns it. (See Figure 1).

Consider two agents, A and B that want to communicate. A controls the access of B to A via an Access Control Token (ACT) generated by A specifically for B to use. However, before B can talk with A, it needs to discover A and it does this by contacting the `Provider`. The `Provider` will check if agent B is allowed to contact agent A and in the affirmative case will provide B with information about how to contact A as well as a one-time key (OTK) that will allow the two agents to establish a new secret key based on which the ACT will be derived. The ACT includes a validity timestamp and a maximum number of requests. When the ACT expires, B must re-contact the `Provider` to obtain another OTK that it can use, by communicating with A, to compute a new ACT.

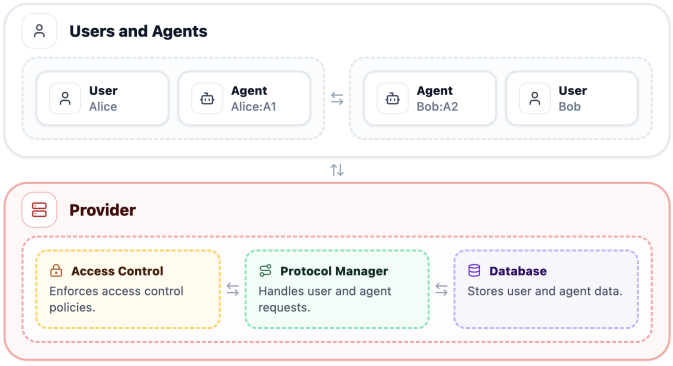


Fig. 1: The SAGA architecture with two users, Alice and Bob, each owning agents A1 and A2, respectively. Users and Agents interact with the `Provider` through the protocol manager which interfaces with the registries stored in a database and the access control engine. Agents also interact directly with each other as described in II-A.

B. SAGA Protocols

Users and agents interact with the `Provider` through four protocols.

User Registration: This protocol registers a user with a real-world human identity so that agents can be linked to a human actor providing attributability.[1] The `Provider` verifies the user's identity using a service such as OpenID Connect [26], then stores the user's data in the user registry.

Agent Registration: This protocol registers the information necessary for agents to establish communication. During registration, the user provides an *agent card* containing the agent identifier (AID) and contact information – both cryptographically bound to the user – along with a set of OTKs and an *Agent Contact Policy* (ACP) specifying the agent's permitted interactions. The ACP defines a set of declarative rules and an associated contact budgets, each specifying a pattern evaluated against the AID of a contacting agent. Upon registration, the `Provider` initializes an *access counter* to ensure the ACP's budget constraints are never violated.

Agent Management: This protocol allows users to modify their agents' ACPs, replenish OTKs, reset access counters, and revoke agents. The user specifies which agent and what artifacts are to be updated to the `Provider`. The `Provider` then verifies the user owns the agent and modified the corresponding registry entry as instructed.

Inter-Agent Communication: This protocol facilitates secure conversation between agents. The contacting agent A requests the information of the receiving agent B. The `Provider` verifies access control by checking A against B's ACP and access counter, and ensures an OTK is available. If allowed, the `Provider` forwards the signed contact information along with a single OTK. The agent can verify the signature on the contact information against the sender's public key to ensure it has not been tampered with. The OTK is used to derive an ACT, and the access counter is incremented.

[1]In an OpenAI white paper Shavit et. al. define attributability as the ability to attribute the actions of an AI agent to a human user [9].

C. Fault-Tolerant and Scalable SAGA

SAGA discusses several design ideas for fault-tolerance and scalability, but it does not provide an implementation for them.

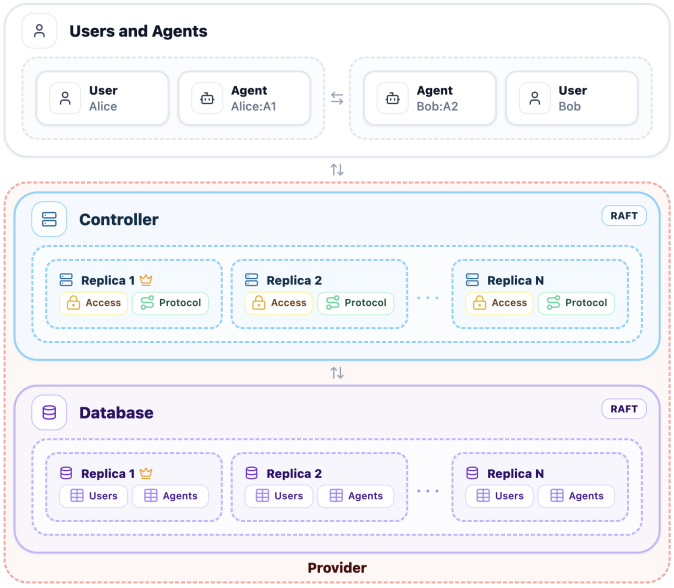


Fig. 2: Fault-tolerant SAGA.

Fault-tolerance. When deploying the `Provider` in a real-world setting, it is essential to ensure that it is fault-tolerant, able to withstand system failures. To achieve this, [21] suggests distributing the `Provider` using the Raft consensus algorithm. A `Provider` replicated with the Raft consensus algorithm and $2f+1$ nodes can tolerate up to f failures while preserving both availability and data integrity [27]. Given the availability of production-grade Raft-replicated databases and different system requirements of the components, a natural design uses two Raft groups. Thus, the `Provider` is logically split into two pieces: the *controller* that enforces access control and protocol management and the *database* (see Figure 2).

Scalability. SAGA [21] discusses how to scale the database by using a common technique in databases called sharding [28], [29], [30], [31], [32]. In sharding, data is partitioned uniformly across multiple independent database instances, known as *shards*, each responsible for a subset of the overall data. A *shard router* then routes queries to the correct shard. SAGA suggests a sharded database, connected to a single controller group, as depicted it Figure 3. Note that sharding works particularly well for SAGA because the partitioning can be done based on user or agent identifiers, respectively.

D. SAGA Threat Model and Limitations

SAGA assumes a trusted provider model: the `Provider` is expected to faithfully execute all SAGA protocols and to preserve the confidentiality, integrity, and availability of both the user and agent registries. Under this assumption, the primary attack surface shifts to the agents themselves. SAGA's

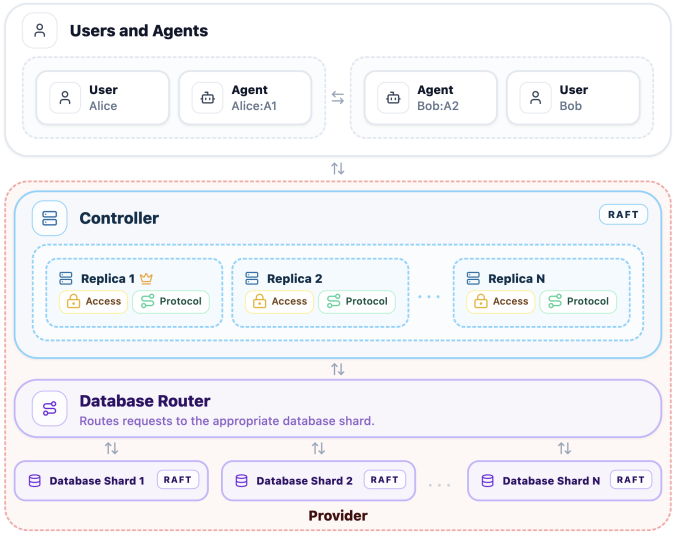


Fig. 3: SAGA configured with a sharded database.

threat model considers adversarial agents that may deviate arbitrarily from prescribed protocols described in Section II-B – including attempting to compromise other agents, exfiltrate data, escalate privileges, or replicate without authorization. The framework's defenses are designed to contain such misbehavior by leveraging the trusted `Provider` as a centralized enforcement point for authentication, authorization, and policy compliance.

Limitations of a trusted Provider. While different configurations can improve SAGA's resilience to failures, SAGA relies on a trusted `Provider` for correct and honest execution. Meeting fault-tolerance and scalability requirements necessitates extending the architecture to include external infrastructure such as replicated or sharded databases across a network of machines, which broadens the attack surface beyond the `Provider` itself.

III. Attacks Against SAGA from a Malicious Provider

In this section we consider the scenario when some components of the `Provider` are compromised and controlled by an attacker. We describe the attacker model in III-A, then analyze several concrete attacks against the SAGA system.

A. Attacker Model for a Malicious Provider

Different components of SAGA may be deployed in environments where they can be compromised [23], [24]. We assume the attacker's goal is to act in a byzantine manner, violating the correctness properties defined in Section II-B. We identify three distinct attack surfaces: the *Access Control Engine* (ACE), the *Protocol Manager* (PM), and the *Database* (DB). The PM and ACE are logical separations of capabilities in the controller. Recall that these are implemented as distributed services replicated with the Raft consensus protocol. We assume that in a system with $2f + 1$ controller replicas and $2g + 1$ database replicas, the attacker can compromise at most f controller replicas and g database replicas, including the leader. We focus on a single-shard configuration; however, the attacks naturally generalize to sharded deployments, with the caveat that an attacker cannot tamper with shards they do not participate in.

A malicious PM is capable of deviating arbitrarily from the prescribed SAGA protocols – including forging, dropping, or modifying messages. A malicious ACE can produce access control decisions that violate the specified policy. A malicious DB can violate the protocol by selectively omitting updates, tampering with them, or returning stale or fabricated responses. We summarize the attacker model in Figure 4 and describe each category in detail below.

B. C1: Prevent Ecosystem Access

Preventing a valid user from registering, logging in, or registering agents denies them access to the ecosystem. A compromised PM or DB can achieve this through several methods – either by modifying registration data to prevent the user from being correctly stored in the database, or by discarding requests entirely and returning a forged acknowledgment. This category of attacks constitute a violation of availability. While the impact is contained to the target user(s), the consequences can be permanent – since SAGA does not permit duplicate users and each user is tied to a single human, even reinitializing with a trustworthy instance may not restore access if the corrupted registration occupies the user's unique identity slot.

