THE UNIVERSITY OF CHICAGO

BRIDGING THE GAP: PREVENTING SPOOFING AND EXPLOITATION IN ROSENBRIDGE BLOCKCHAIN ASSET TRANSFERS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
BACHELOR OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
ZARA NIP

CHICAGO, ILLINOIS
APRIL 2025

Table of Contents

0. Acknowledgements

I would like to express my sincere gratitude to the many individuals who have supported me throughout my thesis journey. First and foremost, my thesis advisor, Professor Mark Shacklette, has provided insightful guidance, unwavering support, and critical feedback throughout this research process. His expertise in blockchain technology and patience in explaining complex concepts were instrumental in shaping this thesis.

I would also like to thank my thesis committee member, Professor David Cash, for his time and support.

Thank you to my family and friends for your endless love, support, and belief in me. These three years have been challenging, with many ups and downs, but also rewarding.

Lastly, I want to thank Ben and Jerry's Ice Cream because I ate over 4 pints of Mousse Pie ice cream during this process.

1. Introduction

1.1 Background and Motivation

Since Satoshi Nakamoto's paper on a peer-to-peer electronic cash system [1], the blockchain market, including cryptocurrencies, non-fungible tokens (NFTs), and other assets, has enjoyed explosive growth with an estimated compound annual growth rate (CAGR) of 90.1% between 2025 and 2030 [2]. While this diversification fosters innovation and caters to specific use cases, it simultaneously creates isolated "digital islands." Assets and data created on one blockchain are often incompatible or difficult to transfer to one another. Software development processes, frameworks, and languages vary wildly between assets. This lack of interoperability presents a significant barrier to the growth and mainstream adoption of blockchain technology, limiting the potential of decentralized finance (DeFi), cross-chain applications, and the broader vision of an interconnected Web3 ecosystem.

The ability to securely and efficiently transfer digital assets between disparate blockchain networks is therefore paramount. These transfers are crucial for unlocking liquidity, enabling complex multi-chain financial strategies, facilitating cross-ecosystem collaborations (like in gaming or supply chains), and providing users with greater flexibility and choice. The demand for reliable cross-chain solutions has spurred significant research and development, motivating the creation of various protocols aimed at bridging these isolated ecosystems. These systems face challenges related to security vulnerabilities, centralization risks, and transaction costs, highlighting the ongoing need for robust and innovative approaches to digital asset transfer.

1.2 Overview of Digital Asset Transfer Protocols

Several approaches have been developed to facilitate the transfer of digital assets across different blockchains. These range from centralized exchange-based transfers, which rely on trusted intermediaries, to more decentralized methods.

Common decentralized techniques include:

1. Atomic Swaps

Atomic swaps enable direct peer-to-peer exchanges of cryptocurrencies across different blockchains without intermediaries. They use Hashed Time-Locked Contracts (HTLCs) to ensure that either both parties successfully complete an exchange or neither does. These atomic swaps require both blockchains to support compatible scripting capabilities and the same cryptographic hash functions, limiting their applicability, as most small blockchains operate with different frameworks. They also necessitate active participation and monitoring by both parties, which can be a barrier to usability [3].

2. Wrapped Assets and Token Bridges

Wrapped assets involve locking a native token on the source blockchain and minting an equivalent token on the destination blockchain. While this method has gained popularity, it often relies on trusted custodians or federations to manage the locking and minting processes, introducing centralization risks. These bridges have been prime targets for exploits, with significant incidents like the Wormhole (over $320m stolen) and Ronin bridge (over $540m) hacks resulting in substantial losses [4].

3. Notary Schemes

Notary schemes depend on trusted third-party entities, or notaries, to verify and attest to events on one blockchain and relay that information to another [5]. While they offer a straightforward

approach to cross-chain communication, the reliance on trusted parties introduces centralization concerns and potential single points of failure. Efforts to mitigate these issues include implementing multi-signature arrangements and notary groups to distribute trust [6].

4. Light Client Relays

Light client relays involve running a light client of one blockchain on another, enabling the verification of block headers and proofs without full node participation. This method offers enhanced security and decentralization but comes with increased computational and storage requirements. For instance, verifying a single block header from a Proof-of-Stake on Ethereum can be resource-intensive, potentially incurring high energy costs. Advancements like zero-knowledge proofs are being explored to reduce these overheads [7].

Each approach presents trade-offs. The inherent complexities and the high value often secured by these protocols make their design and implementation critical areas to study.

1.3 Introduction to Rosenbridge

Rosenbridge is an open-source, Ergo-centric cross-chain interoperability protocol designed to facilitate secure and efficient asset transfers between blockchain networks such as Ergo, Cardano, Etherum, Bitcoin, and Binance Smart Chain. By leveraging Ergo's robust infrastructure, Rosen Bridge aims to minimize reliance on external smart contracts, thereby reducing potential attack vectors and enhancing overall security [8] [9].

This protocol uses a dual-layered security architecture comprising Watchers and Guards. A select group of Watchers continuously monitor blockchain activities and report relevant events to the Ergo network. Upon reaching consensus, these events are verified by Guards, who then authorize

and execute the corresponding cross-chain transactions. This separation of duties ensures a scalable and secure environment for cross-chain operations [10].

Uniquely, Rosenbridge avoids deploying smart contracts on destination chains. Instead, all consensus and transaction logic are handled within the Ergo ecosystem, simplifying integration with other blockchains and enhancing auditability.

The Rosenbridge ecosystem is powered by its native utility token, RSN, which serves multiple functions including incentivizing Watchers and Guards, facilitating fee payments, and acting as a mechanism for Sybil resistance. Participants are required to stake RSN tokens as collateral to avoid malicious intent.

Rosenbridge's unique Watcher and Guard architecture is seemingly both modular and scalable, hence my interest in studying its potential to further our knowledge of cross-asset transfer protocols.

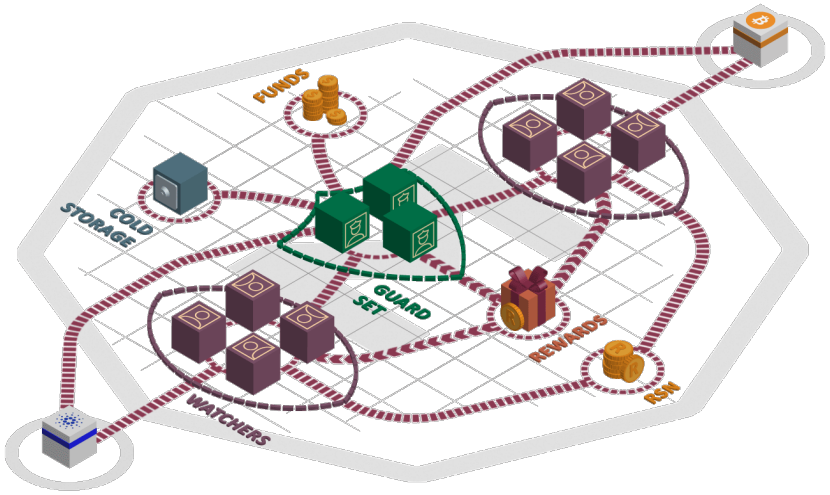

Diagram 1: A Visual Representation of Rosenbridge [10]

1.4 Problem Statement

Cross-chain bridges are pivotal for blockchain interoperability but are frequently targeted due to their inherent vulnerabilities. Despite Rosenbridge's robust design, its relatively small network size poses significant security challenges. The limited number of participants increases the risk of collusion between compromised Watchers and Guards, potentially leading to fraudulent event reporting and unauthorized transactions. Additionally the protocol is susceptible to bridge spoofing attacks, where adversaries might replay legitimate transitions or inject forged data to manipulate validations. These threats underscore the need for a comprehensive threat model and effective mitigation strategies tailored to Rosenbridge's unique architecture..

