IET Blockchain

The Institution of Engineering and Technology

ORIGINAL RESEARCH OPEN ACCESS

CommTLC: An Alternative to Reduce the Attack Surface of HTLCs in Lightning Networks

Prerna Arote | Joy Kuri

Department of Electronic Systems Engineering, Indian Institute of Science, Bangalore, Karnataka, India

Correspondence: Prerna Arote (prernaarote@iisc.ac.in) | Joy Kuri (kuri@iisc.ac.in)

Received: 24 October 2024 | **Revised:** 14 January 2025 | **Accepted:** 21 February 2025

Funding: No funding was received for this research.

ABSTRACT

The Payment Channel Network is widely recognized as one of the most effective solutions for handling off-chain transactions and addressing blockchain scalability challenges. In the Lightning Network, a popular off-chain mechanism, multi-hop payments are facilitated through Hashed Time-Locked Contracts (HTLCs). However, despite its broad adoption, HTLCs are susceptible to various attacks, such as Fakey, Griefing, and Wormhole attacks. In these attacks, adversaries aim to disrupt transaction throughput by exhausting channel capacity or stealing routing fees from honest nodes along the payment path. We propose a scheme called CommTLC, which leverages Pedersen commitments and signatures to detect and punish/prevent adversaries in Fakey, Griefing and Wormhole attacks. We implement the proposed scheme and analyse its security within the universal composability (UC) framework. Additionally, we compare the performance of CommTLC with the latest schemes, MAPPCN-OR and EAMHL+. The results demonstrate that CommTLC outperforms both MAPPCN-OR and EAMHL+ in communication overhead, with only a slight increase in computational overhead compared to MAPPCN-OR. Furthermore, the adversary detection time for Fakey, Griefing and Wormhole attacks using CommTLC is reduced to just a few milliseconds—specifically, less than 112 ms for a payment path involving five users.

1 | Introduction

The Bitcoin blockchain is one of the most popular immutable distributed ledgers, offering decentralization and pseudonymity [1]. As of 25 May 2024, Bitcoin processes approximately 8.35 million transactions daily, with a total transaction volume of 1.01 billion, highlighting its rapid growth in usage [2]. Despite Bitcoin's widespread adoption in the payment space, it faces challenges such as low throughput, long confirmation times, and high transaction fees, all of which contribute to scalability issues within the Bitcoin blockchain.

To overcome this issue, several layer-2 solutions [3–5] enable users to perform transactions off-chain, making Bitcoin more scalable. The Payment Channel Network (PCN) is one such layer-2 solution, with the Lightning Network [6] being the most widely used PCN. The Lightning Network allows nodes to connect through payment channels, enabling the payer and payee to conduct multiple transactions off-chain. However, a major drawback of PCN-based solutions is that parties must lock funds upfront. When the payer and payee do not have an already-established ('direct') channel between them, multi-hop channels can be utilized. A multi-hop channel consists of several direct channels concatenated to form a payment path, starting with the payer (the first node in the path), passing through intermediate nodes and ending with the payee (the last node in the path). Payments on a multi-hop channel rely on Hashed Time-Locked Contracts (HTLCs) [6], as shown in Figure 1. HTLCs enable conditional payments, meaning the payee receives funds only if certain conditions are met within a specified safety period, known

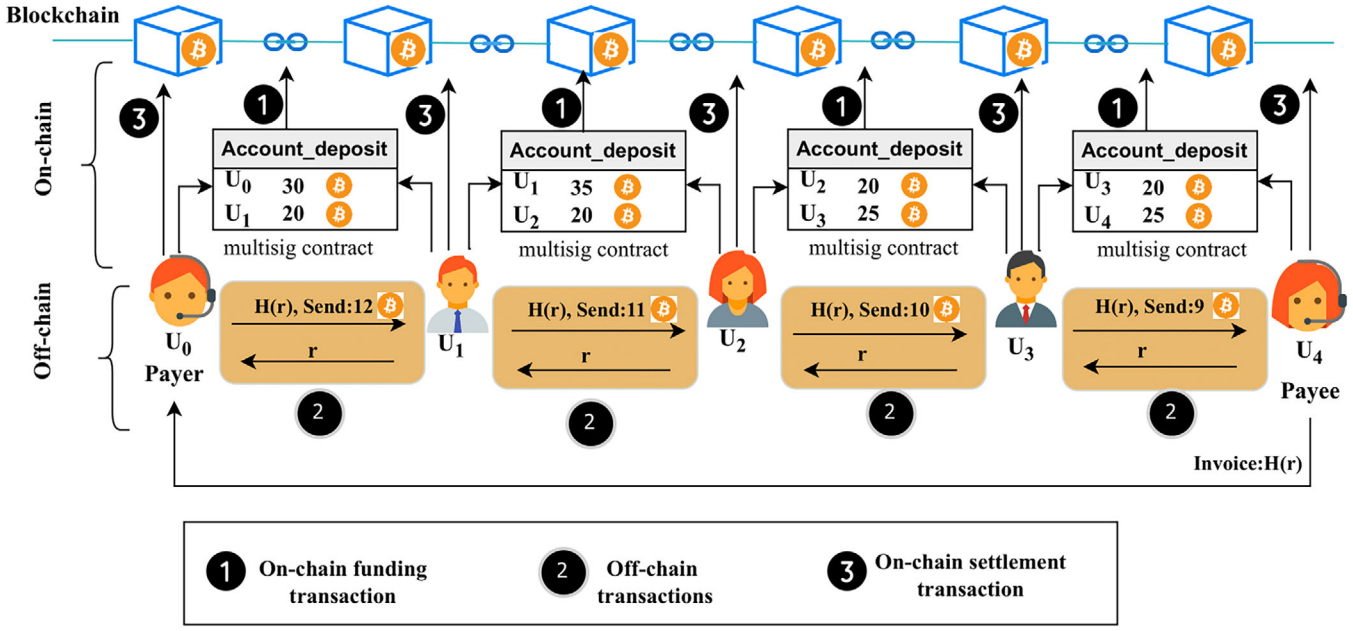

FIGURE 1 | Multi-hop payment procedure in payment channel network.

as the time-lock delta. If the conditions are not met within this time frame, the transaction amount is refunded to the payer.

Despite its widespread adoption, HTLC has been the target of several major cyberattacks, including Fakey attacks, Congestion attacks, Griefing attacks and Wormhole attacks [7–11]. **In this paper, we focus specifically on Fakey, Griefing and Wormhole attacks**. Fakey and Griefing attacks lead to contract failures, causing the payment amount to remain inaccessible until the contract's expiration period, effectively resulting in a denial-of-service (DoS) attack. In contrast, the Wormhole attack, carried out by colluding malicious nodes along the path, bypasses honest intermediaries and deprives them of their rightful routing fees. In all these attacks, the adversary's goal is to throttle transaction throughput by locking funds in one or more payment paths simultaneously, depleting channel capacity and stealing the routing fees from honest nodes along the payment path.

These attacks are possible due to certain vulnerabilities in HTLCs. One key issue is that the same payment hash is used in all HTLC contracts along the payment path. As a result, colluding nodes with access to the hash pre-image can bypass other intermediate nodes and claim the payment. Additionally, any two intermediate nodes can determine if they are on the same path by sharing the payment hash. There is also no mechanism to verify whether the payment hash provided by the payee matches the one used by the payer when setting up the contract. This creates an opportunity for the payer to manipulate or falsify the payment hash. Another vulnerability arises when the payee, having the pre-image, refuses to reveal it to unlock the payment, as there is no way to verify what payment hash was provided by the payee.

By taking advantage of these issues in HTLCs, an adversary can successfully launch these attacks. It is indeed required to detect such adversaries and prevent them from launching such harmful attacks. Many existing solutions focus on providing privacy guarantees to HTLCs and preventing Wormhole attacks [12–14]. However, none of these solutions address the detection of such adversaries. In an earlier version [15] of the proposed solution, we could detect adversaries in Fakey and Griefing attacks only by the receiver using hash-based commitments. That approach fails to detect adversaries involved in Griefing attacks by intermediate nodes and Wormhole attacks, and it does not prevent any of these attacks due to the same commitment being locked in each contract. The objective of this work is to **detect and punish/prevent** these adversaries, without violating privacy properties of PCN. Thus, this work is different from existing solutions.

We propose a protocol, named CommTLC (Commitments-based Time-Locked Contract), in which the contract between two parties uses distinct **commitments** instead of a single payment hash (as in HTLC). Initially, the payee selects a secret message and generates a Pedersen commitment using this secret message. Next, the payee shares this commitment along with a signed version of its commitment with the payer. Upon receiving this, the payer selects a different secret message, commits it and shares its commitment, a signed version of its commitment, and also the signed version of payee's commitment with the payee. After exchanging the Pedersen commitments, in the payment lock phase, the payer forms an onion routing packet such that each contract locks distinct but interdependent Pedersen commitments between every pair of users on the path. The **additive homomorphism property** of Pedersen commitments is utilized to unlock commitments in the payment release phase. Additionally, some partial opening information is shared with the intermediate nodes via this onion routing packet. We note that existing solutions assume that *the payer shares information private to each intermediate node through a secure and anonymous channel ('the secure and anonymous channel assumption')* [16]. **CommTLC does not depend on the secure and anonymous channel assumption**.

TABLE 1 | A summary of CommTLC's objectives.

Attacks	Adversary	Attack detection	Attack prevention	Adversary detection
Fakey attack	Payer	●	◐	●
Griefing attack	Payee or Intermediate nodes	●	◐	●
Wormhole attack	Intermediate nodes	●	●	●

Note: (●) denotes that the stated characteristic is achieved by CommTLC, while (◐) denotes that it is partially achieved.

1.1 | Our Contribution

- We propose the CommTLC protocol, which detects Fakey, Griefing and Wormhole attacks and adversaries launching them, and prevents Wormhole attacks (a summary is provided in Table 1).
- We implement the CommTLC scheme using the Lightning Network Daemon (LND) implementation of the Lightning Network and evaluate its performance in terms of communication and computational overhead. We also compare its performance with the MAPPCN and EAMHL+ schemes.
- We analyse the security of CommTLC using the UC framework.

We discuss prior literature in Section 2. Section 3 provides the necessary background information about the Lightning Network and potential attacks on HTLCs. The proposed solution is described in detail in Section 4. Implementation and evaluation details are presented in Section 5. Section 6 analyses the security of the proposed protocol. Finally, we conclude the work in Section 7.

2 | Related Work

The Lightning Network, one of the most prominent PCNs, uses HTLCs for multi-hop payments. However, HTLCs are vulnerable to attacks such as Fakey, Wormhole, and Griefing. Additionally, they compromise privacy properties in PCNs, including identity leakage of both the payer and payee, balance privacy, path privacy and more. To address these privacy concerns, various studies have proposed privacy-preserving PCNs, such as those presented in references [12, 17, 18]. Prior works like [19] and [20] explore cryptographic techniques for providing privacy guarantees with respect to user's location in the context of location-based service providing applications. These papers are using cryptographic mechanisms such as zkSNARK and Diffie-Hellman Key Agreement protocol to make user's location private. However, our focus is on addressing inherent vulnerabilities in the HTLC mechanism without imposing any additional privacy requirement on HTLCs. Next, we provide comparison of these PCN-related solutions in Table 2.

