LayerZero

Ryan Zarick Bryan Pellegrino Isaac Zhang Thomas Kim Caleb Banister

LayerZero Labs

Abstract

In this paper, we present the first *instrinsically secure* and *semantically universal* omnichain interoperability protocol: LayerZero. Utilizing an *immutable* endpoint, append-only verification modules, and fully-configurable verification infrastructure, LayerZero provides the security, configurability, and extensibility necessary to achieve omnichain interoperability. LayerZero enforces strict application-exclusive ownership of protocol security and cost through its novel trust-minimized *modular security* framework which is designed to universally support all blockchains and use cases. Omnichain applications (OApps) built on the LayerZero protocol achieve frictionless blockchain-agnostic interoperation through LayerZero's universal network semantics.

1 Introduction

Blockchain interoperability represents an ever-growing challenge as the diversity of chains continues to grow, and the importance of connecting the fragmented blockchain landscape is increasing as applications seek to reach users across a progressively wider set of chains. We present LayerZero, the first omnichain messaging protocol (*OMP*) to achieve a fully-connected mesh network that is scalable to all blockchains and use cases.

In contrast to the monolithic security model of other cross-chain messaging services, the LayerZero protocol uses a novel modular security model to *immutably* implement security. This approach can still be extended to support new features and verification algorithms. *Intrinsic security* against censorship, replay attacks, denial of service, and in-place code modifications is designed into *immutable* Endpoints. Less fundamental *extrinsic* aspects of security (e.g., signature schemes) are isolated into independently-immutable modules. As a protocol, LayerZero is not bound to any infrastructure or blockchain; all components other than the endpoint can be interchanged and configured by applications built on LayerZero.

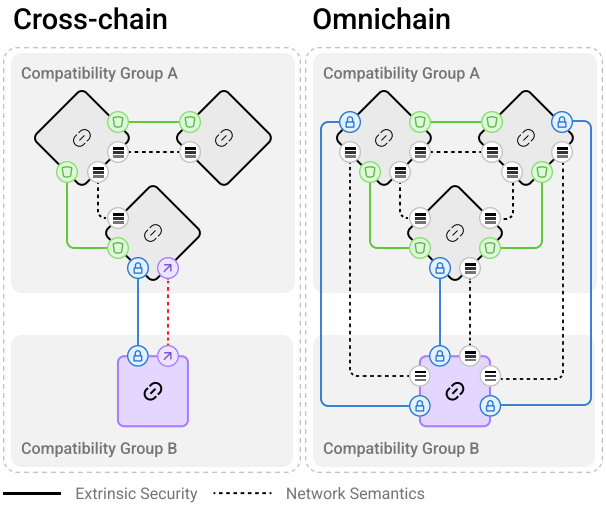

Figure 1: The omnichain fully-connected mesh network has universal network semantics for all connected chains and security specialized to each link.

We illustrate the omnichain fully-connected mesh network in Figure 1. Each chain is directly connected to every other chain, and while the *extrinsic security* (Section 2.1) may be different for different chain pairs (illustrated by the colored solid lines), the guarantees of eventual, lossless, exactly-once packet delivery should be uniform and never change.

LayerZero's network channel semantics, including execution features, configuration semantics, censorship resistance, and failure model, are universal. These universal semantics allows application developers to easily architect secure, chain agnostic omnichain applications (*OApps*).

The remainder of this paper is organized as follows. First, Section 2 explains the overarching fundamental

	Integrity Layer	Failure model
OMP	Channel validity	Packet censorship
		Packet replay
		Buggy updates
		Invalid reconfiguration
	Channel liveness	Denial of service
		Infrastructure health
		Administrator health
	Data validity	Cryptographic attack
		Malicious infrastructure
	Data liveness	Data loss on blockchain

Table 1: We divide protocol integrity into four properties. Data liveness depends on the underlying blockchains and cannot be secured by the OMP.

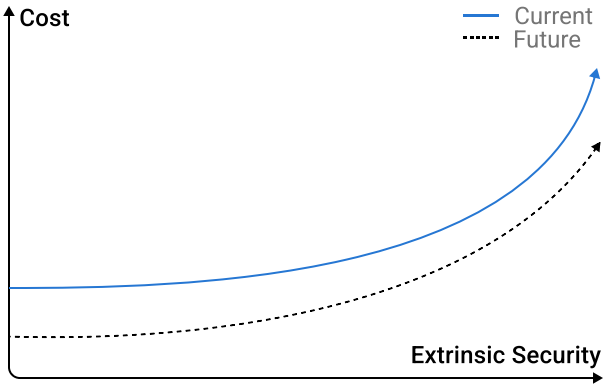

Figure 2: The pareto frontier of extrinsic security vs cost continually changes due to advancements in technology.

principles underlying the LayerZero protocol. Section 3 describes the protocol design and highlights how each component is architected for security. Finally, Section 4 presents examples of how LayerZero can easily be extended to support a wide range of additional features in a blockchain-agnostic manner.

2 Principles

The responsibilities of an OMP can be condensed into two requirements: intrinsic security and universal semantics. Existing messaging services fail to implement one or both of the above requirements and thus suffer from two fundamental deficiencies: monolithic security and overspecialization. In the remainder of this section, we contextualize security and semantics within the cross-chain messaging paradigm, describe how existing cross-chain messaging systems fall short of these goals, and outline how LayerZero is designed from the ground-up to overcome these shortcomings.

2.1 Security

The first and most important requirement of OMPs is that they should be secure. We divide security into *intrinsic* and *extrinsic* security, and while all messaging systems implement extrinsic security, few provide intrinsic security. Intrinsic security refers to protocol-level invariants of lossless (censorship resistance), exactly-once (no replay), eventual (liveness) delivery. Extrinsic security encompasses all other security properties, such as signature and verification algorithms.

Most existing messaging services have taken an ad-hoc approach to security, continuously updating a single monolithic end-to-end security model to accommodate chains as they are added to their network. These services invariably utilize forced, in-place updates to a shared security model, and thus cannot provide long-term security invariants for OApps to build upon. To provide long-term security invariants in LayerZero, we chose instead to modularize security and enforce strict immutability for all modules.

Table 1 illustrates how we divide protocol integrity into *channel* and *data* integrity. Each of these integrity layers is further subdivided into validity and liveness properties. We more formally define intrinsic security to cover channel validity and liveness, and *extrinsic* security to cover data validity. In this paper, we refer to the extrinsic security configuration as the *Security Stack*. Monolithic shared security systems force the same Security Stack on all applications, while isolated security systems allow a different Security Stack per OApp.

