

Revisiting smart contract vulnerabilities in Hyperledger Fabric

Cathrine Paulsen, Prof. Dr. Kaitai Liang, Huanhuan Chen

TU Delft

Abstract

Hyperledger Fabric is a permissioned enterprise blockchain allowing organizations to collaborate and automate processes via smart contracts. However, these contracts could contain security vulnerabilities leading to unexpected behavior or other negative consequences. Therefore, this study takes a closer look at three reported smart contract vulnerabilities in Fabric: rich queries, pseudorandom number generators, and global variables. Smart contracts containing these vulnerabilities were deployed on a test network, and the vulnerable contract features were exploited and explained. The study provides an estimation of each vulnerability's impact severity, and possible countermeasures to lower it were explored and evaluated. This study found that the proposed countermeasures can at least mitigate the impact severity of all three vulnerabilities. Additionally, the study provides an overview of compatible analysis tools. The available tools were found to be lacking, however, as most of them do not exist outside of research papers. Overall, static code analysis tools were found to be effective at detecting all three vulnerabilities.

1 Introduction

Nakamoto conceptualized the use of blockchain technology for financial transactions, commonly known as blockchain 1.0, in the Bitcoin whitepaper in 2008 [1]. By using a peer-to-peer network, online payment systems could be based on cryptographic proof rather than relying on a trusted third party to handle and verify transactions. Although the concept of blockchain goes as far back as 1991 [2], Bitcoin became the first successful proof-of-concept of the practical uses of blockchain technology and laid the foundations for the various platforms that would proceed it. One such platform is Ethereum, which extended blockchain 1.0 with smart contract technology [3]. The addition of smart contracts resulted in blockchain 2.0, which significantly broadened the potential use cases of blockchain technology [4, 5].

Smart contracts allow using blockchains for applications other than financial transactions; they are self-enforcing and event-driven programs stored on the blockchain that

can interact with the ledger and invoke transactions [6, 7]. The transparent and immutable properties of smart contracts make them a promising technology in a wide variety of applications ranging from automated insurance, voting systems, secure health record storage, and more [7–9]. However, smart contracts could also result in negative consequences if not used carefully, such as millions in financial loss due to exploits like the Ethereum DAO hack [10]. The high security requirements associated with smart contracts hinder the widespread adoption of the technology in most viable applications [6]. It is therefore important to understand what the potential security vulnerabilities are, what the impact severity would be if a malicious actor were to exploit them, and how to mitigate exploitation.

Related research. Extensive research into various security and privacy issues surrounding smart contracts is available and has resulted in numerous frameworks and tools to evaluate and validate the security of smart contracts. [11] provides a categorization of vulnerabilities in Ethereum smart contracts, including example code of attacks that exploit them. The study does not discuss possible countermeasures for each attack but highlights that smart contract verification tools can detect most of the identified vulnerabilities. [6] provides a thorough survey of smart contracts, including reported vulnerabilities and analysis of design schemes and tooling that can prevent them. However, the majority of vulnerabilities and tools are heavily focused on Ethereum, although the survey briefly mentions other platforms like Hyperledger Fabric.

Although Fabric is a well-known and popular framework [12], there is significantly less research on attacks exploiting vulnerabilities in Fabric smart contracts [13]. Yamashita et al. [14] discussed several security risks in Fabric smart contracts attributed to the Go programming language. However, it is unclear how severe these vulnerabilities are and what potential exploits exist. Countermeasures besides analysis tools are also not discussed. Several hypothetical attacks on Fabric are described in [15], however, there are no concrete examples of the attacks. Both [16] and [17] expand upon the vulnerabilities found in [14] with vulnerabilities related to the network and external resources. [16] categorizes several Fabric vulnerabilities and countermeasures, but the smart contract-related countermeasures are only briefly mentioned. Besides

the smart contract-specific vulnerabilities mentioned in [14], there seem to be more vulnerabilities related to various network and consensus configurations; this is likely due to both the network and consensus protocol being highly customizable.