In reference [12], the authors were the first to identify that HTLC contracts are not privacy-preserving and are vulnerable to Wormhole attacks. Later, they proposed a privacy-preserving solution using the Paillier encryption scheme and non-interactive zero-knowledge proofs (NIZKPs) to defend against Wormhole attacks [17]. While this was the first off-chain protocol to address both privacy and concurrency in PCNs, the solution had a high computational cost due to the use of NIZKPs. Another drawback of this solution is that the sender must maintain a secure channel with each intermediate node in the payment path. It is important to note that establishing a secure channel requires knowledge of the peer's IP address, an assumption that may not always hold in a PCN setting. Furthermore, the existence of a secure channel reveals the sender's identity to the nodes along the path. This solution is neither secure against Fakey and Griefing attacks nor does it detect adversaries launching these attacks [17].

Later, in 2019, the authors in reference [16] proposed an efficient off-chain solution using chameleon hashes (CHTLC) to address the payment path leakage problem. The CHTLC scheme leverages chameleon hashes, which are less computationally expensive than NIZKs. However, it retains the drawbacks of reference [17] and is also vulnerable to key exposure attacks. The first solution to relax the secure channel assumption between the sender and intermediate nodes—while also addressing the identity disclosure of the payer and payee—was introduced by reference [13]. Their approach was based on elliptic curve group operations, though it was not implemented at the time. Despite offering some improvements, it neither detects adversaries nor prevents the attacks outlined in Section 3.2. In 2021, the same authors proposed another solution called neo-HTLC, which provides privacy guarantees without requiring a secure channel between the sender and intermediate nodes [18]. While the sender's identity was preserved in this approach, the solution was not compatible with the Sphinx onion packet format. To address this compatibility issue, the authors have also proposed key-based time-locked contracts (KTLCs), which use a symmetric key encryption-based protocol [18]. However, KTLC relies on the secure channel assumption, potentially exposing the sender's identity to nodes along the payment path. Additionally, neither neo-HTLC nor KTLC detect adversaries launching Fakey, Griefing or Wormhole attacks.

Although these solutions offer privacy guarantees, they either introduce high computational and communication overhead or remain insecure against Fakey, Griefing and Wormhole attacks. To address these issues, the authors in reference [21] proposed a lightweight solution using a simple hash function, onion routing and a sidechain. This approach preserves the identities of both the payer and the payee, while also preventing Griefing and Wormhole attacks. However, their solution requires the payee to deposit the routing fees (charged by intermediaries along the path) into the sidechain. Moreover, it remains vulnerable to Fakey attacks. The authors who identified the Fakey attack also suggested a countermeasure using symmetric encryption [22]. However, their solution does not detect adversaries launching Fakey attacks. In symmetric encryption, a shared secret key is independent of the message, which benefits the malicious sender.

TABLE 2 | Comparison of existing solutions in PCN.

Scheme	Cryptographic primitives	Attack detection and prevention			No SC ∗	Adversary detection	Drawbacks
		Fakey	Griefing	Wormhole			
HTLC	Hash function	✗	✗	✗	✓	✗	Security and relational anonymity is violated
AMHL	NIZK, HOWF, ECDSA	✗	✗	✓	✗	✗	Complex key management and reveals sender's identity
CHTLC	Chameleon Hash	✗	✗	✓	✗	✗	Reveals sender's identity and susceptible to key exposure attack
MHTLC	NIZK, Hash function	✗	✗	✓	✗	✗	Complex key management, high computation and communication overhead, and reveals sender's identity
MAPPPCN	ECC	✗	✗	✓	✓	✗	Not implemented
n-HTLC	Hash, CS	✗	✗	✓	✓	✗	Incompatible with onion routing and less efficient
KTLC	Symmetric encryption	✗	✗		✗	✗	Less efficient due to encryption technique used
Hybrid Multihop	Hash function	✗	✓	✓	✓	✗	Payee must deposit routing fees into sidechain.
HTLC-GP	Hash function	✗	✓	✓	✓	✗	Payee must deposit before payment initiation
AMHP (AMHL+, EAMHL+)	Bilinear maps	✓	✗	✓	✓	✗	AMHL+ has high communication cost and EAMHL+ has high computation overhead. All payments are of same value, thus privacy is leaked

Note: In this table, No SC ∗ denotes "No secure channel assumption with intermediate nodes."

A recent work by the authors in reference [14] achieves privacy in PCNs without relying on the secure channel assumption. Their solution is secure against Fakey and Wormhole attacks, though it remains vulnerable to the Griefing attack. To ensure privacy, the authors use bilinear maps, but this approach introduces significant computational overhead. Moreover, their solution assumes that the payment value remains consistent across all nodes, meaning intermediaries do not alter any routing fees. However, this assumption is unrealistic in real-world scenarios.

The authors in reference [23] propose a solution to prevent the Griefing attack. Their approach involves two contracts: a payment contract and a penalty contract. The receiver must lock funds in the penalty contract beforehand, and if the receiver refuses to provide the required pre-image, all other nodes along the payment path are compensated from the penalty contract. However, this solution neither detects adversaries nor prevents attacks, and it also requires the receiver to lock funds in the penalty contract before receiving any payment from the sender.

Thus, there is a need for a solution that: (a) is secure against Fakey, Griefing and Wormhole attacks and (b) can detect, punish or prevent adversaries from launching these attacks.

3 | Background

This section provides a technical overview of the Lightning Network's channel construction, payment routing, detailed payment processing and some of the possible attacks on Lightning nodes in multi-hop payments.

3.1 | Lightning Network

We first discuss how channels are established between Lightning nodes and then describe payment routing using HTLCs.

3.1.1 | Channel Construction Details

A node (say, U_0) joining the Lightning Network for the first time must establish a network connection to a node that is already connected to Lightning's peer-to-peer overlay network. Each node in the Lightning Network is identified by a long-term secp256k1 public key [6]. The initial key exchange handshake, along with all inter-peer communications, is authenticated and encrypted based on the Noise protocol framework [24]. The node can establish a new payment channel with a neighbouring node (say, U_1) by

sending an 'open_channel' message. The neighbouring node U_1 responds to this request by sending 'an accept_channel' message. Through this exchange, both peers, U_0 and U_1, negotiate the channel capacity (α) and the initial balance of the channel (υ_0). Using this exchanged information, the initiating peer U_0 issues a funding transaction and a refund transaction. After the receiving node U_1 responds with a 'funding_signed' message, U_0 broadcasts the funding transaction to the Bitcoin network. This funding transaction is recorded on-chain, as shown in Step 1 of Figure 1.

Once the funding transaction is confirmed (with at least six confirmations) on the blockchain, the channel is opened. Thus, the channel with identifier $c_{<U_0,U_1>}$ becomes ready for future payments [6].

Furthermore, if U_0 wants to act as a node forwarding payments for others, it can announce its existence by sending a 'node_announcement' message and its channel's existence by sending a 'channel_announcement' message in the Lightning overlay network. These messages also contain the 'cltv_expiry_delta' field, which declares the maximum time a node is willing to have its funds locked in the payment channel. This announcement of the node and channel helps inform other peers in the network about the new node's (i.e., U_0's) channel capacity (α) and associated routing fees (rf).

3.1.2 | HTLC

A multi-hop payment is facilitated using the HTLC protocol. HTLC is a type of contract used to make conditional payments in the Lightning Network. HTLC locks an amount to a specific multisig address. The locking script in HTLC employs two main locks: a hashlock and a timelock. The HTLC locks an amount into a multisig-based contract that can be spent in three different ways by either the sender or the receiver.

First, the receiver can claim the amount by providing the pre-image for the hashlock used in the contract before a certain time or before a specific block height is reached. If the receiver fails to unlock the amount before the specified time (i.e., CheckLockTimeValue [CLTV] delta), then the sender can redeem the amount using the second method, which involves providing a valid signature after the CLTV delta has expired.

Third, the locked amount can be spent immediately by anyone (the sender or receiver) who provides a revocation key. Next, we will examine how multi-hop payments are processed using these HTLCs.

3.1.3 | Multi-Hop Payment Procedure

To understand payment routing in a multi-hop payment process, let us assume that user U_0 is already connected to several other nodes in the Lightning Network. Additionally, U_0 has established one or more payment channels with these nodes, allowing U_0 to send or receive payments. To make a payment to user U_4, to whom U_0 does not have a direct channel, it must find a suitable path in the network. Finding an appropriate path depends on various

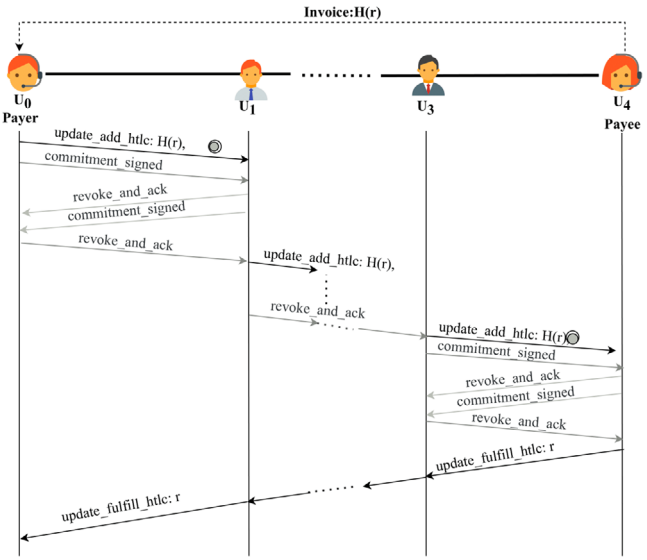

FIGURE 2 | Message exchanges during payment routing in Lightning Networks.

factors, such as a path with sufficient channel capacity, adequate liquidity (i.e., the available balance of the node after subtracting the channel reserve and any pending HTLCs committed by the node) low fees, and short timelocks. Once the path is determined, the node must set up the corresponding HTLCs to complete the payment.

We illustrate the sequence of exchanged messages between these nodes in Figure 2. Initially, U_0 and U_4 are supposed to communicate through a private channel over which U_4 can send an invoice to U_0. This invoice contains information about U_4's public key, the amount to be paid and the payment hash $H(r)$ of a randomly generated secret r (generated by U_4). Using this invoice and publicly available routing information, the payer U_0 determines a suitable path (say, a path with channel identifiers $[c_{<U_0,U_1>}, \ldots, c_{<U_3,U_4>}]$) to U_4 using source routing. Note that the Basis of Lightning Technologies (BOLT) specifications do not cover the behaviour of the source routing algorithm [25]. However, in source routing, route selection utilises routing fees and past payment success factors based on a modified version of Dijkstra's shortest path algorithm. If a path with sufficient channel capacities is not found, then U_0 is expected to try finding a suitable path. For now, let us assume that a path exists from U_0 to U_4 through the intermediate user nodes U_1, U_2 and U_3 as shown in Figure 2.

Using this path, U_0 initiates the HTLC construction with its next neighbouring user node, U_1. The HTLC construction enables a series of conditional payments, which can either be claimed by providing the pre-image r to the 'payment hash $H(r)$' or timed out after the expiration of a certain lock-time field. The payer, U_0, must calculate the amount of values υ_i each intermediate node should forward, as well as the remaining time-lock values t_i for each outgoing hop. These time-lock values decrease as they approach the payee, U_4. Next, U_0 encodes this information in an onion routing packet that follows the Sphinx packet format [26]. This packet format allows the user to construct the packet using multiple layers of encryption. Each layer of wrapping information

helps determine the next hop user, the amount to forward and the remaining time-lock value.

