Smart Contract Security (Defi)

Manmeet Singh Brar
20UCS112

Table Of Contents

1.Introduction

Blockchain technology has revolutionized the way we think about digital trust and secure transactions. By providing a decentralized and tamper-proof ledger that is maintained by a network of participants, blockchain has made it possible to conduct secure and transparent transactions without the need for intermediaries [1]. There are several key aspects of blockchain technology that makes it unique. Some of which include Decentralisation, Immutability and Transparency. By decentralized we mean that the blockchain is not controlled by a single authority or intermediary. Instead anyone can become a node of the blockchain who has a copy of the ledger. Once added on the blockchain cannot be removed or altered and is visible for everybody to verify, these properties of immutability and transparency makes blockchain an ideal technology for applications that require transparency and accountability[1]. Blockchain has several use cases beyond digital currencies,including decentralized applications, supply chain management, voting systems,identity verification systems and Decentralized Finance(DeFi). All these use cases are enabled by the use of smart contracts which are a crucial part of blockchain.

Smart contracts are self-executing computer programs that are designed to automatically enforce the terms of a contract[1].These programs are written in programming languages such as Solidity and deployed on blockchain where they are run. Once deployed to the blockchain, they are executed by the network of nodes that make up the blockchain. Smart contracts have played a significant role in the growth and development of blockchain technology. Prior to the introduction of smart contracts,blockchains were primarily used for storing and transferring cryptocurrencies. While this was a revolutionary development in its own right,it was limited in its scope and potential applications. Smart contracts changed this by enabling the creation of decentralized applications (dApps) that can perform complex functions beyond just transferring cryptocurrencies. Smart contracts allow for the creation of self executing computer programs that can automatically execute the terms of a contract when certain conditions are met. This enables the creation of dApps that can automate a wide range of processes,from financial transactions to supply chain management to digital identity management. The ability to create dApps with smart contracts has opened up new opportunities for the use of blockchain technology, making it more versatile and appealing to a wider range of industries and applications. This has led to the development of new ecosystems and networks built on top of blockchain technology, such as DeFi,NFTs and more. While smart contracts offer many benefits, such as automation and transparency, they also pose unique security challenges. Smart contracts security is critical in ensuring that the contracts are not vulnerable to

exploitation or attack. Vulnerabilities in smart contracts can result in significant financial losses for users of the contract and damage the reputation of the blockchain network as a whole. There are various types of smart contract vulnerabilities,including coding errors,logic errors and external attacks which we will be discussing in Section 2 of this report.

One of the most promising applications of smart contracts is Decentralized Finance(DeFi) [11]. DeFi is an umbrella term for a new class of financial applications that are built on top of blockchain networks and leverage smart contracts to enable decentralized financial services[9]. DeFi supplies transparent, uncensorable, and decentralized financial services and products. DeFi brings the traditional finance or CeFi(Centralized finance) to the blockchain providing transparency, immutability etc [9]. DeFi supports most of the products and services available in CeFi: asset exchanges, loans, leveraged trading, decentralized governance voting and stablecoins. The use of the key aspects of blockchain in DeFi are what differentiates it from CeFi[9].It adds Transparency, Control and Accessibility to DeFi which are not present in CeFi. DeFi is a fast-growing movement which is composed of a variety of applications and protocols that enable decentralized financial services.In the context of DeFi, smart contract security is particularly crucial since DeFi applications rely heavily on smart contracts to execute financial transactions and automate various financial services.Any vulnerability or exploit in a smart contract could to significant financial losses for users, undermining the trust and credibility of the DeFi ecosystem. Therefore,it is essential to conduct comprehensive security audits of smart contracts and ensure that they adhere to best practices in smart contracts development. We will be doing this in Section 3. These audits should include testing for logical security vulnerabilities and ensuring that the smart contract's code is robust,efficient and secure.By doing so, we can improve the security and resilience of the DeFi ecosystem and ensure that it continues to grow and innovate in a safe and secure manner.

In this report, we will provide an overview of the current state of research on smart contract security focusing on DeFi. We will start by reviewing the literature on manual and automatic security testing of smart contracts, and discuss the design-related and Solidity-related security issues that can arise in DeFi protocols. We will also examine the design of existing DeFi protocols, including cross-chain protocols, decentralized exchanges (DEXs), and decentralized oracles, and discuss the security issues that are specific to these protocols.Finally, we will present a security audit report and provide our conclusions on the current state of smart contract security in DeFi, as well as suggestions for future work in this area.

2.Literature Review

We will examine the different approaches to smart contract security testing, including manual and automatic testing. Manual testing and Automatic testing are both important for ensuring security of smart contracts. We will be looking into the two broad types of vulnerabilities design related issues and solidity related bugs with respect to Manual and Automatic testing. Studying the design of existing DeFi protocols is important as it allows us to to understand how these protocols are constructed and how they work in practice,which can provide insight into potential vulnerabilities and areas for improvement.It can also help us identify common design patterns and best practices that can be applied to future DeFi protocols to improve their security. Three of the fundamental building blocks of DeFi are cross-chain functionality, decentralized exchanges(DEX's), and decentralized oracles[11]. Cross Chain functionality is the ability of interacting with other blockchains[3]. This is essential for building a comprehensive DeFi ecosystem that can support a wide range of assets and services[12]. Decentralized exchanges(DEXs) are a critical component of the DeFi ecosystem. DEXs allow users to trade cryptocurrencies and other digital assets in a decentralized and non-custodial way[10]. Unlike traditional centralized exchanges,which hold user's assets and act as intermediaries in trades[8]. Decentralized oracles are another essential component of the DeFi. They are responsible for providing off-chain data to the smart contracts, enabling them to perform various functions, such as triggering payments and executing trades.Understanding their design can help us gain a better understanding of the overall DeFi landscape and identify potential areas of concern. Additionally, since these protocols are often interconnected and interdependent, understanding their design can help us identify potential systemic risks and vulnerabilities that may affect the entire DeFi ecosystem.We will be studying the existing protocols and their security and trust assumptions in Subsection 3.DeFi ecosystem consists of multiple smart contracts which work in a complementary manner to provide users with a seamless and decentralized financial experience.This makes smart contracts a crucial part and the programming language it is written in is also very crucial as language related bugs can cause great financial losses.Therefore, it is important to study language related bugs to ensure robustness. We will be studying Solidity related security issues in Subsection 4. Solidity is a high-level programming language used to write smart contracts on the Ethereum blockchain.Solidity has become the most widely used for writing smart contracts on Ethereum blockchain.Solidity has many underlying security issues that developers need to be aware of while writing a secure smart contract. The combination of manual and automatic testing is the best approach to ensure both the design and language related vulnerabilities.

2.1 Manual Testing

Manual testing is an important part of the smart contract development process. It involves the assistance of a human tester who manually executes testing steps to identify potential issues or bugs in the code. One example of manual testing for smart contracts is code audits which we will be doing in Section 4, where developers and/or auditors go over every line of contract code to ensure that it is written in a way that is secure and functional.The main objective of manual testing is to ensure the correctness of the code,identify vulnerabilities,and improve the overall quality of code.Design related issues in smart contracts are related to the overall design of the contract and how it interacts with the blockchain network[4,5].These issues may include logical error, security vulnerabilities, and inefficient use of resources.On the other hand, Solidity related issues are related to the specific programming language used to develop the smart contracts.Manual testing also includes the readability, maintainability and modularity of the code.A well designed and well structured smart contract is more likely to secure and less prone to errors.Manual testing is also better for understanding the design protocol of the smart contract which helps in identifying potential vulnerabilities better and improve the overall quality of the code.Conducting manual testing of smart contracts demands significant expertise and a significant commitment of resources,including time and money. Furthermore, manual testing may be vulnerable to issues arising from human error.During manual testing, a tester can carefully examine the smart contract's code,which can reveal any inconsistencies, design flaws or solidity related bugs/flaws.Additionally,manual testing can allow the tester to identify the parts of the code that may be difficult to maintain or modify in the future.

2.2 Automatic Testing

Automatic testing is a type of software testing that involves the use of tools to run tests automatically without human intervention. There are several types of automatic testing,including unit testing,integration testing,and end-to-end testing.Unit testing involves testing individual functions and methods within the smart contract to ensure that they function as expected. Integration testing involves testing how different components of the smart contract work together to ensure that they are integrated properly. End-to-end testing involves testing the entire smart contract as a whole to ensure that it works correctly from start to finish. Automatic testing tools are used to identify design and Solidity related issues in smart contracts.These tools can automate the process of testing and analysis, making it faster and less prone to human error. There are several tools available in the market that can automatically detect design flaws and coding issues in smart contracts. These tools use various methods such as static analysis, dynamic analysis,and symbolic execution to identify vulnerabilities in the

code.One such tool is the Slither static analysis tool, which can analyze the smart contract code and provide feedback on issues such as code complexity, potential bugs, and security vulnerabilities. Other tools include Mythril, Oyente, and Manticore, which can also be used for automated testing and analysis of smart contracts. These tools can help developers and auditors to identify potential issues and mitigate them before deploying the smart contract.Automated testing can be done quickly and with minimal effort.It can be scaled up easily to accommodate large and complex smart contracts making it ideal for testing in the DeFi space where contracts are very complex.We will be studying and using some of these automatic tools in the future.

