Delft University of Technology, Bachelor Seminar of Computer Science and Engineering

An evaluation of the reentrancy vulnerability on GoQuorum-based smart contracts

Sara Op den Orth [1] , **Prof.Dr. Kaitai Liang**[1] , **Huanhuan Chen (PhD)**[1]
[1]TU Delft

Abstract

Within the context of the Ethereum blockchain protocol, reentrancy is a well-known and well-researched smart contract vulnerability. However, when considering GoQuorum, an Ethereum soft fork, barely any research discussing smart contract vulnerabilities exists. This report aims to partly fill this research gap by evaluating the reentrancy smart contract vulnerability in the context of a GoQuorum network. First, the reentrancy attack was demonstrated and its attack features evaluated. Then any known countermeasures were collected. Moreover, it was proposed that some GoQuorum features may also be used as mitigation techniques. Finally, each countermeasure was assessed and categorized. Of all the methods, the checks-effects-interactions pattern is the most direct way to deal with the reentrancy vulnerability. To maximize contract security, however, it is advised to use a combination of the specified prevention and mitigation techniques.

1 Introduction

In 2008, the Bitcoin white paper [1] was published. The paper described how to use *blockchain* technology to trade a kind of digital currency called *cryptocurrencies* without the need for a trusted third party or central server. The implemented Bitcoin protocol, as of the time of writing (June 2021), has a total value of almost 1 trillion dollars [2]. Three characteristics make blockchain such an attractive technology to use for exchanging cryptocurrencies: decentralization, pseudo-anonymity, and transparency.

The Bitcoin framework initially focused primarily on using blockchain technology to exchange cryptocurrencies. Although this was indeed an exciting use of the technology, adopting the technology to execute programs in a decentralized manner would open up even more opportunities. Nick Szabo was the first to propose this concept in 1997 [3] and coined these pieces of code: *smart contracts*.

The Ethereum protocol [4] was constructed as a blockchain protocol that would integrate smart contracts into its framework from the start. Thus Ethereum opened up a plethora of new possible applications for blockchain technology.

To keep its networks secure, public frameworks like Bitcoin and Ethereum leverage blockchain technology's inherent decentralization and transparency characteristics. However, this setup is not suitable for all blockchain applications. One may find the scalability and flexibility of blockchain attractive, but privacy and security may also be essential. Ethereum *soft forks* are built on top of the original protocol, solving such security and privacy concerns in a variety of ways. These types of forks are still backward-compatible with the standard Ethereum protocol.

The Quorum protocol layer [5] is such an Ethereum fork. Besides reliability and flexibility, Quorum focuses on privacy and security. ConsenSys [6], an Ethereum software company, maintains two Quorum-based Ethereum clients. One of which is GoQuorum [7], a soft fork from the Go-based Ethereum client Geth [8].

Some key differences between GoQuorum and a standard Ethereum protocol client like Geth are discussed next. GoQuorum is a permissioned network instead of permissionless, meaning only network nodes granted permission can access the network. Moreover, GoQuorum uses different consensus algorithms and supports private as well as public transactions. More information about these features can be found in section 2. The above characteristics combined make GoQuorum ideal for corporations that value privacy and access control as well as reliability and decentralization. GoQuorum is already used in various applications, from managing massive agricultural trade operations [9], to optimizing commodity trade finance [10].

So as shown above, GoQuorum is in some ways different from the Ethereum protocol. Smart contracts, however, are still essentially the same. Smart contracts are a type of Ethereum account, deployed via a transaction to the network, and every full node then runs the smart contract code. User accounts can call public functions of the deployed smart contracts via the contract's address. It is important to note that, when triggered by a user account, smart contracts can initiate transactions and call functions of other deployed contracts.

Smart contracts can thus be used to automate behavior, both productive and exploitative. On the one hand, this feature makes smart contracts useful for a variety of industries such as health care, logistics, and telecommunications [11]–[13]. On the other hand, it makes them a convenient tool for malicious actors to automate vulnerability exploitation for profit.

Since the inception of the Ethereum network, there have been countless attacks using vulnerabilities in smart contracts to steal ether. One of the most famous smart contract attacks is 'the' DAO attack, where the attacker collected over 3.5 million ether [14]. This attack was executed in 2016; since then, an increasing list of applications have been created using smart contracts. More people use the technology; thus, more money is involved, making smart contracts an ever more attractive target for attacks. Moreover, as the DAO attack showed, smart contract attacks can lead to enormous financial losses.

The DAO attack used a type of vulnerability called reentrancy; the same vulnerability evaluated in this report. Reentrancy exploits external contract calls combined with incorrect contract state updates to extract ether from the victim. A more detailed explanation of the attack can be found in section 4.

Related Research

Besides the aforementioned DAO attack, there have been many more large- and small-scale attacks, making smart contracts' security and privacy vulnerabilities an interesting research area. In this section the existing smart contract vulnerability studies are discussed.

The collected papers have been divided into three categories. First, the largely Ethereum-based smart contract vulnerability surveys are discussed, followed by studies focused on analysis tools, and finally, the papers relating directly to GoQuorum.

The paper by Atzei, Bartoletti, and Cimoli [15] is one of the most cited in the smart contract vulnerability research area, which is understandable as it contains concise and clear explanations of Ethereum-based smart contract vulnerabilities. However, this paper was published in 2017. Since Ethereum is under constant revision, it is essential to study more recent surveys as well. Software updates will have patched some vulnerabilities, changed the needed attack strategy of other vulnerabilities, or even introduced new ones.

The following four papers also analyze smart contract vulnerabilities, and these are more recently published. In the first of these, Chen et al. [16] analyzed and classified 40 known vulnerabilities, indicating which ones have already been eliminated by a patch in the software and what methods can be employed to either prevent or mitigate any associated risks. The paper by Khan and Namin [17], is the most recent. Moreover, [17] contains the most actual code, demonstrating how a vulnerability is implemented in a smart contract and how to exploit them. As the current study focuses on implementation, [17] was a helpful resource. He et al. [18] analyzes some vulnerabilities, but it does not focus much on the countermeasures. The paper by Wang et al. [19] contains a complete taxonomy of the vulnerability research between 2015 and 2019. The paper lists not only problems but also corresponding solutions.

All of the above papers give good overviews; however, there is not much detail per type of vulnerability. A paragraph is spent on each vulnerability, leaving only enough room to list one or two countermeasures without showing how to implement them.

Similar to the above presented vulnerability surveys, most analysis tools also focus on a wider variety of vulnerabilities. The paper by Mense and Flatscher [20] is an interesting survey comparing such tools.

Other papers present analysis tools that focus on detecting a specific vulnerability. These are also the only papers collected that focus on one specific attack vector, instead of giving an overview of many. Sereum [21], and RA [22] are two especially interesting tools as they are focused on detecting the interest of this paper: reentrancy. Both Sereum and RA are further discussed in section 4. The downside of these types of papers is that they often neglect their explanation of the countermeasures as the new tool will already solve the problem.

All of the above papers consider the Ethereum framework; only [18] mentions GoQuorum. Some smart contract surveys discuss different soft forks, but GoQuorum is never prominently featured in [11], [13], [18], [23]. Only the master thesis by Lagarde [24] mentions GoQuorum in detail. However, Lagarde focuses on authentication and authorization, not smart contract security.

Motivation

As presented above, many papers consider the security or privacy of smart contracts. However, three shortcomings have been identified. In surveys, there tends to be a lack of depth per vulnerability. More insight into each vulnerability is given in papers that present analysis tools, but these lack exhaustive evaluation of possible countermeasures. Finally, almost no research concerns the GoQuorum framework specifically.