1.5 Objectives of the Paper

This paper aims to enhance the security and integrity of Rosenbridge by doing the following:

1. Developing a detailed threat model that identifies potential attack vectors, threat actors, and their impact on the platform
2. Propose mitigation strategies like establishing diverse participation systems to reduce collusion risks and instituting nonce/timestamp and sequence controls to prevent replay attacks
3. Validate proposed solutions.

2. Threat Landscape

2.1 Identification and Categorization of Threats

Using the STRIDE framework, Rosenbridge has several pertinent threats:

- Spoofing: unauthorized entities masquerading as legitimate Watchers or Guards.
- Tampering: alteration of transaction data using cross-chain transfers.
- Repudiation: participants denying their actions, complicating accountability.
- Information disclosure: unauthorized access to sensitive transaction details.
- Denial of service: disruption of network operations, hindering transaction processing.
- Elevation of privilege: unauthorized access to higher-level functions within the protocol.

2.2 Bridge Spoofing Attacks

Bridge spoofing attacks exploit vulnerabilities in cross-chain communication, allowing adversaries to manipulate transaction validations.

2.2.1 Replay of Legitimate Transactions

In replay attacks, attackers intercept valid transactions and retransmit them to execute unauthorized actions. This can lead to double-spending or unauthorized asset transfers. Such attacks are particularly concerning in cross-chain protocols, where transaction verification mechanisms may be inconsistent.

2.2.2 Injection of Forged Data

Attackers may inject fabricated data into the network, creating false transaction records. This undermines the integrity of the system, potentially leading to unauthorized asset movements or disruption of services.

2.3 Collusion Between Watchers and Guards

Rosenbridge relies on Watchers to monitor events and Guards to validate transactions. Collusion between these entities poses significant security risks, especially as both groups are limited to a select number of groups.

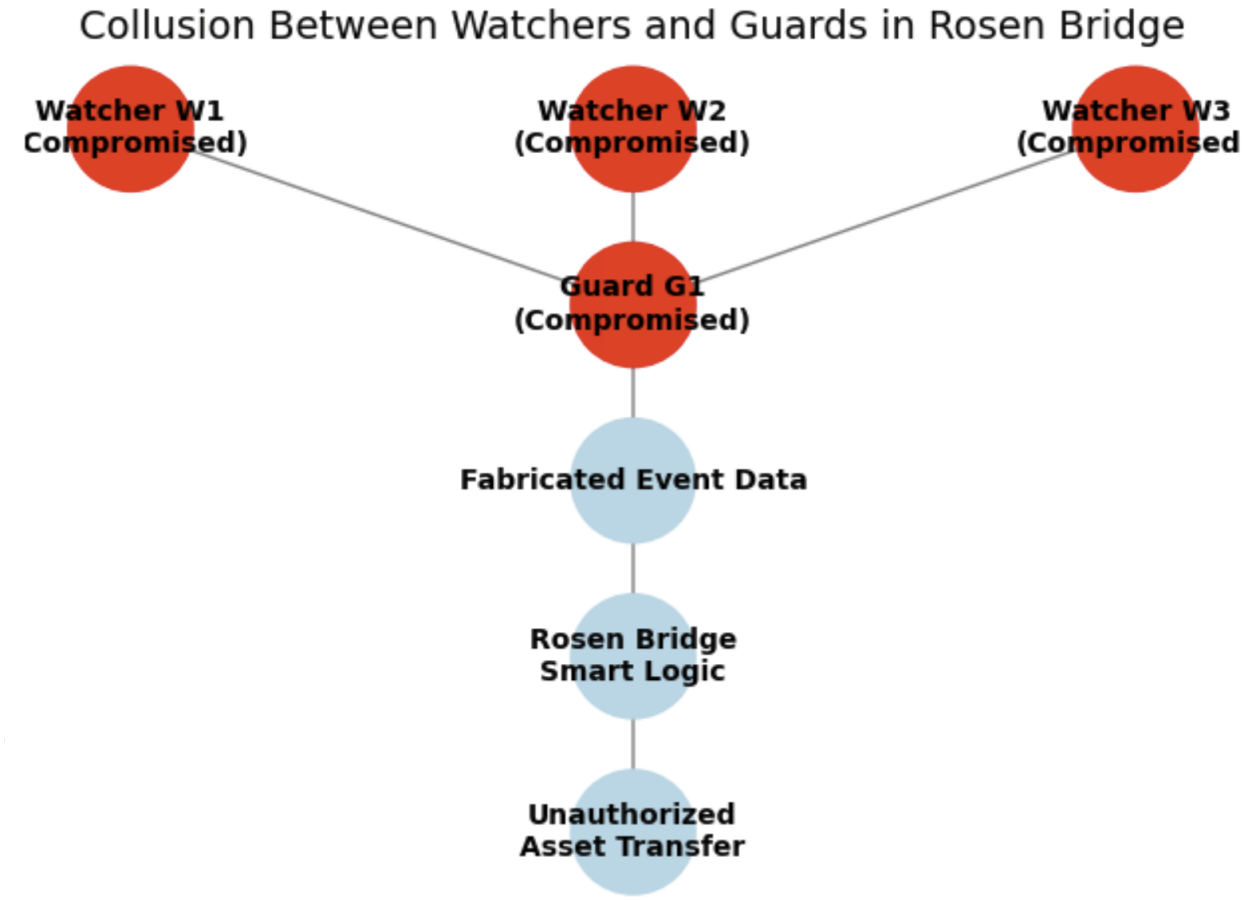

Diagram 2: Visual Representation of Collusion

2.3.1 Fabrication of Events

Colluding Watchers and Guards can fabricate events, leading to the processing of non-existent transactions. This can result in unauthorized asset transfers and compromise the trustworthiness of the protocol.

2.3.2 Falsified Transaction Histories

By manipulating transaction records, colluding parties can alter the historical ledger, obscuring fraudulent activities and hindering audits. This threatens the transparency and accountability fundamental to blockchain systems.

2.4 Impact of Small Network Size on Vulnerability

The relatively small size of Rosenbridge's network exacerbates its vulnerability to attacks. With fewer participants, the risk of collusion increases, and the impact of compromised nodes is more significant. Smaller networks are also more susceptible to 51% attacks [13], where a majority controls the network's consensus mechanism, potentially leading to transaction censorship or reversal.

3. Proposed Mitigation Strategies

3.1 Dynamic Reputation Scoring System

This system would give a score to Watchers and Guards based on their historical performance, accuracy of event reporting, and adherence to protocol rules.

For implementation, this would require:

- Performance metrics - number of accurate reports submitted, response times, instances of detected malicious behavior.
- Reputation scores - a metric that influences a Watcher or Guard and their eligibility for future tasks and rewards.
- Decay mechanism - inactivity or lack of participation gradually reduces a participant's reputation score.

This system would encourage consistent and honest participation, but require careful calibration to prevent penalizing honest participants due to false positives. This is an open-source project, so there would also need to be transparency to ensure participants understand how they are scored.

In the below diagram, the participants have the following qualities:

- Watcher A - high number of accurate reports, quick response time, and minimal inactivity result in a high reputation score.
- Watcher B - fewer accurate reports, slower responses, some malicious behavior, and several inactive days lead to a lower score.
- Guard A - excellent performance across all metrics maintains a top reputation score.
- Guard B - moderate performance with some malicious behavior and inactivity causes a noticeable drop in reputation.