2.3 Design of Existing DeFi Protocols

In this section, we will study the design of three most popular/widely used DeFi protocols. The design of these protocols involves various components such as architecture,data flow,user interface,and security features[11].Studying the design of existing DeFi protocols provides insight into how these protocols work, their strengths and weaknesses,and how they can be improved.It also enables developers to build upon existing protocols to create new and innovative DeFi solutions.By analyzing the design of DeFi protocols,we can better understand the challenges associated with designing financial systems and develop solutions that address these challenges.The three fundamental components of DeFi as stated Cross Chain protocols, Decentralized Exchanges(DEX) and Decentralized oracles. They enable interoperability,liquidity and excess to external data sources[1,10,11]. Understanding how these protocols work, their benefits, and their vulnerabilities is crucial for development of secure and sustainable DeFi platforms.

2.3.1 Cross Chain

Cross Chain technology is a crucial part of the DeFi ecosystem.It is used to enable exchange of information between different blockchain networks, allowing users to enjoy the benefits of multiple blockchains in a single platform.It involves a source blockchain and a target blockchain.The source blockchain is where the transaction is initiated, and the target blockchain is where the transaction is executed.In simpler terms,cross chain technology allows for the interoperability between different blockchains,which opens up new possibilities for decentralized finance.It is imperative that cross-chain communication systems maintain Byzantine Fault Tolerance(BFT)[2]. This is because blockchain systems are designed to withstand node failures, network failures, and malicious actors through the use of BFT consensus mechanisms. Instead of relying on a single entity, blockchain nodes come to a consensus on the validity of proposed transactions.A BFT system has to maintain atomicity, liveness, and safety. Atomicity ensures that transactions are either completely executed or completely rejected.

Liveness ensures that the system keeps making progress despite failures. Safety ensures that transactions are executed correctly and the system maintains consistency.Focusing on these three properties cross chain protocols and cross chain consensus are created[1].Cross chain Consensus is the technique by which nodes or entities on a destination blockchain know that nodes or entities on a source blockchain have come to agreement on some fact. It allows information from a source blockchain to be trusted on a destination blockchain[2].In recent years, the blockchain ecosystem has evolved into a multi-chain world. Various blockchains, like the popular Bitcoin Network or Ethereum, are evolving simultaneously. Given the diverse nature of blockchains and their unique requirements for cross chain consensus, various techniques are employed in cross chain communication. These techniques are based on the core capabilities of atomic swaps, cross chain messaging and pinning[2].

2.3.1.1 Atomic Swaps

Atomic swaps are a type of cross chain technique that facilitate trading between pirates in a trustless environment without the need for any third party or intermediary.[3] Atomic swaps are designed to eliminate counterparty risk by enabling parties to trade assets on different blockchains without requiring them to trust each other. In an atomic swap, two parties agree to trade assets on different blockchains, and the trade is executed simultaneously, ensuring that neither party can back out of the trade. The atomicity property of the swap ensures that either both parties successfully complete the trade or neither party does, thus eliminating the risk of one party losing their assets without receiving the assets they are trading for. Atomic swaps have been used for trading between different cryptocurrencies, and have the potential to enable trading between different types of assets such as stocks and commodities in a decentralized and trustless manner. A special or innovative use of atomic swaps is when the value on the source blockchain is inaccessible and the same value is created on the destination blockchain.This allows value to be transferred across blockchain without the need of counterpart for trading. It is important that the value on the source blockchain is inaccessible as there have been exploits on this.This issue was identified in the PolyNetwork bridge[5].Here the attacker got access to the vault(where the value was stored) privileged roles and changed the owners address.Atomic swaps use Hash Time-Locked Contracts(HTLCs) to ensure that both parties fulfill the terms of the swap[2].

2.3.1.1.1 HTLCs

Hash Time Locked Contracts (HTLCs) have become a popular and widely used technique for implementing atomic swaps. They were introduced as an alternative to

centralized exchanges, providing a trustless environment for trading between two parties without the need for any intermediaries. HTLCs employ hash locks or signature-based locks and timelocks to ensure the atomicity of operations. This technique has been widely adopted in the cryptocurrency space as it offers a secure and decentralized way of exchanging assets across different blockchains[1].In a hash locks-based protocol, the preimage resistance property of hash functions is utilized. The protocol involves two participants, P and Q, who transfer assets to each other via transactions that require the preimage of a hash h := H(r), where r is chosen by P, the initiator of the protocol, usually at random. This technique ensures that only the party who knows the preimage of the hash can unlock and claim the assets[3]. Protocols based on hash locks have a limitation in terms of interoperability, as they require both cryptocurrencies to support the same hash function within their script language. An alternative to this is a protocol where participants P and Q can transfer assets to each other through transactions that require solving the discrete logarithm problem of a value Y := g^y. In this case, the value of y is chosen uniformly at random by P, who initiates the protocol.At last is to use cryptographic challenges, which require a solution to be revealed at a specific time in the future. With this approach, P and Q can commit to the cross-chain transfer by solving one of the challenges. Timelock puzzles are one example of a construction that can be used. These puzzles use inherently sequential functions, where the result is only revealed after a predetermined number of operations are performed.

For swapping using HTLC agreement occurs off-chain, with on-chain consensus used to ratify the off-chain agreement. Consensus is formed between the two parties on the amounts to be exchanged. Consider the scenario where there are two blockchains, Blockchain A and Blockchain B, each with their own tokens, alpha and beta, respectively. Alice wants to exchange her 10 beta tokens for 1 alpha token which Bob has. To make this possible, Alice creates smart contracts on both blockchains, deploys them and generates a random number R. She uses this R to create a commitment value H, using the equation H = MessageDigest(R). Bob then deposits his 1 alpha token into the smart contract on Blockchain A, which was deployed by Alice, indicating his acceptance of the offer. Alice can now withdraw the 1 alpha token on Blockchain A by submitting the value R. Once Bob sees the transaction, he can use the same value R to unlock the 10 beta tokens on the contract deployed on Blockchain B by Alice. A timeout is also enforced on the smart contracts to ensure that Alice and Bob can retrieve their tokens if R is not submitted. However, there are some downsides to this approach. For instance, if the price of alpha tokens falls respective to beta tokens after Bob deposits his token, Alice may not complete the transfer by not submitting the R value. In such cases, Alice and Bob can retrieve their original tokens after the timeout. There is also the safety issue that Bob may lose his funds if he goes offline for a longer duration than

the timeout period before withdrawing his funds or if his transaction is not added to Blockchain B[3,5]. Despite these limitations, the use of HTLCs for atomic swaps is considered secure, efficient, and fast, with no involvement of third parties. Some solutions have been proposed to enhance HTLCs and provide additional on-chain trust anchors, such as XCLAIM and the Lightning Network (LN). [1]

2.3.1.1.2 XCLAIM

The XCLAIM protocol [1] employs a hybrid of HLTCs, collateralization, and escrow parties to achieve non-interactive cross-chain atomic swaps. Additionally, it leverages the BTC Relay protocol to transmit transaction inclusion proofs from Bitcoin to Ethereum. BTC Relay is a tool that enables users on the Ethereum MainNet to take actions based on Bitcoin transactions using the Simplified Payment Verification (SPV) method outlined in the Bitcoin White Paper. SPV relies on transferring Block Headers between different blockchains[3]. This protocol involves various participants, such as the requester, sender, receiver, redeemer, backing vault, and issuing smart contract. Requesters deposit coins to generate tokens, and the redeemer burns the tokens to receive coins. The sender transfers the tokens, and the receiver receives them. The vault smart contract fulfills asset backing requests and verifies correct redemption. The issuing smart contract creates and trades cryptocurrency-backed asset tokens while enforcing the vault's proper functioning.

Atomic swaps are a promising technology for facilitating trustless, decentralized exchanges between parties on different blockchain networks. However, the success of atomic swaps is dependent on the availability of smart contracts, which may not be supported on all blockchain systems. This can make it difficult to carry out atomic swaps, as smart contracts are integral to the process. Even when smart contracts are available, atomic swaps only address the problem of exchanging assets between two entities, and do not fully solve the need for a fully decentralized exchange[7].Furthermore, the current implementation of atomic cross-chain swapping is still in its early stages of development, and the scale of the atomic swapping scheme remains relatively small. Despite this, experts believe that atomic swaps will likely become a seamless and integrated background processing process, while maintaining the core features of blockchain technology. This means that atomic swaps may be used for a wider range of applications in the future, potentially revolutionizing the way that exchanges are carried out on the blockchain.

2.3.1.2 Cross Chain Messaging

Cross Chain Messaging (CCM) enables the transfer of messages or data from one blockchain to another[2]. This allows users to access and transact with assets across different blockchains, which is crucial for interoperability between blockchain networks.To achieve this, CCM involves a multi-step process[3]. First, a user on blockchain A generates a message or transaction with the payload to be transmitted. This transaction is then added to the blockchain A and subsequently relayed to the destination blockchain B.However, nodes on the destination blockchain B must verify that the message received from the source blockchain A can be trusted. Depending on the protocol used, the relayed message may undergo verification to ensure its integrity and authenticity. Once the message is verified, it is then processed on the destination blockchain.CCM is essential for realizing cross-chain interoperability, which is a critical requirement for the growth and success of the blockchain industry[1]. However, the implementation of CCM requires careful consideration and standardization to ensure its security and effectiveness. As the technology continues to evolve, we can expect CCM to play a more significant role in enabling seamless communication and transfer of assets across various blockchain networks[3].There are different types of protocols and products for cross chain messaging all differing in the aspects of scalability,security and decentralization.