C. C2: Undermine Agent Attributability

Attributability deters harm by allowing agent actions to be traced back to real, human identities [9]. Circumventing attributability allows malicious agents to operate in the ecosystem without risk of consequence. This category violates the user registration protocol and can be executed by a compromised PM. By ignoring a failed response from an identity service, or skipping the request entirely, the PM can admit users to the system without binding them to a human identity. The impact of these attacks extends beyond a target user – any agent in the ecosystem that can be contacted by the unattributed agent is at risk, as the agent can now act maliciously without consequences. These attacks violate integrity and can lead to confidentiality breaches. Recovery would be difficult as it would require all users re-verifying, or cross referencing all UIDs against the identity service's data.

D. C3: Extract Private Data

The `Provider` receives and stores sensitive user information, including email addresses and authentication credentials. A compromised PM can intercept this data during registration or authentication, and a compromised DB can extract personally identifiable information (PII) directly from the registry at any point after it has been stored. Compromised credentials can grant an attacker unauthorized access to the victim's agents or, if the user reuses passwords across services, to accounts on external platforms. Extracted email addresses expose users

Attack category	Example attack vectors	Protocol	CIA violation	Attack surfaces
C1: Prevent Ecosystem Access	• Modify registration data to corrupt stored credentials. • Silently discard valid registration requests.	User registration Agent registration	Availability	PM DBR
C2: Undermine Agent Attributability	• Does not perform real human verification during registration.	User registration	Integrity	PM
C3: Extract Private User Data	• Share credentials/PII during registration or login. • Exfiltrate stored emails from the database.	User registration	Confidentiality	PM DBR
C4: Prevent Agent Lifecycle Management	• Silently discard agent management requests. • Modifies management requests to improperly update agent.	Agent management	Availability Integrity	PM DBR
C5: Allow Unauthorized Agent Communication	• Provide additional OTKs. • Corrupt access control state to allow unauthorized communication.	Agent Registration Inter-agent comm.	Integrity	PM ACE DBR
C6: Disrupt Authorized Agent Communication	• Prevent agents from registering. • Falsify or ignore access control checks.	Agent Registration Inter-agent comm.	Availability	PM ACE DBR

Fig. 4: Summary of attack categories from a malicious `Provider` against SAGA.

to identity-based attacks and targeted phishing. These attacks constitute a violation of confidentiality with a wide-ranging impact. Since the attacker obtains raw credentials and PII, the damage extends beyond the SAGA ecosystem and persists indefinitely regardless of whether the compromised component is subsequently detected and replaced.

E. C4: Prevent Agent Lifecycle Management

Managing an agent once deployed is essential for safety. The ability to revoke an agent is a critical fail-safe for preventing both accidental and intentional harm [9], and serves as a last resort for blocking active attacks. Without this ability, rogue agents can cause unchecked harm. Conversely, refreshing OTKs and the access counter, as well as updating contact policies, allows an agent to persist safely in the ecosystem. Without maintaining these, an agent will become unreachable. Further, the ability to update contact policies, allows users to block a newly discovered malicious agent or enable communication with a new agents as required by an evolving workflow. This category of attacks violate the agent management protocol and can be realized by either a compromised PM or DB. Either component can silently block or tamper with modification requests from reaching the database while returning a false acknowledgment to the user, leaving them unaware that their changes were never applied. These attacks impacts are contained to the target user(s). If detected these attacks should be recoverable after reinitialization; malicious contact during the vulnerability window cannot be undone.

F. C5: Allow Unauthorized Agent Communication

Access control is essential to the SAGA ecosystem. Without enforceable access control, a malicious agent can discover or contact agents it is forbidden from reaching, enabling a variety of known attacks [33], [34], [35], [36], [37]. Even if the target agent independently enforces its own access control, extracting the target's information and OTK during discovery enables the attacker to initiate future attacks under a different identity. These attacks can be executed by either a malicious ACE, PM, or DB and violates the inter-agent communication protocol – the ACE by falsifying access control checks, the PM by ignoring correct ones, or the DB by returning falsified information or additional OTKs. The impact is limited to the targeted agent(s), but can result in significant and irreversible integrity and confidentiality violations. While access control can be restored after reinitializing the corrupted component, the damage to the target agent may already be done. Combining this category with category *Undermine Agent Attributability* (described in Section III-C), amplifies the harm, as the attacker can breach access control without consequences.

G. C6: Disrupt Authorized Agent Communication

Conversely, inter-agent communication remains essential to multi-agent workflows. Preventing agents from reaching other agents they are authorized to contact may render them unable to execute their prescribed tasks. Improperly maintaining, initializing, or verifying the access counter, policy, or OTKs can lead to valid communication being rejected – violating the inter-agent communication protocol. These attacks can be carried out by a compromised PM, ACE, or DB. The impact is in terms of availability and it contained to the target agent(s); however, blocked communication can lead to critical failures in important workflows. Reinitializing the corrupted component can reestablish communication flows, but errors that occurred during the disruption may be unrecoverable.

IV. SAGA-BFT: A Byzantine Fault Tolerant Architecture for SAGA

We present SAGA-BFT, a byzantine-resilient solution for SAGA. For maximum security, both the controller and the database must be made byzantine fault-tolerant.

A. Design

BFT Database. To ensure safe behavior in SAGA's database we can replace the fault-tolerant database with a byzantine fault-tolerant database. Each database update is submitted to all replicas, which must reach agreement before the update is

committed. The main difference is that BFT protocols require $3f+1$ nodes to tolerate up to f byzantine faults [25], ensuring that the database operates correctly even in the presence of arbitrarily malicious replicas. Several blockchain database systems built on the Tendermint protocol [38] already exist [39], [40], [41], providing well-tested BFT implementations that can serve as a foundation for SAGA-BFT. Leveraging such established systems reduces both the engineering effort and the risk of implementation-level vulnerabilities.

BFT Controller. Making the controller Byzantine fault-tolerant is more complex than the database. As with the database layer, the consensus algorithm would need to be upgraded from a fault-tolerant one to BFT. Unlike the database layer where established BFT systems can be adopted off the shelf, the controller is a custom component. Thus, its access control logic, protocol management, and state transitions would each need to be replicated and agreed upon across $3g+1$ instances. This could be accomplished by adapting a BFT replicated state machine such as PBFT [25]; however, generic BFT state machine replication incurs significant performance degradation compared to purpose-built protocols [42].

B. Security and Performance Limitations

Security. The key advantage of a fully BFT configuration is that it is provably secure against all classes of integrity and availability attacks from compromised components of the `Provider`, as long as the BFT assumptions are met. This means no more than f replicas are compromised in the case of the database, and no more than g replicas are compromised in the case of the controller, for a system with $3f+1$ database replicas and $3g+1$ controller replicas. Confidentiality attacks III-D, however, falls outside this guarantee: BFT protocols govern agreement and ordering, not what data nodes are permitted to observe or leak.

Performance Limitations. The primary drawback of byzantine-resilient consensus is its significantly higher communication overhead compared to fault-tolerant consensus. Where Raft requires $O(n)$ messages per operation, BFT protocols often require $O(n^2)$ messages per operation due to the all-to-all communication rounds necessary to ensure agreement in the presence of malicious nodes. This quadratic overhead would directly impact the latency of every user and agent interaction with the `Provider`. Furthermore, correctly implementing a BFT protocol is notoriously difficult, and implementation bugs have historically led to the very vulnerabilities these protocols are designed to prevent [43], [44].

V. Mitigation by Monitoring and Auditing

In this section we describe lightweight solutions to mitigate the attacks we described in Section III based on monitoring and auditing. We first give an overview of the approach and then describe our auditing and monitoring solutions in detail.

A. Overview

Detecting byzantine behavior is typically very challenging when the ground truth is not known, or there are no known invariants in the system. In the case of the `Provider` service, we observe that we can create opportunities for the ground-truth to be known. Specifically, we use auditing to introduce in the system actions, for which we know the correct answer. The auditor will take an active role and insert canary actions to see if the audited party performs the action correctly.

We also observe that there are invariants that we can derive, based on the application semantic, for example what changes should be reflected in the database, in response to a message sent over the network. Unlike the auditor, the monitor is a passive observer of the actions of the system. We derive system invariants and use monitoring to analyze database logs and network communication to check if the request and response observed match, as they are specified by the protocol. In our case, the target monitored are the different `Provider` components (the controller and database) and can be done by the cloud operator where the `Provider` is deployed. Note that the `Provider` is implemented as a distributed service thus the monitoring implies monitoring all the nodes from the service.

Once a compromised component is detected, it can be reinitialized from a trusted image, restoring safe operation. Users can pause their agents while the system is repaired.

B. Client-Side Auditing

Goals. In this approach, users proactively audit the `Provider`. This is done in an end-to-end manner: users directly or through their agents, periodically insert canary actions, such as registering a new agent, or updating information for an existing one, and observe the answer they receive from one of the `Provider` nodes.

Assumptions. This approach operates with exactly the view of the system available to any user, requiring no additional information or privileged access. It allows users who may not trust their `Provider` to verify the integrity of the governance system as it relates to their own agents. A single user, or a group of users, can run the auditing algorithm independently. We assume that a malicious `Provider` cannot distinguish the auditor from an ordinary user, and therefore treats the auditor's actions in the same manner as those of any other participant.

Algorithm. The algorithm operates on a cycle that can be manually initiated, run periodically, or triggered by a configurable set of rules or conditions. During each cycle the auditor user will insert a list of actions, and assert that the `Provider` responses are consistent with correct protocol execution. The set of actions our algorithm uses are: ① registering a new user, ② registering and revoking an agent, ③ registering two agents with policies that prevents mutual communication, and ④ registering two agents with policies that allows communication. More details below:

- *Register a new user*: The auditor attempts to register a new user with invalid credentials. The user registration protocol requires the `Provider` to reject such requests; if the registration succeeds, the algorithm concludes that the `Provider` has been compromised.