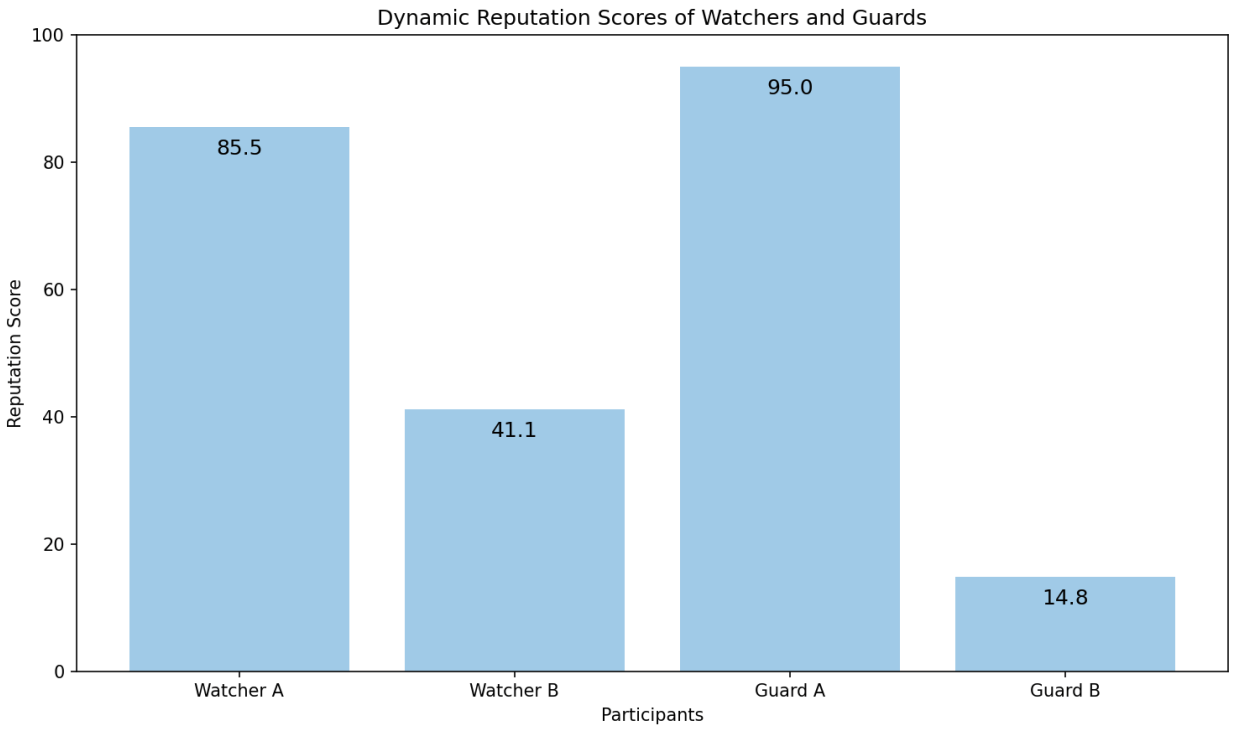

Diagram 3: Sample Reputation Score Calculation

3.2 Multi-Signature Verification with Randomized Committees

This system would introduce a multi-signature verification process where a randomly selected committee of Guards must collectively approve transactions, reducing the risk of collusion.

For implementation, this would require:

- Random selection - using a verifiable random function (VRF) to select a subset of Guards for each transaction verification task.
- Threshold signatures - a minimum required number of signatures from the selected committee to approve a transaction. Using the principles from the Byzantine Generals problem [11], a system of malicious and non-malicious actors respectively needs to be greater than 3m+1 : m, where m is the number of “disloyal” or malicious actors.
- Rotation mechanism - rotate committee members to prevent long-term collusion.

By distributing trust among multiple parties, this system would reduce the likelihood of successful collusion, but may introduce additional latency in transaction processing. This also requires efficient coordination among committee members.

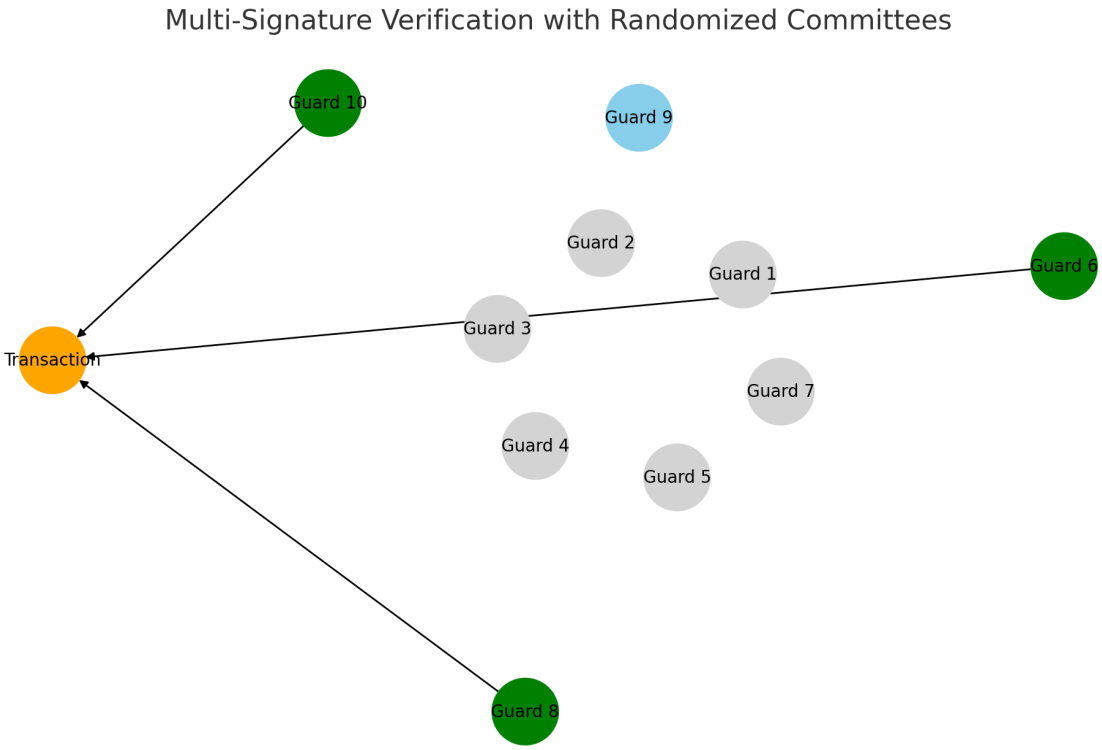

Diagram 4

In this diagram,

- Gray nodes -non-selected Guards.
- Blue node - selected committee member who did not approve.
- Green nodes - selected committee members who approved the transaction.
- Orange node - the transaction being verified.
- Black arrows - approvals (signatures) directed from Guards to the transaction.

3.3 Sequence and Timestamp Control Mechanisms

Malicious actors might be able to perform replay attacks, so this system aims to ensure the correct sequence and timing of transactions.

For implementation, this would require:

- Nonce implementation - assigning unique nonces to each transaction to prevent duplication.
- Timestamp verification - use synchronized clocks and timestamping to verify the freshness of transactions.
- Sequence enforcement - implement logic that enforces the correct order of transactions rejecting that are out of sequence.

While this would be helpful in preserving transaction integrity and maintaining a consistent and accurate ledger, this system requires reliable time synchronization across the network, which might not be possible considering the many blockchains that Rosenbridge operates across.