2.3.1.2.1 Ion Project

The Clearmatics Ion project is a blockchain-based system that aims to facilitate the transfer of block headers from one blockchain to another[2]. This system utilizes a mechanism called "block header transferring," which allows a set of Relayers to transfer block headers from one blockchain to another securely and efficiently.In this system, Relayers act as intermediaries who facilitate the transfer of block headers between two different blockchains. They can initiate the transfer process by selecting the block headers they wish to transfer and obtaining the required signatures.To ensure the security and integrity of the transferred block headers, the system requires that a threshold number of Relayers must sign each block header before it is accepted by the receiving blockchain. This ensures that the block headers are valid and have not been tampered with during the transfer process[3]. Moreover, the Relayers only transfer a block header once the blocks they relate to are deemed to be final. This means that the block headers being transferred are from a finalized block, which ensures that the transactional data contained in them is accurate and cannot be modified.In a specific scenario where the source blockchain is Ethereum, a user initiates a transaction on the source blockchain that triggers an event. Once this transaction is confirmed, the receipt generated by the Ethereum network contains several pieces of information such as the transaction hash, block number, transaction number, and a list of event data.To ensure

the authenticity of the transaction, the event data is incorporated into the transaction hash calculation. Subsequently, the user can execute a transaction on the destination blockchain, providing the necessary parameters related to the source blockchain's transaction. These parameters include the block number, Merkle Proof which demonstrates that the transaction belongs to the block, and the event information.Upon receiving this information, the code on the destination blockchain performs a verification process to ensure the validity of the source blockchain's transaction information. If the information is found to be accurate, the code can then utilize the event data to execute a specific action. Overall, this process allows for the seamless transfer of data and execution of actions between different blockchain networks, thereby enhancing interoperability and expanding the functionality of blockchain technology.

The success of the system in transferring events from the source blockchain to the destination blockchain is heavily reliant on the performance of the Relayers involved. As such, it can be said that the system's effectiveness is dependent on these Relayers to a significant extent. However, it should be noted that this system is not capable of facilitating atomic behavior, meaning there is no guarantee that the events forwarded by the Relayers will be successfully relayed to the destination blockchain[2]. There are various factors that may hinder the relay of events, including technical glitches, network congestion, and other unforeseen circumstances. Furthermore, even if an event is successfully forwarded by the Relayers, there is no guarantee that the transactions will be processed correctly, and higher-level protocols will operate correctly based on the event. This is because the correctness of the processing and operation of higher-level protocols is dependent on the accuracy of the relayed event data. If the data is inaccurate or tampered with, it can negatively affect the operations of the higher-level protocols[4,5]. Therefore, it is crucial to understand the limitations of this system and consider the potential risks associated with using it, such as the possibility of incomplete or inaccurate event forwarding, which may impact the overall functioning of the system. It is also essential to ensure that the Relayers involved in the system are reliable and capable of executing their duties efficiently to minimize the risks associated with this system.

2.3.1.2.2 Cosmos

Cosmos is a revolutionary multi-blockchain system that facilitates communication between individual blockchains, referred to as "Zones," through a central blockchain known as a "Hub"[2]. The Zones and the Hub in the Cosmos system utilize a consensus

algorithm called Tendermint, which is a type of Practical Byzantine Fault Tolerance (PBFT) consensus algorithm[3]. One of the unique features of Cosmos is its ability to allow for heterogeneous blockchain communications, meaning that the Zones can have varying levels of permissioning and can use alternative consensus algorithms. This includes algorithms that offer probabilistic finality, which is a property of some consensus algorithms that allows for a high level of confidence that a block will not be reversed after a certain number of confirmations[1]. Moreover, the Cosmos system allows for entirely different blockchain paradigms, meaning that the Zones can have completely different design structures and functions from one another. This provides a level of flexibility and customization that is not typically available in traditional blockchain systems. Overall, the Cosmos multi-blockchain system offers a new level of scalability and interoperability between individual blockchains, enabling them to communicate with each other seamlessly[6]. Its unique design also allows for greater flexibility and customization, making it a promising solution for enterprises seeking to build robust and adaptable blockchain networks. To function effectively, each individual Zone blockchain in the Cosmos system is required to have a set of validators. These validators are responsible for verifying and authenticating transactions on the Zone blockchain, ensuring its security and integrity. In order for the Cosmos system to work seamlessly, there must be a high level of trust and cooperation between the validators of each Zone blockchain and the validators of the central Hub blockchain. This means that the validators of each Zone must trust the validators of the Hub, and vice versa. This trust is critical as it allows for the validation of transactions across different blockchains in the system. Validators in the Hub blockchain must be able to verify the authenticity and correctness of transactions originating from the Zone blockchains, while the validators of each Zone must trust the validators in the Hub blockchain to ensure the secure transfer of transactions to other Zones.

The communication process of a datagram from Zone 1 to Zone 3 using Cosmos' Inter-Blockchain Communications (IBC) system involves several steps. Firstly, a transaction on the Zone 1 blockchain generates the datagram to be sent to the Zone 3 blockchain. Next, the datagram is included in a block on the Zone 1 blockchain. To communicate the datagram to the Zone 3 blockchain, a Block Commit message is created, which contains the block header of the block that contains the datagram. This message is signed by at least a threshold number of validators and sent to the Hub blockchain. In addition, a Packet message is created containing the datagram along with a Merkle Proof, which proves that the datagram is related to the block in which it is included. The Packet message is also sent to the Hub blockchain. Upon receiving the Block Commit and Packet messages, the Hub blockchain includes the datagram in a block. A node on the Hub blockchain then creates a Block Commit message containing the Hub blockchain's block header for the block that contains the datagram. This

message is signed by at least a threshold number of validators on the Hub blockchain and sent to the Zone 3 blockchain. Finally, the Hub blockchain sends the datagram along with a Merkle Proof in a Packet message to the Zone 3 blockchain. The validators on the Zone 3 blockchain can then send a Block Commit message and an acknowledgement message back to the Hub blockchain. Once the Hub blockchain includes the acknowledgement in a block, validators on the Zone 1 blockchain can check the acknowledgement and be sure that the transaction has been successfully included in the Zone 3 blockchain.

The IBC system, similar to the ION project, has its own set of limitations. One such limitation is that the system does not provide guaranteed atomic behavior, meaning that it cannot ensure that a datagram is included in both the Zone 1 and Zone 3 blockchains. Additionally, there is no guarantee that the validators on the Zone 3 blockchain will submit acknowledgements to the Hub blockchain, potentially causing further issues. Although the Hub blockchain should produce a timeout datagram, there is no assurance that this will occur. Furthermore, the system relies heavily on the trust between the Zone blockchain validators and the Hub blockchain validators. In other words, the Hub blockchain serves as a central point of control for the entire network of blockchains, raising concerns over centralization. Cross-blockchain messaging has crucial challenges for the blockchain industry. As the number of blockchains continues to grow, there is a pressing need to develop solutions that allow for seamless communication between them. However, current cross-blockchain techniques that heavily rely on relayers or communicators to transmit transactions are vulnerable to attacks that can cause significant damage. For instance, an attacker can trick a relayer into sending an invalid message from one blockchain to another, leading to unintended consequences. Such an exploit occurred with the pNetwork pBTC-on-BSC project[5], highlighting the need for better security measures in cross-blockchain communication protocols. Additionally, a short-term 51% attack on the source blockchain can trick the communicator into forwarding messages that will disappear after a reorganization of the source blockchain. Despite the growing need for efficient and decentralized cross-blockchain communication protocols, the number of feasible solutions for generic blockchain interoperability remains small. The creation of a cross-blockchain smart contract requires an inter-blockchain communication protocol that can facilitate decentralized and trustworthy arbitrary data exchange among blockchains. However, developing such protocols poses significant technical challenges, and feasible cross-blockchain smart contracts have a long way to go. In summary, while cross-blockchain communication is essential for the growth and development of the blockchain industry, the current methods have several limitations that need to be

addressed. Future solutions must prioritize security and decentralization while also facilitating efficient and trustworthy data exchange among different blockchains.

