Formal Analysis of IBC Protocol

Qiuyang Wei*†‡, Xufeng Zhao*†, Xue-Yang Zhu*†#, Wenhui Zhang*†
* State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
‡ Hangzhou Institute for Advanced Study, University of Chinese Academy of Sciences, Hangzhou, China
† University of the Chinese Academy of Sciences, Beijing, China
{weiqy,zhaoxf,zxy,zwh}@ios.ac.cn

***Abstract*—Since the inception of Bitcoin in 2008, blockchain technology has had a significant impact on many fields. The lack of effective communication between heterogeneous and isolated blockchains, however, restricts the promotion and ecological development of blockchain industry. In this context, cross-chain technology has rapidly developed and become a new research hotspot. Due to the decentralized nature of blockchain and the complexity of cross-chain scenarios, cross-chain technology faces huge security risks. In this paper, we propose formal analysis of the IBC protocol, one of the most popular cross-chain communication protocols, aiming to help developers design and implement cross-chain technologies more securely. We formalize three main components of the IBC protocol with TLA^+, a temporal logic specification language, and verify some important requirements with model checking tool TLC. The verification results are analyzed comprehensively. Issues found through our formal analysis have been reported to the community, most of which have been acknowledged. We also propose some recommendations for removing potential risks.**

***Index Terms*—Cross-chain, IBC, formal analysis, TLA^+**

I. Introduction

Blockchain is a decentralized distributed ledger that realizes a secure and credible network through cryptography technology [1]. Since the groundbreaking paper [2] published by Satoshi Nakamoto on October 31, 2008, thousands of blockchains have sprung up, sparking a new wave of information technology. Blockchains like Bitcoin [2] and Ethereum [3] are gaining rapid traction with the market cap reaching over a trillion dollars [4], attracting a lot of attention from both industry and academia. Blockchain technology not only brings revolutionary changes to the economic field, but also profoundly impacts politics, society, science, and other aspects [5]. However, most blockchains adopt different technical frameworks and design philosophies for different purposes and applications. For example, Ethereum supports smart contacts and uses Proof of Stake [6] while Bitcoin is primarily designed as a cryptocurrency base on Proof of Work [7]. There is a lack of effective communication methods between heterogeneous and isolated blockchains [8]. With the rapid development of the blockchain industry, this diverse and fragmented ecosystem introduces tremendous and growing demands for the interoperability of data and functionality, eventually motivating the emergence of cross-chain technology.

# Corresponding author.

This work is partially supported by the National Natural Science Foundation of China (No. 62072443).

As a bridge between different independent blockchains, cross-chain technology realizes the function of value and information exchange. However, most cross-chain solutions mainly focus on exchanging assets between public blockchains. For example, Interledger [9] is a payment protocol for sending value across different blockchains; BTC Relay enables users to pay with Bitcoin to use Ethereum decentralized applications [10]. These cross-chain approaches provide convenience and atomicity guarantees for transacting fungible and non-fungible assets in different cryptocurrencies. In order to provide users with higher flexibility and throughput, multi-chain frameworks have emerged, providing reusable data, network, consensus, and incentive layers to create interoperable application-specific blockchains [11]. Multi-chain frameworks require a more foundational and universal method than assets exchange to communicate between parallel blockchains. These methods are called cross-chain communication protocols. Cosmos Network is one of the most popular multi-chain platforms, where separated and sovereign chains are connected with Inter-Blockchain Communication (IBC) [12] protocol. Up to now, the total IBC market cap is more than $7.57B among 46 IBC-enabled chains [13].

Although cross-chain technology connects isolated blockchains and promotes the development of the blockchain ecosystem, it also weakens the security of the blockchain system. Indeed, the exploit of Ronin bridge creates the largest attack in the history of DeFi so far [4]. According to statistics, assets worth of over a billion dollars were stolen or locked in blockchain systems last year due to improper design, implementation, or deployment [14]. Due to bugs in on-chain contracts, Poly Network [15] lost about $600M, and Wormhole [16] was stolen more than $300M. These attacks have caused considerable losses to users and shaken their confidence in cross-chain technology. However, cross-chain technology is too complex for developers to comprehensively and rigorously audit their designs and implementations manually, especially in an untrusted and uncertain operating environment like blockchains.

To the best of our knowledge, this paper is the first to use formal methods to improve the security and reliability of cross-chain communication protocols. We choose the IBC protocol as an example to show how helpful the formal analysis can be in such an uncertain and complex scenario. The IBC protocol includes the core transport, authentication, ordering (TAO) layer, and the application (APP) layers describing packet encoding and processing semantics. At present, our work mainly concerns the core TAO layer. The contributions of this paper are summarized as follows.

- *Formalization of protocol requirements.* We extract and formally interpret the requirements that the IBC protocol is expected to satisfy. These formal interpretations can provide a better understanding to developers and users of the IBC protocol.
- *Formal modeling of the protocol.* We model the core TAO layer of the IBC protocol in TLA^+ [17] based on assumptions that the involved blockchains are secure and the light clients could correctly verify headers and proof submitted by the relayers. We focus on the inter-blockchain security, which is affected by on-chain modules and off-chain relayers. Our model is detailed enough for analysis and still amenable to experiments.
- *Analysis and recommendations of the protocol.* Based on our model, we analyze the current IBC protocol design and identify some important issues, such as incomplete handshake and trapped packets. Issues found through our analysis have been reported to the community, and most of them have been acknowledged. We also propose recommendations against potential risks.

The remainder of this paper is organized as follows. Related work is reviewed in Section II. We present the architecture and key components of the IBC protocol in Section III. We carry out a systematic interpretation of the security assumptions and requirements of the IBC protocol in Section IV. In Section V, we explain the basics of TLA^+ and our modeling methods. We present our analysis and recommendations of the IBC protocol in Section VI. Finally, Section VII concludes.

II. Related Work

A. Cross-Chain and Multi-Chain

Although there have already been some available cross-chain solutions in the blockchain industry, cross-chain is still a new topic in academia. Previous research mainly focuses on organizing and analyzing available cross-chain solutions from different perspectives. Both Cao et al. [18] and Ou et al. [8] proposed a classification of existing cross-chain technologies based on their mechanism. Cao et al. also proposed and tested an atomic exchange cross-chain protocol, while Ou et al. further analyzed existing cross-chain projects according to the technical classification. Robinsom [10] presented cross-chain communication protocols based on the usage scenarios, including value swapping, cross-chain messaging, and blockchain pinning. Belchior et al. [11] focused on blockchain interoperability and discussed the requirements, challenges, and existing attempts to achieve interoperability in different blockchain systems. Some studies were also interested in the design and analysis of different cross-chain atomic exchange methods. Nehaï [19] formally analyzed cross-chain swap protocols. And Pillai [20], [21] paid attention to the burn-to-claim protocol. Herlihy [22] proposed the cross-chain deal as a new notion for distributed exchanges. At the same time, several researchers have also begun to pay attention to the security issues of cross-chain technology. Lee et al. presented the first systematization of the attacks on cross-chain bridges in recent years [14]. Zhang et al. documented the classes of attacks in cross-chain bridges and proposed the Xscope to detect security bugs for defense [23].

