A Study on EOS Blockchain Smart Contracts

Lan H. Nguyen Maria M. Vela Melendez Mikel Santana Maraña
Illinois Institute of Technology
Department of Computer Science
Chicago, IL, USA
{Lnguyen18,mvelamelendez,msantanamarana}@hawk.iit.edu

ABSTRACT

One of the most disrupting technologies of the 21st century is digital currency. Some examples of the most widely deployed digital currencies are Bitcoin and Ethereum. Digital currency relies on an idea of a decentralized network of computers interconnected with each other to form a cryptographically secure blockchain network. Some advantages of digital currencies over traditional currencies are low transaction fees, fast processing, deflationary, and unbanked support. These digital currencies are kept secure through a Proof-of-Work algorithm that is energy intensive – It is estimated that about 120 terawatt-hours of electricity are used annually at tens of billions of dollars. Continuing with an unprecedented growth of distributed, decentralized blockchain networks, a more efficient, automated, self-control, immutable form of binding agreement between parties was born, Smart Contracts. Ethereum has been under the spotlight of cryptocurrency market, which has driven numerous researches into studying the security of its smart contract. This paper hopes to perform a comprehensive study on EOSIO smart contract and its known vulnerabilities. In addition, we also strive to cover several safety concerns that have not yet been addressed.

Index terms:
cryptocurrency, blockchain, smart contract, virtual machine, proof-of-work, proof-of-stake, software security, vulnerability, exploits

1 INTRODUCTION

Society is built on contracts, starting with the implied and explicit social contract that forms the basis of modern Law and Order. Everything works with agreements between the parties. But the enforcement of traditional contracts is complex and the enforcement system is inefficient and unreliable. Back in 2015, Vitalik Buterin, co-founder of Ethereum, when exploring the additional functionalities the Blockchain technology could provide, came up with the concept of Smart Contract. As it will be explained, this could be helpful to solve the previously mentioned problem.

Smart contracts enable trusted transactions and agreements between disparate, anonymous parties without the need for a central authority, legal system or external enforcement mechanism. Intermediaries and, by extension, associated delays and fees are removed.

The terms of a Smart Contract are set in code. They are agreed upon cryptographically and they're enforced by distributed consensus that reliably executes that code. This has numerous applications in reducing the level of trust and third-party participation that is required in numerous business scenarios. A review of these consensus protocols will be conducted.

Traditionally, the most extended used one was Proof-of-Work (PoW). But due to various limitations of PoW, researchers have proposed different promising consensus algorithms to replace the original PoW. Notable consensus are Proof-of-Stake or Proof-of-Share (PoS) and Delegated Proof-of-Stake (DPoS). These consensus algorithms solve a wide array of cryptocurrency problems ranging from power consumption to transaction throughputs. Unlike the founding cryptocurrency, Bitcoin, Ethereum and EOS are equipped with additional features that aim to outperform Bitcoin such as high transaction throughput and notably, the mentioned, smart contracts. It is important to note that a smart contract is embedded in a blockchain and is secured by a blockchain itself. Thus, on a high level, security characteristics remain as secure as a blockchain it resides on. Even though blockchain is deemed to be far more secure than traditional finance, there's still security risks being exploited such as 51% attack i.e., a mining pool could overtake an entire blockchain if they have more than 51% of mining power. With that being said, as a rule of thumb, the more complex and feature-rich a program becomes, the more probability there exists a security or loop-hole that can be exploited by malicious actors.

2 BACKGROUND

The Ethereum project was the first blockchain protocol to install smart contract technology. And nowadays it is the most popular one. This technology can save a huge amount of money for businesses and organizations. As this technology has vast potential, several smart contract platforms, known as "Ethereum's Killers," compete to offer developers the best tools.

EOS is one of many Ethereum alternatives. Its ultimate aim is to be the fastest, cheapest, and the most scalable smart contract blockchain globally. As a result, it aspires to win a large percentage of Ethereum's market share.

Although EOS and Ethereum might seem very similar, there are some clear differences. This section will study each of these blockchain platforms separately to delve into the similarities and, more importantly, the differences that set them apart.

For that, in this background approach, we will introduce the protocols used as consensus to validate the different transactions, what an account is, and how they are managed in Ethereum and EOS. Then the different parts of a transaction and its lifecycle will be analyzed, together with the basic idea of smart contracts. Finally, a brief introduction to the virtual machines used to run the nodes of the network will be explained.

2.1 Ethereum

2.1.1 Ethereum Consensus Protocol.

In every Blockchain network, there must be a consensus protocol.

As mentioned, there are different kinds having all of them something in common: they form the backbone of the network, helping the nodes in the network verify the transactions.

In blockchain, reaching consensus means that at least 51% of the nodes on the network agree on the next global state of the network. It allows distributed systems to work together securely. Consensus mechanisms were used way before blockchain, but in recent years, new ones have been developed to enable crypto economic systems, such as Ethereum, to agree on the state of the network.

Ethereum's, like Bitcoin's, current consensus protocol is Proof-of-work (PoW) [1].

The Proof-of-work is performed by miners, who compete to create new blocks filled with processed transactions. The winner shares the new block with the rest of the network and wins some ETH. The race is won by the computer that solves a mathematical puzzle the fastest, which produces the cryptographic link between the current block and the previous one. Solving this puzzle is the challenge of Proof-of-work.

The network is kept secure because it would take 51% of the network's computing power to defraud the chain. This would require such a large investment in equipment and energy that it would probably spend more than it would gain.

It should be noted that Ethereum is planning an upgrade that will change its consensus protocol to Proof-of-stake (PoS).

We think it would be interesting to comment that, technically, proof-of-work and proof-of-stake are not consensus protocols per se but are often referred to as such for simplicity. They are Sybil resistance mechanisms and block author selectors; they decide who is the author of the latest block. This Sybil resistance mechanism combined with a chain selection rule is a true consensus mechanism. Sybil resistance measures how a protocol behaves against a Sybil attack. Sybil attacks occur when a user or group impersonates many users. Resistance to this type of attack is essential for a decentralized blockchain and allows miners and validators to be rewarded equally based on the resources contributed. Proof-of-work and proof-of-stake protect against this by making users spend a lot of energy or put up a lot of collateral. These protections are an economic deterrent to Sybil attacks. A chain selection rule is used to decide which chain is the "correct" one. For example, Ethereum and Bitcoin currently use the "longest chain" rule, meaning that whichever blockchain is the longest will be the one that the rest of the nodes accept as valid and work with. In the case of Proof-of-Work chains, the longest chain is determined by the total cumulative proof-of-work difficulty of the chain.

2.1.2 Ethereum Virtual Machine.

We will start by recalling a crucial fact: Ethereum has one and only one 'canonical' state at any given block in the chain. But what is a state? Ethereum's state is a large data structure that holds not only all accounts and balances but also a machine state, which can change from block to block according to a predefined set of rules and execute arbitrary machine code. The specific rules of changing state from block to block are defined by the Ethereum Virtual Machine (EVM) [2].

Therefore, EVM is the global virtual computer whose state every participant on the Ethereum network (every Ethereum node) stores and agrees on. Ethereum nodes are the real-life machines that store the EVM state. Nodes communicate between themselves to propagate the modifications occurring in the EVM state. For example, any participant can request the execution of arbitrary code by broadcasting a code execution request from a node. If the code execution is processed, the state of the EVM is changed, and all the nodes of the Ethereum network will be notified of this change and will store the modified state.

EVM works in a very simple manner; it behaves like a mathematical function. For a given input, it returns a deterministic output. The EVM executes as a stack machine with a depth of 1024 items. Each item is a 256-bit word chosen for ease of use with 256-bit cryptography. The EVM maintains a transient memory (as a word-addressed byte array) during execution, which does not persist between transactions. This section will also study how data is stored in EVM. The Ethereum Virtual Machine has three possible options for where to store data:

- **Storage:** Is where all the contract state variables reside. Every contract has its own storage, and it is persistent between function calls and quite expensive to use
- **Memory:** Used to hold temporary values. It is erased between (external) function calls and is cheaper to use
- **Stack:** Used to hold small local variables. It is almost free to use but can only hold a limited amount of values

2.1.3 Ethereum Accounts and Permissions.

Ethereum follows an entire account model. This means that there is actually data that describes token balances and other related information for each account. But what is an account?

An Ethereum account is an entity with an ether (ETH) balance that can send transactions on Ethereum. Accounts can be user-controlled or deployed as smart contracts. Users can initialize accounts, deposit ether into the accounts, and transfer Ether to other users. Accounts and their balances are stored in a big table in the EVM; they are a part of the overall EVM state.

We will review the two types of accounts in Ethereum. The first class is called externally-owned. The characteristics of this type of accounts are: creating them costs nothing, can initiate transactions, and transactions between this type of accounts can only be ETH/token transfers. On the other hand, we have the contract accounts, with the following properties: creating them has a cost because the network storage is being used, can only send transactions in response to a received transaction, and transactions from an external account to a contract account can trigger code, which can execute many different actions, such as transferring tokens or even creating a new contract.