U_0 initiates the payment by sending an 'update_add_htlc' message, which carries the onion packet along with the payment hash $H(r)$ to the next neighbouring node, that is, user U_1. Along the payment path, each intermediate user decrypts the incoming packet, retrieves its payload and forward the HTLC offer to the next hop user. U_1 proceeds only after the new HTLC is incorporated into the new state of the payment channel, and this state change is irrevocable. Since these users share the revocation key of the previous commitment transaction through a handshake involving the 'commitment_signed' and 'revoke_and_ack' messages (refer to Figure 2). Once these pending state updates have been negotiated, U_1 forward the HTLC by attaching the remainder of the onion packet to an update_add_htlc message sent to the next hop, which proceeds in a similar manner. If any of the intermediate users do not agree with the payment due to insufficient balance, going offline or becoming unresponsive, the HTLCs will fail, and the node will inform the other nodes in the path by sending an update_fail_htlc message carrying a failure message. This failure message is onion encrypted and is forwarded along the backward path from the node that detected the failure until it reaches the original payer, U_0. These failures are uncertain; thus, the Lightning Network does not guarantee reliable payments.

Once the HTLC construction reaches the final payee, that is, U_4, it provides the pre-image secret r to the payment hash challenge $H(r)$ via an 'update_fulfill_htlc' message, which is then forwarded in the backward direction by each intermediate node. Consequently, each intermediary node claims its conditional payments, receives the determined fees and finally settles all pending HTLCs. The nodes may close the channel by sending a settlement transaction on-chain, as shown in Step 3 of Figure 1. This action moves the balance from the payment channel to the user's account on the blockchain.

3.2 | Possible Attacks on HTLC During Payment Routing

Here, we discuss some of the prominent attacks on HTLCs during multi-hop payments, which include Fakey, Griefing and Wormhole attacks. Fakey and Griefing attacks are intentionally launched by an adversary to fail the payment and lock the intermediaries' funds until their timelocks expire. In contrast, the Wormhole attack is carried out by malicious intermediate nodes to steal the routing fees from other honest intermediaries present on the routing path. Consider the HTLC example provided in Figure 3a, where U_0 wishes to make a payment to U_4 through intermediate users U_1, U_2 and U_3.

3.2.1 | Fakey Attack

In a Fakey attack, as shown in Figure 3b, a malicious payer (in this case, U_0) ignores the payment hash (the SHA-256 hash of the pre-image r) received from the payee U_4 and instead generates and shares a fake payment hash, denoted as F_H, with other users in the path. User U_1, unknowingly accepts F_H and forward it to next hop user U_2 (Steps 2–3). Similarly, U_2 and U_3 accept and forward F_H further (Steps 4–5). While U_4 possesses the pre-image r, its hash will not match F_H since U_0 has not forwarded the genuine payment hash P_H. As a result, U_4 will not finalise the payment, and the users along the path must wait for their timelocks to expire to redeem the locked funds [22].

3.2.2 | Griefing Attack

A Griefing attack is launched by a malicious receiver, who can be either a payee or any other intermediate node in the path. In Figure 3c, the malicious payee U_4 receives P_H but refuses to provide the pre-image r to U_3 in order to execute a Griefing attack. In this attack, the adversary may go offline or intentionally provide the wrong pre-image to its previous neighbour, ensuring that users along the path can only roll back their locked funds to their respective wallets after the expiration of their timelocks [23].

3.2.3 | Wormhole Attack

In this attack, two malicious users collude to prevent honest intermediate users from participating in a successful off-chain payment. These colluding users steal the transaction relay fees that are intended for the honest intermediate users. Figure 3d describes the Wormhole attack, where users U_1 and U_3 collude with each other. In this scenario, U_3 does not provide the secret r to U_2 to meet the HTLC contract condition in Step 4. Instead, U_3 bypasses its previous neighbour U_2 and shares the pre-image r directly with U_1, as indicated by the left red arrow in Step 7. As a result, the colluding users U_1 and U_3 steal the transaction relay fees that were intended for U_2 [12, 17].

4 | The Proposed Protocol

This section provides a detailed description of the CommTLC payment protocol. We first discuss the essential building blocks required for the proposed scheme, followed by a description of the various phases of the CommTLC. Finally, we provide a detailed discussion on detection of attacks in CommTLC.

4.1 | Building Blocks of CommTLC

We use the Pedersen commitment scheme [27], which is based on the Discrete Logarithm Problem (DLP). The reason for selecting Pedersen commitments is that they possess the additive homomorphic property, allowing us to prove that a secret committed value has not been tampered with, without revealing the secret.

4.1.1 | Pedersen Commitments

The Pedersen commitments scheme uses $\mathbb{Z}_p^*$, where p is a prime, as its basic algebraic structure. Let $\mathbb{G}_q$ be a subgroup of $\mathbb{Z}_p^*$ of order q, and g, h be elements of $\mathbb{G}_q$. A committer U_x commits to a secret message $s \in \mathbb{Z}_q$ by randomly choosing a trapdoor $r \in \mathbb{Z}_q$, generating commitment $c := \text{commit}(s, r) = g^s h^r (\text{mod } p)$. The group $\mathbb{G}$ and elements g, h are chosen by either a trusted third party or a verifier as suggested in reference [27, 28]. Here, we

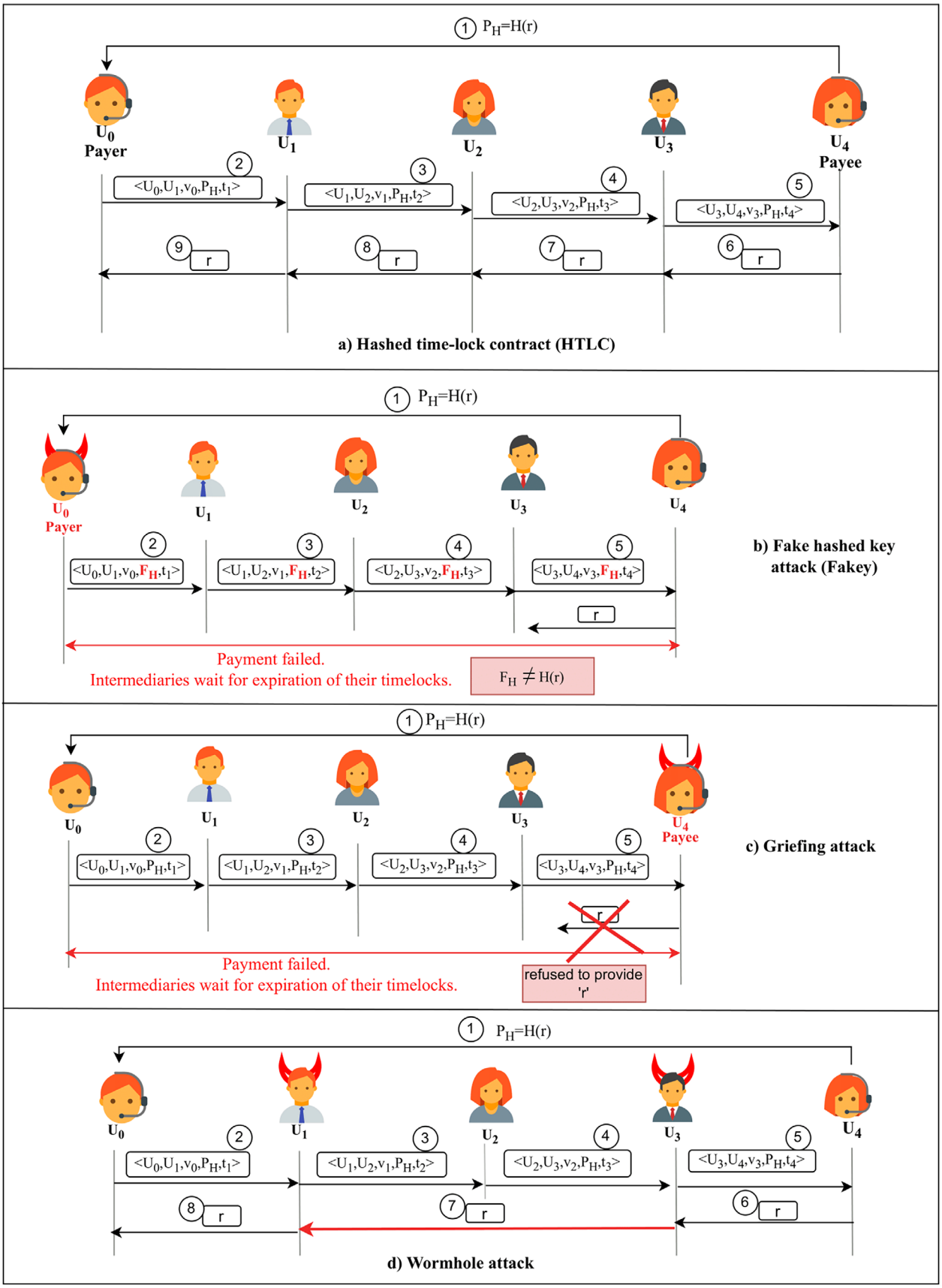

FIGURE 3 | Fakey, Griefing and Wormhole attacks on HTLC during payment routing in a multi-hop PCN. (a) The regular HTLC setup among five users is shown. (b) The malicious payer U_0 launches a Fakey attack, and (c) the malicious receiver launches a Griefing attack. This Griefing attack can also be launched by any intermediate node, which is not depicted here. (d) U_1 and U_3 collude to launch a Wormhole attack.

assume that the group $\mathbb{G}$ and elements g, h are chosen by a verifier and made available to the committer U_x.

The commitment c can be opened by U_y only when U_x provides (s, r) to U_y at later point of time. The Pedersen commitment scheme is unconditionally hiding and computationally binding [27]. The scheme is unconditionally hiding since the verifier cannot learn any information about s from c. The scheme is computationally binding, since the committer cannot use some other $\bar{s}$ to generate the same c, unless the committer is able to solve the DLP [28].

The Additive homomorphic property allows us to multiply two commitments of unknown values and to calculate the commitment of their sum. Assume that, U_x has generated two commitments c_i, c_j where it committed messages s_i, s_j using trapdoors r_i, r_j, respectively. The resulting commitments are $c_i \leftarrow \text{commit}(s_i, r_i)$ and $c_j \leftarrow \text{commit}(s_j, r_j)$. One can generate a new commitment c_k from previous commitments c_i, c_j using the homomorphic property as follows:

$$\begin{aligned} c_k &= c_i.c_j \\ &= \text{commit}(s_i, r_i).\text{commit}(s_j, r_j) \\ &= (g^{s_i}.h^{r_i}).(g^{s_j}.h^{r_j}) \\ &= (g^{s_i+s_j}.h^{r_i+r_j}) \\ &= \text{commit}(s_i + s_j, r_i + r_j) \end{aligned}$$

4.2 | CommTLC Details

This subsection presents CommTLC in detail. We consider an indirect payment from a payer U_0 to a payee U_4, along the payment path $\mathcal{P} := \{U_0 \rightarrow U_1 \rightarrow U_2 \rightarrow U_3 \rightarrow U_4\}$. U_0 wishes to get some service for payment amount υ_3 from U_4 via these intermediate users $\{U_i\}_{i\in[1,3]}$ as shown in Figure 4.

The three primary phases of CommTLC: pre-setup, setup and payment are discussed here.