Besides these gaps in the literature, there are also very few sources available with respect to the deployment and smart contract interaction of GoQuorum frameworks. Even blogs and forums, usually an essential part of implementation research in computer science, are rarely useful with regards to GoQuorum.

Due to time constraints, only one vulnerability could be addressed in depth. It was decided to evaluate the reentrancy vulnerability for this study. Reentrancy is well-known and well-researched in the Ethereum context. So this paper was constructed to do the same for the GoQuorum framework.

Contribution

This paper aims to fill the above-presented gap in research by providing a thorough analysis of the reentrancy smart contract vulnerability and its corresponding countermeasures. Besides listing countermeasures known to work on the Ethereum network, it is also considered whether any GoQuorum-specific features may be used to protect against reentrancy. For each countermeasure, it is indicated which part of the vulnerability the countermeasure addresses and whether the technique is recommended or not. Furthermore, this is one of the first papers providing information on GoQuorum, presenting information on its privacy features, reentrancy resiliency, contract deployment and contract interaction details. The following Research Questions (RQ) have been constructed to guide the study's goals.

RQ: How is the reentrancy vulnerability exploited in GoQuorum-based smart contracts, and what can be done to prevent or mitigate the associated risks?

1. How do GoQuorum and Ethereum smart contracts compare?

2. How can one execute an attack using the reentrancy vulnerability on a GoQuorum network?
3. What are the features of the attack?
4. How can the attack be prevented or mitigated?
5. How do the countermeasures compare, and which should or should not be used?

Structure

The report is constructed as follows. Section 2 discusses the background information needed to understand the rest of the text. Next, section 3 describes the paper's methodology. Then the reentrancy vulnerability and its features are explored, followed by a deep dive into the different available countermeasures in section 4. The section ends by categorizing and comparing the different countermeasures. Section 5 deals with the study's ethical concerns and its reproducibility. The studies limitations and suggestions for future work are discussed in section 6. Finally, section 7 concludes the report, discusses possible improvements, and suggests topics for future work.

2 Background

This section will explain most theory related to GoQuorum needed to understand this study. Furthermore, this section highlights the differences between Ethereum and GoQuorum. As the network side of blockchain technology is not the focus of this research, it is not discussed extensively here. Instead, there are references included whenever a non-essential term is mentioned. Some of the topics discussed are specific to GoQuorum, and others are the same in both GoQuorum and Ethereum. Differences between the two platforms are highlighted. An important difference that is not necessarily relevant to the current study are the different consensus algorithms used in Ethereum and GoQuorum, more information about this can be found in Appendix A.

Section 2.1 explains what a blockchain is. The following section, 2.2, discusses what the Ethereum Virtual Machine (EVM) is and what role *gas* plays in Ethereum and GoQuorum. Section 2.3, explains the different types of Ethereum accounts. And section 2.4 describes the important, Solidity specific, fallback function. Finally, in section 2.5 the private side of GoQuorum is discussed.

2.1 Blockchain

Both Ethereum and GoQuorum are blockchain networks. The blockchain is a kind of distributed ledger; a database shared between multiple connected nodes. Blockchains implement such a distributed ledger using chains of ordered blocks, each containing several transactions. If a block is added to the ledger, all *full nodes* [25] update independently. Full nodes keep track of the complete blockchain, and *light nodes* [25] keep track of part of it. Together, the nodes keeps track of the network's state.

2.2 EVM

The following quote from Ethereum documentation best encapsulates what the EVM is: "At any given block in the chain, Ethereum has one and only one 'canonical' state, and the EVM is what defines the rules for computing a new valid state from block to block." [26]

Transactions, and smart contracts triggered by transactions, can change the state of the blockchain according to the rules set out by the EVM. However, each node should still agree on the state of the network. So every node running the smart contract has to produce the same result: contracts should be deterministic. The EVM was developed to achieve this objective.

Smart contracts are usually written in a high-level language, Solidity [27] in the case of GoQuorum, which is a Turing-complete language. Solidity compiles into EVM bytecode, and each full node runs this bytecode in their EVM.

Running code costs the node owner resources such as electricity. In Ethereum, this "computational effort" is measured in units of *gas*, each bytecode operation is associated with a specific amount of gas. Thus, pure transactions cost less gas than deploying smart contracts, as the latter compiles to many more bytecode operations. Because read operations do not change the state, they do not have to be added to the blockchain and thus run by every node. As a result, these operations are cheaper, and correspondingly, operations that change the state of the blockchain are more expensive.

When sending a transaction, the sender sets an amount of gas the transaction can maximally consume, the gas limit, and sets a price per unit of gas: "$totalTransactionFee = \#gasUnits * gasPricePerUnit$" [28]. The sender has to be able to pay the total transaction fee. If any gas is left at the end of the transaction, it is sent back to the sender, and if not enough gas is provided to 'pay' for the computational effort needed, the entire transaction is reverted, and all ether spend on the gas used thus far is send to miner. The node that adds the transactions to the blockchain, the *miner* [29], gets to keep the gas transaction fee for his/her efforts.

In Ethereum, the gas price has two uses: to motivate miners to run transactions, as they receive the gas fees; and to discourage network spamming. In GoQuorum, however, the gas price is always set to zero. This gas price is feasible because GoQuorum uses different consensus algorithms (so miner motivation is not necessary), and it is permissioned (network spamming is less likely). Although transactions do not cost ether in GoQuorum, transactions still need gas to be executed.

2.3 Ethereum Accounts

There are two types of Ethereum accounts: smart contracts and Externally Owned Accounts (EOAs). EOAs are user accounts controlled by a private key. As mentioned previously, Ethereum accounts can send and receive cryptocurrencies. EOAs can also call functions of deployed contract accounts.

There are several differences between EOAs and smart contracts. The most important one for this study is as follows: where EOAs are limited to the functionalities mentioned above, smart contracts run any *Turing-complete* [30] pieces of code [31]. This versatility makes smart contracts such an asset.

Smart contracts are deployed, by a node, via a transaction and are immutable once deployed. Contract immutability has two caveats: contracts can have a self-destruct option, and it is possible to make smart contracts virtually upgradeable. The latter will be discussed in section 2.4.

A special type of smart contract is the token contract. These token contracts contain a ledger where, per account, there is a note of how many tokens that account owns. There is no limit on what these tokens may represent. The Ethereum community has created several token contract standards, referred to as either EIPs or ERCs, to systemize token interaction.

2.4 Solidity: Fallback Function

Overall the Solidity language is quite similar to JavaScript and Python; however, it does have some peculiarities. For this study, the most important of these traits is the `fallback`[1] function (see line 8 Listing 2). This function has no arguments, no return value, and visibility has to be set to `external`, meaning only functions outside of the contract can trigger the function. Only one fallback function may be defined per smart contract.

Since version 0.6.x of Solidity a similar function called `receive` was introduced [32], which is called if two conditions apply: it is defined, and only ether, no data, is sent. In case of reentrancy, either function would suffice. The fallback function is used in this study as it is compatible with all versions of Solidity and thus more prevalent.

A call to a smart contract invokes the fallback function, assuming that it is defined in one of two situations: (i) the name of the function called is not defined in the contract; (ii) the call's `msg.data` parameter is empty. In the case of (ii), the fallback function has to be marked as `payable`, indicating it may receive ether. Otherwise, an exception will be thrown, and the transaction is reverted. The fallback function serves as a catch-all for any no-data transactions directed at the contract, which enables convenient functionalities such as proxy contracts [33]. However, the fallback function also facilitates reentrancy attacks. Especially for programmers new to Solidity programming, the fallback function introduces possibly unexpected behavior, which is why it is critical to be aware of its existence.