An account is formed by a keypair: public and private keys. Their function is to verify that the corresponding sender has signed a transaction to prevent forgeries. Therefore, users sign their transactions, granting their custody over the funds associated with their account. Ethereum users never really hold Ether, as the funds always stay on Ethereum's ledger; what they hold is their private keys. This prevents malicious actors from broadcasting fake transactions because users can always verify the sender of a transaction.

Ethereum accounts have four fields:

- *nonce:* A counter that indicates the number of transactions sent from the account. This ensures that transactions are

only processed once. This number represents the number of contracts created by the account in a contract account
- *balance:* The number of wei owned by this address. Wei is the smallest denomination of ETH, and there are 1e+18 wei per ETH
- *codeHash:* This hash refers to the code of an account on the Ethereum virtual machine (EVM). Contract accounts have code fragments programmed in, so that they can perform different operations. This EVM code gets executed if the account gets a message call. It cannot be changed, unlike the other account fields. All such code fragments are contained in the state database under their corresponding hashes for later retrieval. This hash value is known as a codeHash. The codeHash field is the hash of an empty string for externally owned accounts
- *storageRoot:* Also known as a storage hash. A 256-bit hash of the root node of a Merkle Patricia trie encodes the account's storage contents (a mapping between 256-bit integer values), encoded into the trie as a mapping from the Keccak 256-bit hash of the 256-bit integer keys to the RLP-encoded 256-bit integer values. This trie encodes the hash of the storage contents of this account and is empty by default

2.1.4 Ethereum Transactions.

An Ethereum transaction refers to an action initiated by an externally-owned account, in other words, an account managed by an individual, not a contract. For example, if Bob sends Alice 1 ETH, Bob's account must be debited, and Alice's must be credited. This state-changing action takes place within a transaction.

Transactions, which change the state of the EVM, need to be broadcast to the whole network. Any node can broadcast a request for a transaction to be executed on the EVM; after this happens, a miner will execute the transaction and propagate the resulting state change to the rest of the network.

Transactions require a fee and must be mined to become valid.

A submitted transaction includes the following information:

- recipient – The receiving address (if an externally-owned account, the transaction will transfer value. If a contract account, the transaction will execute the contract code)
- signature – The identifier of the sender. This is generated when the sender's private key signs the transaction and confirms the sender has authorized this transaction
- value – The amount of ETH to transfer from sender to recipient (in WEI, a denomination of ETH)
- data – Optional field to include arbitrary data
- gasLimit – The maximum amount of gas units that the transaction can consume. Units of gas represent computational steps
- maxPriorityFeePerGas – The maximum amount of gas to be included as a tip to the miner
- maxFeePerGas – The maximum amount of gas willing to be paid for the transaction (inclusive of baseFeePerGas and maxPriorityFeePerGas)

Gas refers to the computation required to process the transaction by a miner. Users have to pay a fee for this computation. The gasLimit and maxPriorityFeePerGas determine the maximum transaction fee paid to the miner.

Nevertheless, a transaction object needs to be signed using the sender's private key. This proves that the transaction could only have come from the sender and was not sent fraudulently. The transaction can be cryptographically proven with the signature hash that it came from the sender and submitted to the network.

TRANSACTION LIFECYCLE

Once the transaction has been submitted, the following happens:

(1) Once a transaction is sent, cryptography generates a transaction hash
(2) The transaction is then broadcast to the network and included in a pool with many other transactions
(3) A miner must pick the mentioned transaction and include it in a block in order to verify the transaction and consider it "successful"
    - If the network is busy the user that generated the transaction may have to wait
(4) The transaction will receive "confirmations". The number of confirmations is the number of blocks created since the block included the transaction. The higher the number, the greater the certainty that the network processed and recognized the transaction
    - Recent blocks may get re-organized, giving the impression the transaction was unsuccessful; however, the transaction may still be valid but included in a different block
    - The probability of a reorganization diminishes with every subsequent block mined, i.e., the greater the number of confirmations, the more immutable the transaction is

2.1.5 Ethereum Smart Contracts.

A "smart contract" is simply a program that runs on the Ethereum blockchain [3]. It is a collection of code (its functions) and data (its state) that resides at a specific address on the Ethereum blockchain.

Smart contracts are a type of Ethereum account. This means they have a balance and can send transactions over the network. However, they are not controlled by a user; instead, they are deployed to the network and run as programmed. User accounts can then interact with a smart contract by submitting transactions that execute a function defined on the smart contract. Smart contracts can define rules as regular contracts and automatically enforce them via the code. Smart contracts cannot be deleted by default, and their interactions are irreversible.

At a fundamental level, a smart contract can be pictured like a sort of vending machine: a script that, when called with specific parameters, performs some actions or computation if certain conditions are satisfied. For example, a simple vendor smart contract could create and assign ownership of a digital asset if the caller sends ether to a specific recipient.

Anyone can write a smart contract and deploy it to the network. The only requisites are: know how to code in a smart contract language and having enough ETH to deploy the contract. Deploying a smart contract is technically a transaction, so a gas fee must be paid the same way that happens in a simple ETH transfer. Gas costs for contract deployment are far higher, however.

Ethereum has developer-friendly languages for writing smart contracts: Solidity and Vyper.

Smart contracts are public on Ethereum and can be considered open APIs. That means smart contracts can be called by other contracts. Contracts can even deploy other contracts.

Smart contracts alone cannot get information about "real-world" events because they cannot send HTTP requests. This is by design. Relying on external information could jeopardize consensus, which is essential for security and decentralization.

Another limitation of smart contracts is the maximum contract size. A smart contract can be a maximum of 24KB, or it will run out of gas. This can be circumnavigated by using The Diamond Pattern.

2.2 EOS

Before going deep into different kinds of vulnerabilities EOS smart contracts can encounter, we first need to introduce different aspects of the EOS blockchain network and its different characteristics to obtain a general idea of how this platform works.

2.2.1 EOS Consensus Protocol.

Moving forward from the PoW used by Bitcoin and Ethereum, the blockchain platform of EOSIO uses a modified PoS called DPoS. DPoS stands for Delegated Proof of Stake. Why delegated? Because each stakeholder of the network can vote up for 30 block producers to act on their behalf as their DPoS delegates.

Once every stakeholder in the network has done their voting, the top 21 elected producers are the delegates to produce and sign blocks. This voting action is done once in a while so the elected producers are not always the same. Also to mention that the weight of each stakeholder is computed as a function of the number of tokens staked and the time elapsed since the EOSIO blocks timestamp epoch [4].

Now that we do know how the delegated term works and what it is used for, let's explain the different layers EOS consensus protocol has. In this case, the protocol is split up into two different ones: Native consensus (aBFT) and the DPoS itself.

(1) **Native Consensus (aBFT):** Decide which blocks (received and synced among the elected producers) will become final. Final means permanently recorded in the block chain. In order to accomplish that there is a two-stage confirmation process, so, two thirds of the super majority of producers from the current scheduled set need to confirm each block twice. As there are 21 block producers, 15 out of the 21 conform two thirds plus one. First confirmation stage proposes a last irreversible block (LIB) and the second one confirms the proposed LIB as final. At this point, the block becomes irreversible

(2) **Delegated Proof-of-Stake (DPoS):** This layer introduces the concepts of tokens, staking, voting... and is also responsible for generating new block producer schedules from the stakeholders voting. The production and signing of the blocks occur in rounds of 126 seconds and that is the time it takes for a block producer to be assigned a timeslot. That time slot lasts 6 seconds per producer where a maximum of 12 blocks can be produced and signed. This layer is enabled by WASM smart contracts

As each producer has 6 seconds to produce/sign 12 blocks, the production time per block is 500ms. Finally, $6x21 = 126$ seconds, that is what each round lasts

2.2.2 EOS Virtual Machine.

As one of the main parts in the EOS blockchain platform we encounter the virtual machine. This one has been constantly developing until the one which exists nowadays. This virtual machine is very different from the EVM (Ethereum Virtual Machine) as it is a web assembly interpreter. Web Assembly is a portable binary code format for executable programs and textual assembly language. Firstly, it was created for web environments even though nowadays can also work standalone. In addition, it supports any programming language and any operating system. The EOS-VM (Virtual Machine) is in charge of synchronously executing smart contract code. EOS VM uses WASM as the final bytecode format for smart contracts in order to run them in the fastest and most efficient way. It has a sandboxed, memory-safe execution environment, and it's efficient. Compilers to WASM are in very stages of development from numerous programming languages. C++ is ahead of the rest of languages and is the most supported language for using EOSIO [5].

This virtual machine is stack-based. For maintaining the stack, it needs two basic attributes: code listing, which has the instruction pointer, and the stack data, which is accessed via the stack pointer. Obviously, the size of the stack has been limited by the eosio.cdt module. In this case, the maximum size it can have is 8KB even if it seems adjustable by the developer [6].