2.3.1.3 State Pinning Techniques

State Pinning refers to a technique of representing the state of one blockchain within another blockchain[1,3]. A common example of this technique involves including the block hash of a private blockchain in a smart contract on Ethereum MainNet. When the block hash from the private blockchain is included in Ethereum MainNet at a specific block number, it signifies that the state of the private blockchain can be represented by that block hash at that particular moment in time. Essentially, State Pinning enables a blockchain to capture and record a snapshot of another blockchain's state, and then securely anchor it to its own blockchain. This technique can be useful for a variety of applications, such as verifying and auditing the state of a private blockchain by a public blockchain, or for enabling cross-chain transactions between two or more blockchains. State Pinning has various use cases, one of which is to provide proof to governmental regulators and other interested parties that the state of a chain has been altered. In situations where a majority of participants in a private blockchain collude to alter the historical state of the chain, State Pinning can be used to detect such alterations. In such a scenario, the colluding participants could produce new blocks on the private blockchain, making it difficult for minority participants to demonstrate the valid state of the blockchain. However, by including the block hash of the private blockchain in a smart contract on Ethereum MainNet, the minority participants can demonstrate the valid state of the blockchain by referencing the Block Hash at a particular block number in Ethereum MainNet. This provides an immutable record of the state of the private blockchain at that point in time, which cannot be altered by colluding participants. In addition to regulatory compliance, State Pinning can also be used in other use cases such as ensuring the integrity of supply chain management systems or verifying the authenticity of digital assets. By providing a secure and decentralized means of storing and referencing blockchain state across multiple blockchains, State Pinning offers significant benefits for the blockchain ecosystem. However, like any other technology, State Pinning also has its limitations and potential security risks, which must be carefully evaluated and addressed. One of the limitations is that it requires trusted third parties to relay the block hash from one blockchain to another, which can lead to centralization and a single point of failure. Moreover, it can only provide a snapshot of the state of the blockchain at a specific point in time and may not be able to provide a full history of the blockchain's state changes[5]. Another potential security risk is the possibility of a

compromised private blockchain. In addition, there is a risk of a replay attack, where an attacker can use a previously pinned block hash to deceive another blockchain into accepting a transaction that was already processed on the pinned blockchain. This can occur if the transaction has already been executed on the private blockchain before the block hash was pinned on Ethereum MainNet. To address these limitations and potential security risks, careful evaluation and implementation of State Pinning is required. It is important to consider the specific use case, the level of trust required between the blockchains, and to implement appropriate security measures such as regular audits and monitoring of the relayers. A specialized use of pinning is to store the final state of a private blockchain prior to the blockchain being archived. Pinning the final state of a blockchain prior to archiving allows the blockchain to be reinstated later if needed at a state which can be verified.

2.3.1.3.1 Merge Mining

Merge Mining is a method that allows a low hashing power blockchain to benefit from the security of a more secure and high hashing power blockchain. In this technique, the block hash of the low hashing power blockchain, such as NameCoin, is included in a more secure and high hashing power blockchain, such as Bitcoin[3]. The process involves Bitcoin miners verifying transactions on both the Bitcoin and NameCoin blockchains. Once the verification is complete, the miner includes the NameCoin block hash in a transaction on the Bitcoin network. This mined transaction can then be included in both the Bitcoin and NameCoin blockchains. The primary advantage of Merge Mining is that it allows smaller and less secure blockchains to benefit from the security of a larger and more secure blockchain, without requiring additional hashing power or resources. Merge Mining can also incentivize miners to verify transactions on smaller blockchains by allowing them to earn rewards on both the larger and smaller blockchains. However, there are also potential risks and limitations associated with Merge Mining. For example, if the same mining pool controls both the high and low hashing power blockchains, it can potentially control the entire mining process and undermine the security of both blockchains. Additionally, Merge Mining can also result in increased blockchain bloat, as both the Bitcoin and NameCoin blockchains include the same transaction data[6]. The technique requires both blockchains to use the same consensus algorithm, which means that they must have the same mining mechanism, such as proof-of-work. It also assumes that all transactions can be viewed by both blockchains, which is not always possible in private blockchain scenarios where transactions must be kept confidential. As a result, merged mining is not suitable for private blockchains where confidentiality is a critical requirement.

2.3.2 Decentralized Exchanges

DeFi applications are smart contracts that offer a wide range of financial services on blockchains .One of the key applications of DeFi is decentralized exchanges (DEXs), which allow users to trade their cryptocurrencies in a decentralized and non-custodial manner[12]. This means that traders can buy and sell assets on DEXs without relying on a central authority[10]. In an automated market maker (AMM) DEX, the exchange rate of each trade is determined by predefined algorithms and market liquidity reserves. This enables DEXes to provide more efficient price discovery and lower trading fees than traditional centralized exchanges. Most of the prominent DEXes in the blockchain ecosystem are AMM DEXes, such as Uniswap and SushiSwap[10]. Unlike traditional centralized exchanges, DEXs operate on a blockchain, which enables users to trade digital assets without intermediaries or a central authority controlling their funds. In a DEX, users maintain control over their private keys and can trade cryptocurrencies with each other directly. Instead of relying on a centralized intermediary to match buy and sell orders, DEXs use a network of nodes to execute trades using smart contracts. These smart contracts are self-executing contracts with the terms of the agreement between buyer and seller being directly written into lines of code. DEXs typically rely on a liquidity pool model, where users can contribute their cryptocurrency holdings to a pool, which is used to facilitate trades between buyers and sellers. In return for contributing to the liquidity pool, users receive a portion of the fees generated from the trades executed on the platform.

The decentralized nature of DEXs provides several benefits over centralized exchanges. For one, users maintain control over their private keys, which means they are less vulnerable to hacks or security breaches. Additionally, DEXs are less prone to manipulation or price manipulation by large institutions or market makers, which can affect prices on centralized exchanges. However, DEXs also have some limitations. They are typically less user-friendly than centralized exchanges and may have lower trading volumes and liquidity, which can lead to wider bid-ask spreads and higher trading fees. Moreover, the lack of regulation in the DEX space can make it more challenging for users to navigate the market and assess the risks associated with trading on these platforms.

2.3.2.1 Uniswap

Automated market makers (AMMs) are software agents designed to pool liquidity and facilitate trading based on predefined algorithms[12]. One specific type of AMM, known as constant function market makers (CFMMs), has become increasingly popular in the world of decentralized finance (DeFi). Uniswap is an example of a CFMM-based AMM that is implemented as a smart contract on a permissionless blockchain. These smart contracts allow anyone to trade tokens without relying on traditional intermediaries like exchanges or brokers. The algorithm used by CFMMs involves maintaining a constant ratio between two assets in a given pool. This constant ratio enables the automated market maker to automatically adjust the price of the assets as the supply and demand change, ensuring that the pool remains balanced. The use of CFMMs in DeFi has enabled decentralized exchanges to operate with low fees, high liquidity, and reduced counterparty risk[12].

2.3.3 Decentralized Oracle

Decentralized oracles play a critical role in many blockchain-based applications that need access to real-world data. They act as intermediaries between blockchain networks and external data sources, allowing smart contracts to interact with off-chain data in a decentralized manner. These oracles leverage a distributed network of nodes or validators to gather, verify, and deliver data to smart contracts on the blockchain. By using multiple independent parties to operate the nodes, the reliance on a single centralized entity is reduced, increasing the security and reliability of the oracle system. One of the key advantages of decentralized oracles is their ability to provide secure and reliable data to smart contracts. Through consensus mechanisms and validation processes, they ensure that the data they provide is accurate, tamper-proof, and verifiable. This allows smart contracts to make informed decisions and take actions based on trusted external data. Additionally, decentralized oracles offer transparency and openness, as the data collection and validation processes are typically transparent and auditable. This allows users to verify the integrity of the data and the performance of the oracle system, enhancing trust and confidence in the overall operation of the oracle. Decentralized oracles also enable greater interoperability between blockchains and the external world. They allow smart contracts to access data from various sources, including APIs, IoT devices, external databases, and other blockchain networks, enabling a wide range of use cases such as DeFi, supply chain management, gaming, and prediction markets. Despite their benefits, decentralized oracles also face several challenges such as potential sources of data manipulation, network latency, and scalability. Nonetheless, they are an essential building block for DeFi applications that

require reliable and decentralized access to external data, enabling greater innovation and expanding the capabilities of blockchain technology beyond its native data.

2.4 Solidity Related Bugs

Solidity is a high-level programming language used to write smart contracts for the Ethereum blockchain. It is an important tool for the development of decentralized applications, and has been used to build a wide variety of projects, ranging from simple token contracts to complex decentralized autonomous organizations (DAOs). However, despite its popularity and usefulness, Solidity is not immune to bugs. In fact, due to the unique nature of blockchain development and the high stakes involved in smart contract execution, Solidity-related bugs can have significant consequences for decentralized applications. There are a number f Solidity related bugs some of which are[13]:

- Unencrypted Private Data On-Chain - sensitive information that is stored in plain text on a public blockchain.

- Message call with hardcoded gas amount - a situation where the contract developer sets a fixed amount of gas for a specific transaction in the contract.

- DoS With Block Gas Limit - A Denial-of-Service (DoS) attack can be launched on the Ethereum network by filling blocks with transactions that require a large amount of gas to execute.

- Requirement Violation - bug that occurs when a smart contract on a blockchain does not meet its intended requirements or specifications.

- Missing Protection against Signature Replay Attacks - Signature replay attacks are a type of security threat that can affect smart contracts on blockchains. In this type of attack, an attacker intercepts a valid signed message from a user and then replays that message to the smart contract to execute a transaction or carry out some other action. This can result in unintended or malicious behavior, such as transferring funds to the attacker's address.

- Incorrect constructor name - In Solidity, the constructor is a special function that is executed during contract deployment and is used to initialize the contract state variables. One common mistake that developers can make is using an incorrect constructor name, which can lead to unexpected results and bugs in the contract.

- Authorization through tx.origin - arises when a smart contract uses the tx.origin global variable to check the originator of a transaction.

- Delegatecall to untrusted callee - a contract delegates a call to another contract without properly verifying the trustworthiness of the callee contract. This can allow the untrusted callee contract to execute arbitrary code in the context of the calling contract, which can lead to unintended consequences and malicious behavior.

- Reentrancy - It is a type of vulnerability in smart contracts that can allow an attacker to repeatedly enter a contract function and potentially steal or manipulate funds.