Polkadot [24] and Cosmos [25] are the most widely adopted multi-chain frameworks. Parachains in Polkadot communicate through the Cross-chain Message Passing Protocol (XCMP) [26] while parallel blockchains in Cosmos are connected utilizing the IBC protocol. Although there are materials such as documents and whitepapers to introduce their design and vision, to the best of our knowledge, there is a lack of research explicitly targeting the security and reliability of cross-chain communication protocols in multi-chain frameworks.

B. Formal Analysis of Protocol

Formal methods are widely used in network protocol verification, ranging from high-level abstract models to concrete implementations. Basin et al. [27] proposed a formal and comprehensive 5G AKA protocol analysis using Tamarin. Kobeissi et al. [28] implemented and analyzed a variant of the popular Signal Protocol with automated verification tools. Zhang et al. [29] presented a security analysis of the QUIC handshake protocol based on ProVerif. There are also studies using formal methods to analyze other aspects of the network. Prabhu et al. [30] combined equivalence partitioning with explicit-state model checking to verify network configuration. TLA^+ [17] is often used to specify distributed and concurrent systems. In network protocol verification, TLA^+ based methods focus more on functional properties rather than encryption-related security properties [27]. Yin et al. [31] utilized TLA^+ to study the correctness of an atomic broadcast protocol that supports additional crash recovery. Akhtar et al. [32] verified the correctness of the Message Queue Telemetry Transport protocol using a TLA^+ based formal method.

There are a few attempts to apply formal methods to the blockchain field. Kuaharenko et al. [33] and Braithwaite et al. [34] are concerned with the consensus mechanism used in the blockchain. Grundmann and Hartenstein [35] attempted to specify functionality and security properties for a payment channel protocol. Overall, research on the application of formal methods to protocols in the blockchain is limited.

III. IBC Protocol

In this section, we illustrate how cross-chain communication is achieved in the IBC protocol, closely following the official standards [36]. To better understand the protocol, we also refer to some implementations of the IBC protocol in different programming languages [37], [38]. We first present the general architecture and afterwards the main components of the IBC protocol.

A. Architecture

Fig. 1 presents the overall architecture of cross-chain communication between two blockchains through the IBC protocol. *Light clients*, *connections*, *channels*, and *modules* are the

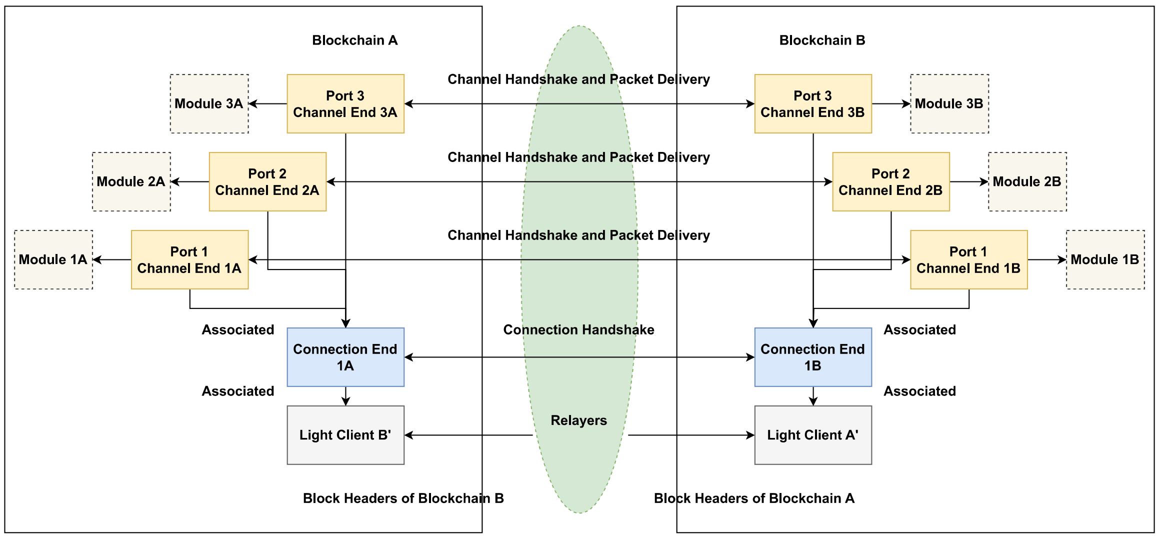

Fig. 1: Architecture of Cross-chain Communication via IBC Protocol

key entities of the blockchain, while *relayers* are off-chain processes that can access the source and target chains.

A light client is the lightweight representation of a blockchain. Instead of storing the entire history of all messages or executing transactions, light clients are designed to efficiently verify the existence of a particular message on the referred blockchain. This concept is not specific to the IBC protocol but is proposed to fulfill the Simplified Payment Verification (SPV) in Bitcoin [2]. In brief, the light client records a sequence of verified block headers of the referred blockchain. To prove that a message exists on the blockchain, a user need to generate cryptographic proof about the message. Then the light client can calculate the root hash through the hash of this message and the generated proof and then compare it with the hash value recorded in its block headers. Cryptographic primitives guarantee that only existing messages with a correct proof can be used to compute a matching root hash [39]. In other words, using light clients allows blockchains to exchange messages without a trusted third party.

Blockchains do not directly send messages to each other. Instead, relayers, the untrusted off-chain processes, are responsible for relaying messages between blockchains by scanning each chain, constructing appropriate datagrams, submitting transactions, and executing them on the corresponding chain as required. The IBC protocol abstracts data structures and handler functions related to cross-chain communication into two layers: connections and channels. The connection is an abstraction consisting of two ends on two separate chains, each associated with a light client representing the opposite chain. Connections utilize the functionality of light clients to provide cross-chain state verification to channels. The channel serves as a conduit for transferring packets between modules on different chains. Each channel is associated with a particular connection, while a connection may have many associate channels. In this way, different channels can share a connection to amortize the cost of cross-chain validation.

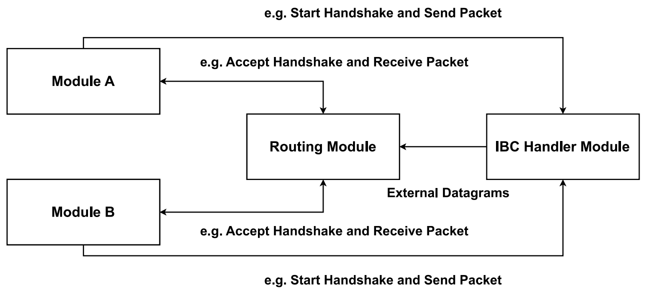
e.g. Start Handshake and Send Packet
Module A
e.g. Accept Handshake and Receive Packet
Routing Module
IBC Handler Module
External Datagrams
Module B
e.g. Accept Handshake and Receive Packet
e.g. Start Handshake and Send Packet

Fig. 2: Layers of Modules

Modules are important components of blockchains in the Cosmos ecosystem. They implement the business logic of applications, similar to smart contracts in the Ethereum ecosystem. The module encapsulates the data that needs to be transmitted into *packets* and sends them to the module on another chain through the IBC protocol. The module needs to bind a port to create a channel to send and receive packets. In practice, however, relayers do not send datagrams directly to the module. A secondary routing module exists between the IBC handler module and other modules (see Fig. 2). Modules need to call the IBC handler individually to deal with handshakes and packets, while the routing module is responsible for receiving external datagrams and managing modules. When a datagram is received, the routing module calls the corresponding module according to the lookup table, simplifying the work of relayers.