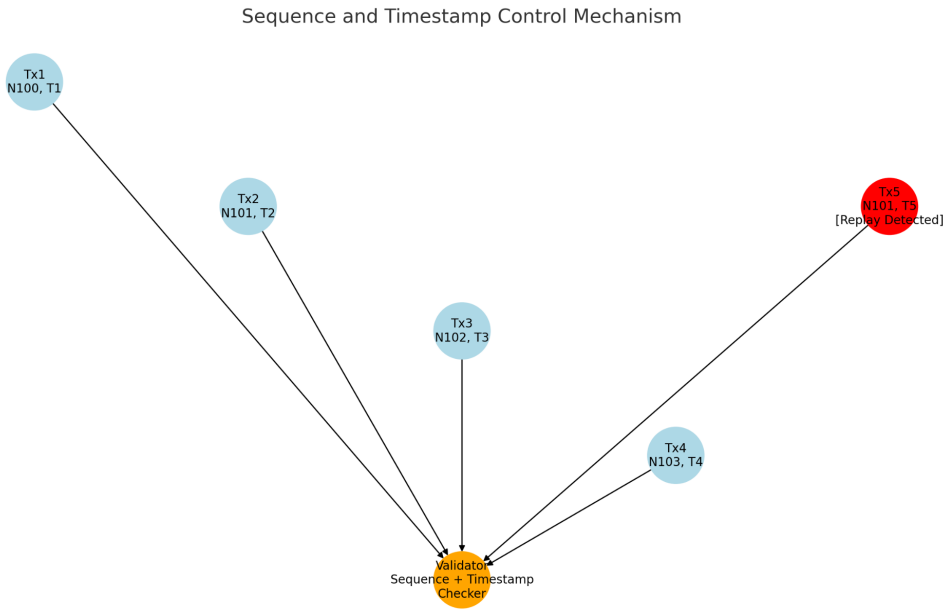

Diagram 5

In this diagram,

- blue nodes - valid, sequential transactions (Tx1 to Tx4), each tagged with a unique nonce (N100 to N103) and increasing timestamps (T1 to T4).
- Red node (Tx5) - a replayed transaction with a reused nonce (N101) and later timestamp (T5)—flagged and rejected by the validator.
- Orange node (Validator) - a logical module that checks nonce uniqueness, timestamp freshness, and transaction order.

3.4 Time-Locked Transaction Queues

This protocol would hold transactions for a predetermined period before execution. Some blockchains, such as Ergo which Rosenbridge operates on top of, already hold transactions, but not all.

For implementation, this would require:

- Queue mechanism - placing transitions in a queue with a time lock before they are processed.
- Dispute window - allow participants to challenge transactions during the time lock if they suspect malicious activity.
- Resolution protocol - establish a protocol for resolving disputes, potentially involving additional verification steps or arbitration.

This also has the potential to create more issues. There have been instances [12] where vulnerable queue implementations have been exploited by malicious actors. If implemented well, this would provide a buffer to detect and prevent fraudulent transactions.

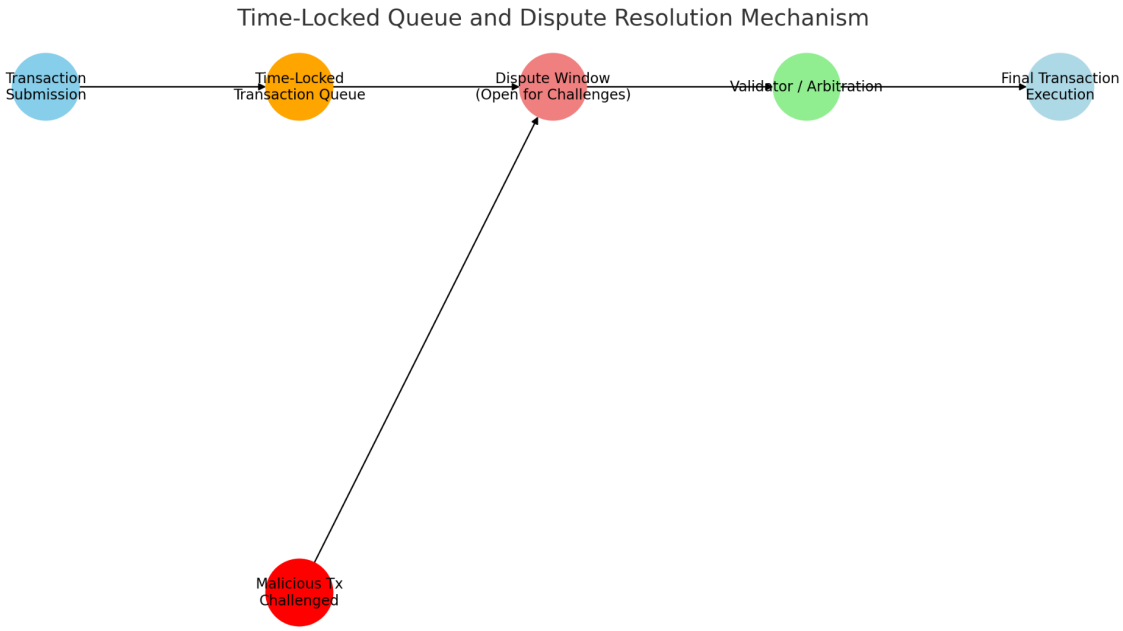

Diagram 6

- Blue node (Tx Submission) - initial submission of a transaction into the system.
- Orange node (Queue) - transaction is held here temporarily before processing.
- Red node (Dispute Window) - time period during which the transaction can be challenged.
- Green node (Validator / Arbitration) - handles any disputes and performs final checks.
- Grey node (Execution) - final approval and execution of the transaction.
- Red node (Malicious Tx) - an example of a challenged and potentially fraudulent transaction flagged during the dispute window.

4. Validation and Evaluation

4.1 Simulation Design and Setup

The goal of my project was to simulate fraudulent activities within Rosenbridge and assess how different attack vectors, particularly those involving fabricated events and collusion, impacted the protocol's integrity and how proposed strategies might prevent or detect these behaviors.

To do this, I used a testnet provided by the Rosenbridge developers in Scala and separately developed a custom simulation environment in Python to replicate the behavior of Watchers and Guards in Rosenbridge, using simplified blockchain models for event reporting and verification. The Rosenbridge code relied on deprecated modules, which made it difficult to customize simulations, whereas the python code allows much more flexibility (code attached in the appendix).

The players simulated include honest Watchers and Guards, colluding Watchers, compromised Guards, simulated bridges and transaction queues.

I aimed to simulate three types of attacks: fabrication of events, where malicious watchers submit fake reports of token locks that never occurred; collusion scenarios, where Watchers and Guards confirm fabricated events leading to asset mints; and replay attacks, re-submission of previously valid events to test redundancy.

All transactions were timestamped and logged. Validation logic includes nonce and sequence checks. Event verification uses simplified consensus modeled as a quorum agreement (⅔ of guards agreeing usually means validation).

4.2 Evaluation Metrics

I employed the following evaluation metrics:

- Detection rate - number of false positives, fabricated or replayed transactions correctly flagged or rejected.
- False acceptance rate - percentage of fraudulent events that bypassed detection and were processed.
- Event confirmation latency - time taken from event submission to confirmation under normal and compromised conditions.
- Attack impact score - cumulative value of unauthorized tokens transferred due to fabricated events.

```
Detection Rate (Correctly Rejected Fraudulent): 30
False Positives (Valid Events Rejected): 0
False Acceptance Rate: 0/30 = 0.00%
Event Confirmation Latency (steps):
  Average: 0.00
  Min: 0
  Max: 0
Attack Impact Score (Total Value of Finalized Fraudulent Events): 0
```

Diagram 7: Sample Output With Low Simulated Watchers and Guards (Low Latency)

4.3 Comparative Analysis with Existing Frameworks

I compared data from Rosenbridge without added safeguards and my own simulation, but was unable to find any vulnerabilities (see appendix for code and output). None of the hundreds of fraudulent transactions successfully bypassed the system.

4.4 Results and Discussion

In my comprehensive simulations, I attempted to exploit potential vulnerabilities in the Rosen Bridge protocol by introducing various fraudulent transactions, including fabricated events, collusion between Watchers and Guards, and replay attacks. Despite these efforts, Rosen Bridge consistently thwarted all malicious attempts, demonstrating robust security mechanisms.

The protocol's architecture, which involves a two-layer verification system with Watchers reporting events and Guards validating them, proved effective in preventing unauthorized asset transfers. Even when simulating scenarios with compromised Watchers and Guards, the fraudulent transactions were identified and rejected.

5. Discussion

5.1 Practicality and Scalability of Proposed Solutions

The simulation results indicate that Rosenbridge’s existing security architecture effectively prevents fraudulent transactions, even under scenarios involving collusion between Watchers and Guards. However, as Ergo and Rosenbridge become more popular, additional security measures proposed in section 3 might prove to be helpful, balanced with the risks of latency and computational overhead.