Intrinsic security can and should be universally secured based on first principles. In contrast, optimal, trustless communication across blockchains is impossible, and the continuous advancement in verification algorithms and blockchain design necessitates *extensibility* and *configurability* of extrinsic security. This is true even in special cases such as L1–L2 rollups; the possibility of hard forks necessitates L2 contract upgradability, thus making the L2 contract owner a trusted entity.

It is infeasible to formally verify the extrinsic security of any nontrivial code given the many underlying layers of execution and the reliance of cryptographic algorithms on the computational intractability of NP problems. As a result, the most practical measure of security is an economic one: a *non-upgradable* smart contract's security is directly proportional to how many assets it has secured and for how long. Thus, the implications of extrinsic security isolation are clear: OMPs must guarantee indefinite access to well-established, extrinsically secure code while still allowing protocol maintainers to extend the protocol. The impossibility of trustless communication thus implies that extrinsic security exists on a

constantly shifting pareto frontier (Figure 2) and should be customizable to OApp-specific requirements.

To provide intrinsic security, the OMP must guarantee that an OApp's Security Stack only changes when the OApp owner *opts in* to the change. This implies that systems designed to allow *in-place* code upgrades can *never* be intrinsically secure. Replaceability of existing code permits the permanent deprecation of well-established extrinsically secure code and the potential introduction of vulnerable code. Current approaches to in-place secure upgrades involve careful testing and audits, but history has shown [1, 3, 4] that this process is not foolproof and can overlook severe vulnerabilities. To guarantee long-term security invariants, OMPs must be architected to isolate each OApp's Security Stack from software updates and other OApps' configurations.

2.2 Universal Semantics

The second requirement of OMPs is universal semantics, or the ability to extend and adapt the network primitive to all additional use cases and blockchains.

Execution semantics (i.e., feature logic) should be both chain-agnostic and sufficiently expressive to allow any OApp-required functionality. A key insight we had when designing LayerZero is that feature logic (execution) can be fully isolated from security (verification). This not only simplified protocol development, but also eliminates concerns about impact to protocol security when designing and implementing *execution features*.

The other aspect of universal semantics is universal compatibility of the OMP interface and network semantics with all existing and future blockchains. The importance of semantic unification cannot be understated, as OApps *cannot* scale if every additional blockchain in the network incurs significant engineering cost to accommodate different interfaces and network consistency models. In practice, an OMP must have a unified interface, transmission semantics, and execution behavior regardless of the source and destination blockchain characteristics.

3 Core protocol design

We divide LayerZero into four components (Figure 3): an immutable *Endpoint* that implements censorship resistance, an append-only collection (*MessageLib Registry*) of onchain verification modules (*MessageLibs*), a permissionless set of Decentralized Verifier Networks (*DVN*s) to verify data across blockchains, and permissionless *executors* to execute feature logic in isolation to the cross-chain message verification context. Tying the components together is the *OApp Security Stack*, which

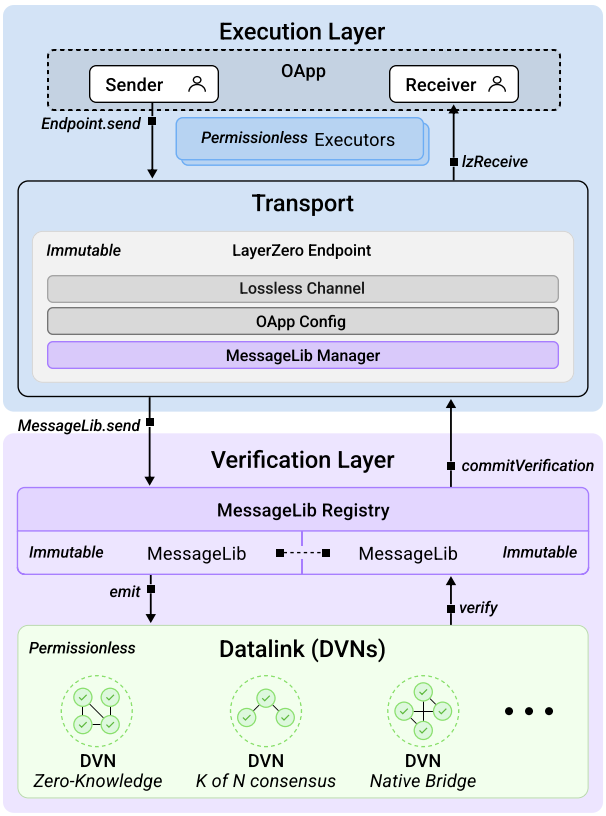

Figure 3: LayerZero is divided into execution and verification layers. The verification layer securely transmits data between blockchains, and the execution layer interprets this data to form a secure, censorship resistant messaging channel.

defines the extrinsic security configuration of the protocol and is modifiable exclusively by the OApp owner.

Messages in LayerZero are composed of a payload and routing information (path). These messages are serialized into *packets* before they are transmitted across the mesh network. Packets are verified by the verification layer on the destination blockchain before they are *committed* into the lossless channel. The packets are then read from the channel and *delivered* by *executing* the *lzReceive* callback on the destination OApp contract.

The LayerZero Endpoint secures channel validity through OApp-exclusive Security Stack ownership in conjunction with an immutable channel that implements censorship resistance, exactly-once delivery, and guaranteed liveness. Endpoint immutability guarantees that no external entity or organization can ever forcibly change the security characteristics an OApp's Security Stack.

Individually immutable MessageLibs collectively form the MessageLib Registry, and each MessageLib is an extrinsically secure interface that verifies packet data integrity before allowing messages to be committed to