4.2.1 | P1: Pre-Setup Phase

This phase is performed between payer (U_0) and payee (U_4). Initially, payee U_4 samples a secret message s_4 and trapdoor r_4 and generates the corresponding commitment $c_4 \leftarrow \text{commit}(s_4, r_4)$. Next, U_4 signs c_4 and shares c_4 and the signed version of it ($\text{SIGN}_{U_4}(c_4)$) with the payer U_0 (Step 1 in Figure 4). Similarly, payer U_0 generates the commitment $c_0 \leftarrow \text{commit}(s_0, r_0)$ and shares (a) c_0, (b) signed version of it ($\text{SIGN}_{U_0}(c_0)$), (c) the signed version of the received commitment c_4 ($\text{SIGN}_{U_0}(c_4)$) and (d) the required information to open c_0, that is, $\{s_0, r_0\}$ with the payee U_4 (Step 2).

4.2.2 | P2: Setup Phase

This phase is performed only by the payer U_0. It chooses three secret messages and corresponding trapdoors, $\{s_i, r_i\} \in Z_q$ where $i \in [1, 3]$ for each intermediate user present in the path $\mathcal{P}$ (Step 3). Note that CommTLC does not allow U_0 to select a secret message for U_4. These secrets $\{s_i, r_i\}$ corresponding to each intermediate user U_i, $i \in [1, 3]$, are shared in the payment phase via onion routing—thus, additional overhead is avoided. Using these generated secrets U_0 computes distinct commitments for U_1, U_2 and U_3 as follows:

$c_1 \leftarrow \text{commit}(s_1, r_1)$, $c_2 \leftarrow \text{commit}(s_2, r_2)$, and $c_3 \leftarrow \text{commit}(s_3, r_3)$. Further, U_0 utilizes these commitments and generates $\phi_4, \phi_3, \phi_2, \phi_1$ as follows — $\phi_4 := c_0.c_4$, $\phi_3 := \phi_4.c_3$, $\phi_2 := \phi_3.c_2$, and $\phi_1 := \phi_2.c_1$

Next, U_0 executes the last phase of CommTLC protocol, the Payment phase.

4.2.3 | P3: Payment Phase

The Payment phase has two sub-phases, Payment – lock and Payment – release.

4.2.3.1 | Payment-Lock. In this phase, payer U_0 uses standard onion routing [29] for passing the information needed by each user U_1 to U_4 present on the path $\mathcal{P}$. Steps 4–7 in Figure 4 show the payment locking phase.

U_0 constructs an onion packet for the next hop user U_1, includes the onion packet in m_0, and sends m_0 via 'update_add_commtlc' request. m_0 is defined as follows: $m_0 = E(E(E(E(z_4, pk_4), z_3, pk_3), z_2, pk_2), z_1, pk_1)$, where $z_i = (\phi_i, \upsilon_{i-1}, t_i, s_i, r_i, U_{i+1})$, $i \in [1, 3]$ and $z_4 = (\phi_4, \upsilon_3, t_4, \text{null})$. Here, $m_{i-1} = E(m_i, z_i, pk_i)$ is the encryption of the message m_i and z_i using public key pk_i of user U_i. This request message signals to U_1 that U_0 would like to establish a new CommTLC contract with it, containing payment amount υ_0 (in millisatoshis [msat]) over this channel. The onion_routing_packet contained in m_0 has information for U_1 about its next hop user U_2. U_1 decrypts m_0, gets m_1 and z_1. Once U_0 and U_1 both agree, U_0 creates a CommTLC contract that is, $\text{CommTLC1}(U_0, U_1, \upsilon_0, \{\phi_1, (s_1, r_1)\}, t_1)$ with its neighbour U_1 (shown in Step 4). Then, U_1 forward message $m_1 = (E(E(E(z_4, pk_4), z_3, pk_3), z_2, pk_2)$ to *itsneighbour* U_2. This continues till user U_4 receives $E(z_4, pk_4)$.

Similar to CommTLC1 created by U_0, other users on the path $\mathcal{P}$: U_1, U_2 and U_3 also create CommTLC contracts, $\text{CommTLC2}(U_1, U_2, \upsilon_1, \{\phi_2, (s_2, r_2)\}, t_2)$, $\text{CommTLC3}(U_2, U_3, \upsilon_2, \{\phi_3, (s_3, r_3)\}, t_3)$, $\text{CommTLC4}(U_3, U_4, \upsilon_3, \phi_4, t_4)$ with their neighbours U_2, U_3 and U_4, respectively (Steps 5–7).

4.2.3.2 | Payment-Release. Once the payment contract CommTLC4 is received by U_4, the payment-release sub-phase begins. U_4 checks the correctness of CommTLC4 by verifying the timelock ($t_4 > t_{\text{now}} + \Delta$) mentioned in the payment contract $\text{CommTLC4}(U_3, U_4, \upsilon_3, \phi_4, t_4)$ received from U_3. If this is correct, then U_4 releases the secret-trapdoor pairs $\{(s_0, r_0), (s_4, r_4)\}$ to U_3 (Step 8). U_3 checks whether the contract condition, that is, $<\phi_4 == \text{commit}(s_0 + s_4, r_0 + r_4)>$ is satisfied or not. If it is valid then U_3 takes $\{(s_0, r_0), (s_4, r_4)\}$ from U_4, adds its secret (s_3, r_3) to it and provides $\{(s_0, r_0), (s_4, r_4), (s_3, r_3)\}$ to U_2 (Step 9). Each intermediate user is supposed to provide secrets received from its right neighbour along with its own secret to its left neighbour.

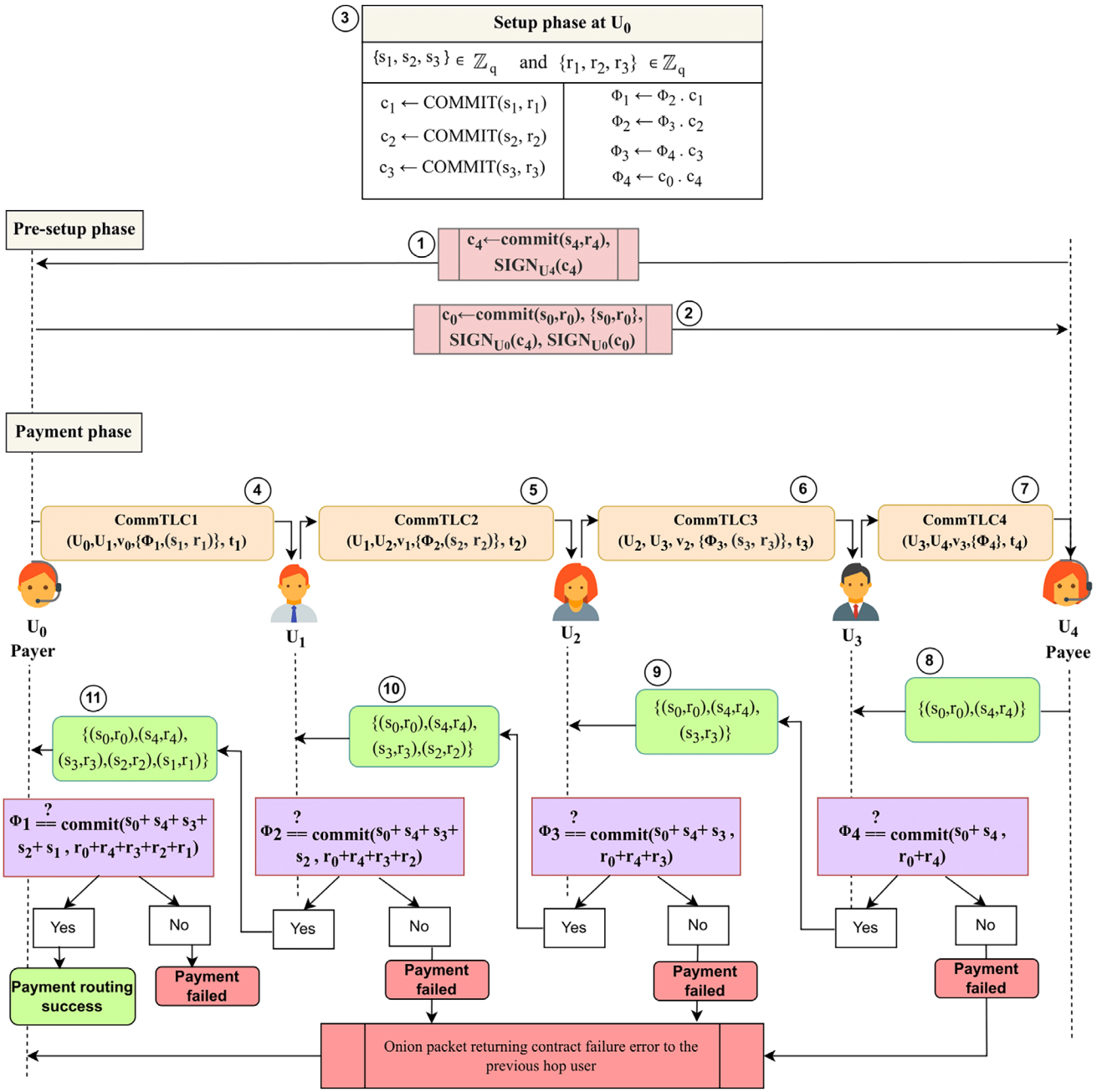

FIGURE 4 | CommTLC protocol overview.

This process continues (as given in Steps 9–11) until original sender receives all secret/trapdoor pairs.

Lastly, user U_1 provides secret–trapdoor pairs $\{(s_i, r_i)\}_i \in \{0, 4\}$ within time t_1 to U_0. If it satisfies the contract condition, $< \phi_1 == \text{commit}(\Sigma_{i=0}^{4} s_i, \Sigma_{i=0}^{4} r_i) >$, then U_0 releases v_0 coins to U_1. Next, each node present in the path $\mathcal{P}$ claims the committed coins from their respective CommTLCs (similar to HTLCs).

4.3 | Detection of Attacks

We discuss here at what stage of the CommTLC protocol, the three attacks (defined in Section 3.2) get detected.

- **Fakey attack**: The malicious payer can launch a Fakey attack by replacing c_0 with $\overline{c_0}$ or c_4 with $\overline{c_4}$ or $c_0.c_4$ with $\overline{c_0.c_4}$ while forming an onion packet. Any of these changes results in a different commitment, that is, $\overline{\phi_4}$ instead of the actual commitment ϕ_4 in step 7 of Figure 4. None of the users present in the path knows about what message they are forwarding to their next hop, and unknowingly, the fake commitment $\overline{\phi_4}$ is being forwarded to payee U_4 via intermediaries. Once the onion packet reaches payee U_4, it can detect that the commitment has been modified because $\overline{\phi_4} \neq c_0.c_4$. Since U_4 has both commitments c_0 and c_4 signed by U_0 from the Pre-setup phase, U_4 can identify that a Fakey attack has been launched by U_0. Moreover, U_0 cannot deny signing c_0 and c_4 due to the non-repudiation property of digital signatures.

To punish U_0, the payee U_4 takes the original c_4, $\text{SIGN}_{U_0}^{c_4}$, and retrieves the IP/TOR address of U_0 using its public key and channel ID, and broadcasts a proof of cheating as < channelID_{U_0}, IP addr_{U_0}, $c_4, c_0, \text{SIGN}_{U_0}^{c_4}, \text{SIGN}_{U_4}^{c_4}, \overline{\phi_4}$, contract _failure_mesg > to the PCN. After receiving the proof of cheating, all nodes in the PCN verify that c_4 is signed by both U_0 and U_4, but observe that $c_0.c_4 \neq \overline{\phi_4}$, which results in a contract failure. A PCN node has enough evidence to determine that U_0 is an adversary who launched the Fakey attack.