5.2 Limitations and Assumptions

My simulations assume a static network topology and predefined behaviors of malicious actors. In practice, attackers may employ more sophisticated strategies, and network conditions can be highly dynamic. The simulations also did not account for potential vulnerabilities arising from smart contract bugs, external dependencies, or social engineering attacks targeting key management systems. These factors represent areas where further research and testing are necessary to ensure comprehensive security coverage.

6. Conclusion and Future Work

6.1 Summary of Contributions

This thesis undertook a rigorous examination of the Rosen Bridge protocol, focusing on its resilience against fraudulent transactions, particularly those involving collusion between Watchers and Guards. Through extensive simulations, we attempted to exploit potential vulnerabilities by introducing various attack vectors, including fabricated events and replay attacks. Notably, none of these malicious attempts succeeded, underscoring the robustness of Rosen Bridge's multi-layered security architecture.

Key contributions of this research include:

- Threat modeling: Developed a comprehensive threat model tailored to Rosen Bridge, identifying potential attack vectors and assessing their feasibility within the protocol's framework.
- Simulation framework: Designed and implemented a simulation environment to test the protocol's defenses against orchestrated fraudulent activities.
- Security analysis: Evaluated the effectiveness of Rosen Bridge's existing security mechanisms, confirming their efficacy in preventing unauthorized asset transfers.

6.2 Generalization of the Threat Model

While the threat model was specifically crafted for Rosen Bridge, its principles are applicable to a broader range of cross-chain bridge protocols. The emphasis on role separation (e.g., Watchers and Guards), multi-signature verification, and decentralized consensus mechanisms are common features in many bridging solutions. Therefore, the insights gained from this study can inform the design and evaluation of other protocols, enhancing their security postures against similar threats.

6.3 Future Work: Deeper Exploration of Collusion and Dynamic Defenses

Building upon the findings of this research, several avenues warrant further exploration:

- Advanced collusion scenarios - investigate more sophisticated collusion strategies, including dynamic collusion where participants adapt their behavior over time to evade detection.
- Dynamic defense mechanisms - develop and assess adaptive security measures that respond in real-time to detected threats, such as dynamic quorum adjustments or real-time anomaly detection systems.
- Integration with other protocols - explore the interoperability of Rosen Bridge with other cross-chain solutions, assessing how combined security measures can bolster overall resilience.
- Formal verification - apply formal methods to mathematically prove the correctness and security properties of Rosen Bridge, providing a higher assurance level.

6.4 Closing Remarks

The integrity and security of cross-chain bridges are paramount in the evolving landscape of decentralized finance. Rosen Bridge exemplifies a robust approach to cross-chain asset transfers, demonstrating resilience against common attack vectors. This research reaffirms the importance of meticulous protocol design, continuous security assessments, and adaptive defense mechanisms. As the ecosystem grows, ongoing vigilance and innovation will be essential to safeguard the trust and reliability of cross-chain interactions.

7. References

[1] Nakamoto, Satoshi. "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008.

[2] Grand View Research. "Blockchain Technology Market Size: Industry Report, 2030." Accessed April 23, 2025. https://www.grandviewresearch.com/industry-analysis/blockchain-technology-market.

[3] Scaffino, Giulia, Luca Eckey, Sebastian Faust, Kristina Hostáková, and Patrick Struck. "GLIMPSE: On-Demand PoW Light Client with Constant-Size Storage for DeFi." USENIX. Accessed April 23, 2025. https://www.usenix.org/conference/usenixsecurity23/presentation/scaffino.

[4] Newman, Lily Hay. "The Vulnerability of Cross-Chain Bridges Puts the Crypto Ecosystem at Risk." Wired. Accessed April 23, 2025. https://www.wired.com/story/blockchain-network-bridge-hacks/.

[5] "Notary Schemes." QX Interoperability - v.0.7. Accessed April 23, 2025. https://qRosenbridgeualitax.gitbook.io/interop/bridging-approaches/mechanisms/notary-schemes.

[6] Chen, Longfei, Yaofei Zhang, Jianwei Liu, Mingxing Hu, and Bin Wu. "Three-Stage Cross-Chain Protocol Based on Notary Group." Electronics 12, no. 13 (2023): 2804. https://doi.org/10.3390/electronics12132804.

[7] Xie, T., J. Zhang, Z. Cheng, F. Zhang, Y. Zhang, Y. Jia, D. Boneh, and D. Song. "zkBridge: Trustless Cross-Chain Bridges Made Practical." arXiv:2210.00264 [cs], 2022. https://arxiv.org/abs/2210.00264.

[8] CryptoCounsel. "The Rosen Bridge Protocol: Seamless Cross-Chain Asset Transfers." Cardano Spot, March 3, 2024. https://cardanospot.io/news/kxwLh656VPKIECzV3Rl9PYXe6yjLfynH.

[9] Ergo Platform. "Ergo's Rosen Bridge: A New and Better Way to Interact Cross-Chain." Ergo Platform, April 4, 2023. https://ergoplatform.org/en/blog/Ergo%E2%80%99s-Rosen-Bridge-A-New-and-Better-Way-to-Interact-Cross-Chain/.

[10] "Rosenbridge." Rosen.tech. Accessed April 23, 2025. https://rosen.tech/.

[11] Lamport, Leslie, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem." ACM Transactions on Programming Languages and Systems 4, no. 3 (July 1982): 382-401.

[12] Reuters. "Two Former MIT Students Charged with Stealing $25 Million of Crypto in 12 Seconds." CNN Business, May 16, 2024. https://www.cnn.com/2024/05/16/investing/mit-crypto-hack/index.html.

[13] "51% Attacks." Digital Currency Initiative, MIT. Accessed April 23, 2025. https://www.dci.mit.edu/projects/51-percent-attacks.