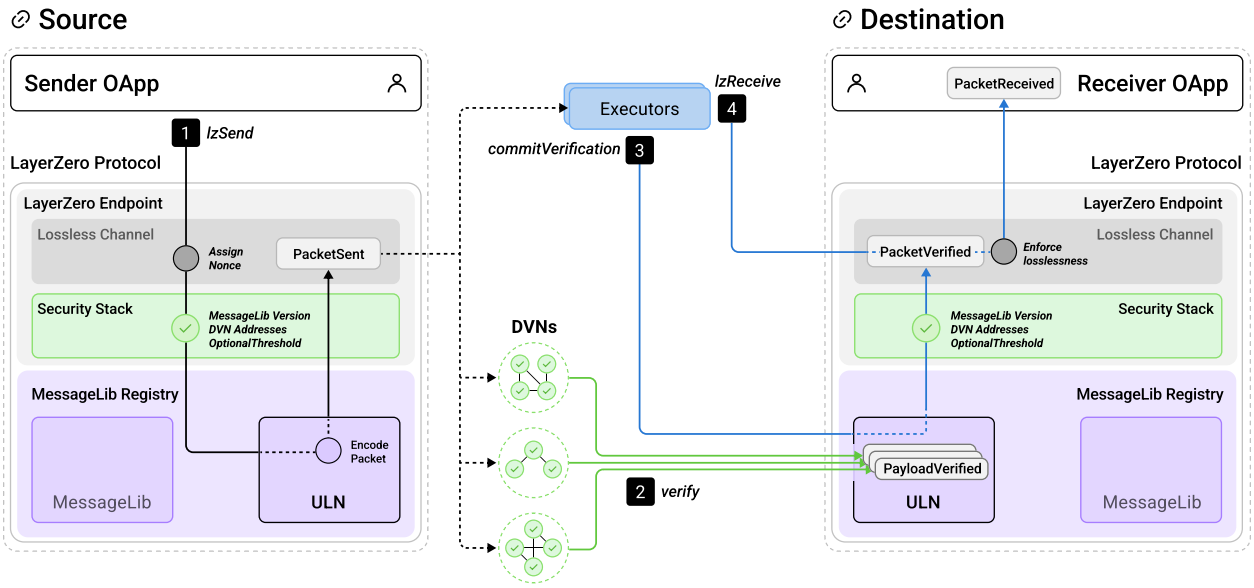

Figure 4: Steps to send a message using LayerZero.

the Endpoint. Existing MessageLibs cannot be modified, thus making the MessageLib Registry append-only, and each Security Stack specifies exactly one MessageLib. Security Stack ownership semantics in conjunction with MessageLib immutability enable applications to potentially use the same Security Stack *forever*.

Each DVN is an aggregation of *verifiers* that collectively verify the integrity of data shared between two independent blockchains. DVNs can include both offchain and/or onchain components, and each Security Stack can theoretically include an unbounded number of DVNs. The underlying DVN structure can leverage any verification mechanism, including but not limited to zero-knowledge, side chains, K-of-N consensus, and native bridges. For brevity, we refer to the abstract collection of MessageLib, DVNs, and other hyperparameters as the Security Stack, and a serialized form of the Security Stack is assumed to be written to each chain.

Channel liveness (eventual delivery) is guaranteed through *permissionless execution* in conjunction with *Security Stack reconfiguration*. Assuming the liveness of the source and destination blockchains, LayerZero channel liveness can only be (temporarily) compromised if (1) the DVNs in the Security Stack experience faults, or (2) the configured executor stops delivering messages. If too many configured DVNs stop verifying messages, the OApp can regain liveness by reconfiguring its Security Stack to use different DVNs. Packet delivery (execution) is permissionless, so any party willing to pay execution gas costs can deliver packets to restore channel liveness.

Interaction between LayerZero components is minimized and standardized to reduce software bug surfaces (Figure 3). LayerZero's modularization and configurability also enables quick prototyping of the protocol on new chains. Using a simple whitelist as a placeholder for MessageLib allows parallel development and testing of all components, expediting expansion of the mesh network to new chains.

3.0.1 LayerZero packet transmission

Before describing each component in detail, we present an overview of how packets are transmitted in LayerZero as shown in Figure 4. The LayerZero mesh network is formed by the deployment by a protocol administrator of a LayerZero Endpoint on each connected blockchain. In this example, the OApp sends a LayerZero message from a *sender* contract to a *receiver* contract across the LayerZero mesh network. For illustration purposes, the MessageLib we use in this example is the *Ultra Light Node* [6] (Section 3.2.1).

During initial setup, the OApp configures its Security Stack on the LayerZero Endpoint on the source and destination blockchains. The MessageLib version configured in the Security Stack determines the *packet version*.

In step ①, the sender calls `lzSend` on the source chain LayerZero Endpoint, specifying the message payload and the *path*. This path is associated with an independent censorship resistant channel, and is composed of the sender application address, the source Endpoint ID, the recipient application address, and the destination Endpoint ID.

The source Endpoint then assigns a gapless, monotonically-increasing nonce to the packet. This

	Field name	Type	
Header	Packet version	uint8	
	Nonce	uint64	
	Source Endpoint ID	uint32	Path
	Sender	uint256	
	Destination Endpoint ID	uint32	
	Receiver	uint256	
	GUID	uint256	
	Message Payload	bytes[]	

Table 2: LayerZero packets are composed of a header and body. The header includes the packet version and path. The body is composed of the actual message payload. Packets are identified by their globally unique identifier (GUID).

nonce is concatenated with the path, then the result is hashed to calculate the global unique ID (GUID) of the packet. This GUID is used by offchain and onchain *workers* (e.g., executors, DVNs) to track the status of LayerZero messages and trigger actions.

The source Endpoint reads the OApp Security Stack to determine the correct source MessageLib (ULN in this example) to encode the packet. The source MessageLib processes the packet based on the configured Security Stack, rendering payment to the configured DVNs to verify the message on the destination MessageLib and optionally specified executors to trigger offchain actions. These DVN and executor identifiers along with any relevant arguments are serialized by MessageLib into an unstructured byte array called *Message Options*. After the ULN encodes the packet and returns it to the Endpoint, the Endpoint emits the packet to conclude the LayerZero `send` transaction.

In step ②, the configured DVNs each independently *verify* the packet on the destination MessageLib; for ULN, this constitutes storing the hash of the packet payload. After a threshold of DVNs verify the payload (see Section 3.2.1), a worker (e.g., executor, DVN, user) commits the packet to the Endpoint in step ③. The Endpoint checks that the payload verification reflects the OApp-configured Security Stack before committing to the lossless channel.

Finally, in step ④, an executor calls `lzReceive` on the committed message to execute the Receiver OApp logic on the packet. Step ④ will revert to prevent censorship if the channel cannot guarantee lossless exactly-once delivery.

3.1 LayerZero Endpoint

The LayerZero Endpoint, implemented as an immutable open-source smart contract and deployed in one or more instances per chain, provides a stable application-facing interface (Table 3), the abstraction of a lossless network channel with exactly-once guaranteed delivery, and manages OApp Security Stacks. The immutability of the LayerZero Endpoint guarantees long-term channel validity by enforcing update isolation, configuration ownership, and channel integrity. The Security Stack is key to LayerZero's channel liveness guarantee, as it mediates the trust–cost relationship between OApps and the permissionless set of DVNs.