2.5 Private Contracts

Besides sending regular public transactions as in Ethereum, GoQuorum adds the option of sending private transactions. These transaction's payload is encrypted before send-off, keeping all contained data hidden from the other nodes on the network. The only difference between deploying a private compared to a public contract is that the `privateFor` parameter has to be set to a list of public keys of participating nodes. Only the nodes specified will be able to decode the message.

The next section introduces the notion of a private state, a concept necessary to enable private transactions. The section then elaborates on the effects this new state has on private contracts.

Private State

To enable private transactions, GoQuorum introduces a private state. The public state is contained in *Merkle Patricia Trie* [34], equivalently as in Ethereum. This state trie should be identical on all nodes, as this is how a blockchain can be decentralized. By adding private transactions, it is now possible for states between nodes to diverge. For example, if node 1 is a participant in a private contract while node 2 is not, their states will be different as they do not have access to the same information.

To remedy this state divergence, GoQuorum introduces the second state trie, the private state trie. Private state tries keep track of all private node interactions and are purposefully divergent on different nodes, enabling both public and private transactions on the same blockchain network. This state separation has several consequences for the contract interactions between public and private contracts, which will be discussed next.

a) Interaction between contract types. The first consequence of the addition of a private state is that there are restrictions on private-to-public and public-to-private function calls. Private contracts cannot change values in public contracts as this would introduce inconsistencies in the public state tries of the nodes. This restriction is necessary for the same reason the private state has to be separated: the public state should not diverge on different nodes. Thus, private contracts can read from public contracts, but they cannot change the contract's state. A similar division has been made for privacy-enhanced private contracts, introduced in the next paragraph. Each type of contract can only interact without restrictions with another of the same type. Between different types, only read operations are allowed.

b) Private contracts and ether. Another quirk of the separated states is that private contracts cannot send or receive ether. It is easiest to see why this is the case by following an example scenario. Say node 1 has 5 ether, it sends 3 ether via a private transaction to node 2.This transaction is recorded only in the private state tries of node 1 and 2. All other nodes will assume 1 still has 5 ether, as they were not privy to the transaction. Node 1 would thus be able to spend ether privately while publicly it still seems to have the original ether balance. So to avoid this, private contracts can only transact tokens, not ether.

Whereas the amount of ether each node owns directly connects to the entire blockchain state, the token contract itself contains the number of tokens each node owns. As contract interaction between different types of smart contracts is already resolved by limiting interaction, as described previously, state divergence is not a problem for token contracts either. The token contract would have to be private for a private contract to interact with it.

c) Privacy enhancements. Another important factor when considering GoQuorum private contracts is the following two vulnerabilities. Each will be referenced by its implemented GoQuorum solution: counter Party Protection (PP) and Private State Validation (PSV) [35]. Both enhancements are explained using an example of two nodes, 1 and 2, that are both participants in the same private contract. The examples are both adaptations from [35].

PP ensures that only nodes participating in the private contract can change the state of that contract. If a third node 3 uncovers the address of the private con-

[1] A monospaced font will be used when referencing a term used when coding. However, when referencing the 'fallback function' concept and no code is directly referenced, the normal fond will be used.

tract between 1 and 2 it possible to send a transaction to the contract with "`privateFor: [<publicKeyNode1>, <publicKeyNode2>]`". Nodes 1 and 2's state will change, while the sender, node 3, is not even a participant of the private contract. As a result, although nodes 1 and 2 are both participants in the same contract, their states will no longer match. When using PP, all contract participants know the complete list of participating nodes. Any transaction that is sent to only a subset of this list is rejected.

The second privacy enhancement, PSV, addresses the possibility of private state divergence. As nodes 1 and 2 are both private contract participants, they should have the same private state regarding this contract. Without using PP transactions, however, it is possible for node 1 to execute a transaction with `privateFor` set to `[]`, an empty list. Node 2 will not be notified of the transaction as its key is not included, but for node 1, the state will change. PSV ensures that every transaction to a private contract is sent to all of that contract's participants.

Evidently, both PP and PSV solve privacy concerns. However, using the privacy enhancements is discouraged by the GoQuorum documentation [35].

The reason for this discouragement is as follows: to be able to implement PP and PSV, the contract deployment has to be fully simulated to determine which contracts are affected by the transaction. This simulation may be a problem for complex contracts as the values used for it may have changed so much by the time the simulation has finished that the transaction has to be reverted. Due to this complication, GoQuorum advises only using PP and PSV enabled transactions when the privacy concerns outweigh the drawbacks.

3 Methodology

To answer the research questions a variety of methods were used, these will be explaind in this section. This section consists of three headings: Background Research (3.1), Exploring Reentrancy (3.2) and Instantiation (3.3).

3.1 Background Research

The objective in this first phase of research was to understand smart contracts, their life cycle, how to deploy one, and especially how to interact with them.

The research started with the Ethereum documentation, as GoQuorum is in many ways identical to Ethereum. As Ethereum is such a widely used protocol, the documentation is extensive, and well-written [36]. Besides the official documentation, the online Ethereum community also provides many blogs and forums with additional information. Some specific topics researched in this part of the process are: Solidity [37], Geth [38] and Web3.js [39]. In order, these topics concern: the smart contract programming language; the Ethereum client implementation GoQuorum builds on top of; and JavaScript libraries that facilitate interaction with nodes.

With a better picture of the inner workings of the Ethereum network, the GoQuorum documentation was easier to understand. It became apparent quickly, however, that GoQuorum was not discussed substantially in blogs or forums, which made setting up and interacting with the network more complex. Aside from the official documentation, the GoQuorum Slack channel [40] proved to be the only valuable GoQuorum specific resource.

3.2 Exploring Reentrancy

Six concepts were identified to be combined into different search queries: smart contract, (Go)Quorum, Ethereum, security, reentrancy and vulnerabilities. Synonyms were added using `OR` search-operators. The queries were used in five search engines and databases: Google Scholar, WorldCat, Scopus, IEEE Xplore, DBLP. However, as explained in the introduction, there are few papers even mentioning either Quorum or GoQuorum. Instead, the focus was on finding papers surveying vulnerabilities in Ethereum-based smart contracts.

Non-scholarly literature was found using the same queries, combined with the Google search engine. Since blogs and forums will contain the most recent developments in the field, these sources can help when implementing a vulnerability from an old paper in the most recent software versions. As non-scientific sources, blogs and forums can also explain vulnerabilities in a more accessible manner.

Reproducing the scenarios described in the papers was still not straightforward due to the scarcity of resources explaining how to deploy and interact with smart contracts in the GoQuorum network. The following section explains that part of the process further.

3.3 Instantiation

Arguably the most significant part of this research process was to test the identified attacks and countermeausures on an actual GoQuorum *testnet*. This process can be divided into five steps:

1. First, a smart contract `Victim` was constructed demonstrating the chosen vulnerability.
2. Then another contract `Attack` was designed exploiting the vulnerability in contract `Victim`.
3. Next, the contract functions needed for contract set up were called.
4. The attack was executed.
5. Then, any countermeasures were collected, and, where relevant, an implementation was tested.
6. And finally, the identified countermeasures were compared.

Steps 3 through 5 were executed in three different test environments; each subsequent configuration increased the number of variables that could cause an issue. The first step was to try and replicate the attack patterns described in the research papers, as these were all Ethereum based, the Remix Ethereum IDE [41] and its Ethereum testnet were used. By first using the above-described setup, the only variable was whether the attack was implemented and executed correctly, not whether the attack works on a GoQuorum network. Then, the attack was tested in Remix while using the GoQuorum plugin. The plugin enables Remix to connect to locally running GoQuorum nodes. This step ensured the attack worked on the locally set up GoQuorum network. However, it removed the possibility of anything related to the contract interactions not working because Remix took care of these elements. Finally, the attack was tested on a local GoQuorum testnet. These extra steps were incorporated to make the most use of the superior logging of the Remix IDE. However, because this is the first paper studying GoQuorum extensively, it was important to interact with the network directly, not via an IDE. Thus Remix was only used for the preliminary tests.