The IBC protocol includes a series of standards. The most important ones are the standards for connections, channels, and packets, which define the data structures and handler functions related to cross-chain communication. Therefore, we further explain these three standards. For better understanding, we simplify the code shown in following sections.

B. Connection Standard

The standard of connections describes how to complete the handshake and version negotiation between two connection ends on two chains. The connection end is a data structure that stores the connection state and information. There are four datagrams involved in connection handshake: `ConnOpenInit`, `ConnOpenTry`, `ConnOpenAck`, and `ConnOpenConfirm`, each of which corresponds to a handler function that is executed when the associated datagram is submitted in a transaction. For connection handshake, the processes of datagrams are simply encapsulations of handler functions.

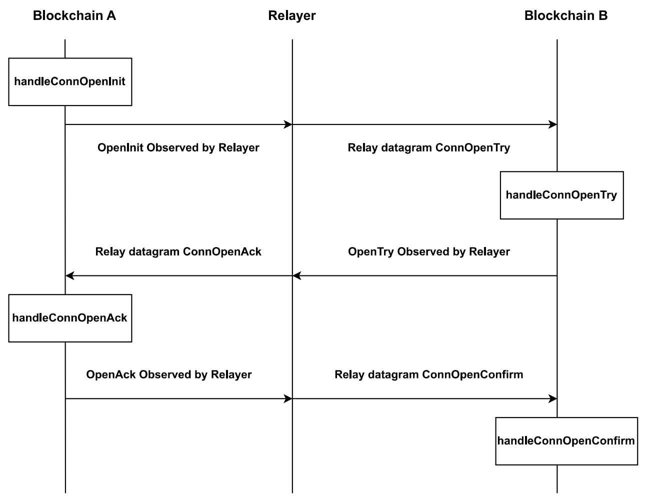
Blockchain A
Relayer
Blockchain B
handleConnOpenInit
OpenInit Observed by Relayer
Relay datagram ConnOpenTry
handleConnOpenTry
Relay datagram ConnOpenAck
OpenTry Observed by Relayer
handleConnOpenAck
OpenAck Observed by Relayer
Relay datagram ConnOpenConfirm
handleConnOpenConfirm

Fig. 3: Connection Handshake

The connection handshake consists of four steps (see Fig. 3). In the following, we focus on the initialization and transitions of connection ends during the handshake (for example, chain A wants to establish a connection with chain B).

1) *OpenInit.* An actor (e.g. an end user) on chain A calls the handler function `handleConnOpenInit` to initialize a connection attempt on chain A, which creates an `INIT` connection end with a unique identifier (e.g. `conn-a`). The connection end does not know its counterparty at this point.
2) *OpenTry.* When an `INIT` connection end is observed on chain A, the relayer relays the connection attempt to chain B by sending `ConnOpenTry` datagram along with the proof of chain A state. If the state of chain A is successfully verified by its client, Chain B creates a `TRYOPEN` connection end with a unique identifier (e.g. `conn-b`) and specify `conn-a` as its counterparty.
3) *OpenAck.* The relayer relays the acceptance of a connection attempt and proof of chain state from chain B back to chain A. In this step, connection end `conn-a` updates its state to `OPEN` and specify `conn-b` as its counterparty connection end.
4) *OpenConfirm.* The relayer sends the corresponding datagram and proof to chain B to confirm the establishment of the connection, after which both connection ends are in state `OPEN`.

Except that step `OpenInit` can actively initiate a connection handshake attempt, both chains can only proceed to the next step after receiving the corresponding datagram relayed by off-chain relayers and verifying that the counterparty chain is in the correct state. In addition, connections once opened cannot be closed, and identifiers cannot be reallocated.

C. Channel Standard

Similar to the connection, a channel is an abstraction of two channel ends. And the channel is established between ports, which are exclusively owned by modules. Therefore, the process of channel handshake datagrams consists of the handler module's handler function and the corresponding module's callback function.

There are three types of channels:

- *Unordered.* For unordered channels, packets can be delivered in arbitrary order, which may differ from the order in which they were sent.
- *Ordered.* For ordered channels, packets must be delivered in the order they were sent. If a packet in the sequence times out, the channel is closed, and all subsequent packets are not be received.
- *Ordered_allow_timeout.* This channel is a less strict version of the ordered channel. If a packet times out, the channel is not closed and the remaining packets continue to be processed in order.

Channel opening handshake is very similar to connection handshake, which also defines four datagrams: `ChanOpenInit`, `ChanOpenTry`, `ChanOpenAck`, and `ChanOpenConfirm` and requires four steps: `OpenInit`, `OpenTry`, `OpenAck`, and `OpenConfirm`. Therefore, we do not elaborate on its details.

The major difference is that the channel is allowed to be closed. And the channel cannot be reopened once closed. Channel closing handshake involves two steps:

1) *CloseInit.* As long as the channel end is initialized and not closed, the module that owns the channel can actively send a datagram `ChanCloseInit` to close it.
2) *CloseConfirm.* When a channel end is observed to be closed, the relayer notifies the counterparty module to close the channel by submitting a datagram `ChanCloseConfirm`.

D. Packet Standard

Packets encapsulate application data and are transmitted through channels. The following steps explain the life cycle for a packet sent from module 1 on blockchain A to module 2 on blockchain B.

1) Light clients and ports setup. The associated light clients are created on both chains, and each module is bound to a port.
2) Establishment of a connection and a channel, optimistic sending.
   a) Connection handshake from chain A to chain B is initialized by module 1.
   b) Channel opening handshake from module 1 to module 2 is initialized associated with the connection end newly created.

    c) Packet can be optimistically sent using the initial channel end.
3) Completion of the handshake. Connection handshake and channel opening handshake are successfully completed in order.
4) Packet reception. Module 2 receives the packet based on the channel order type and whether it has timed out.
5) Acknowledgement and timeout. If an acknowledgment relayed from module 2 is received, module 1 can execute the acknowledgment process. The timeout process can be executed if the sent packet has exceeded the specified height or timestamp or the counterparty channel has been closed.

Fig. 4 and Fig. 5 further demonstrate the state transitions of chain A and chain B starting from sending packets.

State 1 is the initial state of chain A before sending a packet. The variable `commitments` is used to store the commitment of the sent packet, while variable `nextSequenceSend` and variable `nextsSequenceAck` respectively record the sequence numbers of the next sent and acknowledged packets. Module 1 can call function `sendPacket` to send a packet, which adds the corresponding commitment to variable `commitments` and increases variable `nextSequenceSend` by 1 (see State 2). For unordered channels, calling function `acknowledgePacket` or `timeoutPacket` only deletes the corresponding commitment (see State 3). For the other two kinds of channels, `acknowledgePacket` deletes the corresponding commitment and increase variable `nextsSequenceAck` (see State 4), while `timeoutPacket` results in different states (see State 4 and State 5). When encountering a timeout, the ordered channel is directly closed while the other one only skips the packet. When the packet is not receivable due to module 2 closing the channel, module 1 can call function `timeoutOnClose` to delete the corresponding commitment and the channel state may have already been closed because of the closing handshake.