This virtual machine also has three different daemons which accomplish different actions:

- **Nodeos:** The core service daemon. Used to run nodes in the EOSIO network. It processes smart contracts, validates transactions, produces, records and confirms blocks in the EOSIO blockchain
- **Cleos:** Is the command line interface used in order to give instructions to the virtual machine
- **Keosd:** Key manager daemon. Local wallet file for secure storage of secure keys

2.2.3 EOS Accounts and Permissions.

EOSIO is an account based platform. Every action that is taken, every change that is made, every piece of data, every token that is stored or sent is done so with the authorization of an account or a group of accounts. Accounts are controlled by key pairs and store EOS tokens in the Blockchain.

Accounts can hold smart contract code. Pushing an action to such an account activates the smart contract code for that action.

Regarding the account in EOSIO blockchain platform, an account identifies in a unique way a participant in the platform. They also represent smart contract actors that push and receive actions to and from other accounts.

We must also present another concept regarding accounts. The Application Binary Interface (ABI) describes a smart contract's actions and tables and how to interact with them. The ABI also indicates the input each action takes. In general, the ABI can be examined for an account to figure out how to interact with a smart contract, in order that when an account calls a smart contract's action it does it with the adequate inputs.

Currently, the ABI is a JSON, written manually or generated by *abigen* [7], and posted to state along with the smart contract code. Actions do not have to be included in the ABI to be exposed.

Related to the accounts another term comes up: permissions. These are used to authorize actions and transactions to other accounts. Each of the permissions of an account is linked to an authority table which contains the threshold that must be reached in order to allow a certain action to be executed on behalf of that permission.

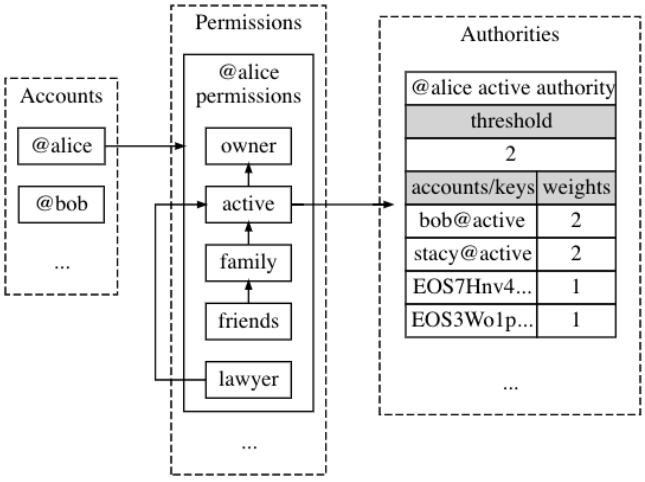

Figure 1: Example of an authority table [8]

The attached [Figure 1] is an example of the authority table of Alice´s *active* permission. This permission, together with the *owner* one, is the default one of every EOSIO account. *Owner* is the parent and *active* is the child, as they are structured in a hierarchical way. Usually, the one used to execute actions is *active*. Regarding the thresholds, the account *bob* with the *active* permission is able to, he alone, execute (and sign) an action whose receiver will be the *active* permission of *Alice* because his weight is of two and the threshold it needs is also two. The threshold is used in order to execute an action under certain permission [8].

In addition, there exist two kinds of accounts; the official ones created by the EOSIO platform itself and the ones created by the DApp developers.

Each account in the EOSIO blockchain platform is identified as a long number and it has a human readable attribute of 1 to 12 characters long in order to have enough options of accounts name for each user and not using twice the same. This number is not the user's public key.

We can state that EOSIO has a powerful account and permission model, and the key pairs which control accounts can be updated.

Resources

Once we have reviewed the structure of EOS accounts, we will proceed to explain the resources EOS accounts use. These resources are implemented and managed through eosio.system Smart Contract.

Just like in any computing system, applications on EOS need resources to run. There are three resources built in EOSIO:

- **RAM:** Used to store data on the blockchain. All the smart contract code, all token balances, all address information, in general, all data that is used by a smart contract is stored in RAM. It is paid for by the payer account specified in the contract. In some cases, this will be the contract account, the smart contract or business behind it. In other cases, this is the user account. Moreover, it should be commented that up until and including EOSIO 2.0, the only way to hold information in state is using a multi-index table which consumes the RAM resource
- **CPU:** Refers to the processing time it takes to execute some code or execute a smart contract action. Obviously, more complicated actions with more instructions will require more processing power and so, will expend more CPU. CPU is built in microseconds and there is an absolute limit of 30 millisecond. Therefore, we shouldn't create actions which are too complex. For example, we can't loop through an action an arbitrary number of times and assume that it will complete because we might hit a limit, and the action can no longer be performed because that loop exceeds the 30 milliseconds limit
- **NET:** Represents the bandwidth used for transaction data transfer. Traditionally, resource challenges on EOSIO public networks have mostly been related to CPU and RAM, not NET. This may change in the future since there are proposals for technologies that do things like store files using this third resource

We will continue explaining how these presented resources are managed. Resource Management models can vary.

On public networks, in general, resources and accounts are managed by system contracts. These are special accounts with contract code that are managed by a multi-signature authority. System contracts can be upgraded with a two-thirds majority of the block producers running the system. So, if a majority, a two-thirds majority, of the block producers has agreed to upgrade a system contract, the upgrade gets carried out through a multi-signature transaction. In this set up accounts generally stake the network asset to CPU/NET and purchase RAM directly.

2.2.4 EOS Transactions.

Regarding the transactions in EOS, they are the basic unit to be verified and packaged in blocks. Transactions are composed of one or multiple actions and the action is the basic unit to trigger a function. In addition, notifications are used to notify a target account of the current action being executed. In order for a transaction to succeed all the actions on it must succeed, so they are executed atomically and in a predefined order.

Smart contracts in EOSIO consist of a set of actions. Several actions are already implemented in the smart contracts by default for different purposes: account creation, producer voting, token operations… and the developers can create new, replace them or modify their functionalities.

TRANSACTIONS LIFECYCLE

Once a transaction is ready to be executed several steps need to be followed in order the whole process to be completed. The lifecycle of a transaction is the following [9]:

(1) **Create transaction:** Instantiate a transaction object and push the related action instances into a list within the transaction instance. The action instance contains the receiver

account, name of the action, list of actors and permission levels that must authorize the transaction

(2) **Sign transaction:** Transaction must be signed by a set of keys sufficient to satisfy the accumulated set of explicit *actor::permission* pairs specified in all the actions enclosed in a transaction. Linkage is done through authority-tables. The signing key is obtained by querying the associated wallet (wallet is where EOS keys are stored locally) with the signing account on the client where the app is being run. The needed parameters for the transaction signing process are: transaction instance to sign, set of public keys from which the associated private keys within the app wallet are retrieved, and the chain id
After the transaction digest (SHA-256 calculation) is computed, the transaction is finally signed with the private key associated with the signing account´s public key. The signature is then added to the signed transaction instance

(3) **Push transaction:** After the transaction is signed, a packed transaction instance is generated and pushed from the app to the local node. This node relays the transaction to the active producing nodes for signature verification, execution and validation

(4) **Verify transaction:** This process is a two-fold. First, the public key associated with the accounts that signed the transaction are recovered (this is cryptographically possible thanks to ECDA or elliptic curve digital signature algorithm). Second, the public key of each actor specified in the list of action authorizations from each action of the transaction is checked against the set of recovered keys
Then, each satisfied *actor::permission* is checked against the associated minimum permission required for that *actor::context::action*. Finally, the authority table (explained in account management) for the receiver account´s permission that matches each actor´s permission within the action context is checked

(5) **Execute transaction:** A chain database session is started and a snapshot is taken as it allows to rollback any changes in case of failure. To execute the transaction, each action associated is dispatched for execution. Context free actions are executed first and then, regular actions

(6) **Finalize transaction:** A corresponding action receipt is produced for each action. This receipt contains a hash of the corresponding action instance, and a few more attributes. From the action receipts a transaction receipt is generated and pushed to the signed block, along with other transaction receipts [9]

2.2.5 *EOS Smart Contracts.*

As it is already known, smart contract is the software that runs on the top of the blockchain nodes. The smart contracts expose executable actions which are functions that perform any specific operations. EOS smart contracts are written in C++ and then converted to Web Assembly bytecode, as that is what the EOS virtual machine understands.

As smart contracts are stored in the accounts of the different users of the network, a dispatcher which handles the calls to different actions is needed. That dispatcher will be the apply function

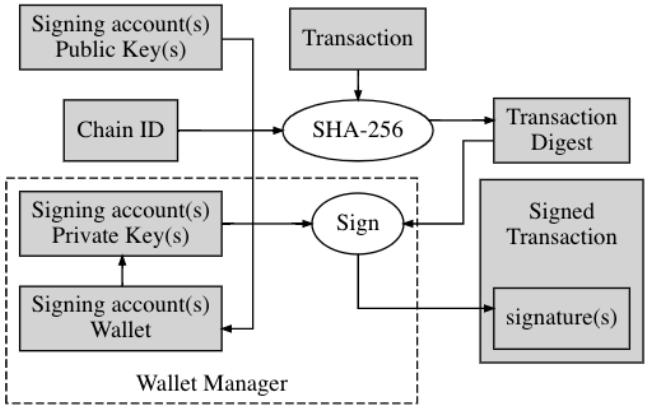

Figure 2: Transaction digest computation [9]

which is available in every smart contract and is the entry function. As input parameters, there is the receiver which is the account where the dispatcher is acting. Then, the code is the account in which the code is being executed, and finally action, which is the name of the invoked function, depending on its value one part of the code or another is going to be executed.