Three of the, by ConsenSys provided, setup techniques were tested for this project: the 7Nodes example code [42], the GoQuorum Wizard [43], and the developer quickstart [44]. All projects were set up using Docker [45] containers to improve the project's reproducibility. The 7Nodes example is meant to familiarize the user with the basics of GoQuorum. The codebase configurations can not be easily modified, so this setup was not ideal for testing purposes. The GoQuorum wizard and development quickstart resulted in quite similar setups. Between these two, the developers on the Slack channel [40] recommended using the development quickstart, so this was used. Further detail about the network setup can be found in the Gitlab repository [46].

The most difficult part of this project was to connect and interact with the deployed GoQuorum network. Although it was possible to connect to any running GoQuorum nodes using the Geth `attach` command, the Web3.js version used on the Geth console was old: version 0.20.7, while the newest version is 1.3.4 (as of June, 2021). So instead, any node interaction was executed via a JavaScript file that could use the newest version of Web3.js. The code snippets presented in this report were highlighted using [47].

Once deployed, the quickstart wizard provided several tools to help with debugging and network interaction. Cakeshop [48] is another Consensys project. It can be used to inspect and interact with the network. However, the tool did not have extensive documentation, so it was difficult to integrate into the workflow. The provided Quorum Reporting tool [49] had similar problems. In the end, neither tool proved to have a significant advantage over using Web3.js scripts. The exact code can be found in the project repository [46].

4 Reentrancy: Characteristics and Resolutions

Next, the reentrancy attack is thoroughly evaluated. Section 4.1 defines the reentrancy vulnerability. Section 4.2 the reentrancy attack is demonstrated and its features evaluated. Then the prevention and mitigation techniques are listed in section 4.3, followed by a comparison of the different methods in section 4.4. Finally, section 4.5, discusses any side-effects of the countermeasures.

4.1 Reentrancy evaluation

A smart contract `A` is vulnerable to a reentrancy attack if another contract `B` can re-enter the contract unexpectedly. 'Unexpectedly' is the critical term in this sentence. In normal smart contract interaction, reentrancy is a vital part of operations, as pointed out by Rodler et al. [21]: when contract `A` calls contract `B` to withdraw ether, `B` sends the appropriate amount of ether to `A`, re-entering contract `A`. So reentrancy is necessary for a contract to function normally; it becomes a problem, however, if a contract is not prepared for the re-entrant, as will be demonstrated below.

4.2 Demonstration

The reentrancy attack will be demonstrated in two contexts, public and private. This division is needed as public contracts can use ether, while private contracts cannot. The section ends by evaluating the attack features of the reentrancy attack, which are the same for both types of contract.

```
contract Victim {
    mapping(address => uint) public shares;

    function withdraw() external {
        (bool success,) = msg.sender.call{
            value: shares[msg.sender]}("");
        if (success)
            shares[msg.sender] = 0;
    }

    function donate(address to) payable
        external {
        shares[to] += msg.value;
    }
}
```

Listing 1: Single-function reentrancy victim. *A smart contract vulnerable to reentrancy via the withdraw function.*

```
contract Attack {
    Victim public dao;

    constructor(address addr) {
        dao = Victim(addr);
    }

    fallback() payable external {
        dao.withdraw();
    }
}
```

Listing 2: Single-function reentrancy attack. *A contract able to exploit the reentrancy vulnerability in Listing 1.*

Public Contract

The reentrancy example shown in Listing 1 and 2 is the most basic form of the vulnerability using public smart contracts. The example is an adapted version of the simple DAO contract given in [15]. The contracts shown have been shortened. The complete code resides at the Gitlab repository [46]. The code on Gitlab shows added functionality to facilitate debugging, and `Attack` has additional functions and variables so the owner can extract the collected funds.

Figure 1 shows the different transactions and function calls needed for the setup and execution of the reentrancy attack. First, the attacker, Mallory, identifies a contract with a reentrancy vulnerability such as Listing 1. Once other nodes have deposited ether, the process can begin. Mallory deploys a contract similar to Listing 2, giving the address of the `Victim` in the `constructor`. Mallory calls `Victim.donate` with the `Attack` address as a parameter. This transaction will also contain some amount of ether to donate. When Mallory triggers the `fallback` of the `Attack` contract, the function calls `Victim.withdraw` which starts the exploitation cycle. Next, `withdraw` calls `Attack` to send the money owed. This call triggers the `fallback` on line 16, starting the cycle anew.

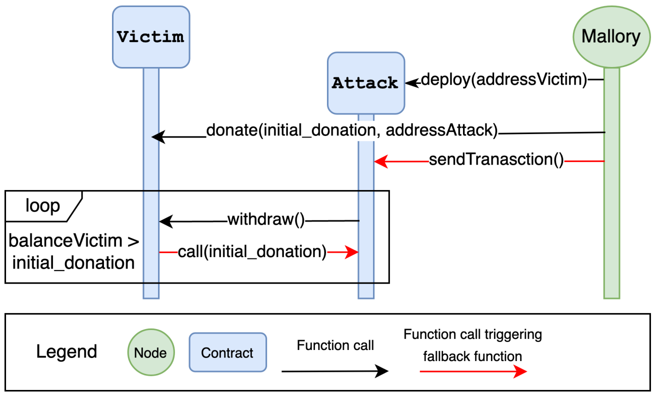

Figure 1: Sequence diagram of a reentrancy attack. *From left to right the colored shapes represent, Listing 1, Listing 2 and the attacker node, Mallory. Each arrow signifies a function call, the chronological ordering starts at the top. The loop shows the `Attack` contract repeatedly reentering the `Victim`.*

Every iteration, the amount initially donated to the `Attack` address is sent back until the balance of `Victim` contains less than the original donation.

The loop ends if one of three conditions apply, the transaction throws an out-of-gas exception, the EVM memory stack if full or `Victim` has no more ether [15]. As stated by Atzei et al: “In all cases an exception is thrown: however, since call does not propagate the exception, only the effects of the last call are reverted, leaving all the previous transfers of ether valid.” [15] Further details about the features of this attack are discussed at the end of this section.

The attack described is called single-function reentrancy. This type of reentrancy is the simplest form of reentrancy and can be detected relatively easily. Rodler et al. [21] identified three other patterns of reentrancy attacks: cross-function reentrancy, delegated reentrancy and create-based reentrancy. Their basic pattern and effects are the same as the presented single-function reentrancy; how exactly these other patterns look is outside of the scope of the current research. It is important to know, however, that these three patterns are more difficult to detect for analysis tools. Paragraph 4.3.d discusses further which analysis tools can detect these types of reentrancy.

Private Contract

As mentioned, private contracts can not use ether, which means the above-presented reentrancy demonstration would not work for a private contract. They can, however, use tokens. For a private contract to be vulnerable to a reentrancy attack, it would have to use a token compliant with the ERC-777 standard. The most-used token standard, ERC-20, would not work for this purpose as it has no fallback-like components, so it is not vulnerable to reentrancy. The ERC-777 standard, however, adds a feature called *receive hooks* [50]. Accounts and contracts can receive and react to these hooks, similar to a fallback function. It is theorized that a private contract transacting ERC-777 compliant tokens could contain reentrancy vulnerabilities. As mentioned previously, neither or both contracts should be private; otherwise, the two can only read values, not call the other’s functions. Due to time constraints, this construction has not been tested, and it is only assumed to work.