State 7 is the initial state of chain B. Variable `receipts` stores the receipts for the received packets, while variable `acks` stores the acknowledgments that will be sent to chain A for processing, and variable `nextSequenceRecv` records the sequence number of the next received packet. Depending on the channel type and timeout situation, calling function `recvPacket` results in different state transitions (see State 8, 10, 11). Only after successfully receiving the packet function `writeAcknowledgement` can be called to generate an acknowledgment (see State 9, 12).

IV. Security Assumptions and Requirements

A. Security Assumptions

Our analysis is based on the following security assumptions for participants and entities of the IBC protocol.

1) Assumptions on blockchains: The IBC protocol provides a mechanism for independent blockchains to communicate with each other, which is responsible for ensuring interchain reliability and is indifferent to Byzantine errors within the blockchain, such as failure of the consensus mechanism. In addition, the IBC protocol applies to heterogeneous blockchains of different consensus mechanisms, including probabilistic-finality consensus algorithms. Thus, we assume that the involved blockchains are secure.

2) Assumptions on light clients: Similar to SPV used in Bitcoin [10], the light client used in the IBC protocol can combine a block header and a cryptographic proof to verify the inclusion or non-inclusion of particular values at particular paths in the blockchain state. Cryptographic primitives guarantee that incorrect values cannot be used to get a correct root hash to match one of the block headers. In order to record a list of correct block headers for the associated chain, the light client needs to validate whether the target chain has confirmed the block headers submitted by relayers. Validity prediction reflects the behavior of the associated blockchain and its consensus. Considering the above, we assume that the light client can correctly verify the block header and proof submitted by the relayer.

In our analysis of the IBC protocol, we mainly focus on the security issues brought by off-chain relayers and on-chain modules in handshake and packet transmission. We aim to explore the behavior of modules and relayers as much as possible to analyze the security risks that users may face thoroughly.

B. Requirements

We extract and interpret the requirements that the IBC protocol should meet from official standards.

- Connection Properties. Connection handshake indicates mutual negotiation and preparation for subsequent channel handshake and packet delivery.
  - *Only the appropriate handshake actions can be executed in order.* To complete the connection handshake correctly, the handshake must strictly follow the order of `OpenInit`, `OpenTry`, `OpenAck`, and `OpenConfirm`. The local connection end can step forward only if the counterparty connection end is verified in the correct state.
  - *Initiated handshake negotiations will eventually be completed.* Participants bear the costs of on-chain transactions and off-chain relays. The connection end that has not completed the handshake will freeze in the current state, which not only wastes occupied resources but also interferes with the normal use of users. Thus, each initiated connection attempt is supposed to be established with the negotiated version.
  - *The created connection ends on both chains eventually contain the consistent states.* If the connection is eventually established, both connection ends should have matching states, such as counterparty connection identifier, counterparty client identifier, version, etc.

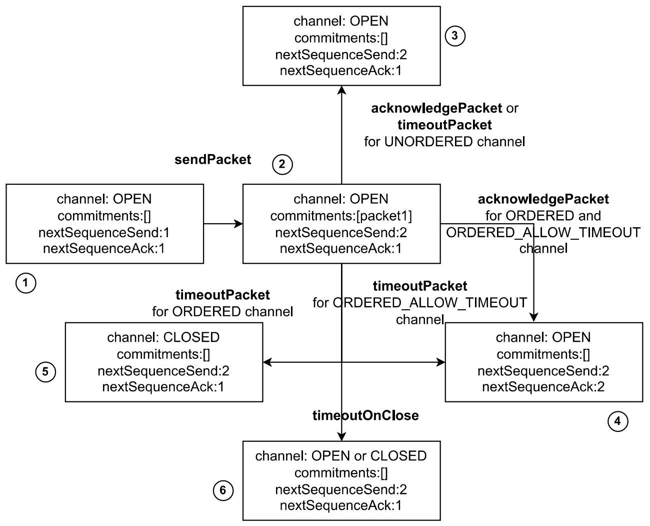

Fig. 4: State Transition of Blockchain A

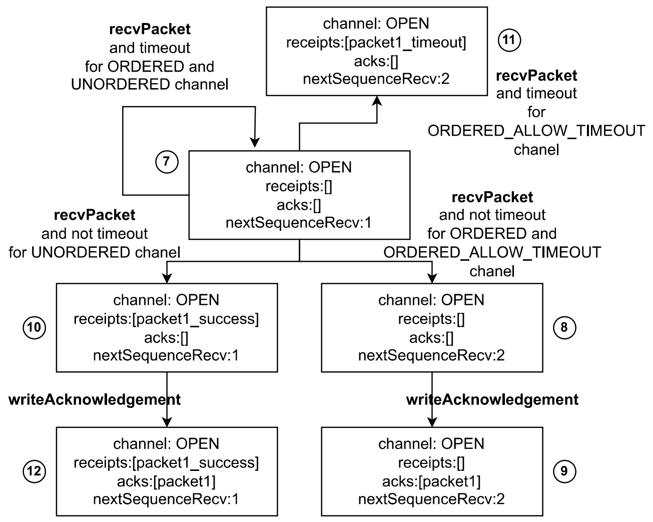

Fig. 5: State Transition of Blockchain B

- Channel Properties. Channels have a handshake process similar to Connections, but channels are allowed to close while connections are not.
  - *Only the appropriate handshake actions can be executed in order.* Opening handshake must follow the order of `OpenInit`, `OpenTry`, `OpenAck`, and `OpenConfirm`. However, as long as the current channel is not closed, either party can initiate a closing handshake, which follows the order of `CloseInit`, `CloseConfirm`.
  - *If no one initiates a closing handshake, the opening handshake will eventually be completed. Otherwise, the closing handshake will ultimately be completed.* If no one closes the channel, both channel ends should end up in the `OPEN` state. Otherwise, they should be `ClOSED` state.
  - *The created channel ends on both chains eventually contain the consistent states.* Both channel ends should have a consistent state, such as counterparty channel identifier, counterparty port identifier, version, etc., regardless of whether they have completed the opening or closing handshake.
- Packet Properties. Packet processing is the foundation of the upper-layer application logic. Therefore, packet transmission requires a correct result to ensure the proper operation of the upper-layer application.
  - *Only the appropriate actions for packets can be executed in order.* Only packets that have been sent can be received or time out. And only received packets can be acknowledged.
  - *For ordered and ordered_allow_timeout channels, the packet sent first will be received first.* Except for unordered channels, packets should be processed in the order they were sent.
  - *A packet cannot be received or time out simultaneously.* Only packets that have not timed out can be received, and only unreceived packets can be declared timeout to indicate that the packet is no longer valid.
  - *For every packet sent, it either ends up being received or eventually times out.* Each packet sent should have a definite and unique result so that it can be processed correctly at the application layer.

V. Formal Model

In this section, we describe how to model the IBC protocol with TLA$^+$. The complete code has been made public [40].