OApps call `send` on the Endpoint to queue a message to be sent through LayerZero, specifying the `path` (Table 2), the message payload, and an optional byte array (Message Options) containing serialized options to be interpreted by MessageLib. Message Options is purposely unstructured, improving extensibility as we discuss in Section 4. The complement to `send` is `lzReceive`, which is executed on the destination chain to consume the message with the specified `GUID`. On the destination chain, the Endpoint handles calls to `lzReceive` and `getInboundNonce`, enforcing lossless exactly-once delivery to protect the integrity of the message channel. `lzReceive` delivers the verified payload of this message to the OApp, provided the message can be losslessly delivered. `getInboundNonce` returns the highest losslessly deliverable nonce, computing the highest nonce such that all messages with preceding nonces have been verified, skipped, or delivered. To handle erroneously sent messages or malicious packets, OApps either call `clear` to skip delivery of the packet in question or `skip` to skip both verification and delivery.

In addition to `clear` and `skip`, the Endpoint provides two convenience functions `nilify` and `burn`. Nilify invalidates a verified packet, preventing the execution of this packet until a new packet is committed from MessageLib; this function can be used to proactively invalidate maliciously generated packets from compromised DVNs. Burn is a convenience function to allow OApps to `clear` a packet without knowing the packet contents; this is useful if a faulty Security Stack commits an invalid hash to the endpoint, or if an OApp needs to clear a nilified nonce.

3.1.1 Out-of-order lossless delivery

We lay out two non-negotiable consistency requirements for LayerZero's message channel: *lossless* and *exactly-once* delivery. Censorship resistant channels *must* be lossless, and exactly-once delivery is required to prevent replay attacks. Both of these requirements are crucial for network integrity, and are guaranteed by the protocol provided the underlying blockchain is not faulty.

Channels in LayerZero are necessarily separated and isolated by `path` (Table 2), as any lossless channel shared by two different OApps must sacrifice channel valid-

Routine	Arguments	Description
`send`		Sends a message through LayerZero.
	`path`	The path of the channel through which to send the message.
	`payload`	Data transmit to the receiver.
	`Message Options`	Byte array of arguments to be interpreted by MessageLib (optional).
`getInboundNonce`		Returns the largest nonce with all predecessors received.
	`path`	The path corresponding to the message channel.
`skip`		Called by the receiver to skip verification and delivery of a nonce.
	`path`	The path of the channel to skip a nonce on.
	`nonce`	The nonce of the message to skip (must be the inbound nonce + 1).
`clear`		Called by the receiver to skip a nonce that has been verified.
	`path`	The path to skip a nonce on.
	`guid`	The GUID of the nonce to skip.
	`message`	The contents of the message to skip.
`lzReceive`		Called by the executor to receive a message from the channel.
	`path`	The path of the channel to receive a message from.
	`nonce`	Output parameter for the nonce of the received message.
	`GUID`	The GUID of the received message.
	`message`	The message to receive.
	`extraData`	Any extra data requested by the receiver.
`nilify`		Called by the executor to receive a message from the channel.
	`path`	The path of the channel to nilify a message on.
	`nonce`	The nonce of the packet to nilify.
	`payloadHash`	The hash of the payload to nilify.
`burn`		Called by the executor to receive a message from the channel.
	`path`	The path of the channel to burn a message on.
	`nonce`	The nonce of the packet to burn.
	`payloadHash`	The hash of the payload to burn.

Table 3: LayerZero core messaging API.

ity or liveness; an adversarial application can trivially deny liveness of a shared channel by refusing to verify a packet, but allowing other applications to forcibly skip the malicious OApp's packets constitutes censorship. Each channel maintains a logical clock implemented by a gapless, strictly monotonically increasing positive integer nonce, and each message sent over the channel is assigned exactly one nonce. On the destination Endpoint, each nonce is mapped to exactly one verified payload hash (Section 3.2), and the channel enforces that each delivered payload corresponds to the verified hash of the relevant nonce. LayerZero guarantees that the delivery of a packet implies all other packets on the same channel with lower nonces are delivered, deliverable, or skipped.

On a given channel, suppose two messages m_k and m_{k+1} are assigned positive integer nonces k and $k+1$ respectively. The gapless nonce guarantees that $\neg(\exists m_j \ s.t. \ m_k \rightarrow m_j \wedge m_j \rightarrow m_{k+1})$ where $\rightarrow$ is the happens-before relation. More informally, there can be no packet with a nonce between k and $k+1$. This is the weakest possible, and by extension most flexible, condition for losslessness. Stronger conditions (e.g., strictly in-order delivery) can be imposed on top of this abstraction if desired.

Censorship resistance is implemented by enforcing that no nonce can be delivered unless all previous nonces have been committed or skipped. For example, a packet with nonce N can only be delivered if all packets with nonce $1,\ldots,N-1$ are either committed or explicitly skipped by the receiver. We term the largest nonce that can be executed to be the *inbound nonce*.

Lossless and exactly-once delivery can be achieved using strictly in-order verification and execution, as demonstrated by Zarick et al. [6]. However, delivery order enforcement can result in artificial throughput limits on certain blockchains and complicates offchain infrastructure. In LayerZero we relax this ordering constraint, implementing out-of-order delivery that maintains channel integrity and does not introduce any additional onchain computational overhead.

The only efficient onchain implementation of an uncensorable channel with lossless, exactly-once, out-of-order delivery is to track the highest *delivered* nonce,