Attack Features

This paper identifies two attack features. The first identified feature is the dependency of the contract’s control flow on a state variable. The second feature is the call function’s gas limit. The same attack characteristics apply when deploying `Victim` and `Attack` as private contracts.

It was considered to add another reentrancy attack feature: the fallback function. Without the fallback function, the `Victim` would have to call another contract function by name. This call would imply some level of knowledge about the called contract, and it would not be logical to call a malicious contract on purpose. Thus the fallback function is an important feature of the reentrancy attack.

There are two reasons the fallback function was not categorized as an attack feature in this study. First, the function could only be changed via an update to the protocol, which is outside of the programmer’s power. Moreover, no sources corroborated the idea to remove or drastically change the fallback function, and thus this idea was deemed a viable solution.

Finally there are the vulnerabilities introduced when using private contracts without privacy enhancements. In short, these vulnerabilities are as follows: non-participant nodes can change the state of a private contract, and the private state between participating nodes may diverge. Further details concerning these weaknesses can be found in the background section. These may not be attack features of the vulnerability itself, but they are important to note as attack features of the private contracts.

4.3 Prevention and Mitigation methods

The techniques that prevent or mitigate the reentrancy vulnerability are divided into seven components. The first four methods are the most cited [13], [16], [51] and seemingly also most used. Next is a proposition to use naming conventions as a mitigation technique. All of the previous techniques can be applied to both public and private GoQuorum smart contracts. The final two techniques are specific to GoQuorum however.

For the first three methods, a code example is provided demonstrating how to adjust `Victim.withdraw` to use the proposed technique. Each of these techniques have been tested in the previously presented single-function reentrancy attack, and did indeed prevent the attack. Further details are included with the complete code [46]. The other methods are more related to using certain configurations, so these implementations are not shown.

```
function withdraw() external {
    uint temp = shares[msg.sender];
    shares[msg.sender] = 0;
    msg.sender.call{value: temp}("");
}
```

Listing 3: Correct state variable update solution. *By updating the state before any external contract calls the withdraw function is no longer vulnerable to reentrancy.*

a) Correct state variable update. Listing 3 shows how to implement `Victim.withdraw` using the *checks-effects-interactions pattern* [52]. By first updating the state variable that determines the function's control flow, any reentrancy would result in a call with an ether value set to zero. So when the same attack method is used, only the donated amount is withdrawn. The withdraw function now behaves as expected.

b) Gas limit. The original withdraw function uses the call function, which either forwards all remaining gas or the amount of gas configured in the function parameters [32]. So when Mallory triggers the fallback function of `Attack`, she sets the gas amount as high as possible to ensure the completion of as many exploitation cycles as possible. If the call function is left to forward all gas, her plan succeeds.

To mitigate this problem a programmer may use one of two other functions to send ether: `send` and `transfer`; both have a gas stipend of 2300 [32]. This amount is not enough to pay for many EVM operations. Updating a state variable, for example, would be too expensive. Therefore, some sources recommend using either `send` or `transfer` [16], [53], [54], to avoid another contract being able to reenter repeatedly. Listing 4 shows this approach.

There are two flaws to using such gas-limited functions, however. First of all, it may sometimes be desirable for the `Victim` contract to forward more gas to allow for an expensive fallback function. Limiting the amount of gas is also not a good solution as the gas costs of EVM operations may be changed by an update to the protocol [55]. So operations could unexpectedly become cheaper, enabling reentrancy where reentrancy was not expected to be possible. Combined with the immutability of smart contracts, this would be quite problematic.

```
function withdraw() external {
    payable(msg.sender).transfer(shares[msg.
        sender]);
    shares[msg.sender] = 0;
}
```

Listing 4: Gas limit solution. *The reentrancy vulnerability in 1 is mitigated by using the* `transfer` *function instead of* `call`*. The former introduces limit of 2300 gas.*

c) Mutex/Guard. Another technique is to add a guard or mutex to the function, as can be seen in Listing 5. The guard is activated on line 5 and only deactivated once the transfer of ether concludes. When the guard is active, no one can enter the function code, which protects it from reentrancy. It is, however, imperative to ensure that the guard is released at some point; otherwise, an attacker could ensure the guard is always activated, and the contract would not be able to function as intended [54]. This technique has been incorporated into the Open Zeppelin libraries as the `ReentrancyGuard` [56].

d) Analysis tools. It is now explored whether the use of analysis tools is a good technique to avoid reentrancy. There are dozens of analysis tools that identify weaknesses in smart contracts (e.g. Oyente [57], Securify [58], Mythril [59]). And most of these can indeed detect single-function reentrancy (e.g. Listing 1). However, as mentioned in section 4.2, there are more complex forms of reentrancy that are more difficult to detect by analysis tools. As pointed out by [21] some reentrancy attacks can only be detected by fully simulating contract execution, as is done during dynamic code analysis.

```
1  bool reentrancyMutex = false;
2
3  function withdraw() external {
4      require(!reentrancyMutex);
5      reentrancyMutex = true;
6      (bool success,) = msg.sender.call{value:
           shares[msg.sender]}("");
7      reentrancyMutex = false;
8      if (success) {
9          shares[msg.sender] = 0;
10     }
11 }
```

Listing 5: Mutex/guard solution. *The addition of the guard on line 4 ensures the function can only be entered again once the call is completed.*

To be able to catch all reentrancy vulnerabilities, the tool would have "to execute all combinations of possible re-entry points" [60, line 22]. In short, the more complex the contract, the more difficult it is to detect all reentrancy vulnerabilities.

Besides these more generic tools that try to detect many types of vulnerabilities, there are also specialized reentrancy detection tools: RA [22] and Sereum [21], both can detect most reentrancy attacks. The former improves upon the latter by using dynamic instead of static analysis, which, as mentioned previously, is better at detecting reentrancy.

The dynamic analysis tool RA has three major drawbacks, however. First, it is a more costly type of analysis due to the symbolic execution taking up considerable resources. Moreover, for dynamic analysis, the programmer needs to have information on how exactly the attack is executed. And finally, in their paper, Chinen et al. admit that even RA does not detect all reentrancy vulnerabilities.

e) Naming convention. The next idea is to add 'untrusted,' or something similar, to every function, that either calls another contract or calls a function already marked as 'untrusted' [61]. So one would uses `untrustedWithdraw` instead of `withdraw`. Especially in more complex contracts where many functions are called inside other functions, this method ensures that the programmer is aware of which functions may be vulnerable to reentrancy.

f) (Enhanced) permissioning. This mitigation technique is the first that is specific to GoQuorum. As mentioned in the introduction, GoQuorum is a permissioned network. Permissioning makes GoQuorum networks more trustworthy as the network administrator dictates who can and can not access the network.

Additionally, GoQuorum has an enhanced permissioning model [62], enabling the network to be subdivided into several organizations and sub-organizations. Each organization has nodes, EOA's, and a list of job functions. Enhanced permissions regulate the extent to which each sub-organization or even node has access to the network.

This model can mitigate the risks of a reentrancy vulnerability by only giving nodes access to a contract if truly necessary. The theory is that, if only specified nodes have access, there is less chance of one of the nodes being a malicious actor and mounting an attack on the contract. It may still be possible that a trusted node turns out to be malicious or a node gains unexpected access to the network. However, permissioning can still be used as an extra safeguard against reentrancy and vulnerability exploitation.

g) Private contract. The final technique involves GoQuorum private contracts. Private contracts are only vulnerable to reentrancy attacks when using an ERC-777 compliant token. Thus, switching from using public contracts and ether to using private contracts and tokens can already significantly reduce the risk of deploying a contract with a reentrancy vulnerability. Moreover, private contracts reduce the number of nodes with access to the `Victim` contract, similar to the previous method, further decreasing the risk of reentrancy. However, private contracts without the privacy enhancements discussed in section c have vulnerabilities as well.