A. *TLA$^+$ and TLC*

TLA$^+$ is a high-level specification language that is based on linear temporal logic. TLA$^+$ is usually used to model distributed and concurrent systems. In TLA$^+$, the system is represented as a state machine, and the execution of the system is equivalent to a series of states, which is called a behavior. The transitions between states are represented by actions over unprimed variables (old state) and primed variables (new state). There are special actions called stutter actions, where all variables retain the previous values. There is no formal distinction between a state machine and a property in TLA$^+$, both described by temporal logic formulas. Temporal operators mainly include two types: $\Box$ and $\Diamond$, and the former means that a formula is true in all states, while the latter means that a formula is true in at least one state.

The system specification is usually in the form of $Init/\backslash\Box\,[Next]\_vars/\backslash Fairness$. The predicate $Init$ represents all possible initial states of the system, while $Next$ defines a disjunction of all possible system actions. And $vars$ represents the tuple of all variables in the system. The formula $\Box\,[Next]\_vars$ indicates that either the system will execute according to the transitions specified by $Next$, or the previous state will remain unchanged. Usually, we need to constrain $Fairness$ to avoid the system being stuck in an infinite stutter state or some actions never being executed.

TLA$^+$ and its tools are usually used to eliminate design-level errors that are difficult to detect or correct at the code level. TLC is a common explicit-state model checker for TLA^+ that can check safety and liveness properties. In this paper, We use the VS Code extension of TLC [41].

B. Modeling Choices

Since explicit-state model checking is very time-consuming, we make some modeling choices to make the model detailed enough for analysis and amenable to experiments.

1) Abstraction of blockchains and light clients: Assuming the involved blockchains are secure, we do not delve into the implementation details of the blockchain. The blockchain in our model is just a host state machine that stores IBC protocol related data structures following standards. The implementation details of light clients and proof exceed the scope of IBC protocol standards. The role of the light clients is to ensure that the datagram submitted by relayers cannot be forged. In the model, we ensure that the relayer cannot forge non-existent messages by requiring the relayer to correctly describe the state of the counterparty chain.

2) Modeling layers of connections, channels, and packets: As we mentioned in Section III-A, we mainly focus on connections, channels, and packets. They have independent behaviors but are related. Connections are the most fundamental part of the three, providing state verification assistance for channels. Therefore, connections must be established before channels can be established. A connection remains in the same state once established, while a channel can be closed anytime after its establishment. Channels are the conduit for packet transmission. The result of sent packets depends on the states of the channel. Specifically, the module can optimistically initiate a channel opening handshake when the associated connection end has been created while the connection handshake has not yet been completed. However, all subsequent steps of channel handshake require that the associated connection end state is already `OPEN`. Therefore, the channel opening handshake can only be completed after the connection handshake is completed. Similarly, after the channel end is created, the module can optimistically send packets. However, receiving, acknowledging, and timing out packets all require that the associated channel end state is `OPEN`. Besides, modules can also declare packets sent timeout based on the state of the counterparty channel end being `CLOSE`. In summary, we model the IBC protocol from two perspectives: connections, channels&packets. We mainly focus on whether the connection handshake can always be successfully completed, and whether the packets sent can always have a definite and correct outcome regardless of the state of the channel.

3) Encapsulation of datagrams submission and execution of handler and callback functions: When a datagram is submitted to the routing module in a transaction, the associated handler function will be executed, including the corresponding module's callback function (if any). As mentioned, model checking is a state traversal method that is very time-consuming. It is too expensive and unnecessary to fully model the datagram submission (in a message queue form), the handler function execution, and the callback function execution. In order to simplify modeling and evaluation, we encapsulate the three parts into an action in TLA^+, which executes the whole corresponding handler function if the input parameters are correct, otherwise it does not change any variables.

4) Simulation of packet timeout: Modules determine whether the packet can be executed by comparing the current height of the destination chain with the specified timeout height in the packet. Due to the abstraction of the blockchain structure, we cannot directly model the packet timeout mechanism through this approach. However, we aim to verify whether the protocol can operate correctly in any scenario. So we can use a data structure to record whether each packet sent is determined to time out. The value of this data structure will be specified during model initialization and includes all possible scenarios. At the same time, in order to ensure that all packets will eventually time out, the values of packets in this data structure may be modified as timeout every time the model advances. Fig. 6 shows the code of this data structure, where `[Seqs -> BOOLEAN]` means a set of functions that map from the set `Seqs` to `BOOLEAN`, and `PacketIsTimeouts` is a set of structures.

```
PacketIsTimeouts ==
    [
        chainA: [Seqs -> BOOLEAN ],
        chainB: [Seqs -> BOOLEAN ]
    ]

Init ==
    /\ packetIsTimeOut \in PacketIsTimeouts
```

Fig. 6: Code of Variable packetIsTimeOut in TLA^+

C. TLA^+ model of IBC Protocol

In TLA^+, systems are modeled in a modular structure. For the IBC protocol, we mainly model it into two parts: chain and environment. The chain represents the blockchain, mainly including the data structures and handler functions related to connections, channels and packets. The environment represents an abstraction of possible behaviors of modules and relayers.

In the part of chain, we model the IBC protocol related data structures in a manner suitable for TLA^+. For example, we use arrays to store the connection and channel ends and use array indexes as identifiers, rather than searching for strings and storage paths in protocol standards. We convert the pseudo code in the standard into TLA^+ expressions based on semantics, and simplify variables and statements related to the implementation of blockchains, such as block heights, storage paths, etc. Fig. 7 shows the simplified code for the function `sendPacket`, and Fig. 8 shows the corresponding TLA^+ code. Since the statements in TLA^+ are not executed sequentially, we adjust the order of the conditional statements and other statements and define the required temporary variables in advance through the `LET` expression to enable the handler function in TLA^+ to execute all corresponding operations when the condition is met. At the same time, we add a condition `seq <= MaxSeq` to limit the verification scale and use an auxiliary variable `packetLog` to facilitate property description in TLA^+ code.

The environment needs to provide proof of the state of the counterparty chain when calling handler functions. In the modeling of connections, proof includes the state of the counterparty connection end. In the modeling of channels and packets, the proof contains the state of the counterparty channel end and data structures related to packet sending, receiving, and acknowledgment.

Our modeling simplifies the data structure and handler functions while preserving the semantics and behavior of the protocol, which is easier to verify by model checking tools.

```
function sendPacket(
  sourcePort,
  sourceChannelr,
  timeoutHeight,
  data):{
    channel = getChannel(sourcePort, sourceChannel)
    require(channel != null)
    require(channel.state != CLOSED)

    connection = getConnection(channel.connection)
    require(connection != null)

    require(timeoutHeight != 0)
    client = getClient(connection.clientIdentifier)
    latestClientHeight = client.latestClientHeight()
    require(latestClientHeight < timeoutHeight)

    sequence = getNextSequenceSend(sourcePort,
    ↪  sourceChannel)
    setNextSequenceSend(sourcePort, sourceChannel,
    ↪  sequence+1)
    setPacketCommitment(sourcePort, sourceChannel, sequence,
    ↪  hash(hash(data), timeoutHeight))
}
```

Fig. 7: Code of Function sendPacket

```
HandleSendPacket(sourcePort, sourceChannel) ==
    LET
        channel == getChannelEnd(sourcePort, sourceChannel)
        connection == getConnection(channel.connectionID)
        seq == getNextSeqSend(sourcePort, sourceChannel)
        log == LogEntry(sourcePort, sourceChannel, seq,
        ↪  "send")
        commit == Commit(sourcePort, sourceChannel, seq)
    IN
    /\ seq <= MaxSeq
    /\ channel /= nullChannelEnd
    /\ channel.state /= "CLOSED"
    /\ connection /= nullConnectionEnd
    /\ chainStore' = [chainStore EXCEPT
                      !.nextSequenceSend =
                          [chainStore.nextSequenceSend
                              EXCEPT ![sourceChannel] =
                              ↪  seq+1],
                      !.commitments =
                          [chainStore.commitments
                              EXCEPT ![sourceChannel] = @
                              ↪  \union {commit}]]
    /\ packetLog' = Append(packetLog, log)
```

Fig. 8: Code of Function sendPacket in TLA$^+$

D. Formalizing Requirements in TLA^+

In TLA$^+$, the properties are described by temporal logic formulas. There is a special temporal property called action property, which is the constraint on system actions. Action properties are usually expressed in the form of $\Box[P(x', x)]\_x$, which means that either the value of variable x remains unchanged or $P(x', x)$ is true, where $P(x', x)$ describes the relationship between the new and old values of variable x.