- **Griefing attack**: This attack is similar to the Fakey attack except that the adversary is the payee. The malicious payee can be either U_4 or any other intermediate user $U_i, i \in [1, 3]$. If U_4 is malicious, then it can either refuse to provide secrets to unlock ϕ_4 or provide incorrect secret values to U_3. This results in contract failure, and payer U_0 receives an error message enclosed in the onion packet. The packet also contains a field 'failure_src_index', which helps U_0 to identify the adversary, U_4, who intentionally provided incorrect or no secrets to U_3. Additionally, U_0 has U_4's signature on the commitment c_4, allowing U_0 to detect the Griefing attack and the adversary responsible for it.
  If the payee is any intermediate node U_i, $i \in [1, 3]$ and provides incorrect secrets to U_{i-1}, then U_{i-1} cannot unlock the payment and sends a contract failure error message with the received secrets along the backward path. This error message is transmitted via onion routing. When the onion packet containing the error message reaches U_0, who possesses the secrets from each intermediate node, U_0 can verify that the contract between U_i and U_{i+1} was executed successfully, but the contract between U_i and U_{i-1} has failed. As a result, U_0 can detect that U_i is malicious and has launched the Griefing attack.
  To punish U_i, U_0 broadcasts proof of cheating to the PCN, similar to the punishment process in a Fakey attack.
- **Wormhole attack**: Suppose U_3 and U_1 are malicious and collude to launch a Wormhole attack in CommTLC. When U_3 shares unlocking information with U_1, bypassing intermediate user U_2, the CommTLC1 contract between U_0 and U_1 fails. U_1 cannot unlock the payment from U_0 because the secrets provided to U_0 do not include U_2's secret. Consequently, U_1 sends a contract failure error message to payer U_0. At this point, U_0 can detect that U_2's (s_2, r_2) (secret/trapdoor) pair is missing since U_0 had provided secret message pairs to each intermediate user during the Setup phase. By computing the commitments using the secret message pairs received from U_1 (in the backward direction) up to the user who sent the contract failure message, U_0 identifies the absence of (s_2, r_2). In this case, due to the payment failure, none of the intermediate nodes receive routing fees, thereby preventing the Wormhole attack in CommTLC.

5 | Performance Analysis

In this section, we discuss the experimental setup, implementation and deployment integration details. Furthermore, we evaluate the performance of the CommTLC scheme in terms of communication and computational overhead.

5.1 | Experimental Setup

We implement the CommTLC scheme using btcd nodes running on a separate Bitcoin network (btcd v0.23.1). Lightning nodes are set up over btcd nodes using the LND implementation in testnet mode, with Go v1.22.1 as the programming language.

The cryptographic operations required by CommTLC are implemented using Python and the Charm library [30]. Pedersen's commitment is implemented using modular exponentiation instead of elliptic curve cryptography (ECC), though an ECC-based implementation can be easily adapted. All experimental results are obtained from a computer equipped with an Intel Core i7-11700 processor, 16 GiB of RAM and 1 TB of disk capacity, running the Ubuntu 20.04 operating system.

5.2 | Implementation Details

The lightning nodes communicate using various messages specified in the BOLT [31], which we reference for demonstration purposes. We set up a host machine running a btcd node connected to the Bitcoin network in testnet mode. Next, we installed four virtual machines (VMs), each running LND instances on different ports. Specifically, VM0 represents user U_0 (the original payer), VM1 represents U_1, VM2 represents U_2 and VM3 represents U_3 (the payee). The btcd node needs to be synchronized with the Bitcoin testnet. The current block height of the Bitcoin testnet can be verified using the Bitcoin Testnet Explorer.[1] At the time of writing, the block height was 3,051,011.

After synchronization, we run LND instances on each user's virtual machine. Next, we create a wallet for each user: U_0, U_1, U_2 and U_3. Upon wallet creation, each user obtains a public/private key pair. We then generate Bitcoin addresses for each user using the 'Pay to Nested Witness Key Hash (NP2WKH)' script. These Bitcoin addresses are used to receive funds in each user's wallet through the Bitcoin testnet faucet.[2] These funds are necessary to create payment channels for the users. U_0, U_1, U_2 and U_3 receive 0.98601954, 0.87601598, 0.64567680 and 0.66450456 bitcoins into their respective wallets: wallet_u0, wallet_u1, wallet_u2 and wallet_u3, from the testnet faucet. However, these are unconfirmed balances. Each user must wait for the transaction to be confirmed to convert the unconfirmed balance into a confirmed one in their wallet.

Next, we connected node U_3 to U_2, U_2 to U_1 and U_1 to U_0. We verified these connections by checking their peer lists. Then, we opened a channel between U_0 and U_1, depositing 150,000,000 msat, where U_0 holds a stake of 100,000,000 msat and U_1 holds 50,000,000 msat. U_0 holds a slightly higher balance to cover the routing fees. The channel creation generates a funding transaction, and the opened channel is announced to other nodes in the network by broadcasting a 'channel_announce' message. Similarly, two more channels are opened: one between U_1 and U_2, and another between U_2 and U_3, with each user holding a stake of 50,000,000 msat.

Once all channels are set up, we configure the routing path at U_0. U_0 informs U_3 that it wishes to make a payment in exchange for

a service from U_3. In response, U_3 generates an invoice for U_1, which contains c_3, $\text{SIGN}_{U_3}(c_3)$ and the 'payment_req'. Next, U_0 uses the received "payment_req" and calls the send_payment() function, which sends an 'update_add_commtlc' message to U_1. The 'update_add_commtlc' message includes the 'onion_routing_packet' field. In the CommTLC contract, we replace the 'payment hash' field from the HTLC contract with the 'product of two commitments'. Following the nested structure of onion routing, the 'onion_routing_packet' field carries 'update_add_commtlc' for the channels between U_1 and U_2, and U_2 and U_3. Once the 'update_add_commtlc' message reaches its destination, U_3 provides the secret–trapdoor pairs (s_0, r_0) and (s_3, r_3) in the 'update_fulfil_commtlc' message to claim the funds locked in the CommTLC contract between U_2 and U_3. An additional 32 bytes are used in the data part of the 'update_fulfil_commtlc' message to supply the secret–trapdoor pairs.

5.3 | Deployment Integration Details

This subsection addresses the integration of CommTLC with the Lightning Network's LND implementation, highlighting challenges, upgrade paths, backward compatibility and network configuration.

5.3.1 | Integration Challenges and Solutions

Key challenges include extending Bitcoin Script to introduce new opcodes, updating LND's cryptographic utilities for Pedersen commitments and modifying the LND database (channeldb) to store commitment metadata. To overcome these challenges, we extended Bitcoin Script to support on-chain commitment operations, ensuring auditability and trust without relying on off-chain mechanisms. We update the script logic by modifying inputs/script_utils.go and htlc.go files to implement Pedersen-specific logic. We also introduced new opcodes for commitment generation (op_commgen), multiplication (op_commmul) and validation (op_commverify) by modifying the opcode.go file in the btcd repository. For Cryptographic Support, we extended lnd/input and lnd/crypto files for compatibility with the secp256k1 curve. For routing and payment, we updated the lnwallet, lnwire and htlcmanager modules to support CommTLC while preserving HTLC functionality.

5.3.2 | Network Configuration and Parameter Selection

To enable commitment-based routing, we added a configuration option (–enable-commitment-routing) in the config.go file, allowing nodes to opt into CommTLC functionality. Updates were made to the lnwire and htlcmanager modules to handle the product of commitments and to the lnwallet/channel.go and lnwallet/script_utils.go scripts for adding, settling and failing CommTLC payments. We defined a 33-byte commitment size and adjusted the cltv_expiry_delta to 42–45 blocks, which is 2–5 blocks more than the 40 blocks used in HTLCs.

5.3.3 | Backward Compatibility

To ensure backward compatibility, we enabled LND to support both HTLC and CommTLC. Conditional logic in link.go processes payments based on the type, while a feature flag introduced during the channel reservation process (lnrpc/lightning.proto, lnrpc/rpcserver.go, lnwallet/channel_reservation.go) identifies the HTLC type. The pathfinding algorithm was modified to prioritize a single HTLC type for multi-hop payments, preventing incompatibility-related failures.

5.3.4 | Incremental Upgrade Path

The changes were implemented incrementally and validated through rigorous unit and integration testing, including end-to-end tests added in itest/lnd_test.go to ensure smooth deployment and reliable functionality.

Thus, this integration enables LND to support CommTLC alongside HTLC.

5.4 | Evaluation

In this subsection, we assess the communication overhead and computation costs of CommTLC. Additionally, we compare the performance of CommTLC with the latest schemes, MAPPCN-OR [32] and EAMHL+ [14].

5.4.1 | Communication Overhead

Let there be $n+1$ users $\{U_0, U_1, \ldots, U_n\}$ present in the payment path. The communication overhead is determined by the amount of data exchanged by the users on a payment path during the execution of CommTLC. This overhead falls into four categories: Pre-setup, Setup, Payment-lock and Payment-release, as shown in Table 3. The overhead incurred during the Payment-lock phase is common to other payment channel-based solutions that use onion routing, such as [13, 14, 32]. Therefore, the remaining three phases—Pre-setup, Setup, and Payment-release—introduce additional overhead specific to CommTLC. In CommTLC, there is no communication overhead associated with the Setup phase, so the communication overhead incurred by CommTLC is specifically due to the Pre-setup and Payment-release phases.

Table 3 compares the communication overhead (theoretically estimated) of CommTLC with MAPPCN-OR and EAMHL+, where $|m|$ denotes the bit length of m; |CM| denotes the commitment length which commits to a secret value; |OnionEnc| denotes the bit length of the ciphertext generated by the onion encryption.

In CommTLC, during the pre-setup phase, the size of a signed commitment is considered to be 64 bytes for 128-bit security. Thus, signed commitments are regarded as being of constant size. In the Payment-lock phase, secret/trapdoor pairs are transmitted via an onion packet, so the communication overhead in the payment-lock phase of CommTLC is just n|OnionEnc|. Similarly, MAPPCN-OR sends a random value r_i through the onion packet,

TABLE 3 | Theoretical comparison of communication overhead in CommTLC, MAPPCN-OR and EAMHL+.

Phase	CommTLC	MAPPCN-OR	EAMHL+
Pre-setup phase	$2\|\mathrm{CM}\| + \|\mathbb{Z}_q\| + c$	—	—
Setup phase	—	$\|\sum_{i=0}^{n-1} r_i\|$	$\|G_T\|$
Payment-lock	$n\|\mathrm{OnionEnc}\|$	$n(\|\mathrm{OnionEnc}\| + \|P\| + \|Q\|)$	$n(\|\mathrm{OnionEnc}\| + \|G_T\| + \|G_1\|)$
Payment-release	$n\|\mathbb{Z}_q\|$	$n\|\mathbb{Z}_q\|$	$n\|G_1\|$

TABLE 4 | Theoretical comparison of computational overhead in CommTLC, MAPPCN-OR and EAMHL+.