4.4 Categorization

To compare the different prevention and mitigation techniques, each method has been assigned one of three categories: feature, access, and awareness. Each category represents an aspect of the vulnerability that the solution method addresses. The first category includes methods that directly address one of the aforementioned attack **features**. Three methods relate to the **access** restriction of vulnerable function; these are included in the second category. The final category considers methods that increase the programmers **awareness** of possible vulnerabilities. Table 1 shows which solutions belong to which category and which solutions are mitigation versus prevention techniques. Even a technique marked as preventative will only work if used correctly and for all reentrancy vulnerabilities in the contract. The table can also serve as a reference for the letters corresponding to each solution

Table 1: Categorized prevention and mitigation techniques. *Each presented technique is categorized into addressing one of three vulnerability aspects: attack features, function access, and vulnerability awareness. Moreover, M marks a mitigation technique and P a prevention technique.*

		feature	access	awareness
a	Correct state variable update	P		
b	Gas limit	M		
c	Mutex/Guard		P	
d	Analysis tools			M
e	Naming convention			M
f	(Enhanced) permissioning		M	
g	Private contract		M	

Attack Feature

Prevention techniques a and b both address an identified attack feature.

Solution a is a proper prevention technique, as it solves the root of the problem. When using this technique it is important to update all state variables before calling external contracts, not just those used in the call.

As described earlier, solution b is no longer considered a viable preventative, as the amount of gas per EVM operation may be subject to change at any time. Although b should not be relied upon, it is still believed that incorporating some gas limit can at least mitigate a reentrancy attack by reducing the number of times an attacker could reenter.

Function Access

The solutions in the second category all relate to the amount of access a malicious actor has to the re-enterable code. Well-placed guard modifiers (c) prevent anyone from reentering a function before it is done executing external calls. The guard has to be placed around every external contract call, also those that first go via another contract function. The risk when using this method is that the programmer may lose track of which function call will end up calling an external contract.

Solution f and g restrict access as well, but in a different way then c. These two methods reduce the number of nodes that have any access to the contract, thus decreasing the chance that one may be a malicious actor. These methods are not as effective as c, however. Because they do not prevent reentrant access, instead, they just cut back on all nodes' access in general. Both of these techniques are only mitigative. Moreover, when using g it is vital to understand the private contract vulnerabilities.

Vulnerability Awareness

The final two methods deal with the awareness the contract programmer has concerning the danger or reentrancy attacks.

The first method to increase awareness is using analysis tools to identify contract vulnerabilities (d). It is indeed advised to use all resources available to increase one's contract's security. In this paper, however, the reader is implored to treat tools as a backup and not a crutch. As can be read in [21], [22], most analysis tools can detect simpler forms of reentrancy, however, the more complicated the situation, the more difficult a vulnerability is to spot. Analysis tools may even give a false sense of security.

Another way to increase vulnerability awareness is by using mitigation technique e. So although the technique does not directly tackle the vulnerability, it is recognized that a programmer can only defend against an attack if he/she is aware of where the exploitable vulnerability exactly is. Technique e can be effective against reentrancy in more complex situations by ensuring one knows exactly which functions are vulnerable to reentrancy.

4.5 Side effects

Solution methods a, c, e have basically no side effects. These techniques call for some simple code adjustments, but not anything that would significantly impact smart contract performance. As mentioned before, introducing a gas limit (b) may introduce unwanted restrictions on the external contract's fallback functions. Using analysis tools (d), on the other hand, will cost extra resources. And finally, methods f and g do not have any side effects on performance. However, as specified, it restricts nodes from accessing the contract, which may not be wanted. Although there would be added overhead, and some transactions may completely fail if contracts were to be used with privacy enhancements

5 Responsible Research

This section will consider two components of responsible research: ethics and reproducibility. The former component deals with the agreed-upon system of moral principles used in the current piece of scientific literature. The latter component covers the reader's capability to recreate this study.

5.1 Ethics

The moral principles guiding this paper are all based on doing no undue harm to others. Three such ethical concerns are addressed: privacy and security when handling data, influence on other users of the blockchain network, and responsible disclosures of bugs.

In this report, no data is collected, so no data must be processed and stored securely. There are also no user surveys used, so there are no concerns regarding anonymity and privacy. Essentially there is no way others are impacted through the correct or incorrect handling of data.

The project could, however, negatively impact other people if the tests were done on the Ethereum *mainnet*, which is the deployed main network through which all Ethereum nodes connect. Testing an attack on smart contracts deployed on the mainnet could have legitimate negative consequences to users. In the case of Ethereum, all testing was done via a private testnet set up by the Remix online IDE. As GoQuorum is a permissioned network, there was no mainnet to connect to, so a local private testnet had to be configured regardless. Section 3.3 describes details regarding this setup process.

A more interesting part of the research ethics stems from the fact that this report considers current smart contract vulnerabilities. Anyone may exploit these vulnerabilities and launch an attack on a deployed smart contract. This research does not, however, contain any *zero days* (undisclosed, unidentified exploits). If this were the case, the report publishing process should follow responsible disclosure guidelines.

The attack discussed in this report, reentrancy, is well known and well studied already. Moreover, the private contract vulnerabilities are also already discussed in the GoQuorum documentations. In short, there is no additional risk to existing smart contracts by publishing this study.

5.2 Reproducibility

The first part of this section discusses the classic example of reproducibility: citations. The following paragraph considers the often scarce referencing regarding the implementation side of the research. The final passage is dedicated to the reproducibility of the network setup.

Besides scientific papers, references may include documentation, forums, blogs, software, or tools. The latter two are referenced by either linking its website or project repository.

Citations are often the sole focus when considering the reproducibility of a report. In computer science, however, there is usually also a technical side to a research paper. Researchers frequently neglect this facet of reproducibility, as discussed in [63].

Moreover, because experts with a considerable amount of experience write papers, there is often an assumption that the reader has a similar level of know-how. Moreover, in papers that do reference some code repository, the actual steps needed to run the code are regularly not included. Of all the papers cited, only [15] referenced complete and working code examples; other papers only displayed shortened versions of smart contract code. None of the mentioned papers demonstrated more extensively how they set up and interacted with their network. Assuming that these steps are easily solved does not consider platforms about which not enough resources are available. When a paper concerns less popular software, as is the case here, describing the steps taken to set up and interact with the network is especially important.

This paper tries to assist this part of the recreation process by providing complete code examples and full explanations on setting up the presented scenarios. The description of how to set up a network is essential for GoQuorum as the code generated when setting up a testnet is severely lacking in comments. This absence of documentation persists in at least the three setup guides presented in 3.3.

Although the GoQuorum developers did not provide their setup guides with adequate documentation, the guides do facilitate the repeatability of the project by providing the same setup and configurations no matter the underlying system by using Docker images. As mentioned previously (3.3), for any Ethereum testing, the Remix IDE and provided testnet was used, which also standardizes the configurations for increased reproducibility.

6 Discussion and Future work

Although the study provided insights into reentrancy countermeasures for both public and private contracts, the study has several shortcomings. All testing was done in a small development network, not a complete deployed network. Furthermore, due to a lack of time and resources, there was no research into other vulnerabilities.

For future research, the same countermeasures should be tested in a more complete network. Moreover, the mitigation techniques should be tested on effectiveness. Then any other known Ethereum vulnerabilities should be evaluated in a GoQuorum context. Furthermore, there should be studies that look into novel GoQuorum-specific vulnerabilities. And more generally, other Ethereum soft forks should be considered similarly.