We use TLA$^+$ to formalize the requirements listed in Section IV-B. A requirement may be expressed in multiple formulas. For example, the requirement *Only the appropriate handshake actions can be executed in order* involves multiple constraints, such as the state cannot be rolled back, and the `OPEN` state can only be transited from `INIT` and `TRYOPEN`. Formula 1 and Formula 2 are two of its formulas.

$$\begin{aligned}
&SafaConnInit == \\
&\quad \forall\, chainID \in ChainIDs, clientID \in ClientIDs, \\
&\qquad connectionID \in ConnectionIDs : \\
&\quad LET\ connectionEnd == \\
&\qquad queryConnection(chainID, clientID, connectionID)\ IN \\
&\quad \Box\,(connectionEnd \neq nullConnectionEnd \\
&\qquad \land connectionEnd.state = \texttt{INIT} \\
&\quad\ \rightarrow \neg\,(\Diamond\ (connectionEnd.state = \texttt{UNINIT})))
\end{aligned} \tag{1}$$

$$\begin{aligned}
&SafeConnOpen == \\
&\quad \Box\,[\forall\, chainID \in ChainIDs, clientID \in ClientIDs, \\
&\qquad connectionID \in ConnectionIDs : \\
&\quad LET\ connectionEnd == \\
&\qquad queryConnection(chainID, clientID, connectionID)\ IN \\
&\quad \lor\ connectionEnd'.state = connectionEnd.state \\
&\quad \lor\ connectionEnd'.state = \texttt{OPEN} \\
&\qquad \rightarrow (connectionEnd.state = \texttt{INIT} \\
&\qquad\quad \lor\ connectionEnd.state = \texttt{TRYOPEN})]\_vars
\end{aligned} \tag{2}$$

Formula 1 is a regular temporal property composed of two operators, which means that the connection end cannot return to an uninitialized state from the `INIT` state. `chainID`, `clientID`, and `connectionID` represent the identities of the connection end, light client, and chain. `LET` operation is similar to redefinition in programming languages and `queryConnection` is an auxiliary function to access variables. Formula 2 is an action property that indicates if the current connection state is `OPEN` and is different from the previous state, the previous state can be inferred as `INIT` or `TRYOPEN`. For the complete formulas of the requirements, the reader may refer to our code [40].

VI. Evaluation and Analysis

A. Results

TABLE I: Verification Results of Requirements

No	Category	Requirement	Result
1	Connection	Only the appropriate handshake actions can be executed in order.	✓
2	Connection	Initiated handshake negotiations will eventually be completed.	×
3	Connection	The created connection ends on both chains eventually contain the consistent states.	✓
4	Channel	Only the appropriate handshake actions can be executed in order.	✓
5	Channel	If no one initiates a closing handshake, the opening handshake will eventually be completed. Otherwise, the closing handshake will ultimately be completed.	×
6	Channel	The created channel ends on both chains eventually contain the consistent states.	✓
7	Packet	Only the appropriate actions for packets can be executed in order.	✓
8	Packet	For ordered and ordered_allow_timeout channels, the packet sent first will be received first.	✓
9	Packet	A packet cannot be received or time out simultaneously.	✓
10	Packet	For every packet sent, it either ends up being received or eventually times out.	×

We use the model checking tool TLC to verify our models and requirements. TLC will stop the state search when a counterexample is discovered. In this case, we can explore more violations of the requirements by modifying the expression formula of the violated requirement and trying to fix the cause of the problem. Up to now, we only check for the communication between two chains, which has well expressed the properties that the IBC protocol should meet and can also easily be extended to more chains. The numbers of connection ends, channel ends, and packets on each chain are configurable parameters in our model.

We depict the verification results of requirements in Table I. The symbol $\times$ denotes that this requirement is violated and the model checking tool gives a counterexample, while the symbol ✓ means that the correctness of the property is proved by the model checker. As shown, the initialized connection and channel opening handshake may not be completed successfully. In some cases, the sent packet may be neither acknowledged nor time out.

B. Discussions and Recommendations

In this subsection, we explain the reasons and provide detailed fix recommendations for violated requirements.

1) Incomplete handshakes: The initialized connection handshake and channel opening handshake are supposed to be completed. However, the violations of Requirements 2 and 5 indicate that the handshakes may not be completed due to the lack of assignable identifiers or mismatched ends.

Taking the connection as an example, the creation of a connection end requires the allocation of a unique identity, whether in the `OpenInit` or `OpenTry` step. It is easy to imagine that we can only create a limited number of connection ends because identity numbers are limited. Thus, only a limited number of connection handshakes can be successfully completed. However, the situation worsens because connection ends need to be created in both `OpenInit` and `OpenTry` steps. If the identities are already used up in the `OpenInit` step, the chain cannot perform `OpenTry`, and no connection handshake can be completed. For example, there are chains A and B, each of which can allocate two identities. If chains A and B both perform `OpenInit` twice, there are now two connection ends in the `INIT` state on each chain. However, at this point neither chain can perform `OpenTry` to create a `TRYOPEN` state connection end pairing with the other chain. All connection endpoints will be frozen in the `INIT` state forever, and no one can take a step forward.

The above issue is because the creation of the connection end is always assigned a new identifier, and the number of assignable identifiers is limited. However, there may be enough identities to avoid situations where identities are exhausted, and the handshake cannot be completed. Therefore, we need to have the model detection tool explores more states to find more harmful counterexamples.

Since creating a connection end in the `OpenInit` and `OpenTry` steps is necessary to complete the handshake, we have refined Requirement 2 to verify whether the handshake can be completed if the `TRYOPEN` connection end has been established. However, the answer is no. The `OpenTry` step only verifies that the counterparty end is indeed in the `INIT` state and does not check whether a `TRYOPEN` state end has been created and paired with it. Thus, repeating the `OpenTry` step (due to concurrent competition or repeated delivery by relayers) creates multiple ends in the `TRYOPEN` state, but only one of them will match the `INIT` counterparty end in the `OpenAck` step to complete the handshake. The remaining ends will freeze in the `TRYOPEN` state and no subsequent operations can be performed.

To determine the sufficient conditions for the completion of the connection handshake, we further refine Requirement 2 and successfully verify that if the `TRYOPEN` state connection end has been created and paired with an `INIT` state connection end, the connection handshake can be completed.