Phase	CommTLC	MAPPCN-OR	EAMHL+
Pre-setup phase	$2\mathrm{T}_{\mathrm{Comm.gen}} + 3\mathrm{T}_{\mathrm{Comm.sign}} + 2\|\mathbb{Z}_q\|$	—	—
Setup phase	$(n-2)(\|\mathbb{Z}_q\| + \mathrm{T}_{\mathrm{Comm.gen}}) + (n-1)\mathrm{T}_{\mathrm{Comm.mul}}$	$T_{\sum_{i=0}^{n-1} r_i}$	T_{pair}
Payment-Lock	$(n-1)(\mathrm{T}_{\mathrm{PKY.Enc}} + \mathrm{T}_{\mathrm{PKY.Dec}})$	$n(\mathrm{T}_{\mathrm{PKY.Enc}} + \mathrm{T}_{\mathrm{S.Enc}} + \mathrm{T}_{\mathrm{PKY.Dec}} + \mathrm{T}_{\mathrm{S.Dec}} + \mathrm{T}_{\mathrm{smec}} + \mathrm{T}_{\mathrm{eca}})$	$n(\mathrm{T}_{\mathrm{PKY.Enc}} + \mathrm{T}_{\mathrm{S.Enc}} + \mathrm{T}_{\mathrm{PKY.Dec}} + \mathrm{T}_{\mathrm{S.Dec}}) + 2n\mathrm{T}_{\mathrm{exp}}$
Payment-Release	$(2n+2)\mathrm{T}_{\mathrm{exp}}$	$n(\mathrm{T}_{\mathrm{smec}} + \mathrm{T}_{\mathrm{eca}})$	$n(\mathrm{T}_{\mathrm{exp}} + \mathrm{T}_{\mathrm{pair}})$

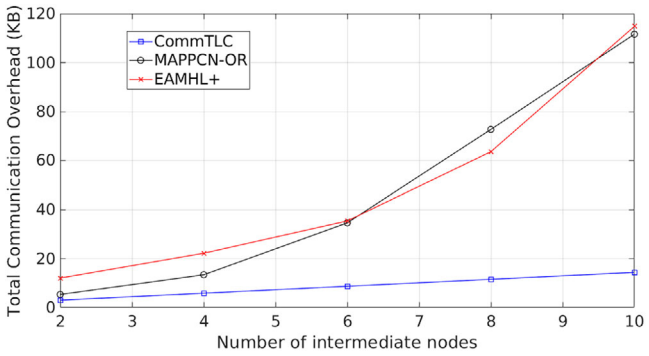

FIGURE 5 | Total communication overhead in CommTLC, MAPPCN-OR and EAMHL+.

but additional elements computed using elliptic curve points are sent separately, resulting in higher communication overhead. In contrast, in EAMHL+, random numbers are selected by the intermediate users, and these random numbers, along with some group elements and the onion packet, cause additional overhead. The experimental evaluation of the total communication overhead in CommTLC, MAPPCN-OR, and EAMHL+ is presented in Figure 5. The figure demonstrates that the communication overhead in CommTLC is significantly lower than in MAPPCN-OR and EAMHL+, making CommTLC more suitable for practical use.

5.4.2 | Computational Cost

Let there be $n+1$ users $\{U_0, U_1, \ldots, U_n\}$ present in the payment path. Similar to communication overhead, computational overhead also falls into the following four categories: Pre-setup, Setup, Payment-lock and Payment-release. We compare the total computational overhead of CommTLC with MAPPCN-OR and EAMHL+, as theoretically determined in Table 4.

We use $\mathrm{T}_{\mathrm{Comm.gen}}$ to represent the time overhead for generating the commitment for a single secret message, $\mathrm{T}_{\mathrm{Comm.sign}}$ for the time overhead of signing a commitment and $\mathrm{T}_{\mathrm{Comm.mul}}$ to represent the time overhead for computing the product of two commitments in the finite field. The encryption and decryption overheads of public key encryption in onion encryption are denoted by $\mathrm{T}_{\mathrm{PKY.Enc}}$ and $\mathrm{T}_{\mathrm{PKY.Dec}}$, respectively. Likewise, $\mathrm{T}_{\mathrm{exp}}$ represents the time overhead for modular exponentiation in the finite field.

Additionally, MAPPCN-OR and EAMHL+ use $\mathrm{T}_{\mathrm{S.Enc}}$ to represent the symmetric encryption overhead and $\mathrm{T}_{\mathrm{S.Dec}}$ for symmetric decryption overhead. $\mathrm{T}_{\mathrm{pair}}$ represents the time overhead of bilinear pairing operations. In MAPPCN-OR, $\mathrm{T}_{\mathrm{smec}}$ and $\mathrm{T}_{\mathrm{eca}}$ represent the time overhead of scalar multiplication and addition operations on the elliptic curve, respectively.

The pre-setup and setup phases introduce additional overhead in CommTLC, but the payment-lock phase causes less overhead compared to other methods. In the payment-lock phase of CommTLC, the overhead is only due to public key encryption and decryption, whereas in MAPPCN-OR, there is additional overhead from symmetric key encryption and decryption, as well as scalar multiplication and addition operations on the elliptic curve. Therefore, the computational overhead in the payment-lock phase of CommTLC is lower than that of both MAPPCN-OR and EAMHL+. In the payment-release phase, CommTLC uses the additive homomorphic property, so the addition operation in the group takes constant time. The overhead is caused only by modular exponentiation. On the other hand, in the same phase, MAPPCN-OR has comparatively lower overhead because all operations are performed on the elliptic curve. EAMHL+, however, incurs high overhead due to operations like pairing and field inversion over the finite field. We present the experimental evaluation of computational overhead in Figure 6. Although CommTLC has lower computational overhead in the payment-lock phase, its total computational overhead is slightly higher

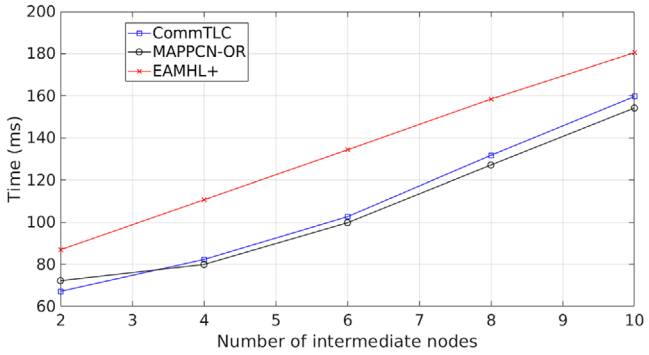

FIGURE 6 | Total computational overhead in CommTLC, MAPPCN-OR and EAMHL+.

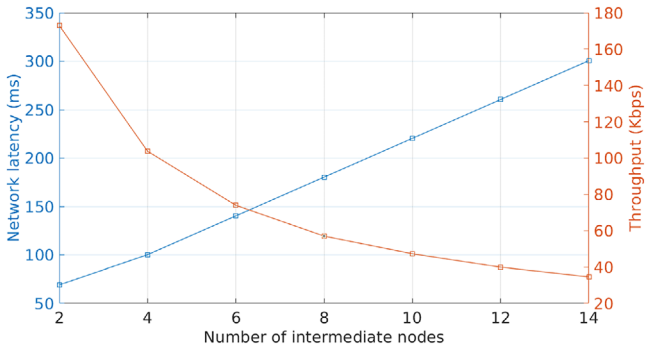

FIGURE 7 | Impact of load on network latency and throughput in CommTLC.

than that of MAPPCN-OR but significantly lower than that of EAMHL+.

To analyse network latency and throughput under stress conditions, we (1) increased the number of open channels from 50 to 200 for all the nodes present on the path and (2) varied processing and queuing delays at each intermediate hop. The resulting graph of network latency and throughput versus number of intermediate nodes in CommTLC for the current setup with 14 intermediate nodes is shown in Figure 7. The bulk of the time needed to serve a packet in CommTLC arises from the processing of packets in onion routing (encryption/decryption) and the setup/verification time in CommTLC contracts. We have found that the time from receiving an onion packet from previous neighbour to sending it to the next neighbour with four active HTLCs is 20.04 ms. We observe that network latency increases with the number of hops. For two intermediate nodes, each having four active HTLCs, the latency is 69.1 ms, while for 14 intermediate nodes, it is 300.60 ms. However, CommTLC remains practical, as LND typically uses an average of three to five hops.

5.4.3 | Adversary Detection Time

We present the time taken by an adversary to launch a Fakey attack, a Griefing attack and a Wormhole attack in Table 5. In a Fakey attack, where U_0 is malicious, the attack is detected by U_4 in 65.28 ms in our setup with five users from U_0 to U_4. In a Griefing$_i$ attack with $i = 4$, where the adversary is U_4, the detection time is 85 ms. However, when the adversary is U_2 (for $i = 2$), the detection time increases to 101 ms. Lastly, in the case of a Wormhole attack, where U_1 and U_3 are colluding malicious nodes, the detection time is 111.5 ms.

TABLE 5 | Time taken to detect an adversary launching Fakey, Griefing and Wormhole attacks in CommTLC estimated for a setup with three intermediate users.

Attacks	Adversary detection time
Fakey	65.28 ms
Griefing$_i$	85 ms for $i = 4$, 101 ms for $i = 2$
Wormhole	111.5 ms

Note: In Griefing$_i$, the i-th user in the payment path is malicious and attempts to carry out a Griefing attack.

6 | Security Analysis

In this section, we provide the security model and show that the proposed CommTLC payment protocol is secure in the Universally Composable (UC) security paradigm, as defined by Canetti [33].

The Universal Composition Theorem: Let ρ, ϕ and π be protocols such that ϕ is a subroutine of ρ, π UC-emulates ϕ and π is identity compatible with ρ and ϕ. Then, the protocol $\rho^{\phi\to\pi}$ UC-emulates ρ [33].

Definition of UC-emulation: Given a security parameter λ, the protocol Π UC-emulates protocol $\mathcal{F}$ if for any computationally bounded adversary $\mathcal{A}$, there exists a probabilistic polynomial time (PPT) simulator $\mathcal{S}$ such that for any PPT environment $\mathcal{Z}$, the ensembles IDEAL$_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ and REAL$_{\Pi,\mathcal{A},\mathcal{Z}}$ are computationally indistinguishable.

Before applying the Universal Composition theorem to the CommTLC protocol, we describe the real-world entities, the ideal functionality model and the adversary model for the CommTLC protocol.

We closely follow the terminology introduced by Canetti [33], in which a protocol is viewed as a set of interacting machines. Furthermore, each machine is assigned an identity ID, a communication set C and a program μ that it executes.

6.1 | Real World Entities

We have five user machines with identities ID$_0$, ID$_1$, ID$_2$, ID$_3$ and ID$_4$. Each machine implements the CommTLC protocol; we denote the CommTLC protocol by Π_{Commtlc}.

6.2 | Adversary Model

We assume that there exists a PPT adversary $\mathcal{A}$, capable of efficiently corrupting users in the payment channel network. $\mathcal{A}$

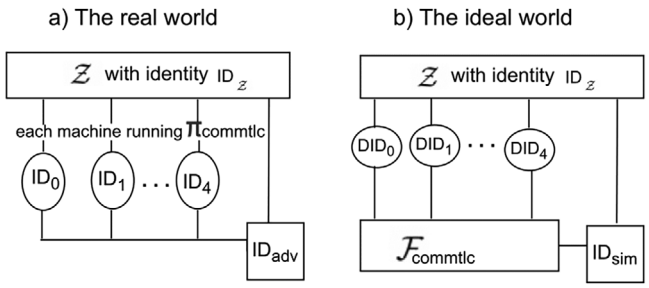

FIGURE 8 | The ideal and real worlds in CommTLC using the UC framework. In the real world, the environment $\mathcal{Z}$, with identity $\mathrm{ID}_{\mathcal{Z}}$, provides input to and receives output from protocol machines with identities ID_0, ID_1, ID_2, ID_3, and ID_4. $\mathcal{Z}$ can also interact with adversary $\mathcal{A}$, which has the identity ID_{adv}. In the ideal world, the oval shapes represent dummy user machines $(\mathrm{DID}_0, \ldots, \mathrm{DID}_4)$ that pass inputs directly from $\mathcal{Z}$ to $\mathcal{F}_{\mathrm{commtlc}}$ and relay outputs from $\mathcal{F}_{\mathrm{commtlc}}$ back to $\mathcal{Z}$. The simulator $\mathcal{S}$, with identity ID_{sim}, acts as the adversary in the ideal world, interacting with both $\mathcal{Z}$ and $\mathcal{F}_{\mathrm{commtlc}}$.