7 Conclusion

This research aimed to evaluate the well-known smart contract vulnerability, reentrancy, for the first time in the context of a GoQuorum network (a soft fork of the standard Ethereum protocol). As one of the first, this study includes extensive information on GoQuorum smart contract interactions. Most importantly, seven methods were identified that would either mitigate or prevent reentrancy in GoQuorum-based smart contracts. Each of these techniques either addressed a feature of, access to, or awareness of the vulnerability.

If one wants to write a GoQuorum-based smart contract not vulnerable to reentrancy, the following advice may be followed. Above all, the checks-effects-interactions pattern should be used whenever a call is made to an external contract. This technique prevents any reentrancy attack. It is advised not to rely on a gas limit to stop reentrancy attacks as the amount of gas per bytecode operation may be subject to change. However, they can still be used as a mitigation technique. Adding a guard to the vulnerable function will, though, prevent any reentrancy. Analysis tools can be used to detect reentrancy if one realizes the weaknesses of each tool and does not rely on them to fix one's code. To keep track of which functions are vulnerable, name all functions 'untrusted' that call either another contract or a function that does so. When using GoQuorum, and only part of the network has to be privy to the contract, either enhanced permissioning or private contracts may be used. For the latter, one should be aware of the inherent limitations. It is advised to use a combination of techniques to ensure the safest possible situation and recognize that the more complicated the contract, the bigger the chance that a vulnerability will go undetected.

Acknowledgments The author thanks the GoQuorum Slack channel members for their GoQuorum expertise and her peer group for their feedback and opinions.

References

[1] S. Nakamoto. (2008). "Bitcoin: A peer-to-peer electronic cash system," [Online]. Available at: https://bitcoin.org/bitcoin.pdf (Accessed on 05/05/2021).

[2] B. Bambrough, "As bitcoin's total value nears $1 trillion, these crypto prices are leaving bitcoin in the dust," *Forbes*, Feb. 18, 2021. [Online]. Available at: https://www.forbes.com/sites/billybambrough/2021/02/18/as-bitcoin-total-value-nears-1-trillion-these-crypto-prices-are-leaving-it-in-the-dust/ (Accessed on 05/06/2021).

[3] N. Szabo, "Formalizing and securing relationships on public networks," *First monday*, 1997.

[4] Ethereum Foundation. "Ethereum," [Online]. Available at: https://ethereum.org/ (Accessed on 06/19/2021).

[5] ConsenSys. "Quorum whitepaper," [Online]. Available at: https://raw.githubusercontent.com/ConsenSys/quorum/master/docs/Quorum%5C%20Whitepaper%5C%20v0.2.pdf (Accessed on 05/01/2021).

[6] Consensys, [Online]. Available at: https://consensys.net/ (Accessed on 06/19/2021).

[7] ConsenSys. "Goquorum," [Online]. Available at: https://github.com/ConsenSys/quorum (Accessed on 06/19/2021).

[8] Hyperledger. "Besu ethereum client," [Online]. Available at: https://github.com/hyperledger/besu (Accessed on 06/19/2021).

[9] Covantis, [Online]. Available at: https://www.covantis.io/ (Accessed on 06/19/2021).

[10] Komgo, [Online]. Available at: https://www.komgo.io/ (Accessed on 06/19/2021).

[11] T. Hewa, M. Ylianttila, and M. Liyanage, "Survey on blockchain based smart contracts: Applications, opportunities and challenges," *Journal of Network and Computer Applications*, p. 102 857, 2020.

[12] S. Dhaiouir and S. Assar, "A systematic literature review of blockchain-enabled smart contracts: Platforms, languages, consensus, applications and choice criteria," in *International Conference on Research Challenges in Information Science*, Springer, 2020, pp. 249–266.

[13] S. Rouhani and R. Deters, "Security, performance, and applications of smart contracts: A systematic survey," *IEEE Access*, vol. 7, pp. 50 759–50 779, 2019.

[14] Coindesk. (Jun. 25, 2016). "Understanding the dao hack," [Online]. Available at: https://www.coindesk.com/understanding-dao-hack-journalists (Accessed on 05/06/2021).

[15] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *International conference on principles of security and trust*, Springer, 2017, pp. 164–186.

[16] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A survey on ethereum systems security: Vulnerabilities, attacks, and defenses," *ACM Computing Surveys (CSUR)*, vol. 53, no. 3, pp. 1–43, 2020.

[17] Z. A. Khan and A. S. Namin, "A survey on vulnerabilities of ethereum smart contracts," *arXiv preprint arXiv:2012.14481*, 2020.

[18] D. He, Z. Deng, Y. Zhang, S. Chan, Y. Cheng, and N. Guizani, "Smart contract vulnerability analysis and security audit," *IEEE Network*, vol. 34, no. 5, pp. 276–282, 2020.

[19] Z. Wang, H. Jin, W. Dai, K.-K. R. Choo, and D. Zou, "Ethereum smart contract security research: Survey and future research opportunities," *Frontiers of Computer Science*, vol. 15, no. 2, pp. 1–18, 2021.

[20] A. Mense and M. Flatscher, "Security vulnerabilities in ethereum smart contracts," pp. 375–380, 2018.

[21] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," *arXiv preprint arXiv:1812.05934*, 2018.

[22] Y. Chinen, N. Yanai, J. P. Cruz, and S. Okamura, "Ra: Hunting for re-entrancy attacks in ethereum smart contracts via static analysis," in *2020 IEEE International Conference on Blockchain (Blockchain)*, IEEE, 2020, pp. 327–336.

[23] B. Hu, Z. Zhang, J. Liu, Y. Liu, J. Yin, R. Lu, and X. Lin, "A comprehensive survey on smart contract construction and execution: Paradigms, tools, and systems," *Patterns*, vol. 2, no. 2, p. 100 179, 2021.

[24] M.-J. Lagarde, "Security assessment of authentication and authorization mechanisms in ethereum, quorum, hyperledger fabric and corda," M.S. thesis, DEDIS at EPFL, Lausanne, 2019. [Online]. Available at: https://www.epfl.ch/labs/dedis/wp-content/uploads/2020/01/report-2018_2-marie-jeanne-security-assessment.pdf (Accessed on 05/05/2021).

[25] Ethereum Foundation. (Jun. 11, 2021). "Nodes and clients," [Online]. Available at: https://ethereum.org/en/developers/docs/nodes-and-clients/#full-node (Accessed on 06/19/2021).

[26] Ethereum Foundation. (May 12, 2021). "Ethereum virtual machine (evm)," [Online]. Available at: https://ethereum.org/en/developers/docs/evm/ (Accessed on 05/23/2021).

[27] Solidity Team. "Solidity," [Online]. Available at: https://soliditylang.org/ (Accessed on 06/19/2021).

[28] Ethereum Foundation. (Jun. 7, 2021). "Gas and fees," [Online]. Available at: https://ethereum.org/en/developers/docs/gas/#top (Accessed on 06/10/2021).

[29] Ethereum Foundation. (Mar. 30, 2021). "Mining," [Online]. Available at: https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/mining/ (Accessed on 06/19/2021).

[30] Wikipedia contributors. (May 4, 2021). "Turing completeness — Wikipedia, the free encyclopedia," [Online]. Available at: https://en.wikipedia.org/w/index.php?title=Turing_completeness&oldid=1023112232.

[31] Ethereum Foundation. "Ethereum accounts: Key differences," [Online]. Available at: https://ethereum.org/en/developers/docs/accounts/#key-differences (Accessed on 06/10/2021).

[32] Solidity by Example. "Sending ether (transfer, send, call)," [Online]. Available at: https://solidity-by-example.org/sending-ether/ (Accessed on 07/09/2021).