This function is used by every smart contract anytime a transaction or an action calls for a specific function, and so, some security measures must be taken into account as some of the well-known EOS vulnerabilities start in this function. This function and the security issues it has will be explained afterwards.

2.3 EOS vs. Ethereum

After carrying out the extensive study presented in the previous points, we proceed to bring together the concepts explained to compare Ethereum and EOS. We will now comment on the similarities and the differences of these platforms and draw conclusions based on these.

As we explained, Ethereum leverages the proof-of-work mining algorithm where each node computes a puzzle. This puzzle is difficult to solve, and any miner wanting to contribute needs to connect to the network. The miners compete and receive Ether as a reward. Since the block generation time is directly proportional to the transaction rate, Ethereum implementation has reduced the time from 10 min (in Bitcoin) to 15 seconds. Besides, tons of computing power and energy is wasted with PoW consensus protocol.

EOS uses Delegated proof of stake (DPoS), which is more robust as stakeholders vote for the witnesses. This lowers the block generation time as transactions can be confirmed faster. In PoS, a user holding the coins cannot validate transactions but can vote to determine the verifier of transactions. Therefore, irrespective of the number of miners, EOS can produce new blocks every 500 milliseconds.

Regarding scalability, implementing DPoS gives EOS an enormous advantage against Etherum. Even though considerable improvements, using PoW as consensus protocol has resulted in performance issues for Ethereum. Ethereum's network is heavily congested because of its time to process transactions. This is the main reason why Ethereum is currently transitioning to Proof-of-Stake,

and it is believed that it will become Ethereum's consensus protocol in the future upgrade that will take place next year.

On the other hand, both platforms use a stack-based virtual machine. However, in EOS case, the maximum size of the stack is 8KB, whereas in EVM stack's the maximum size is 32KB [2].

Concerning accounts, both Ethereum and EOS follow a full account model. Ethereum's account model is based directly on keypairs; an account is a key. This means if a user has his private key controlling a public key to which he receives tokens, that is his account and controls his token balances.

On EOSIO, things are different. Accounts are identified in the system as a long number, and it has a human-readable representation between one and 12 characters. This number is not the user's key; it is not related to a key pair, one or more key pairs control it, and those keypairs can be updated.

Finally, in EOSIO smart contracts are stored in the accounts, that means that each account can own as much as one smart contract. In Ethereum the smart contracts are a different type of account where users can ask to execute certain functions.

As for transaction fees and resource usage, EOS and Ethereum work completely differently.

EOS follows an ownership model. This means that the users are the owners of the resources provided. We are pointing to the previously presented resources: CPU, RAM, and NET bandwidth. Therefore, there is no need to pay rent on an EOS blockchain. EOS intends to be a decentralized operating system rather than a decentralized supercomputer, which can be used by DApp developers to create and code various DApps.

By staking and locking up EOS tokens, users are provided an equivalent amount of resources in NET and CPU bandwidth in return. However, since RAM is a limited resource, the action of staking does not provide RAM to users. So, to obtain it, it is necessary to purchase it directly from the RAM marketplace.

Since these resources are scarce, EOS doesn't want users to hold on to their tokens for too long. The company behind EOS has explicitly mentioned in the platform constitution that EOS members who don't use their tokens for three years would get their accounts terminated.

Ethereum follows a different approach. Ethereum operates on a rental resource mechanism, which implies paying transaction fees to use an Ethereum blockchain. The miners are responsible for putting transactions inside their blocks. They must use their computational power to validate smart contracts to do so. In general, any transaction initiated on the Ethereum blockchain network carries transaction fees that must be paid in ETH.

This fee acts as fuel for completing the transactions and ensures the network's security. Ethereum aims to build a worldwide supercomputer that will rent out its computation power to developers all over the world. This 'computational power' is named 'Gas,' and to execute each step of a smart contract, a certain amount of gas must be spent. The more Gas you pay, the faster a transaction will be processed.

The main difference regarding the smart contracts is that in Ethereum they are independent from the users or the accounts, and in EOS they are a mandatory part of the account, that means they are a snippet of code stored inside the blockchain network node.

Nevertheless, the programming language in which they are written is also different. Ethereum is mainly programmed in Solidity, while the most common programming language in EOSIO is C++ as the conversion to Web Assembly is quite well established.

To conclude this section, we present an adapted summary table [10], as we believe it is visually helpful and may be of great interest to compare the main characteristics offered by the blockchain platforms we are discussing.

Feature	Ethereum	EOS
Currency	ETH	EOS
SC Language	Solidity	C++
Bytecode	EVM	Web Assembly
Transactions per Second	30	4,000
Block Production Rate	10-15 seconds	0.5 seconds
Consensus Algorithm	PoW	Delegated PoS
Miner Fee	Gas Fee	Stake Model

Table 1: Ethereum vs. EOS [10]

Finally, after the comparison, we can state that although EOS presents better characteristics of flexibility, scalability, and transaction costs, Ethereum is still ahead with a considerable difference. However, we must consider the disadvantages that Ethereum presents are gaining more and more weight; the network is congested, the increasing carbon footprint caused by PoW and the continuous complaints from developers about the high price of transactions (gas fees).

On the other hand, the numerous advantages EOS presents are why even though there are many competitors to Ethereum, EOS stands out among the rest and is considered by many as the only real threat against Etherum. In addition, Ethereum is finalizing the details for its new upgrade, which aims to solve the problems it suffers. If this upgrade meets the announced expectations, Ethereum may never be dethroned. However, if these improvements do not satisfy the problems afflicting the network, EOS could definitely become the Ethereum killer.

3 MOTIVATION FOR EOS VULNERABILITY STUDIES

Given that cryptocurrency is one of the newest disruptive technologies, the security of transactions involving them is still in the spotlight.

As it has been already stated, Ethereum was the first cryptocurrency to launch smart contract functionalities. Moreover, it is the most popular implementation of the protocol. Therefore, there is a wide range of open-source projects that leverage Ethereum's smart contracts for its operations. This has led to a considerable amount of research regarding the topic, which has enabled the identification of vulnerabilities, hence ensuring to some extent the security of Ethereum smart contracts.

However, it is not the case for EOS. Despite the fact that in recent years EOSIO has become one of the most popular blockchain platforms, there is a lack of in-depth research on its ecosystem, especially on issues related to security and potential vulnerabilities.

Table 2: EOSFuzzer vs. EVulHunter [11]

Vulnerability	Total	EOSFuzzer			EVulHunter		
		Reported	FP	FN	Reported	FP	FN
Block Info. Dependency	82	2	0	1	N/A	N/A	N/A
Forged Transfer Notification	82	4	0	0	12	10	2
Fake EOS Transfer	82	2	1	0	9	8	0

One of the main causes is that there aren't many open-source EOS's smart contract projects for researchers to perform their studies.

Nevertheless, Ethereum has become a victim of its own success. The network is heavily congested, has a large carbon footprint, and developers complain of high Gas fees. By contrast, the EOS system aims to specialize in scalability, speed, and flexibility. This has resulted in EOS being considered "the Ethereum killer".

Ethereum will soon see its Serenity upgrade roll out, which addresses those scalability problems with ETH 2.0. Ethereum 2.0 is a set of updates that will transition Ethereum to a more efficient consensus architecture, PoS, making Ethereum consume at least 99.95% less energy. In the case Ethereum isn't able to solve scalability issues, EOS will outperform Ethereum.

In conclusion, the lack of previous research on EOS vulnerabilities and given the improvement that EOS brings to Ethereum's Achilles heel, scalability, result in the development of this project, which, given the reasons stated could be of great interest in the near future

4 CONTRIBUTIONS

This paper hope to make the following contributions to the community:

(1) First of all, we have conducted an intensive study of the EOS platform, in order to find and understand the possible vulnerabilities
(2) On the other hand, we have performed an in-depth study to find the differences between Ethereum and EOS, focusing on the ones related to Smart Contracts
(3) After deeply studying the EOS blockchain platform, we have focussed on collecting all the existing smart contract vulnerabilities in order to warn future dApp developers about the issues they might have if the code is not perfectly checked, or if some of the practices seen in the vulnerabilities of these papers are carried out
(4) Discussing potential vulnerabilities that are outside the scope of a smart contract's code i.e., social concerns, fairness, etc.