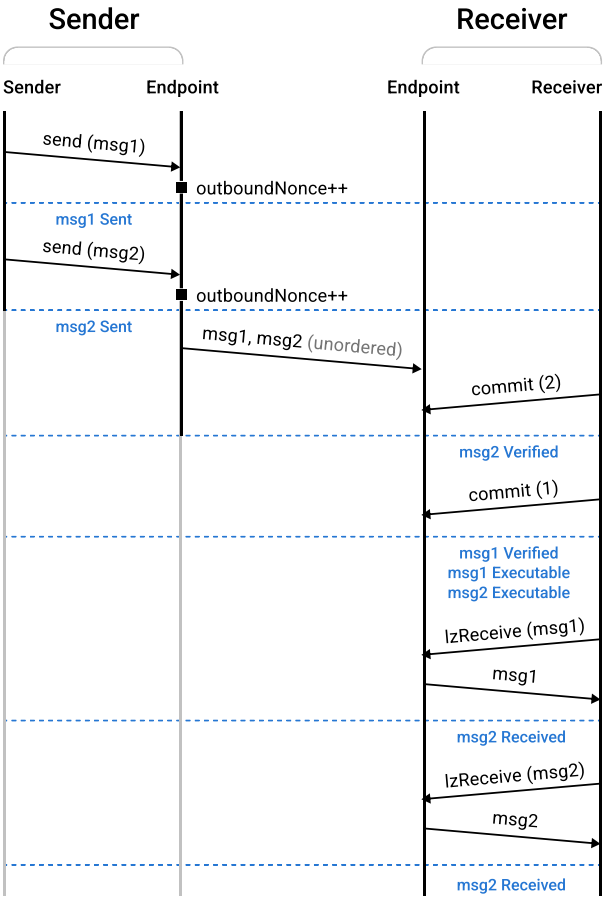

Figure 5: A packet is *Sent* after source transaction increments the nonce, *Verified* after it is committed into the Endpoint, and *Received* after delivery (execution).

which we term the *lazy inbound nonce*. The lazy inbound nonce begins at zero, and packets can only be executed if all packets starting from the lazy inbound nonce until the packet nonce are verified. Every time a packet is delivered (or skipped), the lazy inbound nonce is updated to the maximum of the current lazy inbound nonce and the nonce of the delivered packet.

The only other theoretical algorithm that can achieve efficient lossless and exactly-once delivery is updating the lazy inbound nonce upon *verification* rather than execution. This does not work in practice, because a single packet commit could result in an arbitrarily large update in the lazy inbound nonce. For example, if nonce 2 through 1000 have been committed before nonce 1, the commit of nonce 1 must iterate through 1000 nonces to update the lazy inbound nonce. This creates a situation where if the number of iterations exceeds the limits of the underlying blockchain, the corresponding channel will permanently lose liveness.

On the other hand, updating the lazy inbound nonce on execution can indeed run into computational limits, but permissionlessly retrying execution at a lower nonce will succeed. It is possible for an uninterrupted sequence of undeliverable messages (i.e., messages that will always revert on execution) to cause temporary loss of channel liveness if the sequence is longer than the blockchain iteration limit. This scenario can easily be rectified by the OApp owner calling `clear` (Table 3) to skip execution of these undeliverable packets.

To enforce exactly-once delivery, we flag each packet after it is successfully received. In LayerZero, this is implemented by deleting the verified hash of a packet from the lossless channel after it is delivered and disallowing verification of nonces less than or equal to the lazy inbound nonce.

We illustrate the lossless channel in Figure 5 through a simplified view of the lifecycle of a LayerZero packet. In this example, the OApp asynchronously sends two messages from the source chain to the destination chain, and each packet can be in one of three states: *Sent*, *Verified*, or *Received*. A packet is Sent after the source Endpoint assigns a nonce to the packet, after which the requested DVNs verify it on the destination MessageLib. The packet transitions into the Verified state after an executor calls `commitVerification` (shown as "commit" in Figure 5), which checks that the packet has been verified by the OApp Security Stack.

Once a packet and all preceding packets have been committed, the executor calls `lzReceive` to deliver the packet and, barring reversion of the transaction, the packet transitions into the Received state. If one or more preceding nonces have not been committed, the lossless channel will revert to prevent censorship.

3.2 MessageLib

The MessageLib Registry is a collection of MessageLibs, each of which are responsible for securely emitting packets on the source chain and verifying them on the destination MessageLib. Each standalone MessageLib implements extrinsic security, necessitating adaptation to underlying environmental changes and precluding a fully immutable design of the MessageLib Registry. MessageLib verifies the *payload hash* of each packet, committing the verified payload hash to the endpoint after the extrinsic security requirement (e.g., DVN threshold) is fulfilled.

To provide extensibility for extrinsic security while protecting existing OApps against in-place updates, we structure the MessageLib Registry as an *append-only* registry of immutable libraries, each of which can implement any arbitrary verification mechanism so long as it conforms to the protocol interface. This design avoids the trap of validation lock-in that most messaging ser-

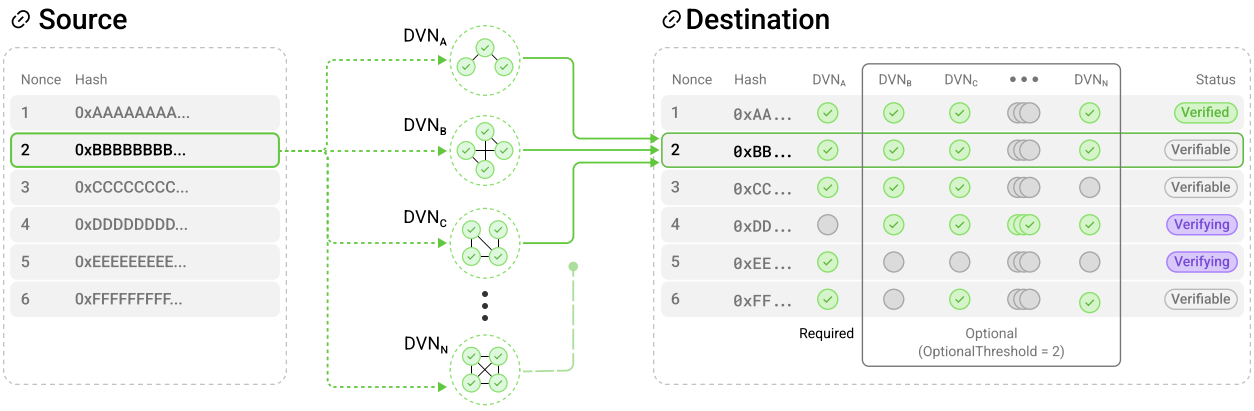

Figure 6: The Ultra Light Node enforces (onchain) the configured required DVNs, optional DVNs, and OptionalThreshold. Verification is neither lossless nor ordered, and messages can be committed to the channel as soon as the Security Stack is fulfilled.

vices fall into, and ensures that LayerZero can be extended to take advantage of the most secure and efficient verification algorithms in every scenario.

This design may seem counterintuitive at first, as it precludes any in-place software updates and thus appears to prevent the protocol admin from addressing software bugs. However, giving a single entity the power to unilaterally fix issues in-place also gives them the ability to introduce new vulnerabilities. This in turn invalidates any long-term protocol security invariants, as a malicious change in the code can easily violate them. Our decision to commit to immutable libraries in MessageLib presents higher barriers to introducing new code and bugfixes, but is crucial in realizing intrinsic security in LayerZero.