[33] E. Gesheva. (Mar. 26, 2020). "Solidity 0.6.x features: Fallback and receive functions," [Online]. Available at: https://blog.soliditylang.org/2020/03/26/fallback-receive-split/ (Accessed on 07/09/2021).

[34] Ethereum Foundation. (Nov. 6, 2020). "Patricia tree," [Online]. Available at: https://eth.wiki/fundamentals/patricia-tree (Accessed on 06/22/2021).

[35] ConsenSys. (May 19, 2021). "Privacy enhancements," [Online]. Available at: https://docs.goquorum.consensys.net/en/latest/Concepts/Privacy/PrivacyEnhancements (Accessed on 05/31/2021).

[36] Ethereum Foundation. (May 6, 2021). "Ethereum development documentation," [Online]. Available at: https://ethereum.org/en/developers/docs/ (Accessed on 05/23/2021).

[37] Ethereum Foundation. "Solidity," [Online]. Available at: https://docs.soliditylang.org/en/v0.8.4/ (Accessed on 05/23/2021).

[38] Ethereum Foundation, *Geth*, version 1.10.3, May 5, 2021. [Online]. Available at: https://geth.ethereum.org/ (Accessed on 06/15/2021).

[39] ChainSafe, *Web3.js*, version 1.3.4. [Online]. Available at: https://web3js.readthedocs.io/en/v1.3.4/ (Accessed on 06/15/2021).

[40] ConsenSys. "Goquorum slack inviter," [Online]. Available at: https://inviter.quorum.consensys.net/ (Accessed on 06/15/2021).

[41] Ethereum Foundation, *Remix - ethereum ide*, version 0.12.0. [Online]. Available at: remix.ethereum.org/ (Accessed on 06/15/2021).

[42] ConsenSys. (Apr. 20, 2021). "Goquorum projects," [Online]. Available at: https://docs.goquorum.consensys.net/en/stable/Reference/GoQuorum-Projects/ (Accessed on 05/31/2021).

[43] ConsenSys. (Apr. 20, 2021). "Goquorum wizard," [Online]. Available at: https://docs.goquorum.consensys.net/en/stable/HowTo/GetStarted/Wizard/GettingStarted/ (Accessed on 05/31/2021).

[44] ConsenSys. (Nov. 24, 2020). "Quorum developer quickstart," [Online]. Available at: https://docs.goquorum.consensys.net/en/stable/Tutorials/Quorum-Dev-Quickstart/ (Accessed on 05/31/2021).

[45] Docker, version 20.10.5. [Online]. Available at: https://www.docker.com/ (Accessed on 06/15/2021).

[46] Op den Orth, Sara. "Rp-group-28-sopdenorth," [Online]. Available at: https://gitlab.ewi.tudelft.nl/cse3000/2020-2021/rp-group-28/rp-group-28-sopdenorth (Accessed on 06/27/2021).

[47] S. Tikhomirov. (Apr. 22, 2020). "Solidity-latex-highlighting," [Online]. Available at: https://github.com/s-tikhomirov/solidity-latex-highlighting (Accessed on 06/03/2021).

[48] ConsenSys, *Cakeshop*, version 0.12.0. [Online]. Available at: https://github.com/ConsenSys/cakeshop (Accessed on 06/15/2021).

[49] ConsenSys, *Quorum reporting*. [Online]. Available at: https://github.com/ConsenSys/quorum-reporting (Accessed on 06/15/2021).

[50] Open Zeppelin. "Erc777," [Online]. Available at: https://docs.openzeppelin.com/contracts/3.x/erc777 (Accessed on 06/22/2021).

[51] ConsenSys. "Known attacks," [Online]. Available at: https://consensys.github.io/smart-contract-best-practices/known_attacks/ (Accessed on 06/22/2021).

[52] Ethereum Foundation. "Security considerations: Use the checks-effects-interactions pattern." version 0.8.4, [Online]. Available at: https://docs.soliditylang.org/en/v0.8.4/security-considerations.html?highlight=Checks-Effects-Interactions%5C%20pattern#use-the-checks-effects-interactions-pattern (Accessed on 07/07/2021).

[53] S. Verma. (Aug. 4, 2020). "Reentrancy exploit," [Online]. Available at: https://medium.com/coinmonks/reentrancy-exploit-ac5417086750 (Accessed on 06/09/2021).

[54] W. Shahada. (Apr. 24, 2019). "Protect your solidity smart contracts from reentrancy attacks," [Online]. Available at: https://medium.com/coinmonks/protect-your-solidity-smart-contracts-from-reentrancy-attacks-9972c3af7c21 (Accessed on 06/09/2021).

[55] S. Marx. (Sep. 2, 2019). "Stop using solidity's transfer() now," [Online]. Available at: https://consensys.net/diligence/blog/2019/09/stop-using-soliditys-transfer-now (Accessed on 06/09/2021).

[56] Open Zeppelin. "Utilities: Reentrancyguard," [Online]. Available at: https://docs.openzeppelin.com/contracts/2.x/api/utils#ReentrancyGuard (Accessed on 06/21/2021).

[57] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.

[58] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 67–82.

[59] ConsenSys. "Mythril," [Online]. Available at: https://github.com/ConsenSys/mythril (Accessed on 06/22/2021).

[60] S. S. R. G. U. Duisburg-Essen. (Nov. 16, 2018). "Eth-reentrancy-attack-patterns/cross-function.sol," [Online]. Available at: https://github.com/uni-due-syssec/eth-reentrancy-attack-patterns/blob/master/cross-function.sol.

[61] ConsenSys Diligence. "Secure development recommendations," [Online]. Available at: https://consensys.github.io/smart-contract-best-practices/recommendations/#mark-untrusted-contracts (Accessed on 06/09/2021).

[62] ConsenSys. (Sep. 20, 2020). "Enhanced permissions model," [Online]. Available at: https://docs.goquorum.consensys.net/en/stable/Concepts/Permissioning/Enhanced/EnhancedPermissionsOverview/ (Accessed on 06/05/2021).

[63] V. C. Stodden, "Reproducible research: Addressing the need for data and code sharing in computational science," 2010.

[64] Ethereum Foundation. (May 12, 2021). "Proof-of-work (pow)," [Online]. Available at: https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/ (Accessed on 06/21/2021).

[65] Binance Academy. "Proof of authority explained," [Online]. Available at: https://academy.binance.com/en/articles/proof-of-authority-explained (Accessed on 10/09/2020).

[66] ConsenSys. (Dec. 7, 2020). "Ibft consensus overview," [Online]. Available at: https://docs.goquorum.consensys.net/en/stable/Concepts/Consensus/IBFT/ (Accessed on 05/15/2021).

[67] ConsenSys. (Dec. 7, 2020). "Raft," [Online]. Available at: https://docs.goquorum.consensys.net/en/stable/Concepts/Consensus/Raft/ (Accessed on 05/15/2021).

[68] ConsenSys. (Dec. 7, 2020). "Clique," [Online]. Available at: https://docs.goquorum.consensys.net/en/stable/Concepts/Consensus/Clique/ (Accessed on 06/22/2021).

A Consensus Algorithms

Following a consensus algorithm, full nodes validate the transactions and miners add them to the blockchain. Consensus algorithms are used to ensure that dynamic, changing networks can agree on something. In the case of blockchain, the nodes have to agree on the transaction order.

Ethereum currently uses Proof of Work (PoW) algorithms [64], while GoQuorum uses Proof of Authority (PoA) [65]. Where the former is more suited to public blockchains as it is very resistant to manipulation, the latter sacrifices fault resistance for increased transaction throughput. PoA works for GoQuorum because it is a permissioned blockchain. Thus, it is a more trusted environment. GoQuorum networks can be configured using one of three different consensus algorithms: IBFT [66], Raft [67] and Clique [68].