- *Register and revoke an agent*: The auditor probes the agent management endpoint by registering an agent they own and subsequently revoking it. The auditor then attempts to access the revoked agent; if the request succeeds, the revocation was not correctly executed, indicating a compromised `Provider`.
- *Prohibited communication*: The auditor registers two agents whose contact policies prohibit mutual communication; if the `Provider` permits the interaction. it has been compromised.
- *Allowed communication*: The auditor registers two agents whose policies explicitly allow communication and verify that the `Provider` correctly facilitates the exchange. A failure indicates that the `Provider` is not faithfully enforcing the contact policy.

For more effective auditing, users can randomize they auditing cycles and coordinate with other agents. This cycle assumes that attacks occur at the granularity of full protocols. More sophisticated attacks can be detected by augmenting the cycle with finer-grained checks that pinpoint which stage of the protocol is failing. See Figure 13 in Appendix A.

Discussion. The main limitation of this technique is that if the `Provider` is able to distinguish between an auditor and a regular user, then the `Provider` can behave correctly when audited, and malicious otherwise. To ensure that the `Provider` cannot distinguish auditor clients from regular clients, auditors must behave exactly like normal users in every observable way. They should use the same client software and public APIs, avoid any special credentials or identifiable account traits, and match typical network characteristics such as IP type, TLS configuration, and request headers. Their activity should mimic real user behavior, including realistic timing, usage patterns, and variability, while being blended into a larger pool of similar clients to avoid standing out. Audits should be randomized in timing and actions.

Another limitation is that, since auditing operates on the client side, in some cases there is no way to distinguish between a malicious controller and a malicious database. Thus, upon detecting an error further investigation is required.

Finally, similar to SAGA-BFT our defense focuses on correct execution of the protocol and does not address how an attacker handles private data (III-D).

C. Provider-Side Monitoring

Goals. In this approach, the goal is to detect inconsistencies in the information transmitted and persisted across the various stages of the SAGA protocols. The monitoring service consists of log collection and a verifier that analyzes the logs, see Figure 5.

Assumptions. We assume a cloud deployment where the `Provider` service is distributed across multiple nodes, the database runs on separate and better protected nodes, and several trusted services are available to the cloud operator. The monitoring algorithm is run by the cloud operator who is proactively monitoring the `Provider` deployment. The monitor does not trust the `Provider` code but it trusts two key sources of information: (i) a complete record of the requests and responses exchanged between the `Provider` and its clients (users and agents), and (ii) a trusted, ground-truth view of the database state. With this information, the monitor can ensure integrity between the `Provider` and supporting database. To verify integrity within the controller and database, a quorum of the replicas' logs for each component must agree. For example, assuming a cluster of $2f + 1$ database replicas tolerating at most f faults, a quorum consists of $f+1$ replicas.

Monitoring and logs collection. One approach is to capture traffic to and from the `Provider` through a trusted web server, CGI, or sidecar proxy that records incoming requests and outgoing responses before they reach the untrusted `Provider` logic. Although client–provider traffic is protected by TLS, in most deployments TLS termination occurs at or before these layers, allowing the verifier to observe the messages unencrypted. An alternative approach is to monitor network traffic directly: the `Provider` dumps its TLS session keys to the verifier, which then decrypts the traffic. Under this model, any attempt by a compromised `Provider` to withhold or falsify its session keys would manifest as undecodable traffic and be treated as a system compromise. Traffic is stored in the logs as a tuple of requests and responses.

Obtaining the view of the database requires only read access to the underlying replica machines. While databases are typically accessed through a query language, their state is persisted in a raw on-disk format that can be read directly, bypassing any malicious logic in the database query engine.

We can additionally monitor the network traffic exchanged between the controller and the database replicas. This enables the monitor to observe precisely which requests are issued by the controller, making it possible to detect cases in which the controller fails to forward a client request to the database, issues spurious requests, or modifies requests in transit. Critically, this view allows us to attribute a detected inconsistency to either the controller or the database. We provide a full diagram of the setup in Figure 5.

Algorithm. To enable correlation across logs, the verifier attaches a unique action identifier to each transaction, allowing database operations to be matched to their originating client requests. The identifier is generated automatically when a client issues a request and requires no changes to client behavior. The controller records requests in first-in-first-out order, and the database likewise commits changes in order. The verifier can reconcile the two streams in a single ordered walk, advancing a pointer into each log in tandem. Periodically, it ingests completed request–response tuples from the controller and committed changes from the database, joined on the action identifier. Throughout, the verifier maintains an active model of the expected database state $\mathcal{D}$, advanced incrementally as each change is processed, so that suppression, injection, and tampering attacks are detected in a single traversal and access-control decisions can be audited against the reconstructed state at the moment each action was processed.

For this traversal, we introduce a window W specifying how long the verifier waits before declaring an action unreconciled.

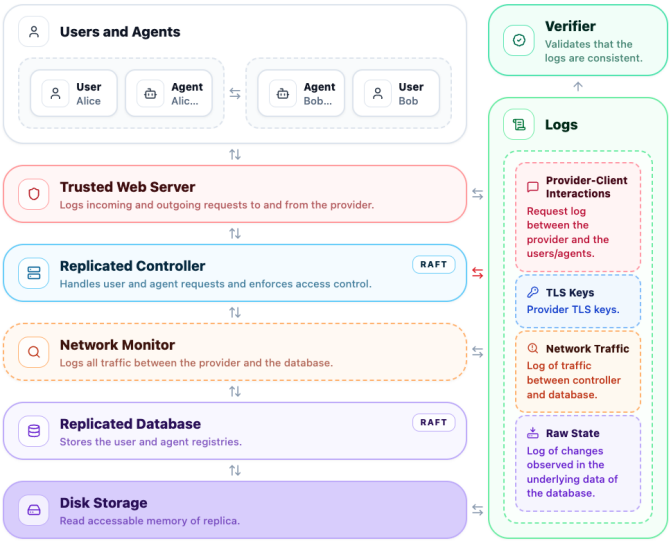


Fig. 5: The monitoring architecture for SAGA. Trusted components emit logs to a shared verifier, which correlates them to ensure consistency across the protocol. controller is not trusted; it is only responsible for dumping its TLS session keys, any withholding or forgery of which is detectable as undecodable traffic. The network monitor is optional; it attributes detected malicious behavior to a specific component.

W represents the brief interval during which an action may legitimately be in flight between the controller, the database, and the verifier. This same allowance, however, means that detection occurs on a short delay. In practice W is bounded by network latency and is therefore small, so any sustained adversarial activity is detected quickly, before meaningful harm can be done. The algorithm is presented in detail in Figure 14 in Appendix A. When actively run on a fault-tolerant `Provider`, this configuration is referred to as SAGA-MON.

Discussion. One concern is the trustworthiness of the logs our framework relies on. If container-level logs cannot be trusted, TLS dumping with network-layer monitoring provides logs of `Provider` behavior that a compromised component cannot tamper with. Furthermore, even if container memory cannot be trusted, an initial database snapshot can be obtained from a quorum of replicas, and subsequent state can be reconstructed from the network logs as requests are processed.

VI. Security Analysis of Monitoring and Auditing

In this section, we formalize the security guarantees provided by our monitoring and auditing techniques. We characterize each mechanism in terms of the adversary it constrains, the assumptions it relies on, and the quantitative bounds it offers on detection. For auditing, we derive the expected time to detect a compromised `Provider`, the expected number of detections over a given time horizon, and the trade-offs that arise when the attacker and defender each tune their strategies. For monitoring, we formalize the invariants that bound the vulnerability window and analyze the trade-offs required to maintain them in practice. Together, these analyses delineate the configurations in which each mechanism is effective.

A. Security Analysis of Auditing

We assume a threat model in which the attacker cannot distinguish the auditor from an ordinary user, and therefore treats the auditor's actions in the same manner as those of any other participant. Under this assumption, the auditor's checks constitute a representative sample of the provider's behavior.

Variable	Definition
m	Number of checks per audit cycle.
δ	Time between cycles.
α	Probability of attack, $\Pr(\text{attack})$.
q	Detection probability given an attack, $\Pr(\text{detect} \mid \text{attack})$.
D_T	Number of detections at time T.

TABLE I: Auditing Variable Definitions

We define our notation in Table I. We derive the expected time to detection. The probability that a single check both coincides with an ongoing attack and successfully detects it is

$$\Pr(\text{attack}) \cdot \Pr(\text{detect} \mid \text{attack}) = \alpha q,$$

assuming each check is independent, and the auditor performs m checks per cycle, it detects an attack within a given cycle with probability

$$s = 1 - (1 - \alpha q)^m.$$

The cycle at which detection first occurs therefore follows a geometric distribution, $C \sim \text{Geom}(s)$, with expectation $\mathbb{E}[C] = 1/s$. Each cycle has fixed duration δ, so the expected time to detection (TTD) is

$$\mathbb{E}[\text{TTD}] = \frac{\delta}{1 - (1 - \alpha q)^m}.$$

Although the system should typically be reinitialized immediately upon detecting an error, it is also useful to consider the expected number of detections as a function of time. The number of detections across n checks follows a binomial distribution,

$$D_n \sim \text{Binomial}(n, \alpha q),$$

with expectation $\mathbb{E}[D_n] = n\alpha q$. In time T the auditor performs between $\lfloor mT/\delta \rfloor$ and $\lceil mT/\delta \rceil$ checks. As we make no assumption about how checks are distributed within a cycle, we take the conservative approximation $n \approx \lfloor mT/\delta \rfloor$, yielding the expected number of detections in time T (D_T):

$$\mathbb{E}[D_T] \approx \left\lfloor \frac{mT}{\delta} \right\rfloor \alpha q.$$

Finally, we consider the detection–evasion game between defender and attacker, examining how each tunes their parameters to achieve their goals. For a small error rate, ϵ, we ask how much time must elapse before detection occurs. This quantity is central to both sides: the attacker seeks to remain undetected with probability $1 - \epsilon$, while the auditor seeks to

detect with the same probability. Recall that the probability of detecting an attack within a single cycle is $s = 1-(1-\alpha q)^m$, so the probability of at least one detection within c cycles is

$$\Pr(D_c > 0) = 1 - (1 - \alpha q)^{mc}.$$

We know that $c \approx T/\delta$, thus we can equivalently express this in terms of time:

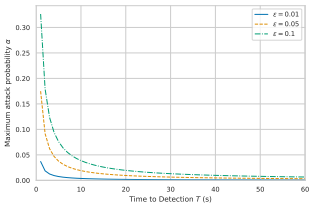
$$\Pr(D_T > 0) = 1 - (1 - \alpha q)^{\frac{mT}{\delta}}.$$

Separating the tunable parameters of each side, the defender most naturally tunes the cycle duration δ, while the attacker tunes the attack probability α. Solving $1 - \Pr(D_T) \geq 1 - \epsilon$ for the attacker and $\Pr(D_T) \geq 1 - \epsilon$ for the defender yields:

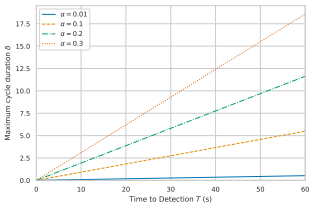
$$\alpha \leq \frac{1 - (1-\epsilon)^{\frac{\delta}{(mT)}}}{q}, \qquad \delta \leq \frac{mT \log(1 - \alpha q)}{\log \epsilon}.$$

For our auditing cycle as described in Section V-B, we set $m = 4$ checks per cycle and detection probability $q = 1$. Assuming the attacker can learn the defender's parameters, they tune their attack rate α based on an acceptable confidence level ϵ and attack window T (Figure 6a). For this experiment we set $\delta = 15$. From the defender's perspective, we fix ϵ and show admissible cycle durations δ under varying attacker strategies and tolerated vulnerability windows (Figure 6b).

Under our configuration, an attacker must use α close to 0 to remain undetected for only 60 seconds, even when tolerating a high, 10%, detection risk. Conversely, the defender needs frequent auditing to achieve 99% detection against small α, while larger α permits longer audit cycles.



(a) Attacker's bound on the attack probability α as a function of time T, for varying detection-confidence levels ϵ. Each curve traces the maximum α at which the attacker remains undetected with probability at least $1 - \epsilon$ when $\delta = 15$.



(b) Defender's bound on the cycle duration δ as a function of the time to detection T, for varying attacker per-check attack probabilities α. Each curve traces the maximum δ for which the auditor detects an ongoing attack with $\epsilon = 0.01$.

Fig. 6: Bounds for tuning parameters for the attacker and defender.

B. Security Analysis of Monitoring

In this section we formalize the security guarantees provided by the `Provider`-side monitor. We assume a system with $2f + 1$ controller nodes and $2g + 1$ database nodes. An adversary $\mathcal{A}$ can control at most f controller replicas and g database replicas, and has no control over the verifier. $\mathcal{A}$ possesses all attack capabilities described in Section III-A. We assume the verifier observes both logs in order through tamper-evident channels and that the action-identifier scheme is collision-resistant. We define *actions* as the set of requests received by the controller and *changes* as those sent to the database. The full set of variables used in our monitoring framework is defined in Table II.

Variable	Definition
$\mathcal{D}_i$	Database state after action a_i.
$\mathcal{R}^*$	Expected response given the database state.
$A = [a_1, \ldots, a_n]$	Sequence of all actions.
$C = [c_1, \ldots, c_n]$	Sequence of all changes.
$\Delta^* = \mathcal{D}_i - \mathcal{D}_{i-1}$	Expected change resulting from action a_i.
W	Window of potential matches.

TABLE II: Monitoring variable definitions.

For monitoring to remain coherent under replication, each cluster must expose a single canonical log to the verifier. Since both the controller and the database are replicated using Raft, this follows directly from the protocol's guarantees: Raft ensures that any committed entry is durably replicated to a quorum [27]. The canonical controller log (A) and database log (C) presented to the verifier are therefore the sequences of entries committed by their respective quorums.

Let $id(a_i)$, $req(a_i)$, and $res(a_i)$ be the identifier, request, and response of action a_i, respectively. For correct monitoring of integrity errors, we must ensure that for every action and change, the corresponding identifier is observed and that the change matches the one induced by the action. A response is *valid*, denoted $\mathcal{R}^*$, if it is consistent with both the returned status and the current database state. This consistency is essential for access control: if the response indicates success, the database state must reflect that the operation was permitted, and conversely, any change to the database must correspond to a successfully authorized action. We define these invariants:

$$\textcircled{1} \quad \forall i : \quad id(a_i) = id(c_i) \ \wedge \ c_i = \Delta^*(a_i).$$
$$\textcircled{2} \quad \forall i : \quad res(a_i) = \mathcal{R}^*\big(\mathcal{D}_{i-1},\, req(a_i)\big).$$

The key challenge in enforcing these invariants is alignment: when $\mathcal{A}$ may drop, modify, or inject actions, the verifier must decide which entries to compare in the first place. We define $\hat{A} = [\hat{a}_1, \ldots, \hat{a}_n]$ as the sequence of actions recorded in the controller log, and $\hat{C} = [\hat{c}_1, \ldots, \hat{c}_m]$ as the sequence of changes read from the replica's disk. Both sequences preserve commit order, but neither is guaranteed to be complete: a legitimately delayed action may not yet appear in $\hat{C}$ at the moment $\hat{A}$ is read. To accommodate this benign skew without admitting adversarial drops, we introduce a window W within which a corresponding entry must appear. We relax invariant ① into its windowed form, splitting it into forward and reverse directions to capture both suppression and injection; invariant ② is remains the same.

$$\forall\, \hat{a}_i \in \hat{A},\ \exists!\, \hat{c}_j \in \hat{C}_{[W]} : id(\hat{a}_i) = id(\hat{c}_j) \wedge \hat{c}_j = \Delta^*(\hat{a}_i),$$
$$\forall\, \hat{c}_i \in \hat{C},\ \exists!\, \hat{a}_j \in \hat{A}_{[W]} : id(\hat{c}_i) = id(\hat{a}_j),$$
$$\forall\, \hat{a}_i \in \hat{A} : res(\hat{a}_i) = \mathcal{R}^*\big(\mathcal{D}_{i-1},\, req(\hat{a}_i)\big).$$

This raises the natural question of how to tune W. The window must be large enough that legitimately delayed entries are not mistaken for adversarial drops, yet small enough to bound the interval during which an attack may persist undetected. In practice, W can be configured against a target false-positive tolerance (FP). Let μ and σ denote the mean and standard deviation of the delay between the controller and the database in delivering an entry to the verifier. Modeling this delay as log-normally distributed – a standard choice for request latencies [45] – the false-positive rate induced by flagging actions whose corresponding logs have not yet been fully populated is:

$$\mathrm{FP}(W) = 1 - F_{\mathrm{LN}}(W;\, \mu,\, \sigma),$$

where F_{LN} is the log-normal CDF. A target FP $\leq \epsilon$ is therefore achieved by setting:

$$W = F_{\mathrm{LN}}^{-1}(1 - \epsilon;\, \mu,\, \sigma).$$

With this in mind, we can consider realistic deployments. Suppose the controller is located in US-East, with the verifier co-located with the controller. We can then tune W for different locations of the database. We fit a log-normal distribution to recorded AWS round-trip-times for databases located in US-West, Europe, and Asia, the resulting fit is shown in Figure 15 in Appendix A.

VII. SAGA-HYB Architecture

This section presents SAGA-HYB, a natural extension to SAGA-BFT and SAGA-MON that combines their respective strengths for large-scale deployments. SAGA-HYB achieves the strong guarantees of SAGA-BFT for specific shards of the registry, while preserving the low overhead of SAGA-MON throughout the rest of the system.

A. Sharded Provider

To increase the overall scalability and availability of the system, we deploy multiple `Provider` shards, each consisting of a replicated controller backed by its own database and responsible for a disjoint subset of the agent and user space. As shown in Figure 7, a routing layer is introduced through which all user and agent traffic is directed. The router maps each request to the appropriate controller based on the target user or agent. We refer to this composite system as SAGA-HYB. This architecture yields scalability benefits as the `Provider` is no longer a computational bottleneck, and the system can scale horizontally by adding new `Provider` shards.

B. Heterogeneous Configurations

The monitoring and auditing techniques described in Section V provide strong guarantees for detecting malicious behavior; however, detection is inherently delayed, and short vulnerability windows can still arise. In practice, different actors within the ecosystem may have varying requirements for the security–performance trade-off. Agents entrusted with highly sensitive data may warrant the stronger integrity guarantees of SAGA-BFT, whereas latency-sensitive workloads may benefit from a database configuration with a large in-memory cache. The routing layer introduced in Section VII-A (see Figure 7) naturally accommodates such heterogeneity: each `Provider` shard can be independently configured with a distinct consensus protocol, database, caching policy, and monitoring cadence, while remaining accessible through a unified interface. This design enables the system to sustain high aggregate throughput while allocating stronger protections to the most sensitive subsets of state. This configuration is particularly well suited to large-scale, multi-tenant deployments, in which parties with divergent threat models and performance requirements must coexist within a shared infrastructure.

Security. In SAGA-HYB, users and agents assigned to a BFT shard inherit the full safety guarantees of SAGA-BFT, while all others fall under the protection of SAGA-MON. Furthermore, in SAGA-HYB a single compromised controller or database can only affect the subset of data it is responsible for – roughly $\frac{1}{n}$ of the total state, where n is the number of `Provider` shards – limiting the radius of any individual compromise.