- Integer overflow and underflow

- Function default visibility - In Solidity, functions have a visibility modifier that determines who can call them. The visibility modifier can be set to public, private, internal, or external. If no modifier is specified, the function's visibility defaults to public.

Solidity-related bugs can have severe consequences in decentralized finance (DeFi) products. These bugs can lead to loss of user funds, smart contract hacks, and other security issues. Therefore, it is crucial to thoroughly check and test for these bugs in any DeFi product to ensure its security and reliability. Developers should follow best practices for writing secure smart contracts, including using the latest version of Solidity, avoiding hard coded values, properly handling input validation and error handling, and performing thorough testing and auditing. By implementing these measures, developers can reduce the risk of Solidity-related bugs and ensure that their DeFi products are secure and trustworthy.

3. Security Audit Report

A smart contract security audit report is a comprehensive assessment of the security and reliability of a smart contract. Smart contract audits are performed by specialized security firms or individuals who are trained to identify vulnerabilities and weaknesses in the code of the contract. The report contains a detailed analysis of the smart contract's source code and outlines any potential security risks that may exist. The report also includes recommendations and suggestions for improving the security of the smart contract. Smart contract security audits are essential for any decentralized application or DeFi product that relies on smart contracts to execute transactions and manage

funds. A security audit helps to ensure that the smart contract is safe, secure, and reliable and that it functions as intended. A smart contract security audit report can help developers identify and address potential security risks before deploying the smart contract on a live blockchain network. The audit process involves a thorough review of the smart contract's code and its functionality. The auditor examines the code for potential vulnerabilities and exploits, such as reentrancy attacks, overflow errors, and other common security risks. The auditor also examines the smart contract's interaction with other contracts and external systems to ensure that it functions as intended and that it does not introduce any security risks. After completing the audit, the security firm or auditor provides a detailed report of their findings, including any vulnerabilities or security risks identified during the audit. The report may also include recommendations and suggestions for improving the security of the smart contract. In order to conduct a security audit of any blockchain project, it is important to first understand the protocol and how it works. This includes understanding the various components of the protocol, how they interact with each other, and how they handle user input and external data. Without a clear understanding of the protocol, it can be difficult to identify potential security vulnerabilities and to develop effective testing strategies.
We will be looking at Wormhole, a Cross Chain product’s protocol before its security audit report. Wormhole is a cross-chain protocol that facilitates message passing between multiple blockchain networks, such as Ethereum, Solana, Binance Smart Chain, Polygon, and more. It has become widely popular due to its efficiency and reliability as a token bridge in the cryptocurrency space. Despite its popularity, Wormhole's underlying design is straightforward and uncomplicated, focusing on a simple generic message protocol that enables the exchange of information between blockchains. The beauty of Wormhole lies in its ease of use. It provides a smooth and hassle-free way for different blockchains to communicate with each other, allowing for the transfer of digital assets between them. Users can conveniently move their tokens and non-fungible tokens (NFTs) from one blockchain to another without the need for complex and time-consuming procedures. Moreover, Wormhole is highly regarded for its security and reliability. The protocol is designed to ensure that all transactions are secure and tamper-proof. It also features a robust set of tools and functionalities that allow users to monitor the movement of their assets and ensure the smooth operation of the bridge. As of January 2023, Wormhole connects 20 different blockchain networks, including Ethereum, Solana, Binance Smart Chain, Polygon, Aptos, and more[9].

The process of simple cross chain interactions on Wormhole can be broken down into three fundamental steps. Firstly, an action is performed on Chain A. Secondly, the resulting Verifiable Action Approval (VAA) is obtained from the Guardian Network. Finally, the VAA is utilized to perform an action on Chain B. To make this process possible, Wormhole operates a network of guardians, which as of September 2022, can

be set up to a maximum of 21 nodes[9]. The guardians function as the protocol's oracle component, and their observations and signatures are critical to ensuring the security and reliability of the Wormhole ecosystem. Multiple smart contracts are deployed on each participating chain, including a core bridge contract, a token bridge contract, and an NFT bridge contract. These contracts enable the communication and transfer of digital assets between different chains. Additionally, relayers play a crucial role in Wormhole's cross-chain interactions by facilitating the communication between different chains. VAAs are the core of Wormhole's technical infrastructure. They represent a collection of independent observations and signatures from multiple guardians, which provide a proof that a state has been observed and agreed upon by the majority of the network. VAAs are vital for ensuring the security and reliability of Wormhole's cross-chain interactions, and they are the messages that have been verified and signed by the guardians. After a VAA is created by obtaining signatures from 2/3 guardians, it is stored within the guardian network indefinitely. Each VAA is uniquely identified by its emitterChain, emittedAddress, and sequence, and can be obtained by querying a guardian node through an RPC API using this information. Although anyone can retrieve VAAs and initiate downstream transactions, it may not be feasible for certain users to do so. For instance, users may not have an account on the target chain or may not have sufficient funds to pay for transaction fees. To solve this problem, Wormhole has introduced bridge relayers that can deliver messages and earn fees. This allows users to transfer their assets across chains without having to worry about the technicalities involved.

Relayers play a key role in the final step of the process -- they can be thought of as the 'write' portion of interoperability, complementing the 'read' portion that Guardians provide.In the context of Wormhole relayer can be defined as any process which delivers a VAA. In most cross chain messaging designs there is a dedicated relaying mechanism which operates inside the protocol’s trust boundaries. This means that the relayer either has an adversarial relationship to the oracle, or the relayer has trust assumptions and contributes to the protocol’s security model. Relayers are usually a trusted party, are often also privileged, and developers are typically forced to use the relayer model built into the protocol. This is where Wormhole differs from other interoperability protocols.In Wormhole, relayers are neither trusted nor privileged. This means relayers cannot jeopardize security, only liveness. Because Wormhole is designed to have a firm trust boundary at the level of the VAA, relayers have exactly the same capabilities as any regular, untrusted blockchain user. In simple terms, Relayers can be put as delivery trucks which deliver VAA’s from the guardians to their destination,and have no capacity to tamper with the delivery outcome. There is no role of Relayers in the security aspect of the protocol, which makes the protocol easy to scale onto new blockchains. As the signing of VAA’s is done by the guardian network

and relayers can be easily incentivised to deliver the VAA's to the new blockchain. Only need would be to deploy the smart contracts. Wormhole protocol consists of core bridge contract, token bridge contract and an NFT bridge contract on all the chains which are registered on Wormhole for interoperability. The token bridge contract and NFT bridge contract are both built on the core bridge contract[13].

Audit Report can be found at last in this report where the Wormhole Protocol and smart contracts are audited and tested.

4. Conclusion and Future Work

We have gone through code and design of the existing widely used DeFi protocols and can conclude that the use of smart contracts continues to grow in the DeFi space, it is crucial to prioritize smart contract security to prevent potential hacks and financial losses. The focus on cross-chain, DEX, and decentralized oracle in DeFi brings new opportunities for decentralized finance but also increases the complexity and potential attack vectors. It is therefore important to apply best practices such as code audits, secure development methodologies, and continuous monitoring to mitigate the risks. As the industry evolves, collaboration between developers, auditors, and researchers will be essential in creating a more secure DeFi ecosystem. Overall, the adoption of smart contracts in DeFi has the potential to revolutionize the financial industry, but it is essential that we prioritize security to ensure its sustainability. Design-related security issues and Solidity-related security issues are two key areas where manual security testing and automatic testing can help identify vulnerabilities. In terms of designs of existing DeFi protocols, cross-chain, DEX, and decentralized oracle designs are three of the most popular. These designs provide innovative solutions for secure cross-chain communication, decentralized exchange, and secure oracle services. However, they also come with their own unique security challenges that need to be addressed. Solidity-related security issues are also a significant concern when it comes to smart contract security in DeFi. Solidity is the programming language used to develop Ethereum-based smart contracts, and it has its own set of security issues. As such, developers need to be aware of these issues and take steps to mitigate them. The security audit report is a critical aspect of ensuring smart contract security in DeFi.

We will be focusing on two important areas: Decentralized Exchanges (DEX) and Decentralized Oracle security. The growth of DeFi has been primarily driven by the success of DEXs like Uniswap, which have provided investors with a new way to trade assets on a decentralized platform. Similarly, Decentralized Oracles like Chainlink have allowed smart contracts to access external data sources in a secure and trustless manner. However, these technologies are not without their security risks and require rigorous auditing to ensure their safety.