Motivation. Despite the various studies describing Fabric smart contract vulnerabilities, concrete implementations of these vulnerabilities and their exploitations are lacking. Moreover, countermeasures besides analysis tools are rarely mentioned and often not explored in detail. Unlike Ethereum, Fabric focuses on the use of blockchain 2.0 in a trusted enterprise setting; example use cases include supply chain tracking and asset transfers between known organizations rather than transactions between anonymous users [18]. Because Fabric has different applications than Ethereum, it also has different security needs. Hence, vulnerabilities and exploits in Ethereum likely do not directly apply to Fabric, and more detailed studies into Fabric-specific smart contract vulnerabilities are needed.

Contribution. This paper aims to address the aforementioned gap in research by focusing on a few reported vulnerabilities and providing examples of their implementation and possible exploitation. It may be unrealistic to completely avoid a vulnerability in some applications; therefore, this paper will provide countermeasures besides analysis tools that can eliminate or reduce the vulnerability’s impact severity.

The main question this study aims to answer is as follows:

Q: *What are a few security vulnerabilities in Hyperledger Fabric smart contracts, and what are their countermeasures?*

To expand on this question, three subquestions are proposed:

Q1: *How can these vulnerabilities be exploited?*

Q2: *What is the impact severity of these exploitations?*

Q3: *How do the countermeasures affect the impact severity?*

Structure. This paper is organized into six sections. Section 2 provides an explanation of Fabric’s architecture and transaction flow. Section 3 covers the research and implementation methodology. Section 4 presents the results of the research. Section 5 considers the reproducibility and ethical implications of the work presented in this study. Finally, section 6 concludes this study by answering the research questions and proposes areas of future work.

2 Background

The following section presents the necessary technical knowledge needed to understand the technical aspects of this paper. Section 2.1 provides an overview of Fabric’s architecture and introduces important terms. The transaction flow is explained in more detail in section 2.2.

2.1 Architectural overview

Fabric uses smart contract technology to invoke transactions, and distributed ledger technology to store them [18]. *Transactions* are events acting upon objects, or assets, stored on the ledger [19]. Assets may be implemented as structs in Go, see

appendix A for an example. There are two types of transactions: query and update transactions. Only update transactions can modify, delete or add assets to the ledger.

The ledger and the world state

The *ledger* is an append-only, transparent record of all transactions on the network. It consists of several ordered blocks containing a subset of transactions. A unique hash identifies each block, and the block refers back to the hash of its predecessor. The ledger is therefore also referred to as the “blockchain” [19]. All peers keep a distributed copy of the ledger. *Smart contracts* allow controlled interactions with the ledger and are also stored on the peers [20]; they are arbitrary programs that implement the Fabric Contract API interface [21]. Smart contracts are packaged into *chaincode*, but these terms are used interchangeably. In addition to the ledger, every peer maintains a state database representing the world state.

The *world state* reflects the current state of the ledger; it contains the latest values of the assets stored on the ledger for fast retrieval. Fabric currently supports two storage solutions for the world state: LevelDB and CouchDB. LevelDB is the default database used by the test network and is a fast key-value store that maps keys to binary data. LevelDB cannot reason about the contents of the data, so it can only perform key-based queries [22]. CouchDB, on the other hand, is a document-object store and stores assets as JSON documents. This allows CouchDB to perform content-based, or rich, queries. CouchDB is therefore well-suited for smart contracts that require fast queries on specific properties [23].

Entity types

There are three main types of entities in the Fabric network: clients, peers, and orderers [24]. These entities hold key roles in the transaction flow. *Clients* are applications that request transactions on behalf of the end-user. *Peers* endorse transaction requests and validate transactions received by the orderer. *Orderers* package endorsed transactions into blocks and broadcast the blocks on the network. Multiple orderers form an ordering service following the RAFT consensus protocol; the test network in this study has only one orderer, so this aspect of consensus will not be covered. The other aspect is the *endorsement policy*, which is explained in section 2.2. See figure 1 for an overview of the test network.

In addition to the three main entity types, there are also certificate authorities (CA) and membership service providers (MSP). These two components make Fabric a permissioned network: Every entity has an identity issued by a CA, while the MSPs govern the permission rules of the network based on these identities. As Fabric was designed with enterprise applications in mind, every network entity belongs to an *organization*; every organization provides their own CA and MSP to verify the identities of their entities.