We argue this append-only design is the *only* way to implement intrinsic security without compromising extensibility. In theory, a single non-upgradable MessageLib that is completely bug-free and perfectly optimized provides intrinsic end-to-end security, but is impractical even in this unrealistic scenario. Changes in the execution environment (e.g., removal or addition of opcodes), consensus mechanism (e.g., validator election protocol), and evolving application preferences for verification algorithms necessitate protocol updates. Any scheme that allows in-place modifications of MessageLibs is inherently not intrinsically secure; OApps are placing their trust in the developers, auditors, or governance structure of the protocol administrator, and there is no practical way to guarantee that the security of updated code will match or exceed the existing MessageLibs. OMPs must allow extensions, but at the same time guarantee that the extrinsic security of previous versions is never impacted by these code additions. Ergo, the only design that provides intrinsically secure updates is to append new versions of the codebase to an immutable registry of library versions.

Each MessageLib operates independently and handles the following tasks: (1) accept the message from the Endpoint, (2) encode and emit the packet (Table 2) to DVNs and executors, paying any necessary fees, (3) verify the packet on the destination chain, and (4) commit the verified message to the destination Endpoint. All other tasks are handled by executors, minimizing the code size of MessageLib and allowing easy addition of features through the implementation and operation of new executors. No party, including the LayerZero admin, is permitted to modify or remove libraries once they are added to the MessageLib registry.

Note that losslessness is enforced in the immutable endpoint (execution layer), *not* in MessageLib (verification layer). As we explain in Section 3.2.1, MessageLib can commit verified packet hashes into the endpoint out of order and with gaps. However, packets cannot be consumed from the lossless channel if there are gaps in the sequence of verified packets.

3.2.1 Ultra Light Node

The Ultra Light Node (ULN) is the baseline MessageLib included in every LayerZero deployment, and allows the composition of up to 254 DVNs through customizable two-tier quorum semantics. ULN implements the minimal set of fundamental features necessary for any verification algorithm and is thus universally compatible with all blockchains. Each OApp Security Stack that is configured to use the ULN includes a set of *required* DVNs (X), *optional* DVNs, and a threshold ($OptionalThreshold$). X DVNs are *required*, and a packet can only be delivered if all X required DVNs and at least $OptionalThreshold$ optional DVNs total have signed the corresponding payload hash. After the necessary DVN signatures have been aggregated on the ULN,

the corresponding packet can be committed to the Endpoint. The required DVN model allows OApps to place a lower bound on the extrinsic security of the verification layer, as no message can be verified without a signature from the most secure DVN in the required set. This design delegates the majority of extrinsic security to the DVNs while still enforcing the Security Stack onchain.

We illustrate an example of the Ultra Light Node verification semantics in Figure 6. The OApp Security Stack includes required DVN (DVN_A) and $N-1$ optional DVNs (DVN_B, DVN_C, ...) with an OptionalThreshold of 1. This gives DVN_A "veto" power, and also requires at least one of the optional DVNs to verify the packet before it can be committed. Nonce 1 has been committed to the messaging channel (*Verified*). Nonces 2, 3, and 6 are committable (verifiable) as the Security Stack has been fulfilled, but are not committed until an executor calls commitVerification. Nonces 4 and 5 are not committable because they have not fulfilled the required verifier set and OptionalThreshold respectively.

This composable verification primitive gives OApps the ability to trade off cost and security, allows OApps to easily configure client diversity in their DVN set, and minimizes the engineering cost of upgrading extrinsic security (no onchain code extensions). The importance of client diversity cannot be understated, as even a non-compromised DVN is subject to buggy code.

3.2.2 MessageLib versioning and migration

MessageLibs are identifiable through a unique ID paired with semantic (`major.minor`) version, and a message can only be sent between two Endpoints if both implement a MessageLib with the same major version. Major versions determine packet serialization and deserialization compatibility, while minor versions are reserved for bugfixes and other non-breaking changes. The packet version of each LayerZero message is mapped to a MessageLib version, which DVNs use to identify which MessageLib to submit packet verification to on the destination blockchain. Each OApp Security Stack specifies the `sendLibrary` and `receiveLibrary` to use for each chain it spans. The `sendLibrary` is the ID and version of the library to use when sending messages, and the `receiveLibrary` is the ID and version of the library to use when receiving messages. This configurability enables OApps to customize Security Stack cost and security based on their individual needs. For rapid prototyping, LayerZero implements an opt-in mechanism to allow OApps to lazily resolve their Security Stack to the defaults chosen and maintained by the LayerZero admin, but OApp owners are strongly encouraged to explicitly set their Security Stack for production applications. Messages cannot be received on a path until an OApp has set their Security Stack or opted into the default Security Stack for that path.

The impossibility of coordinating atomic transactions over an asynchronous network [2, 5] necessitates a live migration protocol when reconfiguring the OApp Security Stack.

Upgrading to a MessageLib with the same major version, but different minor version (e.g., $1.1 \rightarrow 1.2$) is achieved by simply setting the `sendLibrary` and/or `receiveLibrary` to the desired version. Migrating between MessageLibs with different major versions (e.g., $1.2 \rightarrow 2.0$) is more involved. First, the OApp sets a grace period for the old `receiveLibrary` (1.2) during which `receiveLibrary` 1.2 can continue to receive messages even if the `sendLibrary` is configured to 2.0 by the OApp. Next, the OApp sets the `sendLibrary` to the new version (2.0). After this point all new messages carry the new packet version, but both `receiveLibrary` 1.2 and 2.0 can verify messages until the grace period elapses. After the grace period elapses, only 2.0 is authorized to verify messages. If the grace period ends before all version 1.2 in-flight messages are committed to the destination Endpoint, packet delivery will temporarily halt and the OApp must reconfigure `receiveLibrary` back to the previous version (1.2) to commit all in-flight messages before updating `receiveLibrary` to 2.0 again. In this way, LayerZero implements intrinsically secure, frictionless live migration between MessageLib versions.

3.3 Decentralized Verifier Network

The datalink in LayerZero is designed on the fundamental observation that connecting two blockchains without the assumption of synchrony requires communication through one or more third parties [5]. A consequence of the potentially-offchain nature of DVNs is the impossibility of guaranteeing long-term immutability and availability, and as such a permissioned verification model is inherently unable to provide strong guarantees of channel liveness. Thus, we have opted to implement a permissionless, configurable verification model in LayerZero, where anyone can operate and permissionlessly integrate their own DVN with LayerZero. Decentralized Verifier Networks, as the name suggests, are composed internally of a set of verifiers that collectively perform distributed consensus to safely and reliably read packet hashes from the source blockchain; this design importantly allows for client diversity within a single DVN, minimizing the chance of a single faulty verifier causing protocol outages or errors.