References

1. Belchior, R., Vasconcelos, A., Guerreiro, S., & Correia, M. (2021). A Survey on Blockchain Interoperability: Past, Present, and Future Trends. INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal.
2. Robinson, P. (2018). Survey of crosschain communications protocols.
3. Zamyatin, A., Al-Bassam, M., & Zindros, D. (2020). SoK: Communication Across Distributed Ledgers.
4. Zhang, J., & Zhuang, Y. (2021). Xscope: Hunting for Cross-Chain Bridge Attacks.
5. Sztorc, P., & O'Connor, J. (2021). SoK: Not Quite Water Under the Bridge: Review of Cross-Chain Bridge Hacks.
6. Wang, Z., & Li, Y. (2021). SoK: Validating Bridges as a Scaling Solution for Blockchains.
7. Dagher, G. G., Mohler, J., Milojkovic, M., Marella, P. B., & Mohan, G. (2021). SoK: Exploring Blockchains Interoperability.
8. Wormhole Docs. Retrieved from https://book.wormhole.com/
9. Ma, Y. (2021). CeFi vs DeFi — Comparing Centralized to Decentralized Finance.
10. Luu, L., Narayanan, V., Zheng, C., Baweja, K., Gilbert, S., & Saxena, P. (2021). SoK: Decentralized Exchanges (DEX) with Automated Market Maker (AMM) Protocols.
11. Liu, J., & Luu, L. (2021). SoK: Decentralized Finance (DeFi).
12. Uniswap v3 Core. Retrieved from https://uniswap.org/docs/v3/core-concepts/introduction/
13. Wormhole Foundation. (2021). Wormhole GitHub Repository. Retrieved from https://github.com/wormhole-foundation/wormhole

Appendix

A.1 Wormhole Smart Contracts

Following are the sol files of Wormhole protocol [13]:

- Structs.sol - Defining structure of Provider, GaurdianSet, Signature, VM(Message)
- Getter.sol - Provides read-only access to various parameters and data of the Wormhole system
- Governance.sol - Decentralized governance mechanism for the Wormhole protocol
- GovernanceStruct.sol - Defining structs for proposals,voters etc. and other functions that help in governance.
- Implementation.sol - Publishing message event and initializing chain ID .(What is the need to set the Chain ID? ,Won't all chains have different smart contracts so why not hard code it)

- Message.sol - Parse and Verify the signed message.
- Migration.sol - Used as a tool for managing the evolution of the Wormhole system over time.
- Setters.sol - Internal setter functions

Structs.sol

Defines the following structures:

- Provider(Relayer) - Transfers message to the Guardians
- chainId - ID of chain where provider is located
- governanceChainId- ID of chain where governance contract is located
- governanceContract - Address of the governance contract

    - GuardianSet - Set of Guardians which signs messages
    - Keys - Stores the public keys of all Guardians
    - Expiration time - time after the guardian set is invalid

- Signature - Holds values of the signatures(signed using ECDSA)
- r,s,v - values of signature
- guardianIndex - holds index of guardian which signed
- VM(Verified Message?) - Parsed signed message structure

- version - representing the version of vm
- timestamp - for when the msg is created
- nonce - for uniqueness of msg
- emitterChainId - source chain ID
- emitterAddress - source address
- sequence - for ordering of messages
- consistencyLevel - the level of consistency required
- payload - message data
- gaurdianSetIndex - set of guardians that signed
- signatures - signatures of the guardians
- hash - hash of the VM

Messages.sol

Contains functions for verifying and parsing a VM

```
function parseVM(bytes memory encodedVM) public pure virtual returns (Structs.VM memory vm)
```

For parsing the encodedVM into the defined Struct of VM

Bytes of the encodedVM vs Parsed VM

- 1st Byte version
- 2-5 Bytes gaurdianSetIndex
- 6th Byte signersLen (used to create a signatures Struct array)
- 7th Byte 0th signature guardian index
- 8-39 Byte Signatures ‘r’
- 40-61 Byte Signatures ‘s’

- 62th Byte When added with 27 gives ‘v’

  The same is iterated for all signatures(let's say now the index is N)

- Next 4 bytes Timestamp
- Next 4 bytes Nonce
- Next 2 bytes emmitterChainId
- Next 32 Bytes EmitterAddress
- Next 8 Bytes sequence
- Next 1 Byte Consistency Level
- Rest Remaining Bytes Payload

In this function only a single check is made which is for checking the version.

Hash only includes the body after the signatures.

```
function verifySignatures(bytes32 hash, Structs.Signature[] memory signatures, Structs.GuardianSet memory guardianSet) public pure returns (bool valid, string memory reason) {
```

For verifying the signatures signed by the guardians. Takes hash of the body,Signature Struct Array, GuardianSet as attributes and returns bool with a reason string.

Signatures array is traversed and following checks are performed

Uses `ecrecover(hash, sig.v, sig.r, sig.s);` to get the address of the signatory if the signature is invalid it returns 0.

Two checks are made

```
require(signatory != address(0), "ecrecover failed with signature");
```

```
require(i == 0 || sig.guardianIndex > lastIndex, "signature indices must be ascending");
```

Here sig is the current signature.

The Second require statement is for ensuring that the signatures are in ascending order to prevent replay attacks.
If not done, the attacker could create a new combination of the signatures and it would be considered as a valid VM.

Then check is made

```
require(sig.guardianIndex < guardianCount, "guardian index out of bounds");
```

```
if(signatory != guardianSet.keys[sig.guardianIndex]){
    return (false, "VM signature invalid");
}
```

Returns the function if the signature is invalid otherwise the iteration continues.

Returns true if all iterations are completed.

```
function verifyVMInternal(Structs.VM memory vm, bool checkHash) internal view returns (bool valid, string memory reason) {
```

For Verifying VM. Takes VM and bool as attributes and returns bool with a reason string.

First Verification is to verify that the hash is equal to the actual hash of the body.

```
bytes32 vmHash = keccak256(abi.encodePacked(keccak256(body)));

if(vmHash != vm.hash){
    return (false, "vm.hash doesn't match body");
}
```

Next Verification is for checking whether the gaurdianSet has Zero keys

Next check is to verify that the current guardianSet is latest

```
if (vm.signatures.length < quorum(guardianSet.keys.length)){
    return (false, "no quorum");
}
```

Next Check is done by verifying the signatures.

If all these checks are passed then the function returns true.

Setter.sol

Contains all internal setter functions of the wormhole bridge protocol for setting,updating and deleting state variables.

Functions for setting fees,Guardian set updation and other functions to update chain Id's.

Implementation.sol

Contains initialization function and message publish event.

```
event LogMessagePublished(address indexed sender, uint64 sequence, uint32 nonce, bytes payload, uint8 consistencyLevel);
```

This event emits the message which is then relayed to the Guardians for verification and signing.

This event is triggered from

```
function publishMessage(
    uint32 nonce,
    bytes memory payload,
    uint8 consistencyLevel
) public payable returns (uint64 sequence) {
```

Here the required fees is confirmed then sequence function is called

```
function useSequence(address emitter) internal returns (uint64 sequence) {
```

It increments the sequence for the emitter and gives an index to the message. At last the 'LogMessagePublished' event is emitted.

```
  sequence = nextSequence(emitter);
  setNextSequence(emitter, sequence + 1);
```

```
function initialize() initializer public virtual {
```

It is run on deployment to initialize the chain with ID.

GovernanceStruct.sol

Defining Structs governance actions and other protocol structs like TransferFees,GaurdianSetUpgrade.

Parse functions for all upgrade requests.

```
/// @dev Parse a contract upgrade (action 1) with minimal validation
function parseContractUpgrade(bytes memory encodedUpgrade) public pure returns (ContractUpgrade memory cu) {
```

```
/// @dev Parse a guardianSet upgrade (action 2) with minimal validation
function parseGuardianSetUpgrade(bytes memory encodedUpgrade) public pure returns (GuardianSetUpgrade memory gsu) {
```

```
/// @dev Parse a setMessageFee (action 3) with minimal validation
function parseSetMessageFee(bytes memory encodedSetMessageFee) public pure returns (SetMessageFee memory smf) {
```

```
/// @dev Parse a transferFees (action 4) with minimal validation
function parseTransferFees(bytes memory encodedTransferFees) public pure returns (TransferFees memory tf) {
```

```
/// @dev Parse a recoverChainId (action 5) with minimal validation
function parseRecoverChainId(bytes memory encodedRecoverChainId) public pure returns (RecoverChainId memory rci) {
```

Governance.sol

Contains all the governance functions

Emits 2 events which are

```
event ContractUpgraded(address indexed oldContract, address indexed newContract);
event GuardianSetAdded(uint32 indexed index);
```

Contract Upgraded event is emitted from

```
function submitContractUpgrade(bytes memory _vm) public {
```

It is submitted via Governance VM.

First check is ‘!isFork()’

Second check is ‘verifyGovernanceVM()’ which is defined here

```
function verifyGovernanceVM(Structs.VM memory vm) internal view returns (bool, string memory){
```

Multiple checks are made in this verifyGovernanceVM()

First - verifyVM()
Second - If signed by current guardian set

Third - To verify if the governance chain is valid

Fourth - To verify if the contract is the desired one

Fifth - To verify if the governance action has been already performed.

Return true if all checks are passed.

After verifying the governance VM upgrade that is the VM payload is parsed using ‘parseContractUpgrade(vm.payload)

Two checks are made on the upgrade
First - Module check
Second - ChainId

Then governanceAction is set to consumed and the contract has been upgraded.

Next function is to set messageFee

```
function submitSetMessageFee(bytes memory _vm) public {
```

First check is made by verifying the governance VM and then the payload is parsed to upgrade.

Same two checks are made on the upload for module and chainId.

Governance action is set as consumed and update is made.

Next function is for deploying a new ‘guardainSet’

```
function submitNewGuardianSet(bytes memory _vm) public {
```

Vm is parsed and first check is made for verifying the governance VM and upgrade request is parsed from the payload.

Same 2 checks for module and chain are made.

Next check is made to check if the guardian set keys are not empty.

Next check for making sure that the guardian set index is incrementing.

And then governance action is consumed and changes are made.