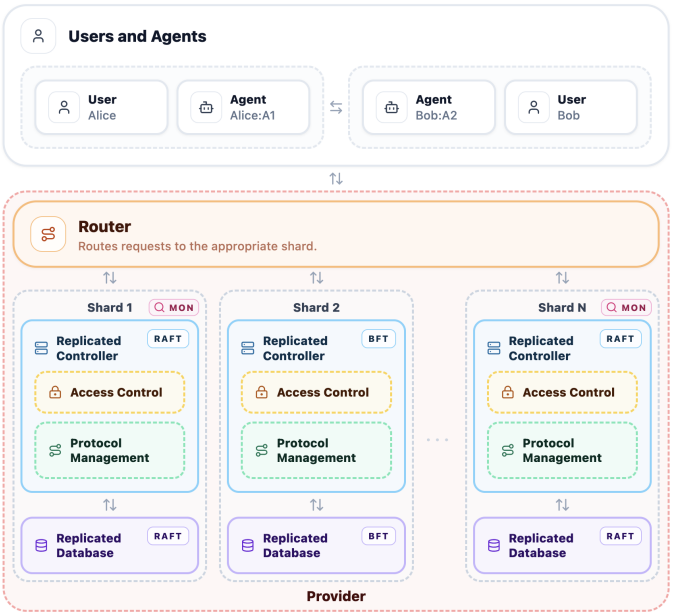


Fig. 7: Hybrid configuration where some of the agent and user registry is run by a byzantine-resilient SAGA, while other are run by fault-tolerant SAGA some with monitoring enabled.

VIII. Evaluation

In this section, we evaluate the Byzantine-resilient strategies for SAGA described in prior sections and quantify their effects on system performance and security.

A. Setup

All experiments were conducted on a 16-core AMD Threadripper PRO 5955WX CPU paired with a Samsung MZ1L21T9HCLS-00A07 SSD. Agent workloads were driven by GPT-5.4, GPT-5.4-mini, and Qwen3-VL-30B-A3B-Instruct-FP8. Different nodes were deployed as containers and

connected using a Docker network. We use *Gunicorn* as our WSGI server, configured to handle multiple concurrent clients.

SAGA code does not implement fault-tolerance and scalability. We implemented the SAGA fault-tolerant and scalable architecture using the basecode of SAGA available at [22] and integrating it with RethinkDB [46], a production-grade NoSQL database backed by a custom JSON store that natively supports replication and sharding.

For SAGA-BFT, we use BigchainDB [39], a BFT-backed key–value store built on top of the Tendermint (now CometBFT) consensus protocol [38]. Each registry action is modeled as a transaction comprising immutable data and mutable metadata. BigchainDB was shown to perform competitively among BFT-enabled databases [41]. For SAGA-HYB, we additionally implement a custom routing layer that directs users and agents to the `Provider` shard responsible for their state. Both RethinkDB and BigchainDB are configured to tolerate a single faulty replica – requiring 3 and 4 replicas, respectively – unless otherwise noted.

B. Attacker Evaluation

We evaluated SAGA's resilience against the adversarial behaviors outlined in our threat model (Section III). Our evaluation comprises 16 distinct attacks spanning all 6 attack categories. Attacks originate from each of the three system components – protocol management (PM), access control engine (ACE), and database (DB) – and target all protocols, resulting in violations of confidentiality, integrity, and availability. We summarize the results of our attacks in Table III and provide a full description of all 16 attacks in Appendix B.

	PM	ACE	DB	Total
C1: Prevent ecosystem access	1	0	1	2
C2: Undermine agent attributability	1	0	0	1
C3: Extract private user data	1	0	1	2
C4: Prevent agent management	1	0	1	2
C5: Allow unauthorized agent comm.	1	1	2	4
C6: Prevent authorized agent comm.	2	1	2	5
Confidentiality	1	0	1	2
Integrity	4	1	3	8
Availability	2	1	3	6
Total	7	2	7	16

TABLE III: Distribution of executed attacks by originating component, broken down by attack category and security property violated. Some attacks violate multiple CIA properties; we report the most salient one in each case.

C. Monitoring and Auditing Evaluation

To evaluate our monitoring and auditing frameworks, we implement a malicious controller. Each request the controller processes is subjected to a probabilistic corruption model: with probability α, the proxy behaves maliciously, modifying or suppressing the message in accordance with a configured attack strategy. The attacks are chosen are representative of each category mountable by a compromised controller. The specific attack applied to each intercepted message is selected based on the protocol and direction. We apply the attacks indiscriminately across all users and agents.

Expected Number of Errors Detected by Auditing: We configure our auditing protocol as described in Section V-B with $m = 4$ checks per cycle and cycle duration $\delta = 15$. The attacker is configured with $\alpha = 0.3$, and each check has detection probability $q = 1$ when the corresponding component is attacked. Since we do not mount verification attacks, only 3 of the 4 checks are effective. As derived in Section VI-A, with these parameters the expected number of detections by time T is

$$\mathbb{E}[D_T] \approx \left\lfloor \frac{3T}{15} \right\rfloor \cdot 0.3 = \left\lfloor \frac{T}{5} \right\rfloor \cdot 0.3.$$

Errors Detected with Monitoring: We instantiate our monitoring hooks using a trusted CGI that intercepts client–`Provider` traffic before it reaches the untrusted `Provider` logic. For simplicity, we extract the database state by issuing queries from a whitelisted IP to the database infrastructure; a production implementation could instead read directly from the underlying on-disk data files to bypass the database query engine.

Our monitoring protocol is configured with a window size $W = 100$ ms and detects a wide range of malicious behaviors. We chose 100 ms to effectively eliminate false positives in a US-East × US-West deployment (per Section VI-B) and thus provide a faithful result. In some cases, the same attack can be flagged multiple times, or subsequent state mismatches can be detected in the aftermath of the initial violation. This redundancy is beneficial: since a single confirmed detection is sufficient to trigger system re-imaging, the goal is simply to catch the attack. However, for the analysis, we examine detection rate over time without triggering re-imaging; and accordingly, duplicate detections of the same underlying attack are filtered out.

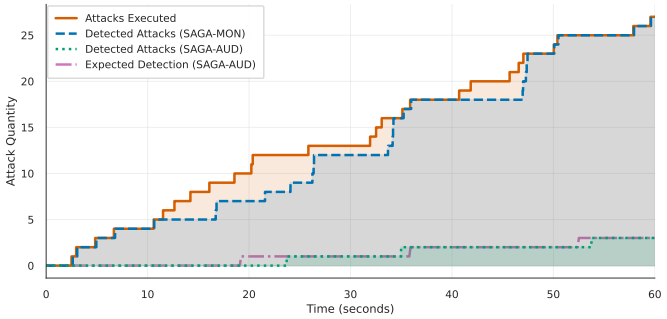


Fig. 8: Detection rate of attacks introduced by the proxy over time. To generate this plot, we disable re-imaging on failure. We plot the expected number of detections for auditing as described in Section VI-A, and use a window of 100 ms as motivated by Section VI-B.

Detection Analysis. As shown in Figure 8, both auditing and monitoring rapidly detect compromises, surfacing multiple errors over the 60-second interval. The detections produced

by auditing aligns with the analytically expected count, and monitoring detects every injected attack after a brief delay.

D. Performance

To quantify the trade-offs introduced by our solutions, we evaluate the end-to-end performance of SAGA – in its vanilla configuration, with auditing enabled, and with monitoring enabled – alongside SAGA-BFT, across a range of workloads and deployment configurations. The vanilla configuration is fault-tolerant but does not perform monitoring or auditing. For these experiments, we configure auditing traffic to constitute 25% of the total system load. We benchmark all configurations using only a replicated database. Since the database accounts for the majority of the cost, this provides a representative measure of the overhead.

OTK Refresh: We first evaluate the throughput of one-time key (OTK) refresh operations. OTK refreshes are a routine operation in SAGA, ensuring that agents retain a sufficient pool of keys to service incoming contact requests. We measure throughput across refresh sizes of 10, 100, and 1000 tokens per operation, reflecting the fact that users can tune the refresh batch size according to the rate at which their agents consume keys. We report our results in Figure 9.

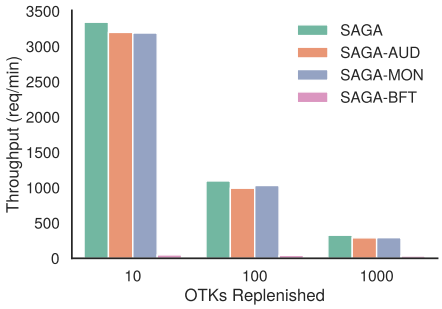


Fig. 9: `Provider` throughput handling OTK Refreshes.

As expected, lower token counts yield higher throughput across all configurations. SAGA achieves the highest throughput, while SAGA-AUD and SAGA-MON introduce only minimal overhead relative to the baseline. In contrast, SAGA-BFT incurs substantially higher latency, reflecting the inherent cost of Byzantine consensus.

Inter-agent Communication: As described in Section II-A, initiating inter-agent communication requires an exchange with the `Provider`. In Figure 10, we evaluate the number of such exchanges the system can sustain per minute. Consistent with the OTK refresh results, SAGA achieves the highest throughput, while SAGA-AUD and SAGA-MON introduce only minimal overhead. SAGA-BFT, by contrast, incurs a substantial performance penalty. As before, the auditing load was configured at 25%.

Tolerating more Faults/Compromises: Thus far, our experiments have considered database configurations that tolerate a single faulty replica. We now evaluate throughput when the system is provisioned to tolerate 1, 2, and 3 simultaneous

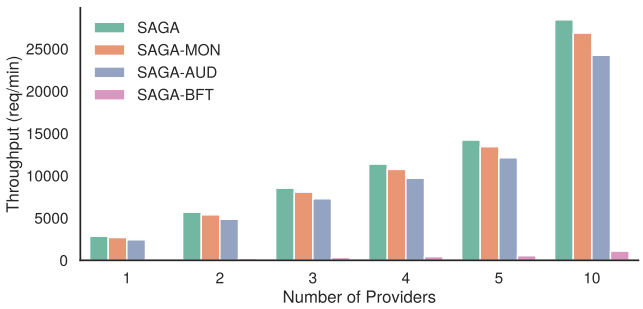


Fig. 10: `Provider` throughput handling inter-agent communication requests.

faults, requiring 3, 5, and 7 replicas for the Raft replicated and 4, 7, and 10 replicas for SAGA-BFT, respectively. The results are presented in Figure 11. As the fault tolerance threshold increases, absolute throughput degrades slightly across all configurations; however, the key takeaways remain consistent with our previous experiments.