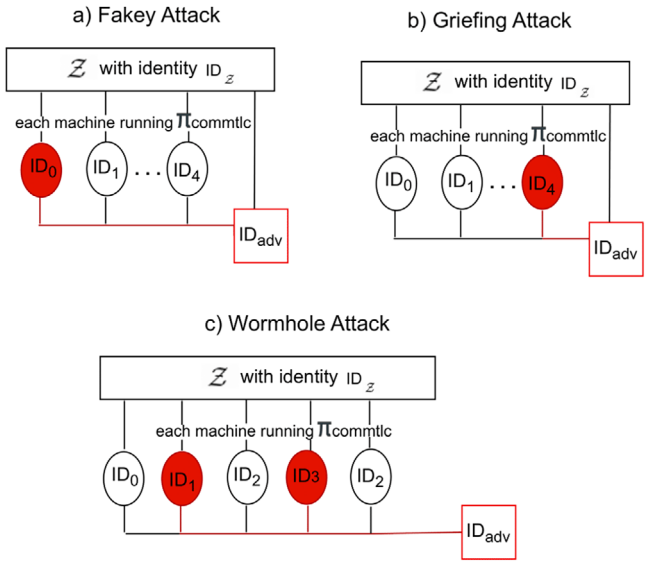

FIGURE 9 | Fakey, Griefing and Wormhole attacks in the real-world setting using the UC framework.

can extract the internal states of corrupted users and intercept all information passing through them. These corrupted users may collude to steal relay fees from honest intermediaries or lock the collateral of other users along the payment path. This prevents the remaining users from participating in any further payment executions by withholding or providing incorrect secret values. $\mathcal{A}$ can also send arbitrary messages while impersonating any corrupted user. However, communication among honest (not corrupted) users takes place over secure channels, making it inaccessible to $\mathcal{A}$. The payer and payee have a secure channel between them, but neither has a secure channel with the intermediate nodes. The intermediate nodes only know their previous and next hop users on the payment path. Here, we focus on internal adversaries, such as the payer, payee or one or more intermediaries.

6.3 | Ideal Function Model

The ideal world functionality $\mathcal{F}_{\mathrm{commtlc}}$ is built on top of other ideal functionalities, utilizing $\{\mathcal{F}_{\mathrm{smt}}, \mathcal{F}_{\mathrm{com}}, \mathcal{F}_{\mathrm{dsa}}\}$ as subroutines. The ideal functionality $\mathcal{F}_{\mathrm{smt}}$ models secure message transmission between two users, ensuring that messages are transmitted securely and remain confidential. $\mathrm{Send}_{\mathrm{smt}}$ interface is defined to send messages via $\mathcal{F}_{\mathrm{smt}}$. Next, the ideal functionality $\mathcal{F}_{\mathrm{com}}$ models a commitment scheme that generates a commitment c, which can be verified later by revealing the secret message and trapdoor. $\mathcal{F}_{\mathrm{dsa}}$ models the secure signing functionality. Additionally, there are dummy user machines, denoted as $\mathrm{DID}_0, \ldots \mathrm{DID}_4$, which relay inputs to $\mathcal{F}_{\mathrm{commtlc}}$ and outputs from $\mathcal{F}_{\mathrm{commtlc}}$ to the caller machine. Finally, following reference [33], we introduce the protocol ρ_{commtlc}, which is a protocol that calls another protocol implementing the functionality $\mathcal{F}_{\mathrm{commtlc}}$.

Figure 8 shows a picture of the real and ideal worlds in CommTLC using the UC framework.

Theorem 1. *Let ρ_{commtlc}, $\mathcal{F}_{\mathrm{commtlc}}$ and Π_{commtlc} be protocols. Then the following are true:*

- *Claim 1 – $\mathcal{F}_{\mathrm{commtlc}}$ is a subroutine of ρ_{commtlc}.*
- *Claim 2 – Π_{commtlc} UC-emulates $\mathcal{F}_{\mathrm{commtlc}}$ in the $\{\mathcal{F}_{\mathrm{smt}}, \mathcal{F}_{\mathrm{com}}, \mathcal{F}_{\mathrm{dsa}}\}$- hybrid model provided that (a) digital signature scheme is computationally unforgeable, (b) Pedersen commitments are secure (under the DLP assumption).*
- *Claim 3 – Π_{commtlc} is identity compatible with ρ_{commtlc} and $\mathcal{F}_{\mathrm{commtlc}}$.*

Proof. Claim-1. Since ρ_{commtlc} calls $\mathcal{F}_{\mathrm{commtlc}}$, $\mathcal{F}_{\mathrm{commtlc}}$ is a subroutine of ρ_{commtlc} and the claim follows.

Claim-2. We consider an adversary corrupting distinct users in Griefing, Fakey and Wormhole attacks, as shown in Figure 9. We then prove that Π_{commtlc} UC-emulates $\mathcal{F}_{\mathrm{commtlc}}$ for each of these attacks. In the real world, the environment $\mathcal{Z}$ provides input1 and input2 to ID_0 and ID_4, respectively. The same inputs are provided to DID_0 and DID_4 in the ideal world by the environment $\mathcal{Z}$. Here, $\mathrm{Input1} = (s_0, r_0), (s_1, r_1), (s_2, r_2), (s_3, r_3)$ and $\mathrm{Input2} = (s_4, r_4)$.

- Griefing-i attack: Here, i represents the user performing the Griefing attack. If ID4 performs the Griefing attack, it is referred to as a Griefing attack by the receiver; otherwise, it is considered a Griefing attack by the i-th user. In this scenario, the adversary can corrupt ID_4 or any other intermediate node, except for ID_0. When the adversary $\mathrm{ID}_{\mathrm{adv}}$ corrupts ID_4 in the real-world setting, the attack is initiated. After receiving inputs from $\mathrm{ID}_{\mathcal{Z}}$, ID_4 generates a commitment $c_4 = \mathrm{commit}(s_4, r_4)$ and sends c_4 along with its signed version to ID_0 via a secure channel. ID_0 follows the CommTLC protocol as detailed in Section 4. When ϕ_4 reaches ID_4, $\mathrm{ID}_{\mathrm{adv}}$ samples $(s_4', r_4') \in \mathbb{Z}_q$ and shares $\{(s_0, r_0), (s_4', r_4')\}$ with ID_3 via ID_4. Next, ID_3 computes $\mathrm{commit}(s_0 + s_4', r_0 + r_4') = \phi_4'$. With very high probability, $\phi_4' \neq \phi_4$ because, due to the 'conditional binding' property of the Pedersen commitment, it is extremely difficult to produce a secret/trapdoor pair (s_4', r_4') that results in $\mathrm{commit}(s_0 + s_4', r_0 + r_4') = \mathrm{commit}(s_0 + s_4, r_0 + r_4)$. Thus, protocol execution halts at ID_3 and it returns 0 as output to the environment $\mathcal{Z}$.

Similarly, in the ideal world, when ϕ_4 reaches the dummy user DID_4, the simulator ID_{sim} samples $(s_4^{''}, r_4^{''}) \in \mathbb{Z}_q$ and provides $\{(s_0, r_0), (s_4^{''}, r_4^{''})\}$ along with ϕ_4 as backdoor input to $\mathcal{F}_{commtlc}$. Next, $\mathcal{F}_{commtlc}$ computes $commit(s_0 + s_4^{''}, r_0 + r_4^{''}) = \phi_4^{''}$ using the $\mathcal{F}_{commit}$ subroutine. With very high probability, $\phi_4^{''} \neq \phi_4$ due to the conditional binding property of the Pedersen commitment. $\mathcal{F}_{commtlc}$ then sends output 0 to DID_3, which forward it to the environment $\mathcal{Z}$. The environment $\mathcal{Z}$ receives output 0 from both the real and ideal worlds with very high probability, making it unable to distinguish between the two outputs. Therefore, $\Pi_{commtlc}$ UC-emulates $\mathcal{F}_{commtlc}$ in the Griefing attack.

- Fakey attack: In a Fakey attack, an adversary corrupts the payer. Here, ID_{adv} corrupts ID_0. After receiving inputs from $ID_{\mathcal{Z}}$, ID_4 generates a commitment $c_4 = commit(s_4, r_4)$ and sends c_4 along with its signed version to ID_0 via a secure channel. The adversary ID_{adv} intercepts and drops both c_4 and the signed c_4 intended for ID_0. It then samples $(s_4^{'}, r_4^{'}) \in \mathbb{Z}_q$ and generates a corresponding commitment $c_4^{'}$. ID_{adv}, which it sends to ID_0. ID_0 continues following the CommTLC protocol. When $\phi_4^{'} = c_0.c_4^{'}$ reaches ID_4, ID_4 shares $\{(s_0, r_0), (s_4, r_4)\}$ with ID_3. Next, ID_3 computes $commit(s_0 + s_4, r_0 + r_4) = \phi_4$. With very high probability, $\phi_4 \neq \phi_4^{'}$ due to the conditional binding property of the Pedersen commitment. As a result, the protocol execution halts at ID_3, which returns 0 as output to the environment $\mathcal{Z}$.
  Similarly, in the ideal world, after receiving the inputs from $ID_{\mathcal{Z}}$, DID_4 forward (s_4, r_4) to $\mathcal{F}_{commtlc}$, which sends c_4 to ID_0. However, ID_{sim} drops this message. Instead, ID_{sim} samples $(s_4^{''}, r_4^{''}) \in \mathbb{Z}_q$ and sends it to $\mathcal{F}_{commtlc}$. The functionality $\mathcal{F}_{commtlc}$ generates a commitment $c_4^{''}$ using $\mathcal{F}_{com}$ subroutine and provides the adversary's signature over it with the help of the subroutine $\mathcal{F}_{dsa}$. It returns $c_4^{''}$ and its signed version to DID_0. When $\phi_4^{''}$ reaches the dummy user DID_4, it sends $\{(s_0, r_0), (s_4, r_4)\}$ and $\phi_4^{''}$ to $\mathcal{F}_{commtlc}$. Next, $\mathcal{F}_{commtlc}$ computes $commit(s_0 + s_4, r_0 + r_4) = \phi_4$ using the $\mathcal{F}_{commit}$ subroutine. Since, $\phi_4 \neq \phi_4^{''}$ with very high probability due to the conditional binding property of the Pedersen commitment, $\mathcal{F}_{commtlc}$ sends output 0 to DID_3, which then forward it to the environment $\mathcal{Z}$. The environment $\mathcal{Z}$ receives output 0 from both the worlds with very high probability, making it unable to distinguish between the two outputs. Hence, $\Pi_{commtlc}$ UC-emulates $\mathcal{F}_{commtlc}$ in the Fakey attack.