5 RELATED WORK

In order to gather all the different vulnerabilities an EOS smart contract developer might come up with, this paper has been based in different sources. One of them is the *EOSafe: Security Analysis of EOSIO Smart Contracts*. This paper was release during November 2019, and analyzes four basic security issues of these network: Fake EOS, fake Receipt, Rollback and Missing Permission Check. Once discovered which the vulnerabilities were, they propose a tool which automatically checks if any smart contract is vulnerable. For that, the tool is designed to have three different modules: Execution Engine, the EOSIO library emulator and the vulnerability scanner. Besides, this tool accepts as input the Control Flow Graph of the contract and the disassembled Wasm instructions. These two parameters are obtained via the security analysis framework for Wasm modules *Octopus*. Regarding the results they obtained from this security analysis tool, they are very promising as from the 52 smart contracts tested (27 vulnerable and 25 non-vulnerable) all of them except 1 was classified as it had to, reaching a success rate of 98.08% [12].

Additionally, it is worth mentioning there's another two published papers identified and proposed solutions to detecting vulnerabilities in EOSIO smart contracts. Those are: *EOSFuzzer* and *EVulHunter*, both of which comes with a detailed explanation of how a certain vulnerability existed, exploited, proposed fix as well as a tool to detect such vulnerability. However, according to the results and comparisons between *EOSFuzzer* [11] and *EVulHunter* [13], it is obvious that *EOSFuzzer* has an advantage edge than *EVulHunter* in terms of accuracy of detection as well as the ability to detect a broader scope of vulnerability in EOSIO smart contracts. According to [Table 2], it is clear that *EOSFuzzer* performed better than *EVulHunter* in terms of effectively and correctly in identifying vulnerabilities in EOSIO smart contracts. On the other hand, *EOSFuzzer* was successfully detected **Block Info. Dependency** vulnerability, while *EVulHunter* is unable to do so [11]. Therefore, in [Table 2], *EVulHunter* contains *"N/A"* values for the **Block Info. Dependency** vulnerability.

6 EXISTING VULNERABILITIES IN EOS SMART CONTRACT

As it has been mentioned in different parts of this paper, EOS has been constantly developing as a huge blockchain platform with much higher transaction throughput compared with the Ethereum network. Even though it has gained a place in the cryptocurrency market, it has not been researched in a very detailed way. Not a lot of papers have been published regarding the different security issues this network might have. And having all the technological advances mentioned, those advances can also lead to different vulnerabilities which can be found in the released smart contracts. In the following pages some of the most common vulnerabilities have been analyzed, mainly taken from different papers such as EOSFuzzer, EOSafe, etc. Some of the issues explained here are already patched and working correctly but some others are not. The security issues which are going to be analyzed are the following: Numerical overflow, authorization check, fake EOS transfer, forged transfer notification, random number generator and rollback attack.

6.1 Numerical overflow

Numerical overflow on arithmetic operations is considered as one of the most common program errors or vulnerabilities found in the real world. Although this issue has been identified a very long time ago, numerical overflow remains one of the most notorious vulnerabilities in software security.

One of a popular case of numerical overflow is the event of integer overflow in Berkshire Hathaway's common stock, ticker BRK.A. During May 2021, BRK.A was worth $435,120.00 per share. However, the NASDAQ exchange employs 32-bit unsigned integers to record and send quotes for stock prices. At this point, given that BRK.A valued at $435,120.00 per share and stored as 4,351,200,000. Unfortunately, it exceeds 32-bit unsigned integers maximum value. Thus, it caused an integer overflow. Therefore, this integer overflow would in turn go past the maximum value, reset back to 0 and continue to add up the remaining. This event, however, displayed to other market participants that BRK.A was currently worth $5,623.2704 [14]. This vulnerability has caused tremendous losses in both financial as well as losses of confidence in the system.

Despite numerical overflow has been identified since the beginning of modern computing, it remains as one of the most common yet critical vulnerabilities in software security. It is empirical to acknowledge that numerical overflow possesses as one of a critical vulnerability in any smart contract program. Especially, numerical overflow has caused a long-lasting financial damage to one of EOS's dApps on EOSIO blockchain network.

During July 2018, EOSFomo 3D, an online game that resides on EOSIO blockchain network (it is important to note that this game was not only resides on EOSIO but also Ethereum blockchain network in a form of smart contract), it was exploited through the use of integer overflow vulnerability, causing a huge financial loss to the publisher. Given that an attacker has successfully exploited this vulnerability [Figure 3], EOSFomo 3D was forced to shut down its operations after the attack [15].

Figure 3: EOSFomo 3D's Homepage After the Attack [15]

According to *Code Sample 1*, the **amount** variable on line 16 is declared as an unsigned integer type with a width of exactly 64 bits. In other words, a variable **amount** can only store a maximum value of 18,446,744,073,709,551,615; any amount beyond that value will cause an overflow. Even in *Code Sample 1*, there's multiple **assert** statements to ensure the validity and correctness of the transfer procedure, however, none of the assert statements cover a

Code Sample 1 Numerical Overflow Vulnerability

```
1  void batchtransfer(symbol_name symbol,
   ↪  account_name from, account_names to, uint64_t
   ↪  balance)
2  {
3      require_auth(from);
4      account fromaccount;
5
6      require_recipient(from);
7      require_recipient(to.name0);
8      require_recipient(to.name1);
9      require_recipient(to.name2);
10     require_recipient(to.name3);
11
12     eosio_assert(is_balance_within_range(balance),
       ↪  "invalid balance");
13     eosio_assert(balance > 0, "must transfer
       ↪  positive balance");
14
15     //Multiplication overflow
16     uint64_t amount = balance * 4;
17
18     int itr = db_find_i64(_self, symbol, N(table),
       ↪  from);
19     eosio_assert(itr >= 0, "Sub-- wrong name");
20     db_get_i64(itr, &fromaccount, (account));
21     eosio_assert(fromaccount.balance >= amount,
       ↪  "overdrawn balance");
22
23     sub_balance(symbol, from, amount);
24
25     add_balance(symbol, to.name0, balance);
26     add_balance(symbol, to.name1, balance);
27     add_balance(symbol, to.name2, balance);
28     add_balance(symbol, to.name3, balance);
29 }
```

case where the boundary of a variable is left unchecked. Therefore, if a smart contract employing this piece of code is vulnerable to numerical overflow vulnerability. In other words, this vulnerability can cause a huge financial loss to a publisher of a smart contract that is vulnerable to this security issue.

To analyze this code better, assuming we have the following value for analysis:

- balance = 4,611,686,018,427,387,904
- Sender initial balance = 1,000,000
- receiver1, receiver2, receiver3, receiver4 balances = 0

With the above value for **balance**, it is clear that it will cause an overflow as the passed-in value for **balance** argument is far beyond the maximum capacity of an unsigned integer with width of 64 bits. Given that the code above did not account for numerical overflow vulnerability, it would trigger the following to happen:

(1) amount = 0
Due to multiplication overflow

(2) Sender balance will be deducted by 18,446,744,073,709,551,616 even though sender only has 1,000,000 in initial balance This means that the sender balance will be overdrawn, resulted with a balance of -18,446,744,073,708,551,616
(3) receiver1 is credited 4,611,686,018,427,387,904
(4) receiver2 is credited 4,611,686,018,427,387,904
(5) receiver3 is credited 4,611,686,018,427,387,904
(6) receiver4 is credited 4,611,686,018,427,387,904

At this point, it is clear that numerical overflow can cause a tremendous amount of damage to a publisher of a smart contract that is vulnerable to this. Notice on line 21, there's an assert statement to ensure that the sender never overdraws its balance. However, it is not working as expected since after multiplication on line 16, **amount** variable will have a value of 0. Thus, the assert statement on line 21 will still evaluate to true. Thus, the transaction will carry on as normal.

Even though this vulnerability is not new, it seems to be repeated quite often. With that being said, EOSIO strongly encourages publishers to develop their smart contracts in a way that is aligned with security guidelines provided for by EOSIO. In addition, this issue can be resolved with validating the passed-in argument to ensure the boundary safety as well as perform verification/assertions on arithmetic computation results prior to interacting with account balances. On the other hand, this type of vulnerability exploits exposure to areas beyond transaction-related procedure such as game related data values in blockchain games (Health, damage, defense, etc.) [15].

6.2 Authorization Check

As of now, *eosio.cdt* v1.7 currently allows contract developers to have the flexibility to perform authorization verifications through the following methods [16]:

```
1. check(has_auth(user), "User is not
↪  authorized.");
2. require_auth(user);
3. require_auth2(user.value, "active"_n.value);
```

The aforementioned methods for authorization checks are similar if not the same in terms of its purpose and functionality. However, the main difference between *require_auth2* with the first two is that *require_auth2* is designed for a more explicit check for authorization. In short, if the same user uses the transaction with a different permission i.e., code, owner but not active, the code execution will halt at *require_auth2* check without processing the rest of the code. Moreover, it is worth noting that **has_auth** only checks for authorization on the account level. Furthermore, **has_auth** does not halt the execution of a program, which, could potentially expose a smart contract to authorization check bypass vulnerability [17].