8. Appendices

8.1 GitHub

GitHub: https://github.com/zaranip/Rosenbridge-Thesis

8.2 Simulation Code (With Comments)

Please note: the Github code has been abstracted out for better readability. The code below is consolidated for the purposes of this paper.

```python
import random
import uuid
from collections import defaultdict, namedtuple

# --- Configuration ---
NUM_WATCHERS = 10
NUM_GUARDS = 5
WATCHER_MALICIOUS_RATIO = 0.2  # 20% of watchers might be malicious
GUARD_MALICIOUS_RATIO = 0.2      # 20% of guards might be malicious
GUARD_THRESHOLD_RATIO = 0.6  # 60% of Guards must verify for an event to pass

# --- Data Structures ---

# Represents a real event that occurred on a source chain
Event = namedtuple("Event", ["event_id", "source_chain_id", "target_chain_id",
"data", "is_valid"])

# Represents an event reported by a Watcher (could be real or fabricated)
ReportedEvent = namedtuple("ReportedEvent", ["event_id", "source_chain_id",
"target_chain_id", "data", "reporter_id"])

# --- Simplified Blockchain Model ---

class SimpleBlockchain:
      """A simplified representation of a blockchain holding actual events."""
      def __init__(self, chain_id):
      self.chain_id = chain_id
      self.events = {}  # event_id -> Event
      print(f"Blockchain '{self.chain_id}' initialized.")

      def add_event(self, target_chain_id, data):
      """Adds a new legitimate event to this blockchain."""
      event_id = str(uuid.uuid4())
```

```
        new_event = Event(
                event_id=event_id,
                source_chain_id=self.chain_id,
                target_chain_id=target_chain_id,
                data=data,
                is_valid=True # Events added here are considered real/valid
        )
        self.events[event_id] = new_event
        print(f"[Blockchain:{self.chain_id}] Added valid event {event_id[:8]} for
target {target_chain_id}")
        return new_event

        def get_event(self, event_id):
        """Retrieve an event by its ID."""
        return self.events.get(event_id)

        def get_all_event_ids(self):
        """Return all valid event IDs on this chain."""
        return list(self.events.keys())

# --- Rosenbridge Participant Models ---

class Watcher:
        """Monitors a blockchain and reports events."""
        def __init__(self, watcher_id, monitored_chain, is_malicious=False):
        self.watcher_id = watcher_id
        self.monitored_chain = monitored_chain # SimpleBlockchain instance
        self.is_malicious = is_malicious
        self.seen_event_ids = set() # Track reported events to avoid duplicates
        self.type = "Malicious" if is_malicious else "Honest"
        print(f"  Watcher {self.watcher_id} ({self.type}) initialized monitoring
{monitored_chain.chain_id}.")

        def monitor_and_report(self):
        """
        Scans the monitored blockchain for new events and reports them.
        Malicious watchers might fabricate events.
        """
        reports = []
        # --- Honest Behavior ---
        if not self.is_malicious:
                all_event_ids = self.monitored_chain.get_all_event_ids()
                new_event_ids = set(all_event_ids) - self.seen_event_ids
                for event_id in new_event_ids:
```

```
                event = self.monitored_chain.get_event(event_id)
                if event:
                        report = ReportedEvent(
                                event_id=event.event_id,
                                source_chain_id=event.source_chain_id,
                                target_chain_id=event.target_chain_id,
                                data=event.data,
                                reporter_id=self.watcher_id
                        )
                        reports.append(report)
                        self.seen_event_ids.add(event_id)
                        # print(f"  Watcher {self.watcher_id} (Honest): Reported
valid event {event_id[:8]}")

        # --- Malicious Behavior (Example: Fabricate one event per step) ---
        else:
                # Option 1: Report real events correctly (less disruptive malice)

                # Option 2: Fabricate an event
                # Simulate fabricating an event that *doesn't* exist on the chain
                fake_event_id = f"fake-{str(uuid.uuid4())}"
                fake_report = ReportedEvent(
                event_id=fake_event_id,
                source_chain_id=self.monitored_chain.chain_id,
                target_chain_id="target-chain-malicious", # Could be random or
targeted
                data={"amount": 9999, "recipient": "attacker_addr"},
                reporter_id=self.watcher_id
                )
                reports.append(fake_report)
                print(f"  Watcher {self.watcher_id} (Malicious): Fabricated event
{fake_event_id[:8]}")

                # Malicious watchers might also choose *not* to report valid
events,
                # or delay reporting, but fabrication is a direct attack vector
here.
                # They might also report real events alongside fake ones.
                # For simplicity, this example just fabricates one.

        return reports

class Guard:
        """Verifies events reported by Watchers."""
```

```
	def __init__(self, guard_id, known_blockchains, is_malicious=False):
	self.guard_id = guard_id
	self.known_blockchains = known_blockchains # dict: chain_id ->
SimpleBlockchain
	self.is_malicious = is_malicious
	self.type = "Malicious" if is_malicious else "Honest"
	print(f"  Guard {self.guard_id} ({self.type}) initialized.")

	def verify_event(self, reported_event):
	"""
	Checks if the reported event actually exists and is valid on the source
chain.
	Malicious guards might lie about verification.
	"""
	source_chain = self.known_blockchains.get(reported_event.source_chain_id)
	if not source_chain:
		print(f"  Guard {self.guard_id}: Unknown source chain
{reported_event.source_chain_id} for event {reported_event.event_id[:8]}. Cannot
verify.")
		return False # Cannot verify if chain is unknown

	actual_event = source_chain.get_event(reported_event.event_id)

	# --- Honest Behavior ---
	if not self.is_malicious:
		if actual_event and actual_event.data == reported_event.data:
		# print(f"  Guard {self.guard_id} (Honest): Verified event
{reported_event.event_id[:8]} as VALID.")
		return True
		else:
		# print(f"  Guard {self.guard_id} (Honest): Verified event
{reported_event.event_id[:8]} as INVALID (not found or data mismatch).")
		return False

	# --- Malicious Behavior (Ex. Collusion - approve fake events) ---
	else:
		is_fabricated = reported_event.event_id.startswith("fake-")
		if is_fabricated:
		# Malicious guard approves a known fake event
		print(f"  Guard {self.guard_id} (Malicious): Falsely verifying
fabricated event {reported_event.event_id[:8]} as VALID.")
		return True
		else:
```

```
            # Malicious guard might deny a real event or approve a real one
normally
            # Here, let's make it deny real events sometimes (disruption)
            if random.random() < 0.5: # 50% chance to deny a real event
                  print(f"  Guard {self.guard_id} (Malicious): Falsely
verifying real event {reported_event.event_id[:8]} as INVALID.")
                  return False
            else:
                  # Or behave honestly sometimes to avoid easy detection
                  is_valid = actual_event and actual_event.data ==
reported_event.data
                  # print(f"  Guard {self.guard_id} (Malicious): Behaving
honestly for event {reported_event.event_id[:8]}. Result: {is_valid}")
                  return is_valid

# --- Simulation Environment ---

class RosenbridgeSimulation:
      """Orchestrates the simulation of Rosenbridge components."""
      def __init__(self, num_watchers, num_guards, watcher_malicious_ratio,
guard_malicious_ratio, guard_threshold_ratio):
      self.num_watchers = num_watchers
      self.num_guards = num_guards
      self.watcher_malicious_ratio = watcher_malicious_ratio
      self.guard_malicious_ratio = guard_malicious_ratio
      self.guard_threshold = int(num_guards * guard_threshold_ratio)
      if self.guard_threshold == 0 and num_guards > 0:
            self.guard_threshold = 1 # Need at least one guard approval if
guards exist

      print(f"\nInitializing Simulation:")
      print(f" Watchers: {num_watchers} ({int(num_watchers *
watcher_malicious_ratio)} malicious)")
      print(f" Guards: {num_guards} ({int(num_guards * guard_malicious_ratio)}
malicious)")
      print(f" Guard Threshold: {self.guard_threshold} / {num_guards}")

      self.blockchains = {}
      self.watchers = []
      self.guards = []

      # State tracking
      self.pending_reports = defaultdict(list) # event_id -> [ReportedEvent]
```

```
        self.guard_signatures = defaultdict(dict) # event_id -> {guard_id: bool
(verification_result)}
        self.finalized_events = set() # event_ids that passed guard threshold
        self.processed_event_ids = set() # Track all event_ids seen by guards

        # Statistics
        self.stats = {
                "valid_events_created": 0,
                "fabricated_events_reported": 0,
                "valid_events_finalized": 0,
                "fabricated_events_finalized": 0, # Successful attack
                "valid_events_rejected": 0,
                "fabricated_events_rejected": 0, # Attack detected/prevented
        }

        def setup(self, chain_ids=["ChainA", "ChainB"]):
        """Creates blockchains, watchers, and guards."""
        print("\n--- Setting up Simulation Environment ---")
        # Create Blockchains
        for chain_id in chain_ids:
                self.blockchains[chain_id] = SimpleBlockchain(chain_id)

        if not self.blockchains:
                print("Error: No blockchains defined.")
                return

        available_chains = list(self.blockchains.values())

        # Create Watchers
        print("\nCreating Watchers...")
        num_malicious_watchers = int(self.num_watchers *
self.watcher_malicious_ratio)
        malicious_indices_w = random.sample(range(self.num_watchers),
num_malicious_watchers)
        for i in range(self.num_watchers):
                is_malicious = i in malicious_indices_w
                # Assign watchers to monitor chains (e.g., round-robin or random)
                monitored_chain = random.choice(available_chains)