This model overcomes two glaring shortcomings of other messaging services: shared security and finite fault tolerance. Existing cross-chain messaging services provide a single shared security configuration that is shared

Type	Structure
Execution gas	[TYPE_1, executionGas]
Gas and native drop	[TYPE_2, executionGas, nativedropAmount, receiverAddress]
Composite	[TYPE_3, [workerID, opType, length, command], ...]

Table 4: Message Options starts with a magic number to identify the options type, followed by type-specific options. Type 1 and 2 are specialized for setting execution gas limits and sending additional native gas tokens as part of an omnichain transaction respectively. Type 3 embeds arguments for an arbitrary set of offchain workers.

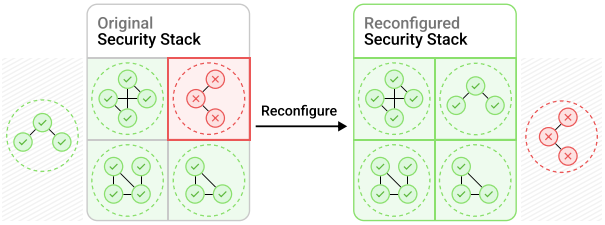

Figure 7: OApps can easily reconfigure their Security Stack to exclude faulty DVNs.

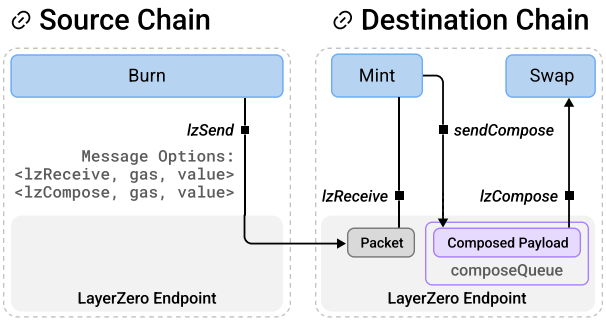

Figure 8: `lzCompose` enables chain-agnostic composition with liveness and safety closures.

across all clients, and this is suboptimal in two ways: it gives OApps no recourse if the protocol-specified verifier set is compromised or faulty, and it requires the majority of OApps to choose between an unnecessarily expensive or excessively risky security configuration for the task at hand. Finite fault tolerance is a problem that plagues all messaging services with a permissioned verification model, as the failure of the entire finite-sized permissioned set of operating organization(s) results in the permanent, unrecoverable failure of the whole protocol.

Through permissionless operation of DVNs, LayerZero is able to provide a practically unbounded degree of fault tolerance. Even if *all* existing DVNs lose liveness from software bugs, security breaches, natural disasters, and/or operational/governance concerns, OApp developers can operate their own DVNs to continue operation of the protocol. OApps can seamlessly reroute traffic through Security Stack reconfiguration, enabling recovery from compromised offchain infrastructure (Figure 7).

3.4 Executors

Implementing and updating extrinsically secure code is resource-intensive due to stringent security testing and auditing requirements. This engineering challenge stands in conflict with our goal of making LayerZero easily extensible to support the needs of a wide variety of omnichain applications. LayerZero solves this problem by separating verification from execution; any code that is not security-critical is factored out into *executors*, which are permissionless and isolated from the packet verification scope. This separation between security-critical and "feature" code between MessageLib and executors respectively provides two main benefits. First, it allows developers to use, implement, and compose feature extensions without considering security; the Endpoint prevents executors from delivering unverified or non-lossless messages, completely isolating packet execution from verification. Second, it decouples security and liveness in LayerZero, ensuring that a faulty executor cannot unilaterally prevent message delivery. By isolating the verification and execution layers in this manner, applications can easily debug which layer an error originated from.

When an OApp sends a LayerZero message, it specifies all offchain workers (e.g., executors, DVNs, etc.) and corresponding arguments through a MessageLib-interpreted byte array called Message Options. The executors then wait for the Security Stack to verify the packet before taking action based on the commands encoded in Message Options.

We purposely left Message Options unstructured (see Section 4), as a more restrictive API only serves to limit the potential capabilities of future onchain and offchain workers. At the protocol level, our primary objective is to provide the highest degree of extensibility, rather than creating an interface that is specialized to the needs of current applications.

The isolation of executors from any verification-related code indirectly improves channel validity, and permissionless execution directly improves channel liveness. This design reduces the code footprint of MessageLib and, by extension, minimizes the potential to

introduce attack surfaces into security-critical code. In addition, permissionless operation of executors ensures that channel liveness can be recovered in the event of executor failure, and fully decouples the liveness of the protocol from any single organization or entity. Once a message is verified by the Security Stack, anyone willing to pay the gas cost can permissionlessly execute the message. This theoretically allows even end-users to manually trigger OApp recovery following executor failure.

4 Extensions

In this section, we illustrate the flexibility of LayerZero through several examples of how the protocol can be extended with additional execution features.

4.1 Message Options

While there is no single standard format for serializing arguments into Message Options, we do not expect developers to write specialized code to support Message Options for every MessageLib. To address this, LayerZero currently defines three standardized formats for Message Options (see Table 4) to facilitate backwards-compatibility between library versions.

Types 1 and 2 specify arguments for a single executor to execute commonly-required functionality, while type 3 encodes a list of (workerID, option) tuples to allow for an arbitrary number of arguments passed to an arbitrary number of workers. Any message delivered by a executor has already been verified by the verification layer, allowing any number of executors to perform arbitrary actions without compromising message integrity.

4.2 Semantically uniform composition

LayerZero defines a universally standardized interface for cross-chain composition: `lzCompose`. Composing the destination chain delivery transaction with other contracts may seem trivial to those familiar only with EVM, which provides a native mechanism for arbitrary runtime dispatch. However, even existing MoveVM-based blockchains do not natively support this feature, invalidating the universality of EVM-style runtime dispatch composition semantics. When composing contracts, the receiver first stores a composed payload into the endpoint (`sendCompose`), after which it is retrieved from the ledger and passed to the composed callback by calling `lzCompose`. This design, while superficially inefficient on EVM-based chains, unifies composition semantics across all blockchains.