Recommendations. If both chains initiate a connection handshake and create a connection end in the `INIT` state, one of the chains should execute the `OpenAck` phase to complete the handshake, avoiding creating two new connection ends in the `TRYOPEN` state to complete the two connection handshakes. The problem of repeatedly creating `TRYOPEN` state connections can be solved by introducing a closure mechanism or labeling these problematic ends to alert users.

We have reported these two issues to the developers and provided repair suggestions [42]. We found that the second issue has already occurred in the actual use of the IBC protocol: only one of the two `TRYOPEN` state channel ends has paired with the `INIT` state channel end to complete the handshake. A user used the channel end that failed to complete the handshake to send a packet, resulting in the transaction being unable to complete or falling back [43]. In the end, the user withdrew the transaction through other means, while the developer only reminded the user but did not take measures to solve the problem completely.

2) Incorrect design of special channels: The `ordered_allow_timeout` channel is a less strict ordered channel that provides users with a more flexible transmission mechanism. However, developers do not design such channels correctly, which may trap packets and prevent them from being received or declared timeout.

Firstly, the existing description of timeout conditions for such channels is incorrect. When receiving packets, regardless of whether they time out, the value of `nextSequenceRecv` will increase by 1. Therefore, the correct timeout condition should be as shown in Fig. 9. But the current condition used in function `timeoutPacket` and `timeoutOnClose` is as shown in Fig. 10, preventing packets from correctly timing out.

Secondly, developers overlook function `timeoutPacket` also needs to be operated on in order. For ordered channels, the reception (function `recvPacket`) and acknowledgment (function `acknowledgePacket`) of packets are required to be ordered, which is guaranteed through the variables `nextSequenceRecv` and `nextSequenceAck`. However, the developer did not consider that the `timeoutPacket` function modifies the variable `nextSequenceAck` for `ordered_allow_timeout` channels, meaning it also needs to be ordered. The following steps show a situation that causes a packet to be neither acknowledged nor declared timeout:

1) Module A sent two packets with sequence numbers 1 and 2, and the current value of `nextSequenceAck` is 1.
2) Module B received the first packet successfully.
3) The second packet timed out, and module A called function `timeoutPacket` for it, which caused the value of `nextSequenceAck` equals 2.

Now Module A cannot acknowledge the first packet because function `acknowledgePacket` requires packet sequence equals `nextSequenceAck`.

```
(nextSequenceRecv  <= packet.sequence ) ||
    (verifyPacketReceipt(packet.sequence, TIMEOUT_RECEIPT)
        == True)
```

Fig. 9: Correct Timeout Condition

```
(nextSequenceRecv  == packet.sequence ) &&
    (verifyPacketReceipt(packet.sequence, TIMEOUT_RECEIPT)
        == True)
```

Fig. 10: Wrong Timeout Condition

Recommendations. The solution is that for ordered channels, packets should be received, acknowledged, and declared timeout in the order they are sent.

We have reported these two issues to the developers, the former has been confirmed [44], and the latter has not yet responded [45]. We have also proposed the correct modifications and are waiting for more inspections.

3) Inappropriate handling of abnormal channel states: The standards allow users to optimistically send packets on initialized channels and expect the channel to open eventually. However, the channel may not open or even abnormally close due to the issues mentioned earlier or incorrect call to function `chanCloseInit`, resulting in packets being trapped.

Suppose a module successively calls function `chanOpenInit`, `sendPacket`, and `chanCloseInit`. In that case, there will be an in-flight packet and a closed channel. The module cannot call `timeoutOnClose` since there is no counterparty assigned to this channel end yet.

Another scenario where abnormal channel close leads to errors is as follows:

1) The channel ends were open in both chains.
2) Module A sent a packet to Module B.
3) Module B received the packet successfully.
4) Module B closed its channel end.

Now Module A has two datagrams on the way and two functions `acknowledgePacket` and `chanCloseConfirm` to call. If `chanCloseConfirm` is called first due to the concurrency between relayers or incorrect behavior of the relayer (the relayer is not trusted), `acknowledgePacket` cannot be called because the channel is closed.

The above issues are due to the developer not fully considering the possible situations during the packet transmission. For example, due to the existence of optimistic sending, the channel end may be in the following states after the packet is sent: created without counterparty, handshaking, and completed. The developer overlooked the first scenario in packet processing.

Recommendations. To address the first issue, the function `timeoutOnClose` can be modified to allow declare packets timeout due to the closure of either the self or counterparty channel ends. Another issue can be solved by checking whether there are any packets that have not been acknowledged before or during the execution of the functions `chanCloseInit` and `chanCloseConfirm`.

We also reported both issues to the developers. For the former question, the developer agrees that this problem exists at the protocol design level, while for the latter, there is no reply yet [45].

4) Others: In addition to serious issues violating the requirements, we also discovered some other issues in the standards, and all the issues had been confirmed by the developers [46]–[48]. Our findings include inconsistencies between documents [46], [47] and code redundancy [48]. We also verified these issues through model checking.

Recommendations. Formal analysis can help developers examine documentation and codes more rigorously and completely.

VII. Conclusion

We have formally analyzed one of the most popular cross-chain communication protocols, the IBC protocol, which supports a prosperous multi-chain ecosystem. Our work includes a detailed analysis of the standards to identify critical requirements, a formal modeling of the protocol with security assumptions, the evaluation using model checking, and a detailed discussion and recommendations of our findings.

While analyzing the standards, we discovered some critical issues that may cause loss to users, especially regarding packet acknowledgement and timeout. Our analysis based on TLA$^+$ also demonstrates the effectiveness of formal methods for complex scenarios such as cross-chain protocols. In addition, our assumptions and choices in the analysis and modeling process can also provide some reference for future research.

In future work, we will extend our research to the application level of the IBC protocol and promote the development of the Cosmos community. We will also study other cross-chain protocols and use our work as a basis for improving cross-chain protocol design.

References

[1] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, and H. Wang, "Blockchain challenges and opportunities: A survey," *International journal of web and grid services*, vol. 14, no. 4, pp. 352–375, 2018.

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized business review*, p. 21260, 2008.

[3] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[4] T. Xie, J. Zhang, Z. Cheng, F. Zhang, Y. Zhang, Y. Jia, D. Boneh, and D. Song, "zkbridge: Trustless cross-chain bridges made practical," *arXiv preprint arXiv:2210.00264*, 2022.

[5] M. Swan, *Blockchain: Blueprint for a new economy*. " O'Reilly Media, Inc.", 2015.

[6] C. T. Nguyen, D. T. Hoang, D. N. Nguyen, D. Niyato, H. T. Nguyen, and E. Dutkiewicz, "Proof-of-stake consensus mechanisms for future blockchain networks: fundamentals, applications and opportunities," *IEEE Access*, vol. 7, pp. 85 727–85 745, 2019.

[7] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 3–16.

[8] W. Ou, S. Huang, J. Zheng, Q. Zhang, G. Zeng, and W. Han, "An overview on cross-chain: Mechanism, platforms, challenges and advances," *Computer Networks*, p. 109378, 2022.

[9] S. Thomas and E. Schwartz, "A protocol for interledger payments," *URL https://interledger. org/interledger. pdf*, 2015.