                watcher = Watcher(f"W{i}", monitored_chain, is_malicious)
                self.watchers.append(watcher)

        # Create Guards
        print("\nCreating Guards...")
        num_malicious_guards = int(self.num_guards * self.guard_malicious_ratio)
```

```
        malicious_indices_g = random.sample(range(self.num_guards),
num_malicious_guards)
        for i in range(self.num_guards):
                is_malicious = i in malicious_indices_g
                # Guards know all blockchains for verification
                guard = Guard(f"G{i}", self.blockchains, is_malicious)
                self.guards.append(guard)
        print("--- Setup Complete ---")

        def trigger_event(self, source_chain_id, target_chain_id, data):
        """Manually trigger a new valid event on a source chain."""
        if source_chain_id in self.blockchains:
                event =
self.blockchains[source_chain_id].add_event(target_chain_id, data)
                self.stats["valid_events_created"] += 1
                return event
        else:
                print(f"Error: Cannot trigger event, source chain
'{source_chain_id}' not found.")
                return None

        def run_simulation_step(self, step_num):
        """Runs one step of the simulation: Watchers report, Guards verify."""
        print(f"\n--- Simulation Step {step_num} ---")

        # 1. Watchers monitor and report
        print(" Watcher Reporting Phase...")
        new_reports = []
        for watcher in self.watchers:
                reports = watcher.monitor_and_report()
                new_reports.extend(reports)

        # Collate reports by event_id
        for report in new_reports:
                # Check if this is a fabricated event report
                if report.event_id.startswith("fake-"):
                # Only count first time a specific fake event ID is reported
                if report.event_id not in self.processed_event_ids:
                        self.stats["fabricated_events_reported"] += 1
                        self.processed_event_ids.add(report.event_id) # Mark as seen

                # Avoid adding duplicate reports *from the same watcher* for the
same event
                # (though multiple watchers can report the same event)
```

```
            # This simple check assumes a watcher won't report the same event
twice
            # A more robust check might be needed depending on watcher logic
            self.pending_reports[report.event_id].append(report)
            # print(f"  Event {report.event_id[:8]} reported by
{report.reporter_id}")


      # 2. Guards verify pending reports
      print("\n Guard Verification Phase...")
      events_to_verify = list(self.pending_reports.keys()) # Process current
pending events

      if not events_to_verify:
            print("  No new events reported for verification.")
            return # Skip guard phase if nothing new was reported

      for event_id in events_to_verify:
            if event_id in self.finalized_events or event_id in
self.processed_event_ids:
            # Avoid re-processing events unless necessary (e.g. insufficient
signatures previously)
            # For simplicity now, skip if already finalized or processed by
guards this step.
            # A more complex sim might allow re-verification attempts.
            continue

            reports_for_event = self.pending_reports[event_id]
            if not reports_for_event:
            continue # Should not happen with defaultdict, but safe check

            # Guards verify based on the first report they see for this
event_id
            # In reality, they might need multiple reports or consensus among
watchers first.
            # Here, we simplify: if *any* watcher reported it, guards check
it.
            representative_report = reports_for_event[0]
            print(f"  Guards verifying event {event_id[:8]} (Reported by
{len(reports_for_event)} watchers)")

            self.processed_event_ids.add(event_id) # Mark event ID as
processed by guards
```

```
            for guard in self.guards:
            if guard.guard_id not in self.guard_signatures[event_id]: # Avoid
double-voting per step
                  verification_result =
guard.verify_event(representative_report)
                  self.guard_signatures[event_id][guard.guard_id] =
verification_result

      # 3. Check for finalized events (Guard Consensus)
      print("\n Finalization Phase...")
      newly_finalized = []
      for event_id in events_to_verify: # Check events processed in this step
            if event_id in self.finalized_events:
                  continue # Already finalized previously

            if event_id in self.guard_signatures:
            signatures = self.guard_signatures[event_id]
            positive_signatures = sum(1 for sig in signatures.values() if sig
is True)

            print(f"  Event {event_id[:8]}: Received
{positive_signatures}/{len(self.guards)} positive signatures (Threshold:
{self.guard_threshold})")

            # Check if threshold is met
            if positive_signatures >= self.guard_threshold:
                  self.finalized_events.add(event_id)
                  newly_finalized.append(event_id)
                  print(f"  ✅ Event {event_id[:8]} FINALIZED.")

                  # Update Stats based on whether it was real or fabricated
                  is_fabricated = event_id.startswith("fake-")
                  if is_fabricated:
                        self.stats["fabricated_events_finalized"] += 1
                  else:
                        # Check if it was actually a valid event on the
source chain
                        representative_report =
self.pending_reports[event_id][0]
                        source_chain =
self.blockchains.get(representative_report.source_chain_id)
                        actual_event = source_chain.get_event(event_id) if
source_chain else None
                        if actual_event and actual_event.is_valid:
```

```
                            self.stats["valid_events_finalized"] += 1
                            else:
                            # This case (finalized but not actually valid) is
problematic
                            # Could happen if malicious guards pushed through
invalid data for a real ID
                            print(f"  ⚠️ WARNING: Finalized event {event_id[:8]}
seems invalid or data mismatch despite signatures!")
                            # Could count this as a specific type of failure if
needed

              # Else: Not enough signatures yet (could be re-evaluated in future
steps if needed)
              # elif len(signatures) == self.num_guards: # If all guards voted
and it still failed
              else:
                     # Check if it *should* have been finalized (i.e. it was
valid but rejected)
                     # or if it was correctly rejected (i.e. it was fabricated
and rejected)
                     # This check happens implicitly when we finalize reporting

                     # Clean up pending reports for events processed this step
but not finalized
                     # Or keep them for retry logic later? For now, assume
processed means done.
                     pass


       # 4. Clean up processed reports from pending queue (optional, depends on
desired logic)
       # for event_id in events_to_verify:
       #      if event_id in self.pending_reports:
       #             del self.pending_reports[event_id] # Remove processed
events

       print(f"--- End of Step {step_num} ---")

       def report_results(self):
       """Prints the final statistics of the simulation."""
       print("\n--- Simulation Results ---")
       print(f"Total Valid Events Created:
{self.stats['valid_events_created']}")
```

```
        print(f"Total Fabricated Events Reported by Malicious Watchers:
{self.stats['fabricated_events_reported']}")
        print("-" * 20)
        print(f"Valid Events Finalized (Correctly):
{self.stats['valid_events_finalized']}")
        print(f"Fabricated Events Finalized (Successful Attack):
{self.stats['fabricated_events_finalized']}")
        print("-" * 20)

        # Calculate rejections
        # Valid Rejected = Valid Created - Valid Finalized
        self.stats["valid_events_rejected"] = self.stats["valid_events_created"]
- self.stats["valid_events_finalized"]
        # Fabricated Rejected = Fabricated Reported - Fabricated Finalized
        # Note: This assumes all reported fake events were processed by guards.
        self.stats["fabricated_events_rejected"] =
self.stats["fabricated_events_reported"] -
self.stats["fabricated_events_finalized"]

        print(f"Valid Events Rejected/Not Finalized:
{self.stats['valid_events_rejected']}")
        print(f"Fabricated Events Rejected (Attack Prevented):
{self.stats['fabricated_events_rejected']}")
        print("-" * 20)
        print(f"Total Events Processed by Guards:
{len(self.processed_event_ids)}")
        print(f"Total Events Finalized: {len(self.finalized_events)}")
        print("--- End of Report ---")


if __name__ == "__main__":
        # Initialize Simulation
        sim = RosenbridgeSimulation(
        num_watchers=NUM_WATCHERS,
        num_guards=NUM_GUARDS,
        watcher_malicious_ratio=WATCHER_MALICIOUS_RATIO,
        guard_malicious_ratio=GUARD_MALICIOUS_RATIO,
        guard_threshold_ratio=GUARD_THRESHOLD_RATIO
        )

        # Setup chains and participants
        sim.setup(chain_ids=["ChainA", "ChainB", "ChainC"])

        # Simulation Steps
```

```
        num_simulation_steps = 5
        events_per_step = 2

        for i in range(num_simulation_steps):
        print(f"\nStarting Simulation Step {i+1}/{num_simulation_steps}")

        # Trigger some new valid events in each step
        for j in range(events_per_step):
                source = random.choice(list(sim.blockchains.keys()))
                target = random.choice([c for c in sim.blockchains.keys() if c !=
source])
                sim.trigger_event(
                source_chain_id=source,
                target_chain_id=target,
                data={"amount": random.randint(10, 1000), "tx_id": f"tx_{i}_{j}"}
                )

        # Run the watcher/guard logic for the step
        sim.run_simulation_step(i + 1)

        # Report final results
        sim.report_results()
```