In [Code Sample 2], it is a typical example of how an authorization check bypass is exploited in a smart contract. Notice on line 20, a developer of this contract has purposely checked for authorization using a ternary operator with **has_auth**. However, as previously mentioned, *has_auth* only checks for authorization on the account level without halting the execution if condition is not matched. At this point, we observed that even though the developer for this contract has intentionally checked for authorization. However, due to improper usage of authorization check in EOSIO smart

Code Sample 2 Authorization Check Bypass Vulnerability

```
1  void token::transfer( account_name from,
2                        account_name to,
3                        asset        quantity,
4                        string       memo )
5  {
6      eosio_assert( from != to, "cannot transfer to
       ↪  self" );
7      eosio_assert( is_account( to ), "to account
       ↪  does not exist");
8      auto sym = quantity.symbol.name();
9      stats statstable( _self, sym );
10     const auto& st = statstable.get( sym );
11
12     require_recipient( from );
13     require_recipient( to );
14
15     eosio_assert( quantity.is_valid(), "invalid
       ↪  quantity" );
16     eosio_assert( quantity.amount > 0, "must
       ↪  transfer positive quantity" );
17     eosio_assert( quantity.symbol ==
       ↪  st.supply.symbol, "symbol precision
       ↪  mismatch" );
18     eosio_assert( memo.size() <= 256, "memo has
       ↪  more than 256 bytes" );
19
20     auto payer = has_auth( to ) ? to : from;
21
22     sub_balance( from, quantity );
23     add_balance( to, quantity, payer );
24 }
```

contract, line 20 is practically useless since it will not halt regardless of the condition of the inputs it received. It is also important to note that the usage of *has_auth* in this case is incorrect as it checks for authorization of a receiver or to. Additionally, when transferring EOS from one account to another, it only makes sense to check authorization on a sender's account to validate the authorization to send. In other words, line 22 and 23 will be executed regardless. This poses a clear image of authorization check vulnerability as the contract will deduct sender's balance and add balance to receiver without performing real authorization verification. In this case, it can be fixed by simply replacing *has_auth(to)* with the following but not limited to:

```
1. eosio_assert(has_auth(from));
2. require_auth(from);
3. require_auth2(from.value, "active"_n.value);
```

6.3 Fake EOS Transfer – Action forwarding

Blockchain has enabled a wide array of innovations in our daily lives. The same applies for smart contracts, enabling the ability of executing an agreement(s) between two or more parties on a particular transaction. In traditional blockchain networks such as

Bitcoin where smart contract is **not yet enabled** prior to its taproot upgrade, a coin or token can typically only be minted from mining activity. However, given an unprecedented use-cases of smart contracts in Ethereum and/or EOSIO network, it has enabled the ability to create a **token** on an existing blockchain infrastructure (Ethereum, EOSIO, etc.) through the use of smart contracts. Typically, one might assume that the *Fake EOS Transfer* vulnerability revolves around an attacker creates a fake token with the same name as the real one, EOS. However, it is not the case for what has happened for **EOSBet**. Though, it is worth noting that any participant can generate their own token on a smart contract enabled blockchain. Therefore, there could potentially be a form of vulnerability where the exploitation of minted/forged tokens to spoof smart contracts that are not properly equipped with necessary mechanisms to discard transactions involve forgery.

Even though in this section, we have briefly mentioned forged tokens. In the case of *Fake EOS Transfer* exploits in **EOSBet**, an attacker exploits flaws in ABI forwarder to bypass a single checkpoint. Consequently, this has led to an attacker gaining direct access to the calling **transfer** function of the contract without having to go through an intermediary, **eosio.token**.

On September 14th, 2018 at around 3:00 AM UTC, EOSBet was attacked by an attacker with an identifier named *aabbccddeefg*. This security breach has caused EOSBet to temporarily halt their operations upon realizing a theft loss of **44,427.4302** EOS [18]. According to historical price data of EOS/USDT, 1 EOS is worth approximately $2.61 in U.S. Dollar on the day EOSBet's security was breached [19]. Therefore, the losses of theft is around $115,956. Per EOSBet team's analysis and statement of the breach, the affected code is shown in [Code Sample 3] with vulnerability was exploited on **line 10**. The aforementioned line of code is a forwarder of any actions taken on a smart contract to the ABI as well as forwarding actions to an internal functions of a smart contract itself. In other words, due to a flawed implementations on line 10, an attacker was able to call internal function of a smart contract.Further dissecting line 10, it only checks for one of the following three items:

(1) code == self
Check if the code of the contract is from "eosbetdice11"
(2) code == N(eosio.token)
Check if the calling contract is **eosio.token**
(3) action == N(onerror)
Check for an onerror coming from the eosio system contract

Since line 10 only checks for an *onerror* coming from the eosio system contract then forward action, this allows an attacker to directly call the **transfer** action without ease. In layman's terms, this allows an attacker to make a bet for free; if outcome of a bet is a losing bet, an attacker lose nothing. If the outcome of a bet is a winning bet, an attacker receives EOS to their wallet. To counter this vulnerability, there's multiple approaches to prevent attacker from exploiting this vulnerability:

- Adding a logic on line 11 such that if the action is a *transfer* action, it will then proceed to validate such that only *transfer* action from **eosio.token** can be forwarded to the smart contract. Moreover, if the *transfer* action of *eosbetdice11* was called directly on line 11, the code execution should not progress any further.

Code Sample 3 Fake EOS Transfer

```
1  // extend from EOSIO_ABI
2  #define EOSIO_ABI_EX( TYPE, MEMBERS )
3  extern "C" {
4     void apply( uint64_t receiver, uint64_t code,
       ↪  uint64_t action ) {
5        auto self = receiver;
6        if( action == N(onerror)) {
7           /* onerror is only valid if it is for the
             ↪  "eosio" code account and authorized by
             ↪  "eosio"'s "active permission */
8           eosio_assert(code == N(eosio), "onerror
             ↪  action's are only valid from the
             ↪  \"eosio\" system account");
9        }
10       if( code == self || code == N(eosio.token) ||
         ↪  action == N(onerror) ) {
11          TYPE thiscontract( self );
12          switch( action ) {
13             EOSIO_API( TYPE, MEMBERS )
14          }
15          /* does not allow destructor of thiscontract
             ↪  to run: eosio_exit(0); */
16       }
17    }
18 }
19
20 EOSIO_ABI_EX(eosio::charity, (hi)(transfer))
```

- Modify line 10 such that if the received action is **transfer** calling directly on *eosbetdice*; but not from **eosio.token**, the code execution should not progress any further
- Note: N() is a preprocessor macro function of C++ for EOSIO smart contract

Code Sample 4 Fake EOS Token – Patch approaches

```
1. Line 11: if( action == N(transfer)) {
eosio_assert( code == N(eosio.token), "Must transfer
↪  EOS") };
2. Line 10: if( ((code == self  && action !=
↪  N(transfer) ) || (code == N(eosio.token) && action
↪  == N(transfer)) || action == N(onerror)) ) { ... }
```

6.4 Transfer Notification

In this section, we will discuss another type of vulnerability that occurs after a **transfer** action was invoked. On a surface, this vulnerability is fairly simple to understand yet requires a comprehensive understanding of how EOSIO's smart contract is operated in order to exploit it. Prior to the analysis and discussion of a code snippet with this type of vulnerability as well as its proposed solution, it is imperative to understand how this vulnerability was exploited in its simplest form.

Code Sample 5 Transfer Notification – Contract Code [20]

```
void transfer(uint64_t sender, uint64_t receiver) {

  auto transfer_data =
  ↪ unpack_action_data<st_transfer>();

  if (transfer_data.from == _self ||
  ↪ transfer_data.from == N(eosbetcasino)){
    return;
  }

  eosio_assert( transfer_data.quantity.is_valid(),
  ↪ "Invalid asset");
}
```

Code Sample 6 Transfer Notification – Attacker Code [11]

```
class contractB : public eosio::contract {
    public:
        contractB(account_name self) :
        ↪ eosio::contract(self) {}
        void transfer(uint64_t sender, uint64_t
        ↪ receiver) {
            require_recipient(N(eosbetcasino));
        }
};
```

With that being said, this **Transfer Notification** vulnerability exists because during a normal EOS transfer, a receiver contract can choose to forward the transfer notification to other accounts through the **require_recipient** field. This *require_recipient* field can be understood as a **carbon-copy or cc** in e-mail. This type of attack can be understood with the following steps:

(1) Attacker controls **two** accounts, A and B
(2) Attacker initializes the attack by first transferring EOS from A to B through the system contract, *eosio.token*
(3) As soon as the transfer is successful, both account A and B will receive transfer notification
(4) However, the contract deployed within account B can then intentionally forward the transfer notification to another contract. In this case, it's **eosbetcasino**
(5) The intention of doing the above procedures is to mislead **eosbetcasino** that is has successfully received EOS from account A because **eosbetcasino** received a transfer notification that was forwarded from account B
(6) At this point the attack is concluded and **eosbetcasino** credited an attacker's account even though attacker never send any EOS to **eosbetcasino**'s account

In order to prevent this vulnerability from being exploited on smart contracts, it is a fairly straightforward fix requiring only minor addition of code. Notice on line 5 of [Code Sample 5], it did not check and validate the destination of the transfer. In other words, if **transfer_data.to** equates to values other than the contract itself (*_self*). It means that the transfer notification involves other parties, not the smart contract. At this point, the code of this smart contract should not proceed any further to prevent loss of funds. A proposed fix for **Transfer Notification Vulnerability** can be done by simply adding an argument such that if the destination of the transfer is not the current contract, then it should ignore the transfer notification. The fix is presented as below:

Code Sample 7 Transfer Notification – Fix

```
//Line 5 of [Code Sample 5]
if (transfer_data.from == _self || transfer_data.from
↪ == N(eosbetcasino) || transfer_data.to != _self {
    return;
}
```

6.5 Random Number Generator

Moving away from previously introduced vulnerabilities that focus more into the technical-side of software security, in this section, we will introduce another type of vulnerability that relies on an exploit of smart contract's code logic. In fact, we consider this vulnerability is a form of logic-based vulnerability since an attacker exploits a smart contract with this type of vulnerability using its logic against itself i.e., its design purpose.