[10] P. Robinson, "Survey of crosschain communications protocols," *Computer Networks*, vol. 200, p. 108488, 2021.

[11] R. Belchior, A. Vasconcelos, S. Guerreiro, and M. Correia, "A survey on blockchain interoperability: Past, present, and future trends," *ACM Computing Surveys (CSUR)*, vol. 54, no. 8, pp. 1–41, 2021.

[12] C. Goes, "The interblockchain communication protocol: An overview," *arXiv preprint arXiv:2006.15918*, 2020.

[13] C. Network. Cosmos Market Capitalization. (2023, May 25). [Online]. Available: https://cosmos.network/ecosystem/tokens

[14] S.-S. Lee, A. Murashkin, M. Derka, and J. Gorzny, "Sok: Not quite water under the bridge: Review of cross-chain bridge hacks," *arXiv preprint arXiv:2210.16209*, 2022.

[15] P. Network. Honour, Exploit, and Code: How we lost 610M dollar and got it back. (2021, Sep 2). [Online]. Available: https://medium.com/poly-network/honour-exploit-and-code-how-we-lost-610m-dollar-and-got-it-back-c4a7d0606267

[16] Wormhole. Wormhole Incident Report. (2022, Feb 5). [Online]. Available: https://wormholecrypto.medium.com/wormhole-incident-report-02-02-22-ad9b8f21eec6

[17] L. Lamport, "Specifying systems: the tla+ language and tools for hardware and software engineers," 2002.

[18] L. Cao and B. Song, "Blockchain cross-chain protocol and platform research and development," in *2021 International Conference on Electronics, Circuits and Information Engineering (ECIE)*. IEEE, 2021, pp. 264–269.

[19] Z. Nehaï, F. Bobot, S. Tucci-Piergiovanni, C. Delporte-Gallet, and H. Fauconnier, "A tla+ formal proof of a cross-chain swap," in *23rd International Conference on Distributed Computing and Networking*, 2022, pp. 148–159.

[20] B. Pillai, K. Biswas, Z. Hóu, and V. Muthukkumarasamy, "Burn-to-claim: An asset transfer protocol for blockchain interoperability," *Computer Networks*, vol. 200, p. 108495, 2021.

[21] B. Pillai, Z. Hóu, K. Biswas, and V. Muthukkumarasamy, "Formal verification of the burn-to-claim protocol for blockchain interoperability."

[22] M. Herlihy, B. Liskov, and L. Shrira, "Cross-chain deals and adversarial commerce," *arXiv preprint arXiv:1905.09743*, 2019.

[23] J. Zhang, J. Gao, Y. Li, Z. Chen, Z. Guan, and Z. Chen, "Xscope: Hunting for cross-chain bridge attacks," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–4.

[24] G. Wood, "Polkadot: Vision for a heterogeneous multi-chain framework," *White paper*, vol. 21, no. 2327, p. 4662, 2016.

[25] J. Kwon and E. Buchman, "Cosmos whitepaper," *A Netw. Distrib. Ledgers*, p. 27, 2019.

[26] Polkadot. Introduction to Cross-Consensus Message Format (XCM). (2023, Mar 13). [Online]. Available: https://wiki.polkadot.network/docs/learn-xcm

[27] D. Basin, J. Dreier, L. Hirschi, S. Radomirovic, R. Sasse, and V. Stettler, "A formal analysis of 5g authentication," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 1383–1396.

[28] N. Kobeissi, K. Bhargavan, and B. Blanchet, "Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach," in *2017 IEEE European symposium on security and privacy (EuroS&P)*. IEEE, 2017, pp. 435–450.

[29] J. Zhang, L. Yang, X. Gao, G. Tang, J. Zhang, and Q. Wang, "Formal analysis of quic handshake protocol using symbolic model checking," *IEEE Access*, vol. 9, pp. 14 836–14 848, 2021.

[30] S. Prabhu, K. Y. Chou, A. Kheradmand, B. Godfrey, and M. Caesar, "Plankton: Scalable network configuration verification through model checking," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 953–967.

[31] J.-Q. Yin, H.-B. Zhu, and Y. Fei, "Specification and verification of the zab protocol with tla+," *Journal of Computer Science and Technology*, vol. 35, pp. 1312–1323, 2020.

[32] S. Akhtar and E. Zahoor, "Formal specification and verification of mqtt protocol in pluscal-2," *Wireless Personal Communications*, vol. 119, pp. 1589–1606, 2021.

[33] V. Kukharenko, K. Ziborov, R. Sadykov, and R. Rezin, "Verification of hotstuff bft consensus protocol with tla+/tlc in an industrial setting," in *Informatics and Cybernetics in Intelligent Systems: Proceedings of 10th Computer Science On-line Conference 2021, Vol. 3*. Springer, 2021, pp. 77–95.

[34] S. Braithwaite, E. Buchman, I. Konnov, Z. Milosevic, I. Stoilkovska, J. Widder, and A. Zamfir, "Formal specification and model checking of the tendermint blockchain synchronization protocol (short paper)," in *2nd Workshop on Formal Methods for Blockchains (FMBC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[35] M. Grundmann and H. Hartenstein, "Verifying payment channels with tla+," in *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 2022, pp. 1–3.

[36] A. et al. Interchain Standards (ICS) for the Cosmos network & interchain ecosystem. (2023, May 27). [Online]. Available: https://github.com/cosmos/ibc

[37] C. R. et al. Interblockchain Communication Protocol (IBC) implementation in Golang. (2023, May 27). [Online]. Available: https://github.com/cosmos/ibc-go

[38] J. K. et al. IBC in Solidity. (2023, May 27). [Online]. Available: https://github.com/hyperledger-labs/yui-ibc-solidity

[39] H. S. de Ocáriz Borde, "An overview of trees in blockchain technology: Merkle trees and merkle patricia tries," 2022.

[40] michwqy et al. public code. (2023, Jun 1). [Online]. Available: https://github.com/michwqy/ibc-tla

[41] tlaplus. vscode-tlaplus. (2021, Jul 12). [Online]. Available: https://github.com/tlaplus/vscode-tlaplus

[42] michwqy et al. ICS 03/04: Some questions about handshake . (2023, Jul 8). [Online]. Available: https://github.com/cosmos/ibc/issues/1001

[43] A. et al. ICS4: Unrecoverable Optimistic SendPacket. (2022, Jan 25). [Online]. Available: https://github.com/cosmos/ibc/issues/645

[44] michwqy et al. ICS04: Something confusing about function timeoutPacket and timeoutOnClose. (2023, Apr 19). [Online]. Available: https://github.com/cosmos/ibc/issues/965

[45] ——. ICS04: some questions about function timeoutOnClose and timeoutPacket. (2023, Apr 26). [Online]. Available: https://github.com/cosmos/ibc/issues/968

[46] ——. ICS03/ICS26: Some inconsistencies. (2023, Apr 13). [Online]. Available: https://github.com/cosmos/ibc/issues/961

[47] ——. ICS18: some possible mistakes . (2023, Mar 1). [Online]. Available: https://github.com/cosmos/ibc/issues/934

[48] ——. Something confusing about ICS03, ICS18. (2023, Apr 13). [Online]. Available: https://github.com/cosmos/ibc/issues/960