Next function is for updating chainId and evmChainId

```
function submitRecoverChainId(bytes memory _vm) public {
```

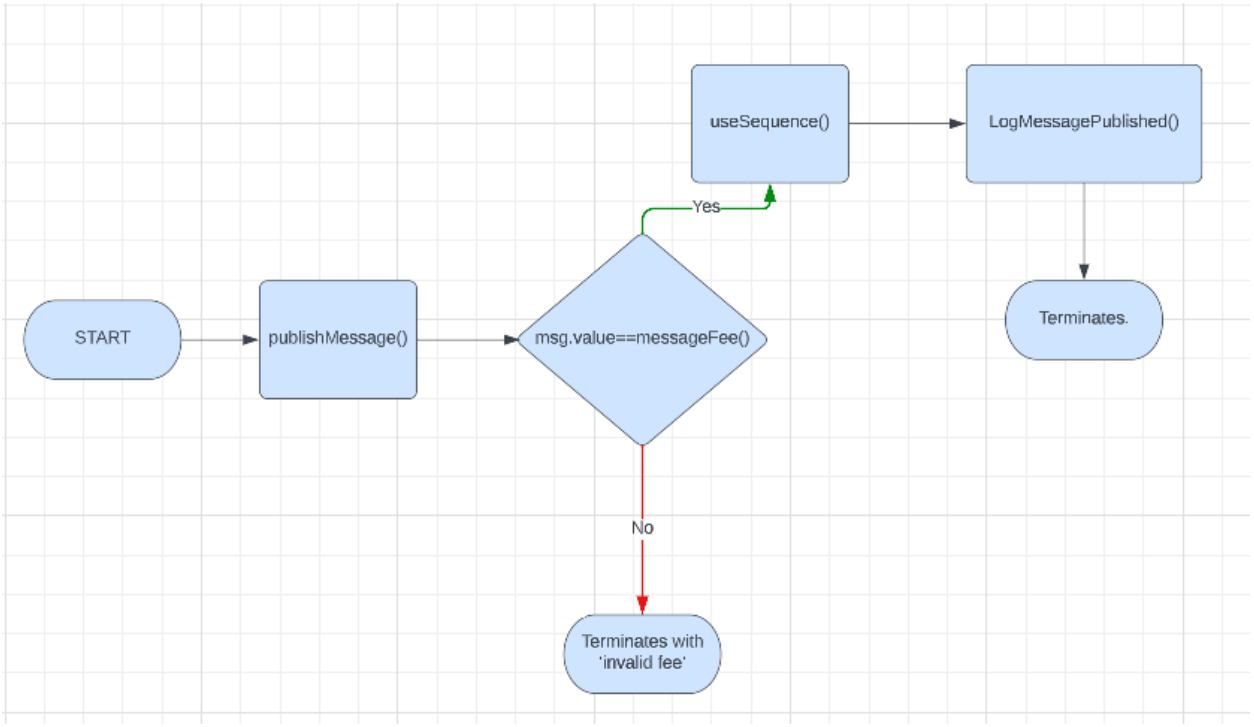
All checks are similar to previous functions.

Last function is

```
/**
 * @dev Upgrades the `currentImplementation` with a `newImplementation`
 */
function upgradeImplementation(address newImplementation) internal {
```

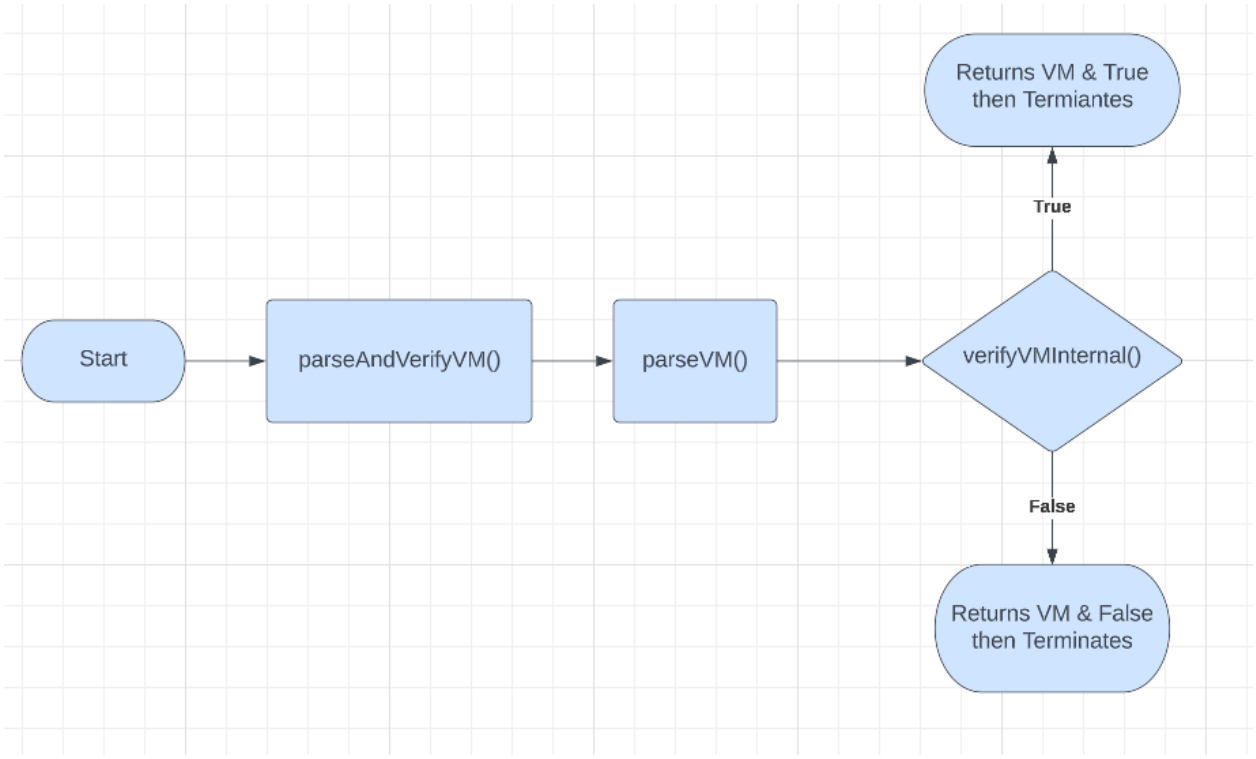
Wormhole Message protocol

The event LogMessagePublish() is emitted upon calling publishMessage() function,this is where the cross chain message originates.



A Require statement is used to make a check for the amount in the transaction with the payload is equal to the messageFee()(messageFee() is the value charged for transferring msg). If it doesn't hold the function is terminated, else the useSequence() is called for setting the sequence of the message for the msg.sender and incrementing it. Then at last the message is emitted using the event LogMessagePublish(). This event is seen by the guardians and when 2/3rd of guardians sign it , it is converted into a VAA which is relayed to the destination chain by the relayers.The destination also has the core bridge contracts deployed.

The function parseAndVerifyVM() in Messages.sol is used for verifying and parsing the VAA which was relayed.



parseVM() is used for parsing the encodedVM into the defined Struct of VM.

In this function only a single check is made which is for checking the version.

verifyVMInternal() is used for Verifying the VM. Takes VM and bool as attributes and returns bool with a reason string.Verifies the guardians signatures with respect to the indexes given to the guardians.This is done for ensuring that the signatures are in ascending order to prevent replay attacks.If not done, the attacker could create a new combination of the signatures and it would be considered as a valid VM. After verifying the VM it returns if it is valid or not and after which the function parseAndVerifyVM() is terminated. The parsed and verified VM can be used as required. The token bridge and NFT bridge are built on these contracts with some extra functions for example wrapEth(),attestToken() etc. The payload can be modified as per choice to have details of the token or NFT for say.

Wormhole - Security and Trust Assumptions

A cross-chain bridge's capacity to function depends on its users' level of trust and confidence in its dependability, security, and transparency, which is determined by security and trust assumptions. A cross-chain bridge may have a number of negative effects, including security vulnerabilities, a lack of transparency, and compliance and regulatory problems, if the security and trust assumptions of the bridge are not appropriately handled.

Security

Guardians

Wormhole guardians are encouraged to act in good faith by the proof-of-authority network design because any cooperation or malicious attacks would only be able to be linked to one of the 19 entities.As these are multi-billion dollar companies who do not want to damage their reputation by signing transactions incorrectly, the fact that a company like Jump Crypto is running Guardians in Wormhole's case is a social kind of security.

Additional Governor Feature

The Governor is intended to be a passive security check that particular Guardians can employ to rate-limit the notional value of assets that can be transferred outside of a specific chain in order to guarantee the reliability of the value kept inside a token bridge.The Governor sets a cap on the number of assets that may be removed from a specific chain in a specific amount of time. This offers a passive security check against external dangers like runtime vulnerabilities or defects in smart contracts. It is a safety precaution that lessens the likelihood and severity of user injury.

Trust Assumptions

Externally verified by Guardians

The proof of authority system used by Wormhole implicitly assumes that Guardians can be relied upon to validate transactions and that more than two thirds of Guardians won't collude at a given moment.

Censorship risk

1/3rd of Wormhole's Guardians can collude to censor a message.

Guardians care about reputation

Wormhole's strategy is based on the idea that the possible upside of collaboration outweighs the damage to its Guardians' good name. If the advantages for 1/3 of the guardians surpass the reputational cost of colluding, however, this might become a serious problem.