`lzCompose` provides a semantically universal standard composition primitive that inherits the same only-once lossless execution semantics of LayerZero messaging, and allows OApps to define a single application architecture that universally scales to all existing and future blockchains. Figure 8 illustrates an example of using `lzCompose` to bridge and swap a token in a single LayerZero transaction.

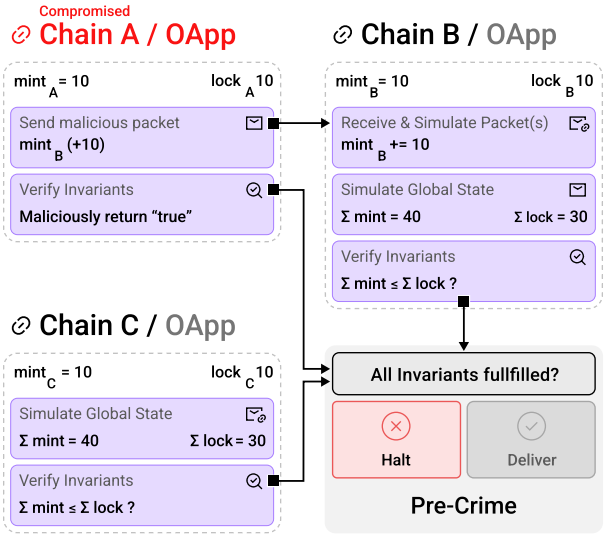

Figure 9: Pre-Crime rejects malicious and malformed messages by checking OApp-specified invariants.

The `lzCompose` primitive is a powerful tool for defining closures for data validity and channel liveness, isolating each composed contract from potential integrity violations by other contracts. Once data is stored using `lzCompose`, the liveness and integrity are "closed", meaning errors in composed contracts can cause loss of liveness or data validity only *within* the closure. Isolation of composed contract faults to the closure scope greatly simplifies reasoning about potential attack surfaces such as reentrancy.

An additional benefit of `lzCompose` is a uniform interface for tracing and analysis of a potentially deep call stack for complex multihop omnichain transactions, giving OApp developers a powerful tool to tackle the potentially daunting task of debugging omnichain code.

4.3 Application-level security

It is impossible to use the Message Options interface to extend the verification scope to include *additional* data, but OApps can use it to detect and filter out verified-yet-malicious messages (e.g., buggy messages that would trigger OApp-level faults). We introduce our novel offchain application-level security mechanism called *Pre-Crime*, which provides an additional layer of application-specific packet filtering on top of the existing LayerZero protocol. Pre-Crime enables any subset of peers (i.e., some or all of an OApp's contracts) to enforce appli-

cation security invariants after simulating the result of packet delivery. The invariant check results are collated by an offchain worker, which halts delivery of the corresponding packet if any peer reports a violated invariant.

Figure 9 illustrates the example of checking the total outstanding token count in a 3-chain token bridge. To use Pre-Crime, the OApp encodes the DVN address and Pre-Crime specific arguments in Message Options. The OApp specifies the invariant that the total minted tokens across all chains $\sum mint$ is less than or equal to the total locked liquidity $\sum lock$. Initially, $mint_A = mint_B = mint_C = 10$, and $lock_A = lock_B = lock_C = 10$. Chain A is compromised and tries to request an additional mint of 10 tokens on chain B without locking any additional assets. Pre-Crime detects this after checking token counts for all chains, isolating the security breach to a single chain (chain A). The receiver can then skip the nonce if necessary by calling `skip` (Table 3). It is important to note that Pre-Crime does not add any additional *protocol* security, and cannot protect data integrity (malicious DVNs or blockchain-level faults).

Pre-Crime and `lzCompose` are just two examples of execution features that are supported by LayerZero. The flexibility of Message Options and the separation between verification and execution enables a LayerZero to be extended to a wide variety of execution features.

5 Conclusion

In this paper, we presented the design and implementation of the LayerZero protocol. LayerZero provides intrinsically secure cross-chain messaging with universal semantics to enable a fully connected omnichain mesh network that connects all blockchains within and across compatibility groups.

By isolating intrinsic security from extrinsic security, LayerZero guarantees long-term stability of channel integrity and gives OApps universal network semantics across the entire mesh network. LayerZero's universal network semantics and intrinsic security guarantees enable secure chain-agnostic interoperation.

Our novel onchain verification module, MessageLib, implements extensible extrinsic security in an intrinsically secure manner. Each OApp has exclusive permission to modify its Security Stack, which defines the extrinsic security (MessageLib, DVNs) of their messaging channel. The immutability of existing MessageLibs ensures that no entity, including the protocol administrator, can unilaterally compromise OApp security.

LayerZero's isolation of execution features from packet verification allows a near-unlimited degree of freedom to implement additional features without affecting security. In addition, the separation of execution and verification in LayerZero reduces engineering costs, attack surfaces, and improves overall protocol liveness. Together, these components create a highly extensible protocol that can provide universal messaging semantics across existing and future blockchains. LayerZero is the protocol that brings consistency and simplicity to a landscape of scattered, ad-hoc messaging services, and lays the foundation for the fully-connected omnichain mesh network of the future.

References

[1] BEHNKE, R. Explained: The nomad hack (august 2022). https://www.halborn.com/blog/post/explained-the-nomad-hack-august-2022.

[2] GMYTRASIEWICZ, P. J., AND DURFEE, E. H. Decision-theoretic recursive modeling and the coordinated attack problem. In *Proceedings of the First International Conference on Artificial Intelligence Planning Systems* (San Francisco, CA, USA, 1992), Morgan Kaufmann Publishers Inc., p. 88–95.

[3] HACXYK. Wormhole $10m bounty. https://twitter.com/Hacxyk/status/1529389391818510337.

[4] THORCHAIN. Post-mortem: Eth router exploits 1 & 2, and premature return to trading incident. https://medium.com/thorchain/post-mortem-eth-router-exploits-1-2-and-premature-return-to-trading-incident-2908928c5fb.

[5] ZAMYATIN, A., AL-BASSAM, M., ZINDROS, D., KOKORIS-KOGIAS, E., MORENO-SANCHEZ, P., KIAYIAS, A., AND KNOTTENBELT, W. J. Sok: Communication across distributed ledgers. Cryptology ePrint Archive, Paper 2019/1128, 2019. https://eprint.iacr.org/2019/1128.

[6] ZARICK, R., PELLEGRINO, B., AND BANISTER, C. Layerzero: Trustless omnichain interoperability protocol. https://arxiv.org/abs/2110.13871, 2021.