Given that some if not most smart contracts rely on block information for its logic to generate random numbers. In other words, **true** random numbers cannot be generated on EOS. Therefore, a contract developer has to design their own random number generator. As developers tend to refer to block information to help generate random numbers. It means that at this point, that contract is exposed to **Random Number** attack. This type of attack exists due to block information being publicly available. Therefore, anyone can easily access to block information and its details using any **EOS Block Explorer** on the internet. In other words, an attacker can generate the exact same random numbers that match the random numbers generated from a smart contract. This exists because of the following reasons:

(1) Block information can be obtained easily via any **EOS Block Explorer**
(2) Block time for EOS is **0.5 seconds**. Plenty of time for an attacker to algorithmically obtain block information and generate a number that matches the number generated from a smart contract
(3) Smart contract source code being open-source is a good thing. However, open-source contract also allows an attacker to examine a contract's logic to generate random numbers

It is imperative to stress that for any type of blockchain network, be it Bitcoin, Ethereum or EOS, there always exists multiple block explorers that are easily accessible on the internet. Furthermore, those block explorers are only seconds away due to the existence of Google search.

In [Code Sample 8] is a code snippet of how EOSDice implemented their *random number generator* to determine the outcome of a gambling game. Due to this vulnerability, an attacker was able

Code Sample 8 Random Number Vulnerability

```
1  uint8_t random(account_name name, uint64_t game_id)
2  {
3      auto eos_token = eosio::token(N(eosio.token));
4      asset pool_eos = eos_token.get_balance(_self,
       ↪  symbol_type(S(4, EOS)).name());
5      asset ram_eos =
       ↪  eos_token.get_balance(N(eosio.ram),
       ↪  symbol_type(S(4, EOS)).name());
6      asset betdiceadmin_eos =
       ↪  eos_token.get_balance(N(betdiceadmin),
       ↪  symbol_type(S(4, EOS)).name());
7      asset newdexpocket_eos =
       ↪  eos_token.get_balance(N(newdexpocket),
       ↪  symbol_type(S(4, EOS)).name());
8      asset chintailease_eos =
       ↪  eos_token.get_balance(N(chintailease),
       ↪  symbol_type(S(4, EOS)).name());
9      asset eosbiggame44_eos =
       ↪  eos_token.get_balance(N(eosbiggame44),
       ↪  symbol_type(S(4, EOS)).name());
10     asset total_eos = asset(0, EOS_SYMBOL);
11     //inline_actiontotal_eos
12     total_eos = pool_eos + ram_eos + betdiceadmin_eos
       ↪  + newdexpocket_eos + chintailease_eos +
       ↪  eosbiggame44_eos;
13     auto mixd = tapos_block_prefix() *
       ↪  tapos_block_num() + name + game_id -
       ↪  current_time() + total_eos.amount;
14     const char *mixedChar = reinterpret_cast<const
       ↪  char *>(&mixd);
15
16     checksum256 result;
17     sha256((char *)mixedChar, sizeof(mixedChar),
       ↪  &result);
18
19     uint64_t random_num = *(uint64_t
       ↪  *)(&result.hash[0]) + *(uint64_t
       ↪  *)(&result.hash[8]) + *(uint64_t
       ↪  *)(&result.hash[16]) + *(uint64_t
       ↪  *)(&result.hash[24]);
20     return (uint8_t)(random_num % 100 + 1);
21 }
```

to exploit a flaw in EOSDice's random number generator. Consequently, EOSDice suffered a loss of 2,545 EOS or $13,500 [11]. In [Code Sample 8], line 13, noticed that EOSDice's contract code is using block's information along with other information that is easily obtained by an attacker. At this point, it means that before an attacker makes a bet, an attacker can use those variables to **predict** a winning number or a random number that was generated by EOSDice's smart contract before the game even starts. In other words, this vulnerability allows an attacker to have a win rate of 100% in a gambling game that is never designed to allow a player to have 100% win rate. It is also important to note that even though EOS's block time is very fast, 0.5 seconds. Nevertheless, 0.5 seconds is more than enough for an attacker to exploit this vulnerability by reading block information, then calculating a winning number and making a bet that always has a winning outcome.

Given that this vulnerability exploits a flaw that is fairly non-technical. Therefore, there's only a limited approach to patch this vulnerability that tends to be non-technical compare to other vulnerabilities:

(1) Contract developer should not fully rely on block information or other methods that are predictable to implement their random number generator. However, they can refer to existing, open-source, well-known random number generator published by the EOSIO developer community
(2) Open-source project is great. However, they should not open-source their random number generator since it allows an attacker to study their code to launch a successful attack

To reiterate, to prevent the *Random Number Vulnerability* from occurring, it is imperative that EOSIO's smart contract developer should refer to a carefully studied existing solution that is open-source for their random number generating processes. In fact, we are fortunate enough to be able to locate a public GitHub repository named **FairDiceGame – How to design a blockchain-based provably fair online dice game**; a carefully studied proposal to ensure fairness & security for random number generating processes [21].

6.6 Rollback Attack

EOS Rollback attack can only happen in gambling DApps, as its basis relies in the generation of random numbers in order to check if the jackpot was hit. In this vulnerability two main actions take part: transfer and reveal [12].

On the one hand, in *transfer* the bet that is received as well as the player´s transfer is added and, on the other hand, *reveal*, the developer uses various methods in order to generate a pseudo-random number for comparing it afterwards with the player´s input.

Even if the developer is going to check in a very detailed way which the input parameters of every action (or at least the most sensitive ones) are, if the gambling game has the following procedure or schema the vulnerability can still happen.

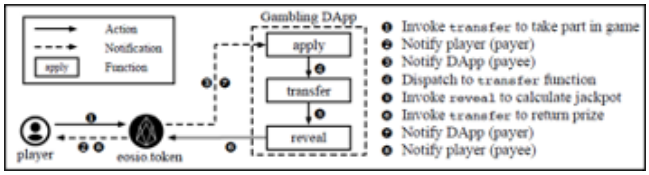

Figure 4: Rollback attack smart contract schema [12]

All the actions are invoked inline, that means, located and loaded from and in a single transaction. In the exact moment the player receives the notification he or she could invoke another inlined action to eosio.token to check his or her balance. In case this balance is lower than what it used to be, it means the player has lost. If he had lost, an assertion statement can be forced in order for the action to fail.

It is well known that the actions in EOS must follow the atomicity; whether all of them are correctly executed or if not they all fail. In this case, after the assertion, the action would fail and all the process would be revoked, so the player can again bet for another number and follow this whole process until he/she hits the jackpot.

In order to detect this kind of vulnerability, the researcher's group who developed the EOSafe vulnerability detector tool, they first filter the transactions that contain at least four actions. In order an action to be suspicious, they have to meet certain conditions [12]:

- First and last action must be invoked among the same smart contract. That last action will be the one determining if rollback is necessary.
- The two actions in the middle must be token transfers through eosio.token and the sender and receiver are arranged opposite to each other.
- One of the counter parties taking part in the execution must be the gambling DApp.
- The number of tokens transferred from the vulnerable smart contract is more than it received.

As this vulnerability can only appear in gambling DApps, from the 53,666 different smart contracts as of November 15th 2019, it can only appear in 17,395 from which 1,149 were vulnerable. That means 2.14% of the candidates. In addition, as money is involved in this vulnerability, the application developers use to patch it first one, and that is why the average time in order to fix it is 4.24 days [12].

This vulnerability could also be related with the *Random Generator Vulnerability*, because if an attacker can know which the randomly generated number is, and can also check if the balance of his/her account has decreased, and if so, revoke any action, in a gambling dApp is a win win situation. Either the developer guesses the random number and copies it for its input, or after checking the balance of its account after the action is able to revoke it.