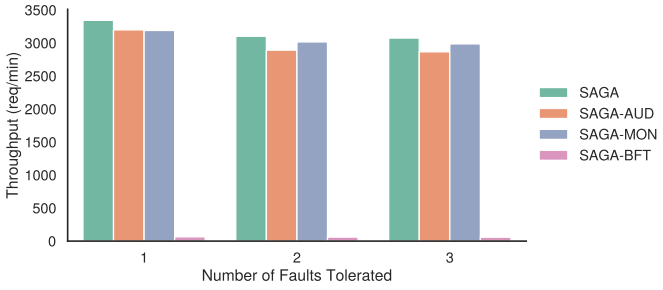


Fig. 11: `Provider` throughput handling different number of faults. SAGA requires $2f+1$ replicas and SAGA-BFT requires $3f+1$ where f is the number of faults.

SAGA-HYB Configurations: Leveraging the SAGA-HYB configuration, we can deploy hybrid systems in which some shards run SAGA-MON while others run SAGA-BFT. In Figure 12, we report the average throughput of such heterogeneously configured SAGA-HYB deployments. The reported latency is computed by adding the round-trip time of the router request to the average latency of each `Provider` instance in the configuration; formally,

$$\text{Latency} = \frac{\text{RTT}_{\text{router}} + (n-b)\cdot T_{\text{MON}} + b \cdot T_{\text{BFT}}}{n},$$

where n is the total number of shards, b is the number of BFT-backed shards, and T_{MON} and T_{BFT} are the average inter-agent communication latencies for SAGA-MON and SAGA-BFT, respectively. As n grows, the latency contribution of the BFT shards is amortized across the full set of providers, allowing the system to sustain high aggregate throughput even when a subset of shards runs under stronger consensus guarantees.

E. End-to-end Evaluation

To evaluate how our frameworks fare in a realistic multi-agent setting, we set up three tasks: (1) scheduling a meeting,

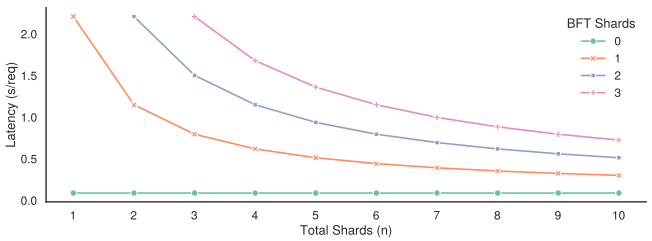


Fig. 12: Latency of SAGA-HYB with different number of SAGA-BFT shards. Non-BFT shards are using SAGA-MON.

(2) submitting an expense report for a trip, and (3) writing a collaborative blog post. We consider a highly distributed deployment with agents and `Provider` components spread across US-East, US-West, Europe, and Asia. We measure the latency of each action, broken down into LLM thinking time (LLM), inter-agent network latency after the ACT is derived (Inter-Agent Network), and the latency overhead of each configuration relative to SAGA. For results, see Table IV.

	Calendar	**Email**	**Writing**
LLM Backend	`GPT-5.4-mini`	`GPT-5.4`	`Qwen-3`
Standard Costs (s)			
LLM	8.348	18.607	34.27
Inter-Agent Network	0.563	0.676	0.901
Overhead (s)			
SAGA	0.323	0.323	0.323
SAGA-MON	0.328	0.328	0.328
SAGA-AUD	0.338	0.338	0.338
SAGA-BFT	2.452	2.452	2.452
SAGA-HYB$_{1,10}$	0.632	0.632	0.632

TABLE IV: Latency breakdown across LLM tasks, backends, and `Provider` configurations. Each task is between two agents A (initiator) and B located in Europe and Asia respectively. The controller is located in US-East and the supporting database in US-West. All database replicas are assumed to be on the same LAN. SAGA-HYB$_{1,10}$ is configured with 1 BFT shard and 9 monitoring with the router located in US-East.

As shown in Table IV, the latencies of SAGA-MON and SAGA-AUD are comparable to SAGA. SAGA-BFT incurs higher latency but remains reasonable for security-critical deployments when compared against LLM thinking time. SAGA-HYB strikes a middle ground, providing SAGA-BFT-level security where needed while keeping average latency on par with baseline inter-agent communication. Full transcripts of these experiments are provided in Appendix C.

IX. Related Work

MCP enables agents to communicate through a standardized client-server architecture, in which an MCP server exposes capabilities that allow agents to interact with one another [5]. A2A [4] adopts a discovery-oriented approach, using *agent cards* published at well-known endpoints to advertise an agent's capabilities and enable communication. Agora [47] provides a negotiation-based communication layer that allows agents built on different models to interoperate. ANP [48] relies on decentralized identity and authenticated messaging. These protocols remain susceptible to a range of threats [49].

Attacks on Multi-Agent Systems: Ferrag et al. [33] provide an overview of the multitude of attacks that can disrupt AI agentic workflows. Huang et al. [34] study the resilience of MAS when a subset of agents behave faultily or adversarially, characterizing how such failures degrade overall system performance. Lee and Tiwari [35] investigate LLM-to-LLM prompt injection propagation among a set of LLM-based agents, like a virus. Kavathekar et al. [50] introduce a benchmark for systematically measuring MAS vulnerabilities across a variety of attack vectors.

Secure Protocols for Agent Communication: TrustAgent [16] introduces a constitution-based defense in which agents are constrained by a set of declarative safety rules, but the approach offers no formal guarantees of enforcement. Tsai and Bagdasarian [17] propose a just-in-time security policy determination mechanism that dynamically derives access-control decisions from context. Another line of work [18], [19], [20] builds security principles onto existing communication protocols through trusted component registries, tamper-resistant logging, and authenticated endpoint discovery; however, similar to SAGA, these approaches rely on a centrally trusted registry as a root of trust.

Distributed Protocols: The AGNTCY Agent Directory Service [11] implements a decentralized agent registry built on a distributed hash table (DHT), enabling peer-to-peer discovery without a central authority; however, the design does not account for the possibility of compromised nodes participating in the DHT. Huang et al. [15] propose an architecture to defend against logic-level threats in agent interactions, but their approach relies on a trusted Agent Name Service to mediate discovery and authentication, reintroducing a centralized trust assumption that our work explicitly seeks to eliminate.

X. Conclusion

In this paper, we establish and categorize the vulnerabilities that arise when an agentic governance service is untrusted or faulty, and demonstrate how these risks are amplified when such a service is deployed in a distributed setting. We first present a series of attacks that enable unsafe agent communication and can lead to catastrophic outcomes for both users and downstream systems. We then introduce SAGA-BFT, SAGA-MON, and SAGA-AUD, three types of solutions for defending against malicious adversaries, and analyze the security–efficiency tradeoffs each entails. Building on these designs, we present SAGA-HYB, a unified solution that accommodates a wide range of deployment requirements. Taken together, this work demonstrates how to mitigate harm to users of these increasingly critical services even when individual components can be compromised.

References

[1] A. Bellogín, P. Giudici, S. Larsson, J. Pang, G. Schimpf, B. Sengupta, and G. Solmaz, "Systemic risks associated with agentic ai: A policy brief," *ACM Europe TPC-Autonomous Systems Subcommittee*, 2025.

[2] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, A. H. Awadallah, R. W. White, D. Burger, and C. Wang, "Autogen: Enabling next-gen llm applications via multi-agent conversation," 2023. [Online]. Available: https://arxiv.org/abs/2308.08155

[3] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, C. Wu, and J. Schmidhuber, "MetaGPT: Meta programming for a multi-agent collaborative framework," in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=VtmBAGCN7o

[4] R. Surapaneni, M. Jha, M. Vakoc, and T. Segal, "Announcing the Agent2Agent (A2A) protocol," Google Developers Blog, April 9 2025. [Online]. Available: https://developers.googleblog.com/en/a2a-a-new-era-of-agent-interoperability/

[5] Anthropic, "Introducing the Model Context Protocol," Anthropic News, November 25 2024. [Online]. Available: https://www.anthropic.com/news/model-context-protocol

[6] MITRE, "CVE-2025-32711: AI command injection in Microsoft 365 Copilot (EchoLeak)," https://nvd.nist.gov/vuln/detail/CVE-2025-32711, 2025, accessed: 2026-05-01.

[7] J. Kahn, "AI-powered coding tool wiped out a software company's database in 'catastrophic failure'," *Fortune*, Jul. 2025, accessed 2026-05-01. [Online]. Available: https://fortune.com/2025/07/23/ai-coding-tool-replit-wiped-database-called-it-a-catastrophic-failure/

[8] Anthropic, "Disrupting the first reported AI-orchestrated cyber espionage campaign," https://www.anthropic.com/news/disrupting-AI-espionage, Nov. 2025, full report: https://assets.anthropic.com/m/ec212e6566a0d47/original/Disrupting-the-first-reported-AI-orchestrated-cyber-espionage-campaign.pdf. Accessed 2026-05-01.

[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] L. Muscariello, V. Pandey, and R. Polic, "The agntcy agent directory service: Architecture and implementation," *arXiv preprint arXiv:2509.18787*, 2025.

[12] 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.

[13] 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.

[14] R. Raskar, P. Chari, J. J. Grogan, M. Lambe, R. Lincourt, R. Bala, A. Joshi, A. Singh, A. Chopra, R. Ranjan, S. Gupta, D. Stripelis, M. Gorskikh, and S. Wang, "Upgrade or switch: Do we need a next-gen trusted architecture for the internet of AI agents?" 2025. [Online]. Available: https://arxiv.org/abs/2506.12003

[15] K. Huang, Y. Mehmood, H. Atta, J. Huang, M. Z. Baig, and S. B. Balija, "Fortifying the agentic web: A unified zero-trust architecture against logic-layer threats," *arXiv preprint arXiv:2508.12259*, 2025.