- Wormhole attack: In a Wormhole attack, the adversary ID_{adv} corrupts both ID_1 and ID_3. Upon receiving inputs from $ID_{\mathcal{Z}}$, ID_0 and ID_4 begin executing the Commtlc protocol. ID_4 generates a commitment $c_4 = commit(s_4, r_4)$ and sends c_4 along with its signed version to ID_0 via a secure channel. When $\phi_4 = c_0.c_4$ reaches ID_4, it shares $\{(s_0, r_0), (s_4, r_4)\}$ with ID_3. ID_3 then computes $commit(s_0 + s_4, r_0 + r_4) = \phi_4$. Since both ID_3 and ID_1 are controlled by the adversary ID_{adv}, the inputs $\{(s_0, r_0), (s_4, r_4), (s_3, r_3)\}$ from ID_3 and $(s_2^{'}, r_2^{'}) \in \mathbb{Z}_q$ sampled by the adversary ID_{adv} are sent directly to ID_1 through the adversary's backdoor channel. ID_1 then sends all received secrets along with its own secret (s_1, r_1) to ID_0. Upon receiving these secrets, ID_0 computes $commit(s_0 + s_4 + s_3 + s_2^{'} + s_1, r_0 + r_4 + r_3 + r_2^{'} + r_1) = \phi_4^{'}$, which does not equal ϕ_4. As a result, the protocol execution halts at ID_0, since $\phi_4^{'} \neq \phi_4$ with a very high probability due to the conditional binding property of the Pedersen commitment. ID_0 returns 0 as output to the environment $\mathcal{Z}$.
  Similarly, in the ideal world, the simulator ID_{sim} samples $(s_2^{''}, r_2^{''}) \in \mathbb{Z}_q$ and sends it as a backdoor input along with other secrets, that is, $\{(s_2^{''}, r_2^{''}), (s_3, r_3), (s_1, r_1)\}$ to $\mathcal{F}_{commtlc}$. $\mathcal{F}_{commtlc}$ already has (s_0, r_0) and (s_4, r_4) from DID_0 and DID_4, respectively. Next, $\mathcal{F}_{commtlc}$ computes $commit(s_0 + s_4 + s_3 + s_2^{''} + s_1, r_0 + r_4 + r_3 + r_2^{''} + r_1) = \phi_1^{''}$ using $\mathcal{F}_{com}$ subroutine. However, $\phi_1^{''} \neq \phi_1$ with a very high probability due to the conditional binding property of the Pedersen commitment. Therefore, $\mathcal{F}_{commtlc}$ returns 0 to DID_0 and DID_0, which relays this output bit 0 to the environment $\mathcal{Z}$.
  The environment $\mathcal{Z}$ has received output 0 from both the worlds with very high probability. Therefore, $\mathcal{Z}$ cannot distinguish between the two outputs. Hence, $\Pi_{commtlc}$ UC-emulates $\mathcal{F}_{commtlc}$ in the Wormhole attack.

Claim-3. $\Pi_{commtlc}$ consist of five user machines with identities ID_0, ID_1, ID_2, ID_3 and ID_4. Similarly, $\mathcal{F}_{commtlc}$ contains five dummy user machines: DID_0, DID_1, DID_2, DID_3 and DID_4. As a result, there exists an identity-preserving injective correspondence between the machines in $\Pi_{commtlc}$ and those in $\mathcal{F}_{commtlc}$. Additionally, $\rho_{commtlc}$ comprises machines with identities DID_0, DID_1, DID_2, DID_3 and DID_4 and a 'caller machine'. Therefore, no machine in $\Pi_{commtlc}$ shares the same identity as a machine in $\rho_{commtlc} \setminus \mathcal{F}_{commtlc}$. Consequently, $\Pi_{commtlc}$ is identity compatible with both $\rho_{commtlc}$ and $\mathcal{F}_{commtlc}$. □

7 | Conclusion

In this paper, we have proposed a scheme, CommTLC, that detects Fakey, Griefing and Wormhole attacks along with identifying the corresponding adversaries launching these attacks. CommTLC ensures that the payer sends the same payment key received from the payee, forces the payee to provide the correct secret/trapdoor pairs and prevents Wormhole attacks. We implemented CommTLC using the LND implementation of the Lightning Network and evaluated its performance in terms of communication and computational overheads. We also compared its performance with the MAPPCN and EAMHL+ schemes. We analyzed CommTLC's security using the UC framework and successfully detected adversaries in all three attacks within 112 ms for a payment path involving five users.

Author Contributions

Prerna Arote: conceptualization, investigation, methodology, validation, writing – original draft, writing – review and editing. **Joy Kuri:** conceptualization, investigation, supervision, validation, writing – review and editing.

Conflicts of Interest

The authors declare no conflicts of interest.

Data Availability Statement

Data sharing is not applicable to this article as no data sets were generated or analysed during the current study.

Endnotes

[1] https://live.blockcypher.com/btc-testnet/

[2] https://coinfaucet.eu/en/btc-testnet/

References

1. E. Androulaki, G. O. Karame, M. Roeschlin, T. Scherer, and S. Capkun, "Evaluating User Privacy in Bitcoin," in *Financial Cryptography and Data Security: 17th International Conference, FC 2013, Revised Selected Papers 17* (Springer, 2013), 34–51.

2. Bitcoin Blockchain Explorer, (2024), accessed May 25 2024, https://www.blockchain.com/explorer/assets/btc.

3. J. Poon and T. Dryja, *The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments* (2016), https://lightning.network/lightning-network-paper.pdf.

4. C. Decker, "On he Scalability and Security of Bitcoin" (PhD diss., ETH Zurich, 2016).

5. R. Khalil, A. Zamyatin, G. Felley, P. Moreno-Sanchez, and A. Gervais, "Commit-Chains: Secure, Scalable Off-Chain Payments," preprint, Cryptology ePrint Archive, (2018).

6. A. M. Antonopoulos, O. Osuntokun, and R. Pickhardt, *Mastering the Lightning Network* (O'Reilly Media, Inc., 2021).

7. A. Mizrahi and A. Zohar, "Congestion Attacks in Payment Channel Networks," in *International Conference on Financial Cryptography and Data Security* (Springer, 2021), 170–188.

8. S. Tochner, S. Schmid, and A. Zohar, "Hijacking Routes in Payment Channel Networks: A Predictability Tradeoff," preprint, arXiv:1909.06890, (2019).

9. Z. Lu, R. Han, and J. Yu, "Bank Run Payment Channel Networks," *IACR Cryptol. ePrint Arch.* 2020 (2020): 456.

10. J. Harris and A. Zohar, "Flood & Loot: A Systemic Attack on the Lightning Network," in *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies* (Association for Computing Machinery (ACM), 2020), 202–213.

11. D. Robinson, "HTLCs Considered Harmful," in *Proceedings of Stanford Blockchain Conference* (2019), https://diyhpl.us/wiki/transcripts/stanford-blockchain-conference/2019/htlcs-considered-harmful/.

12. G. Malavolta, P. Moreno-Sanchez, A. Kate, M. Maffei, and S. Ravi, "Concurrency and Privacy With Payment-Channel Networks," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (ACM, 2017), 455–471.

13. S. Tripathy and S. K. Mohanty, "MAPPCN: Multi-Hop Anonymous and Privacy-Preserving Payment Channel Network," in *International Conference on Financial Cryptography and Data Security* (Springer, 2020), 481–495.

14. Y. Zhang, X. Jia, B. Pan, et al., "Anonymous Multi-Hop Payment for Payment Channel Networks," *IEEE Transactions on Dependable and Secure Computing* 21, no. 1 (2023): 476–485.

15. P. Arote and J. Kuri, "Detect and Isolate an Adversary in Fakey and Griefing-R Attack on Lightning Network," in *2024 IEEE International Conference on Blockchain (Blockchain)* (IEEE, 2024), 563–568.

16. B. Yu, S. K. Kermanshahi, A. Sakzad, and S. Nepal, "Chameleon Hash Time-Lock Contract for Privacy Preserving Payment Channel Networks," in *Provable Security: 13th International Conference, ProvSec 2019, Proceedings 13* (Springer, 2019), 303–318.

17. G. Malavolta, P. Moreno-Sanchez, C. Schneidewind, A. Kate, and M. Maffei, "Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability," Cryptology ePrint Archive, (2018).

18. S. K. Mohanty and S. Tripathy, "n-HTLC: Neo Hashed Time-Lock Commitment to Defend Against Wormhole Attack in Payment Channel Networks," *Computers & Security* 106 (2021): 102291.

19. W. Wu, E. Liu, X. Gong, and R. Wang, "Blockchain Based Zero-Knowledge Proof of Location in IoT," in *ICC 2020-2020 IEEE International Conference on Communications (ICC)* (IEEE, 2020), 1–7.

20. C. Yang, X. Ju, E. Liu, Y. Geng, and R. Wang, "Blockchain-Based Indoor Location Paging and Answering Service With Truncated-Geo-Indistinguishability," *IET Blockchain* 1, no. 2-4 (2021): 105–117.

21. Q. Zhang, S. Cao, Y. Ni, T. Chen, and X. Zhang, "Enabling Privacy-Preserving Off-Chain Payment Via Hybrid Multi-Hop Mechanism," in *ICC 2022-IEEE International Conference on Communications* (IEEE, 2022), 13–18.

22. A. A. Khalil, M. A. Rahman, and H. A. Kholidy, "Fakey: Fake Hashed Key Attack on Payment Channel Networks," in *2023 IEEE Conference on Communications and Network Security (CNS)* (IEEE, 2023), 1–9.

23. S. Mazumdar, P. Banerjee, and S. Ruj, "Time is Money: Countering Griefing Attack in Lightning Network," in *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)* (IEEE, 2020), 1036–1043.

24. T. Perrin, "The Noise Protocol Framework," *noiseprotocol, Protocol Revision* 34 (2018).

25. R. Russel, "Lightning Network," in *BOLT In-Progress Specifications* (2019), https://github.com/lightningnetwork/lightning-rfc.

26. G. Danezis and I. Goldberg, "Sphinx: A Compact and Provably Secure Mix Format," in *2009 30th IEEE Symposium on Security and Privacy* (IEEE, 2009), 269–282.

27. T. P. Pedersen, "Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing," in *Annual International Cryptology Conference* (Springer, 1991), 129–140.

28. R. Metere and C. Dong, "Automated Cryptographic Analysis of the Pedersen Commitment Scheme," in *Computer Network Security: 7th International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security, Proceedings 7* (Springer, 2017), 275–287.

29. D. Goldschlag, M. Reed, and P. Syverson, "Onion routing," *Communications of the ACM* 42, no. 2 (1999): 39–41.

30. J. A. Akinyele, C. Garman, I. Miers, et al., "Charm: A framework for Rapidly Prototyping Cryptosystems," *Journal of Cryptographic Engineering* 3 (2013): 111–128.

31. Lightning Network, *Lightning Network Specifications* (2019), accessed June 11 2024, https://github.com/lightning/bolts.

32. C. Wang, Y. Ren, and Z. Wu, "Multi-Hop Anonymous Payment Channel Network Based on Onion Routing," *IET Blockchain* 4, no. 2 (2024): 197–208.

33. R. Canetti, "Universally Composable Security: A New Paradigm for Cryptographic Protocols," in *Proceedings of 42nd IEEE Symposium on Foundations of Computer Science* (IEEE, 2001), 136–145.