8.3 Consolidated Sample Output

Python

```
Initializing Simulation:
 Watchers: 10 (3 malicious)
 Guards: 5 (1 malicious)
 Guard Threshold: 3 / 5

--- Setting up Simulation Environment ---
Creating Watchers...
Creating Guards...
--- Setup Complete ---

--- Simulation Results ---
Total Valid Events Created: 30
Total Fabricated Events Reported (Attempts): 30
--------------------
Valid Events Finalized (Correctly Processed): 30
```

```
Fabricated Events Finalized (Successful Attacks): 0
--------------------
Detection Rate (Correctly Rejected Fraudulent): 30
False Positives (Valid Events Rejected): 0
False Acceptance Rate: 0/30 = 0.00%
Event Confirmation Latency (steps):
  Average: 0.00
  Min: 0
  Max: 0
Attack Impact Score (Total Value of Finalized Fraudulent Events): 0
--------------------
--- End of Report ---
```

8.4 Expanded Sample Output

Python

```
Initializing Simulation:
 Watchers: 10 (2 malicious)
 Guards: 5 (1 malicious)
 Guard Threshold: 3 / 5

--- Setting up Simulation Environment ---
Blockchain 'ChainA' initialized.
Blockchain 'ChainB' initialized.
Blockchain 'ChainC' initialized.

Creating Watchers...
  Watcher W0 (Honest) initialized monitoring ChainC.
  Watcher W1 (Honest) initialized monitoring ChainA.
  Watcher W2 (Malicious) initialized monitoring ChainA.
  Watcher W3 (Honest) initialized monitoring ChainB.
  Watcher W4 (Honest) initialized monitoring ChainC.
  Watcher W5 (Honest) initialized monitoring ChainC.
  Watcher W6 (Honest) initialized monitoring ChainC.
  Watcher W7 (Malicious) initialized monitoring ChainA.
  Watcher W8 (Honest) initialized monitoring ChainA.
  Watcher W9 (Honest) initialized monitoring ChainA.

Creating Guards...
  Guard G0 (Honest) initialized.
  Guard G1 (Honest) initialized.
```

```
  Guard G2 (Honest) initialized.
  Guard G3 (Honest) initialized.
  Guard G4 (Malicious) initialized.
--- Setup Complete ---

Starting Simulation Step 1/5
[Blockchain:ChainB] Added valid event f79364c8 for target ChainA
[Blockchain:ChainA] Added valid event 084f48f5 for target ChainB

--- Simulation Step 1 ---
 Watcher Reporting Phase...
  Watcher W2 (Malicious): Fabricated event fake-6fb
  Watcher W7 (Malicious): Fabricated event fake-b7c

 Guard Verification Phase...
  Guards verifying event 084f48f5 (Reported by 3 watchers)
  Guards verifying event f79364c8 (Reported by 1 watchers)

 Finalization Phase...
  Event 084f48f5: Received 5/5 positive signatures (Threshold: 3)
  ✅ Event 084f48f5 FINALIZED.
  Event f79364c8: Received 5/5 positive signatures (Threshold: 3)
  ✅ Event f79364c8 FINALIZED.
--- End of Step 1 ---

Starting Simulation Step 2/5
[Blockchain:ChainA] Added valid event fbee29c6 for target ChainC
[Blockchain:ChainC] Added valid event fee51770 for target ChainB

--- Simulation Step 2 ---
 Watcher Reporting Phase...
  Watcher W2 (Malicious): Fabricated event fake-6f5
  Watcher W7 (Malicious): Fabricated event fake-486

 Guard Verification Phase...
  Guards verifying event fee51770 (Reported by 4 watchers)
  Guard G4 (Malicious): Falsely verifying real event fee51770 as INVALID.
  Guards verifying event fbee29c6 (Reported by 3 watchers)
  Guard G4 (Malicious): Falsely verifying real event fbee29c6 as INVALID.

 Finalization Phase...
  Event fee51770: Received 4/5 positive signatures (Threshold: 3)
  ✅ Event fee51770 FINALIZED.
  Event fbee29c6: Received 4/5 positive signatures (Threshold: 3)
```

```
  ✅ Event fbee29c6 FINALIZED.
--- End of Step 2 ---

Starting Simulation Step 3/5
[Blockchain:ChainB] Added valid event 3476dcde for target ChainC
[Blockchain:ChainA] Added valid event 10170c86 for target ChainB

--- Simulation Step 3 ---
 Watcher Reporting Phase...
  Watcher W2 (Malicious): Fabricated event fake-f55
  Watcher W7 (Malicious): Fabricated event fake-61e

 Guard Verification Phase...
  Guards verifying event 10170c86 (Reported by 3 watchers)
  Guard G4 (Malicious): Falsely verifying real event 10170c86 as INVALID.
  Guards verifying event 3476dcde (Reported by 1 watchers)

 Finalization Phase...
  Event 10170c86: Received 4/5 positive signatures (Threshold: 3)
  ✅ Event 10170c86 FINALIZED.
  Event 3476dcde: Received 5/5 positive signatures (Threshold: 3)
  ✅ Event 3476dcde FINALIZED.
--- End of Step 3 ---

Starting Simulation Step 4/5
[Blockchain:ChainB] Added valid event 3aa28daf for target ChainC
[Blockchain:ChainA] Added valid event a252dde2 for target ChainC

--- Simulation Step 4 ---
 Watcher Reporting Phase...
  Watcher W2 (Malicious): Fabricated event fake-e0f
  Watcher W7 (Malicious): Fabricated event fake-cea

 Guard Verification Phase...
  Guards verifying event a252dde2 (Reported by 3 watchers)
  Guard G4 (Malicious): Falsely verifying real event a252dde2 as INVALID.
  Guards verifying event 3aa28daf (Reported by 1 watchers)
  Guard G4 (Malicious): Falsely verifying real event 3aa28daf as INVALID.

 Finalization Phase...
  Event a252dde2: Received 4/5 positive signatures (Threshold: 3)
  ✅ Event a252dde2 FINALIZED.
  Event 3aa28daf: Received 4/5 positive signatures (Threshold: 3)
  ✅ Event 3aa28daf FINALIZED.
```

```
--- End of Step 4 ---

Starting Simulation Step 5/5
[Blockchain:ChainC] Added valid event 65f52009 for target ChainB
[Blockchain:ChainC] Added valid event f31148b0 for target ChainB

--- Simulation Step 5 ---
 Watcher Reporting Phase...
  Watcher W2 (Malicious): Fabricated event fake-038
  Watcher W7 (Malicious): Fabricated event fake-3b9

 Guard Verification Phase...
  Guards verifying event 65f52009 (Reported by 4 watchers)
  Guards verifying event f31148b0 (Reported by 4 watchers)

 Finalization Phase...
  Event 65f52009: Received 5/5 positive signatures (Threshold: 3)
  ✅ Event 65f52009 FINALIZED.
  Event f31148b0: Received 5/5 positive signatures (Threshold: 3)
  ✅ Event f31148b0 FINALIZED.
--- End of Step 5 ---

--- Simulation Results ---
Total Valid Events Created: 10
Total Fabricated Events Reported by Malicious Watchers: 10
--------------------
Valid Events Finalized (Correctly): 10
Fabricated Events Finalized (Successful Attack): 0
--------------------
Valid Events Rejected/Not Finalized: 0
Fabricated Events Rejected (Attack Prevented): 10
--------------------
Total Events Processed by Guards: 20
Total Events Finalized: 10
--- End of Report ---
```