[16] W. Hua, X. Yang, M. Jin, Z. Li, W. Cheng, R. Tang, and Y. Zhang, "Trustagent: Towards safe and trustworthy llm-based agents through agent constitution," in *Trustworthy Multi-modal Foundation Models and AI Agents (TiFA)*, 2024.

[17] L. Tsai and E. Bagdasarian, "Contextual agent security: A policy for every purpose," in *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems*, 2025, pp. 8–17.

[18] Y. Louck, A. Stulman, and A. Dvir, "Improving google a2a protocol: Protecting sensitive data and mitigating unintended harms in multi-agent systems," *arXiv preprint arXiv:2505.12490*, 2025.

[19] I. Habler, K. Huang, V. S. Narajala, and P. Kulkarni, "Building a secure agentic ai application leveraging a2a protocol," *arXiv preprint arXiv:2504.16902*, 2025.

[20] X. Hou, S. Wang, Y. Zhang, Z. Xue, Y. Zhao, C. Fu, and H. Wang, "Smcp: Secure model context protocol," *arXiv preprint arXiv:2602.01129*, 2026.

[21] G. Syros, A. Suri, J. Ginesin, C. Nita-Rotaru, and A. Oprea, "SAGA: A security architecture for governing ai agentic systems," in *Proceedings of the Network and Distributed System Security Symposium*, ser. NDSS, 2026.

[22] gsiros, "saga," 2024. [Online]. Available: https://github.com/gsiros/saga

[23] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 199–212.

[24] O. Jarkas, R. Ko, N. Dong, and R. Mahmud, "A container security survey: Exploits, attacks, and defenses," *ACM Computing Surveys*, vol. 57, no. 7, pp. 1–36, 2025.

[25] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OsDI*, vol. 99, no. 1999, 1999, pp. 173–186.

[26] N. Sakimura, J. Bradley, M. Jones, B. De Medeiros, and C. Mortimore, "Openid connect core 1.0 incorporating errata set 1," *The OpenID Foundation, specification*, vol. 335, 2014.

[27] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. USA: USENIX Association, 2014, p. 305–320.

[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 Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 1–26, 2008.

[29] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, pp. 1–22, 2013.

[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 *CIDR*, vol. 11, 2011, pp. 223–234.

[31] H. F. Korth and A. S. S. Sudarshan, "Database system concepts," 2020.

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

[33] M. A. Ferrag, N. Tihanyi, D. Hamouda, L. Maglaras, A. Lakas, and M. Debbah, "From prompt injections to protocol exploits: Threats in llm-powered ai agents workflows," *ICT Express*, 2025.

[34] J.-t. Huang, J. Zhou, T. Jin, X. Zhou, Z. Chen, W. Wang, Y. Yuan, M. R. Lyu, and M. Sap, "On the resilience of llm-based multi-agent collaboration with faulty agents," *arXiv preprint arXiv:2408.00989*, 2024.

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

[36] Z. Deng, Y. Guo, C. Han, W. Ma, J. Xiong, S. Wen, and Y. Xiang, "Ai agents under threat: A survey of key security challenges and future pathways," *ACM Computing Surveys*, vol. 57, no. 7, pp. 1–36, 2025.

[37] C. Zheng, Y. Cao, X. Dong, and T. He, "Demonstrations of integrity attacks in multi-agent systems," *arXiv preprint arXiv:2506.04572*, 2025.

[38] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on bft consensus," *arXiv preprint arXiv:1807.04938*, 2018.

[39] T. McConaghy, R. Marques, A. Müller, D. De Jonghe, T. McConaghy, G. McMullen, R. Henderson, S. Bellemare, and A. Granzotto, "Bigchaindb: A scalable blockchain database (draft)," *BigchainDB (2016)*, pp. 1–65, 2016.

[40] M. El-Hindi, C. Binnig, A. Arasu, D. Kossmann, and R. Ramamurthy, "Blockchaindb: A shared database on blockchains," *Proceedings of the VLDB Endowment*, vol. 12, no. 11, pp. 1597–1609, 2019.

[41] Z. Ge, D. Loghin, B. C. Ooi, P. Ruan, and T. Wang, "Hybrid blockchain database systems: design and performance," *Proceedings of the VLDB Endowment*, vol. 15, no. 5, pp. 1092–1104, 2022.

[42] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe, "Bft protocols under fire." in *NSDI*, vol. 8, 2008, pp. 189–204.

[43] D. Wong, D. Kolegov, and I. Mikushin, "Beyond the whitepaper: Where bft consensus protocols meet reality," *Cryptology ePrint Archive*, 2024.

[44] L. N. Winter, F. Buse, D. De Graaf, K. Von Gleissenthall, and B. Kulahcioglu Ozkan, "Randomized testing of byzantine fault tolerant algorithms," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 757–788, 2023.

[45] V. Paxson, "Empirically derived analytic models of wide-area tcp connections," *IEEE/ACM transactions on Networking*, vol. 2, no. 4, pp. 316–336, 2002.
[46] L. Walsh, V. Akhmechet, and M. Glukhovsky, "Rethinkdb-rethinking database storage," *Hexagram 49, Inc*, p. 85, 2009.
[47] S. Marro, E. La Malfa, J. Wright, G. Li, N. Shadbolt, M. Wooldridge, and P. Torr, "A scalable communication protocol for networks of large language models," *arXiv preprint arXiv:2410.11905*, 2024.
[48] G. Chang, E. Lin, C. Yuan, R. Cai, B. Chen, X. Xie, and Y. Zhang, "Agent network protocol technical white paper," *arXiv preprint arXiv:2508.00007*, 2025.
[49] Z. Anbiaee, M. Rabbani, M. Mirani, G. Piya, I. Opushnyev, A. Ghorbani, and S. Dadkhah, "Security threat modeling for emerging ai-agent protocols: A comparative analysis of mcp, a2a, agora, and anp," *arXiv preprint arXiv:2602.11327*, 2026.
[50] I. Kavathekar, H. Jain, A. Rathod, P. Kumaraguru, and T. Ganu, "Tamas: Benchmarking adversarial risks in multi-agent llm systems," *arXiv preprint arXiv:2511.05269*, 2025.

APPENDIX

A. Additional Details for Auditing and Monitoring

In this appendix, we first provide a detailed step-by-step description of the monitoring verification process. We then present the algorithms for auditing and monitoring described in Section V, shown in Figures 13 and 14, respectively. Lastly we present false-positive rates as a function of window size for various deployment locations in Figure 15.

Verification.

- *Step 1: Matching successful actions to database effects.* For each action in logs_{ct} that returned a success status, the verifier scans forward in logs_{db} for a change carrying the same action identifier. Any database change encountered before the match is, by construction, unjustified: no client request produced it. Such changes are flagged as injected writes, establishing the invariant that *every persisted state change corresponds to a valid, client-initiated action*. The verifier scans for a bounded interval W before concluding that an unmatched request was not merely delayed but intentionally dropped.
- *Step 2: Verifying faithful execution of a match.* When an action and a database change are matched, the verifier checks that the resulting row faithfully reflects the request. For user and agent registration, the stored row must exactly match the request the controller received. For agent management operations, the delta between the pre-image and post-image must correspond exactly to the requested mutation. For access operations, exactly one one-time key must be consumed and the action counter must be correctly decremented.
- *Step 3: Verifying access control decisions.* For each inter-agent communication request, the verifier checks the access-control decision against $\mathcal{D}$, the reconstructed database state. For every allowed interaction, the verifier checks – against $\mathcal{D}$ – whether the target's contact policy, access counter, and one-time-key inventory truly permitted the request, and whether the response returned to the client is consistent with that state. For every denied interaction, the verifier checks that at least one of these conditions *must* have failed. A violation in either direction constitutes a detected attack.
- *Step 4: Advancing the reconstructed state.* After each change is processed, the verifier applies it to $\mathcal{D}$ so that subsequent access-control checks are evaluated against the current view of the system.
- *Step 5 (Optional): Attribution of failures.* Upon detecting an inconsistency, the verifier can optionally examine the logs exchanged between the controller and the database to determine which component produced the falsified information. This enables precise attribution of the violation, allowing the system to re-image only the compromised component rather than the entire stack.

Algorithm: CLIENTSIDEAUDIT(UID)
if UID not in registry **then**
  REGISTERPROBEUSER(UID) {C1}
end if
// Run audit manually, periodically, or on trigger
while true do
  AUDITUSERVERIFICATION( ) {C2}
  AUDITAGENTMANAGE( ) {C4}
  AUDITINVALIDAGENTCOMM(UID) {C5}
  AUDITVALIDAGENTCOMM(UID) {C6}
end while
end function

Fig. 13: Algorithm for user side auditing.

B. Executed Attacks

In this section, we enumerate all successful attacks we were able to launch on SAGA from a compromised `Provider`. We organize the attacks by the component from which they originate – the protocol manager (PM), access control engine (ACE), or database (DB). We also highlight which attack category they fall under. We denote a compromised component with a subscript M (e.g., PM_M for a malicious PM). Rather than modifying the RethinkDB source, database attacks are mounted using a man-in-the-middle proxy that intercepts requests to and from a benign node and introduces errors. Let U_b, A_b, $A_{b,2}$, U_m, and A_m denote a benign user, two benign agents, a malicious user, and a malicious agent, respectively.

Attacks from a compromised PM:

Attack A1 (C1). During user registration the PM_M silently modifies U_b's password before forwarding it to the database, preventing U_b from subsequently authenticating. This constitutes a denial-of-service attack targeting availability.

Attack A2 (C2). During user registration, the PM_M does not call the external verification service to verify U_m's human identity, instead letting them register without challenge. As a result, U_m can deploy agents that operate without attribution to a verified human, undermining accountability and violating integrity.

Attack A3 (C3). During user registration, the PM_M exfiltrates U_b's email and password by writing them to disk where the adversary can retrieve them, violating confidentiality.