7 SAFETY ISSUES IN EOS SMART CONTRACT

Throughout our journey to research existing vulnerabilities in EOSIO smart contracts, most if not all of the vulnerabilities have an outcome that resulted in a financial loss to an owner/operator of a smart contract. Moreover, vulnerabilities that were previously discussed in this paper are often times involved technical details in terms of how a particular type of vulnerability is found, exploited, studied and proposed patch/fix.

With that being said, there are other safety issues/concerns that are currently existed in EOSIO smart contracts. Therefore, it is imperative for us to develop a discussion on this matter since these safety issues/concerns tend to be overlook. Moreover, these issues have not only resulted in financial loss of a smart contract operator but also imposed an unfair/negative atmosphere for the entire blockchain community as well as the society.

It has come to our attention that most if not all DApps reside on EOSIO blockchain network are catering gambling services to its clients. Due to the nature of blockchain network, participants in a network remains anonymous. Meaning, one way to identify an identity of a wallet is to hope that an anonymous wallet transfer their funds to an exchange that requires *KYC*. Given that an anonymous wallet has transferred funds into an exchange that requires *KYC*, it means could link a wallet to an identity of a particular person. Though, it will not be a definitive answer to the question "Wallet 'A' is John's wallet". Therefore, this has nurtured an environment for criminals to safely transact their illegal activities over the internet.

Continuing with regards to laws and regulations. Most of us understand that gambling businesses operated in *Las Vegas, Nevada* are regulated by a regulatory agencies of the government. Several notable key identifier for a legal gambling businesses are: Can only be serviceable to an individual with age over 21+ years old and slot machine or other gambling games must be configured so that the probability of winning must be fair for the player while remains profitable for business operator. In other words, regulations enable gambling business operator the ability to bar a certain cheating behavior such as card counting. Same applies for gambler, regulations allow gambler to play a gambling game at a regulated venue knowing that the game is fair and not fixed. At this point, since gambling services operated on EOSIO blockchain (Or any other blockchain that supports smart contract) through the use of smart contracts are not regulated. This means that a gambling service can be operated without being enforced regulations by a regulatory agency. In other words, a smart contract operator could develop a gambling game that is closed-source, fixed their gambling game by tweaking their game such that the statistics of gamblers end up winning is near non-existence. Therefore, created an unfair game for gamblers.

Another form of unfair practice that could potentially occurr in EOSIO smart contract platform is involved with the auction activity. Typically, an auction requires a participant to put down their funds to secure their bid. It is important to note that once a participant places their bid, their bid remains un-retractable unless other participant provide a bid higher than current. If their bid ends up a winning bid, that participant wins that particular auction session. However, in smart contract enabled network such as EOSIO, there's a vulnerability that allows bidder to retract their bid at any given time. Therefore, this will create a problem of fairness since a highest bidder is expected to pay for their bid if they end up winning. Moreover, this vulnerability does not only create a fairness issue in auction bidding on smart contract but also deter future participants from participate due to this concern.

8 CONCLUSIONS

Throughout this paper, we have presented a wide array of vulnerabilities in EOSIO smart contracts. Furthermore, we aim to investigate and discuss topics that have not been previously introduced, Other Safety Issues in EOS Smart Contract. Before discussing each of the vulnerabilities, we deemed it is necessary to perform a deep study and analysis of both EOSIO and Ethereum platforms. In other words, we believe this is crucial since we need to better our understanding of how each vulnerability can be a threat to smart contracts and their decentralized applications. It has come to our attention that most, if not all, of the publications regarding the security of blockchain or smart contract, tends to focus heavily toward a well-known platform, Ethereum. However, we believe that it is critical to study and relate vulnerabilities that existed in Ethereum to EOS to further the security of blockchain and smart

contract technologies as a whole. With that being said, we would like to present this paper as an opportunity to spread awareness to the EOSIO developers and any other type of blockchain community that support smart contract. In addition, we would also like to stress the importance of developers being aware of other safety issues that could occur in EOS smart contracts, such as fairness in auctions, gambling, and social safety concern, i.e., under-age gamblers, gambling under the influence, etc.

9 TEAM MEMBER CONTRIBUTION

First of all, we wholeheartedly believe we have worked fantastically together, complementing each other's part, helping the other team members when needed to obtain a considerable interesting paper as a result.

9.1 Lan Nguyen

Summary of tasks performed/responsibilities:

(1) Study current vulnerabilities of EOSIO smart contract and the way current research handle them
(2) Experimenting, re-create/simulate and validate *Numerical Overflow* vulnerability in C++
(3) Study EOSIO's vulnerabilities and its discussions outside of research papers. Several sample links to those discussions are:
    - https://mp.weixin.qq.com/s/WyZ4j3O68qfN5IOvjx3MOg Had to use Google Translate
    - https://www.reddit.com/r/eos/comments/9fxyd4/eosbet_transfer_hack_statement/
    - https://github.com/jc1991x/bet-death-causes
    - https://github.com/DeBankDeFi/fairdicegame
(4) Prepare, organize and write to document on Overleaf for the team
(5) Created a testnet account and attempted to interact with smart contract on EOSIO Testnet (https://testnet.eos.io/deployment)
    Also attempted to use/run the following software for study on EOSIO Smart contracts: Scatter and EOSStudio
(6) Attempted to re-produce EOSFuzzer's results but was unable to do so due to version incompatibility
(7) Studied and presented other safety issues in EOS smart contract

9.2 Maria Milagrosa Vela Melendez

Maria was assigned the task of studying Ethereum. As explained, Ethereum was the first blockchain protocol to introduce smart contracts. And therefore, it is taken as a reference for its competitors, including EOS. That is why we consider that it could be of great interest to study Ethereum in depth to obtain data that would allow us to see what differences exist between EOS and the industry leader. To accomplish her task, Maria also had to analyze the EOS platform. In this part Mikel went deeper, but Maria had to work here as well to be able to carry out the subsequent comparison. Based on the data obtained in this part, some of the vulnerabilities they have explained could be reasoned out. The main elements analyzed in depth in both platforms are those that have been exposed in the paper: virtual machines, accounts, resources, consensus protocols, among others. Finally, after the study of both platforms, Maria made a comparison between Ethereum and EOS, in order to obtain the similarities and, more relevantly, the differences that separate them, and to obtain, from them, some conclusions.

9.3 Mikel Santana Maraña

Mikel´s main goal was to study in a deep way the EOS Blockchain infrastructure. What does that mean? Well, looking for how accounts are managed in EOS, as it is an important part of the permission hierarchy, researching about the new and modern DPoS consensus protocol in order to know how blocks are added onto the final chain. Besides, how the transactions are formed and their lifecycle once they are executed is also important, to see how they are created from different actions, how each action needs different signs from different users in order to be validated by the 21 block producers. Not only that, a deeper look into the EOS virtual machine based on WebAssembly bytecode has also been an important task, where three daemon nodes can be distinguish: Nodeos, Keosd and Cleos. Finally, and to end with the EOS functionalities, how smart contracts work and how they are executed has also been researched. In addition, regarding the vulnerabilities, as Mikel read the EOSafe paper, he has also contributed with the vulnerabilities appearing there: fake EOS transfer, authorization check and rollback attack. Even if this part has been mainly developed by Lan, he also added some of the vulnerabilities appearing there.

REFERENCES

[1] Ethereum Foundation. Intro to ethereum.
[2] Vaibhav Saini. Getting deep into evm: How ethereum works backstage.
[3] Ethereum Foundation. Introduction to smart contracts.
[4] eosio developers. Consensus protocol.
[5] eosio developers. Eos virtual machine: A high-performance blockchain webassembly interpreter.
[6] peosdev. Limited stack size for contracts.
[7] Chris Hager. Creating go bindings for ethereum smart contracts.
[8] eosio developers. Accounts and permissions.
[9] eosio developers. Transactions protocol.
[10] BlockHunters. Eos vs ethereum - comparison of smart contract platforms.
[11] W. K. Chan Yuhe Huang, Bo Jiang. Eosfuzzer: Fuzzing eosio smart contracts for vulnerability detection.
[12] Haoyu Wang Lei Wu Xiapu Luo Yao Guo Ting Yu Xuxian Jiang Ningyu He, Ruiyi Zhang. Eosafe: Security analysis of eosio smart contracts.
[13] Haoyu Wang Lijin Quan, Lei Wu. Evulhunter: Detecting fake transfer vulnerabilities for eosio's smart contracts at webassembly-level.
[14] Vladimir Kushnir. Safe c++: How to avoid common mistakes.
[15] Wei Cai Tian Min. A security case study for blockchain games.
[16] EOSIO Developers. How to perform authorization checks.
[17] MeiTu Tech. Vulnerability detailed explanation | malicious eos contract has the security risk of swallowing user ram.
[18] The EOSBet Team. Eosbet transfer hack statement.
[19] TradingView. Eos historical price data.
[20] EOSBetCasino. Eosbetcasino vulnerable contract code.
[21] onigirisan Erjan Kalybek DeBankDeFi, Tang Hongbo. How to design a blockchain-based game provably fair online dice game.