Validators don't have a bond

The stake of the Guardians is not bound, therefore if they act maliciously, their stake won't be reduced or they won't be punished.Therefore, there is no bonding or slashing mechanism in place to protect user cash.

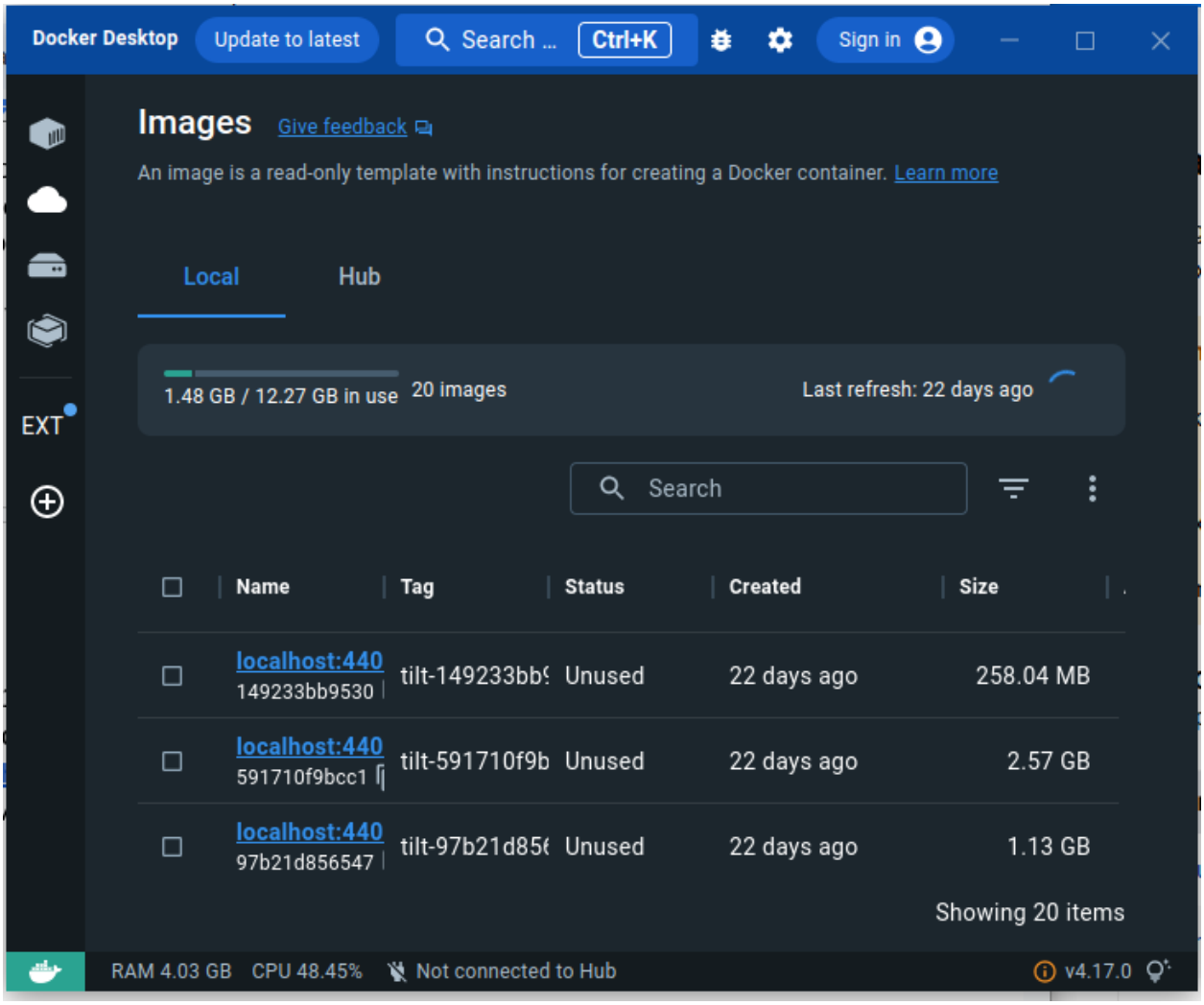
Local Installation Guide and Setup

Install Tilt(Devnet)

Tilt is part of the official Docker ecosystem. It's a tool which allows developers to easily configure a Kubernetes environment for development. It is best to use Tilt functionality in a UNIX-style environment.

Install Tilt (https://docs.tilt.dev/install.html) :

1.Verify that you have Docker and kubectl installed on your machine. If you don't have them installed, you can follow the instructions on https://docs.docker.com/desktop/install/linux-install/.
Also install Docker Desktop which should be ran upon startup.



2.Setup docker as a non-root user with commands

```
sudo groupadd docker
sudo usermod -aG docker $USER
newgrp docker
docker run hello-world //Verify that you can run docker without sudo
```

3.Install Kubectl which allows you to run commands for Kubernetes clusters.Use the following commands:

```
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"

Validate the download by downloading kubectl checksum file using
```

```
curl -LO "https://dl.k8s.io/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl.sha256"
echo "$(cat kubectl.sha256)  kubectl" | sha256sum --check
```

Install kubectl using command

```
sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
```

```
manibrar@brux:~$ kubectl version --client
WARNING: This version information is deprecated and will be replaced with the output from kubectl version --short.  Use --output=yaml|json to get the full version.
Client Version: version.Info{Major:"1", Minor:"26", GitVersion:"v1.26.3", GitCommit:"9e644106593f3f4aa98f8a84b23db5fa378900bd", GitTreeState:"clean", BuildDate:"2023-03-15T13:40:17Z", GoVersion:"go1.19.7", Compiler:"gc", Platform:"linux/amd64"}
Kustomize Version: v4.5.7
```

4.Install ctlptl and create Kind with a local registry

curl -fsSL https://github.com/tilt-dev/ctlptl/releases/download/v$CTLPTL_VERSION/ctlptl.$CTLPTL_VERSION.linux.x86_64.tar.gz | sudo tar -xzv -C /usr/local/bin ctlptl

Create Kind cluster after installation

ctlptl create cluster kind --registry=ctlptl-registry

```
manibrar@brux:~$ ctlptl get
CURRENT   NAME        PRODUCT   AGE   REGISTRY
*         kind-kind   kind      22d   localhost:44031
```

5.Install Tilt using
curl -fsSL https://raw.githubusercontent.com/tilt-dev/tilt/master/scripts/install.sh | bash

Install GO

Use Command to remove any Go installation deleting the /usr/local/go folder and extract the archive you download from https://go.dev/dl/ :
rm -rf /usr/local/go && tar -C /usr/local -xzf go1.20.3.linux-amd64.tar.gz

Add the following line to your $HOME/.profile or /etc/profile (for a system-wide installation):
export PATH=$PATH:/usr/local/go/bin

Check using `go version` command

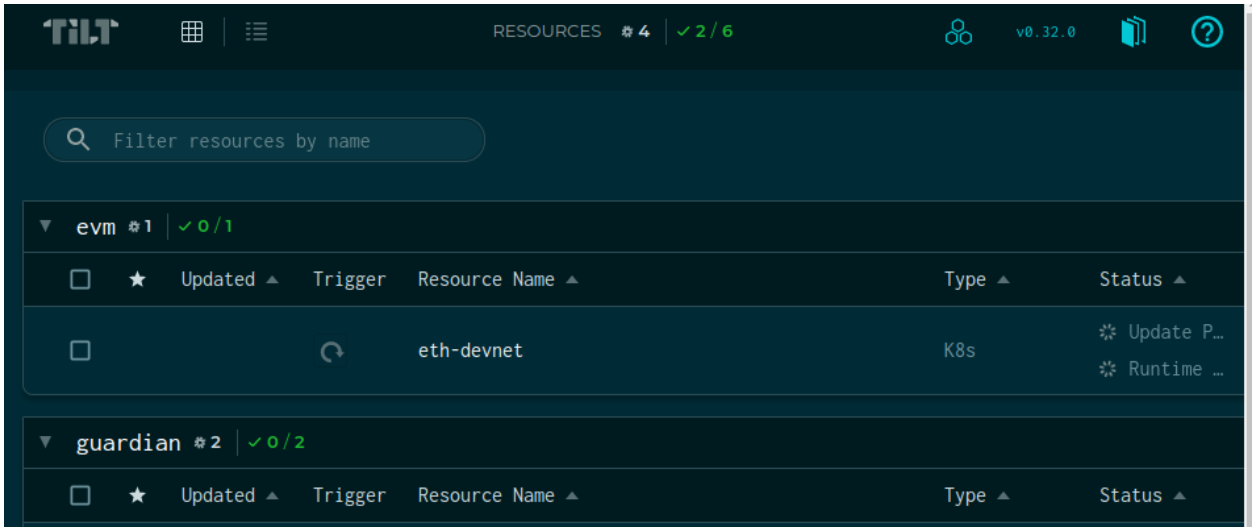
Use Tilt

Clone Wormhole's github repository and then tilt up to start the local server :

git clone --branch main https://github.com/wormhole-foundation/wormhole.git

cd wormhole

tilt up



Dashboard will appear where you can see all the endpoints (using which we can connect to the deployed chains and smart contract). Use other chains which are set to false in the Tiltfile by setting them true.
For example: tilt up -- --algorand=true --solana=true

Use TestNET GUI

Use testnet faucets for experimenting real time token transfer at (https://wormhole-foundation.github.io/example-token-bridge-ui/#/transfer) which can be found in Wormhole's github repository.