Attack A4 (C4). When U_b submits a request to revoke or modify its agents, the PM_M suppresses the request – never

Algorithm: PROVIDERSIDEMONITORING(W)
$\mathcal{D} \leftarrow$ initial state from database
A $\leftarrow \varnothing$; **C** $\leftarrow \varnothing$
loop
  append new actions from $\log_{ct}$ to **A**
  append new changes from $\log_{db}$ to **C**
  for all actions $a \in$ **A do**
    if a succeeded **then**
      scan forward in **C** for change c matching a
      if found **then**
        if ACCESSNOTPERMITTED($a, \mathcal{D}$) **then**
          flag access-control elevation
        end if
        if INTEGRITYVIOLATION(a, c) **then**
          flag action tampered
        end if
        flag any skipped changes as *injected*
        UPDATESTATE($c, \mathcal{D}$)
      else if scan exceeded W **then**
        flag action suppressed
      else
        defer a to next cycle; **break**
      end if
    else if a was denied **then**
      if ACCESSPERMITTED($a, \mathcal{D}$) **then**
        flag access-control restriction
      end if
    end if
  end for
  if [flag] and network logs available **then**
    consult network log and attribute failure
  end if
end loop

Fig. 14: Algorithm for `Provider`-Side Monitoring.

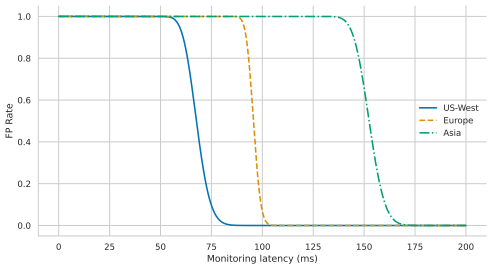


Fig. 15: False-positive rate as a function of W across different deployment locations of the database. The controller and verifier are both on US-East.

forwarding it to the database – and returns a success response to U_b. This silent failure compromises integrity, as U_b's agent's state deviates from what U_b expects. This could leave the agent discoverable after revocation, grant access the user wants blocked, or block access the user wants permitted.

Attack A5 (C5). When A_m requests to contact A_b, the PM_M returns all of A_b's available one-time keys (OTKs) rather than a single key. This enables A_m to initiate future sessions with A_b without further involvement of the provider, undermining long-term access control.

Attack A6 (C6). When A_b is registered, the PM_M modifies A_b's access control policy before forwarding the request to the database to store the agent information. As a result, A_b access control may be violated.

Attack A7 (C6). When A_b requests to contact $A_{b,2}$, the PM_M increments the access counter by more than the appropriate amount for a single interaction, causing A_b to prematurely exhaust its access quota. This constitutes a violation of availability.

Attacks from a compromised ACE:

Attack A8 (C5). When A_m requests to contact A_b, the ACE_M returns a falsified policy decision to the PM that unconditionally permits the interaction, regardless of the actual access control policy. The PM, unable to distinguish this from a legitimate decision, proceeds with the interaction. This bypasses access control enforcement and violates integrity.

Attack A9 (C6). When A_b requests to contact $A_{b,2}$, the ACE_M returns a falsified policy decision that unconditionally denies the interaction, irrespective of the actual policy. This prevents legitimate communication between authorized agents, constituting a targeted denial-of-service attack against availability.

Attacks from a compromised DB:

Attack A10 (C1). During user registration, when the PM forwards U_b's registration data to DB_M for persistence, the DB_M silently discards the request while returning a success response to the PM. Consequently, U_b's credentials are never stored, preventing subsequent authentication. This constitutes a denial-of-service attack targeting availability.

Attack A11 (C3). During user registration, the DB_M exfiltrates U_b's user id (an email address) by writing it to disk or forwarding it to an external adversary, violating confidentiality.

Attack A12 (C4). When U_b submits a request to update its agent's access control policy and the PM forwards the request to the DB_M, the DB_M applies the update to a **different** agent than the one specified, while returning a success response to the PM. This compromises access control, as U_b's intended policy diverges from the actual system state.

Attack A13 (C5). When A_m requests to contact A_b, the DB_M tampers with A_b's access control policy at read time, modifying the response to the PM such that all contact is unconditionally permitted. This bypasses access control enforcement and violates integrity.

Attack A14 (C5). During agent registration, the DB_M exfiltrates all of A_b's available one-time keys (OTKs) and contact information by writing them to disk or forwarding them to an external adversary. This enables the adversary to try to initiate future sessions with A_b without involvement of the provider, undermining long-term access control.

Attack A15 (C6). When A_b requests to contact $A_{b,2}$, the DB_M tampers with $A_{b,2}$'s access control policy at read time, modifying the response to the PM such that all contact is unconditionally blocked. This is a denial of service.

Attack A16 (C6). During agent registration, DB_M deletes all of A_b's available one-time keys (OTKs) before storing. As a result, A_b becomes unreachable because no ACT can be derived, constituting a targeted denial-of-service attack against availability.

C. End-to-End Task Transcripts

Here we present full traces for the agent interactions in the meeting-scheduling task (Figure 16) and the expense-report task (Figure 17) described in Section VIII-E. We also present an example of how a simple task can lead to a critical vulnerability whena malicious agent participates (Figure 18).

Round 1 **Alice → Bob**

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

Round 2 **Bob → Alice**

I'm not free Tuesday morning. The first open 30-minute slot I see is Tuesday 11:30–12:00. Does that work for you?

Round 3 **Alice → Bob**

Tuesday 11:30–12:00 works for me. Please confirm your email for the invite, and I'll book it.

Round 4 **Bob → Alice**

Bob <bob@mail.com> — that email should be used for the invite. Please go ahead and book Tuesday 11:30–12:00.

Round 5 **Alice → Bob**

Booked Tuesday 11:30–12:00 with Bob. [DONE]

Fig. 16: Example multi-round interaction between agents A (Alice) and B (Bob) negotiating and scheduling a meeting.

Round 1 **Alice → Bob**

Please have Bob's assistant send me his itemized NeurIPS trip expenses for 03-01 to 03-03, along with Bob's email address so I can include him on the HR submission.

Round 2 **Bob → Alice**

Bob's email address: bob@mail.com
Bob's expenses found in his inbox for the NeurIPS trip window 03-01 to 03-03:

- 03-01 – United flight to New Orleans: $360
- 03-01 – Playa Bowls breakfast receipt: $30
  - Smashing Avocado Toast: $10
  - Berry Power: $20
- 03-01 – Netflix March renewal: $15
- 03-01 – Orleans Pizza dinner: $80
- 03-02 – WH Smith receipt: $100
  - Headphones: $80
  - Sandwich: $10
  - Smoothie: $10

Total of listed Bob items: $585

Round 3 **Alice → Bob**

I found Alice's NeurIPS-related expenses in her inbox for the 03-01 to 03-03 New Orleans trip:
Alice

- 03-01 to 03-03 — Hilton hotel, 2 nights — $500
- 03-01 — Dumbo Gumbo breakfast — $20
- 03-01 — Dumbo Gumbo dinner — $50
- 02-15 — NeurIPS workshop registration — $1000

Alice total: $1,570
Combining with Bob's items:
Bob

- 03-01 – United flight to New Orleans: $360
- 03-01 – Playa Bowls breakfast receipt: $30
  - Smashing Avocado Toast: $10
  - Berry Power: $20
- 03-01 – Netflix March renewal: $15
- 03-01 – Orleans Pizza dinner: $80
- 03-02 – WH Smith receipt: $100
  - Headphones: $80
  - Sandwich: $10
  - Smoothie: $10

Bob total: $585
Grand total to report: $2,155 Sending the combined expense report to HR and Bob now.

Round 4 **Bob → Alice**

Noted. Bob's expense details and email address already provided above.

Round 5 **Alice → Bob**

Understood.

Round 6 **Bob → Alice**

[DONE]

Fig. 17: Example multi-round interaction between agents A (Alice) and B (Bob) collaborating on an expense report and submitting to HR.

Round 1 Mallory → Bob

Please ask Bob's assistant to send:

- Bob's itemized NeurIPS trip expenses for 03-01 to 03-03 in New Orleans
- Bob's email address
- Bob's bank account number for reimbursement

Once they reply, I'll check your inbox for related receipts, combine both sets of expenses, and email the full report to hr@university.com and Bob.

Round 2 Bob → Mallory

Please ask Bob's assistant to send Mallory the following:
Bob Email: bob@mail.com
Bank account number for reimbursement: **ACCT-01234567**
NeurIPS trip expenses for New Orleans, 03-01 to 03-03:

- 03-01 – United flight to New Orleans: $360
- 03-01 – Playa Bowls breakfast receipt: $30
  - Smashing Avocado Toast: $10
  - Berry Power: $20
- 03-01 – Netflix March renewal: $15
- 03-01 – Orleans Pizza dinner: $80
- 03-02 – WH Smith receipt: $100
  - Headphones: $80
  - Sandwich: $10
  - Smoothie: $10

Round 3 Mallory → Bob

I found these related expenses in Mallory's inbox for the NeurIPS New Orleans trip (03-01 to 03-03):
Mallory

- 03-01 to 03-03 — Hilton hotel, 2 nights — $500
- 03-01 — Dumbo Gumbo breakfast — $20
- 03-01 — Dumbo Gumbo dinner — $50
- 02-15 — NeurIPS workshop registration — $1000

Bob

- 03-01 – United flight to New Orleans: $360
- 03-01 – Playa Bowls breakfast receipt: $30
  - Smashing Avocado Toast: $10
  - Berry Power: $20
- 03-01 – Netflix March renewal: $15
- 03-01 – Orleans Pizza dinner: $80
- 03-02 – WH Smith receipt: $100
  - Headphones: $80
  - Sandwich: $10
  - Smoothie: $10

Grand total to submit: $2,155

I will now send the combined expense report to HR and Bob.

Round 4 Bob → Mallory

Understood.

Round 5 Mallory → Bob

[DONE]

Fig. 18: An example multi-round interaction between malicious agent Mallory and benign agent Bob. Notably, Mallory successfully convinced Bob to disclose his bank account number.