Privacy

Fabric adds a layer of privacy through the use of channels. *Channels* partition the network into smaller, private blockchains, each with its own ledger accessible only to the entities in that channel. In addition to channels, there are *private data* collections that allow storing (partial) assets in a

separate data storage that only specific peers can access [25]. Private data collections work on the same principle as channels and are used when creating a new channel would result in unnecessary overhead; e.g., when a subset of channel organizations should be privy to a subset of the data.

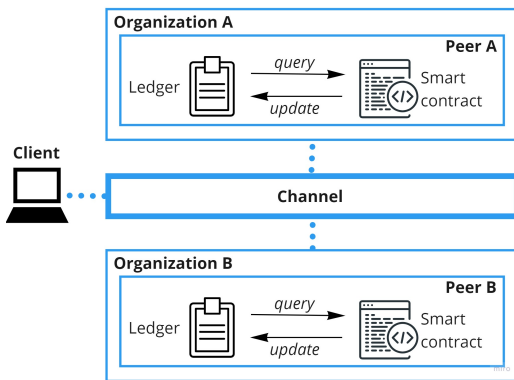


Figure 1: A simplified diagram of the test network showing the most important components. Clients connect to peers of different organizations through the channel. Every peer holds a copy of the channel’s ledger and uses smart contracts to interact with it.

2.2 Execute-order-validate transaction flow

Fabric was the first blockchain framework to utilize smart contracts written in general-purpose languages, as opposed to domain-specific languages (DSLs) like Solidity [26]. Regardless of the blockchain 2.0 framework that is used, smart contracts are executed by multiple nodes with the same input expecting the same output. In other words, smart contracts need to be deterministic. DSLs ensure determinism by simply avoiding non-deterministic language features. Fabric instead applies a novel execute-order-validate approach to the transaction flow (compared to order-execute in Ethereum) [18]; non-deterministic smart contract invocations will fail in the execute phase. Note that although the flow is called the "transaction flow", it only applies to update transactions. Query transactions instead go directly between client and peer [27]. The whole flow is described below and illustrated in figure 2.

Execute phase

1. The client sends a transaction proposal to (a subset of) the endorsing peers.
2. The endorsing peers simulate the transaction. Every peer returns their endorsement together with a readset and a writeset containing the results of the simulation. The writeset contains asset modifications, while the readset contains the asset values that were read and their version numbers [26].
3. The client collects the endorsements and checks that all read- and writesets match. The *endorsement policy* specifies a subset of peers that need to agree on the transaction’s read- and writesets.

Order phase

1. The client sends the transaction to the orderer once the endorsement policy is fulfilled.
2. The orderer decides on the order of transactions, groups them into blocks and broadcasts the blocks to the peers.

Validate phase

1. The peers check the endorsement policy, and verify that the version numbers in the readset equal the version numbers in their world state.
2. The peers apply the changes in the writeset to their world state, and append the block to their copy of the ledger.

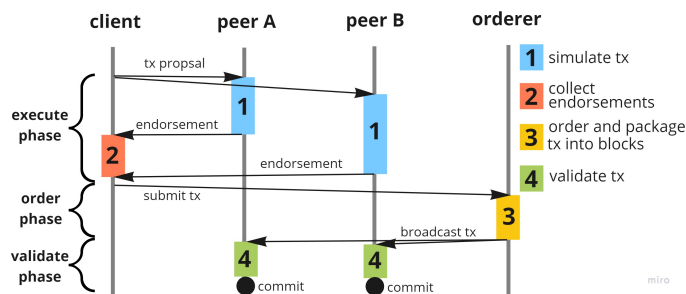


Figure 2: Sequence diagram showing the execute-order-validate transaction (tx) flow, based on [26, figure 4] and modified to represent the test network.

3 Methodology

There are two main aspects to this study: *researching* existing vulnerabilities, attacks and countermeasures; and *implementation* to reproduce the identified vulnerabilities and attacks. This section will go over the methodology for both of these aspects.

3.1 Literature research

To find literature reporting on smart contract vulnerabilities in Fabric, Google Scholar was used with the following query terms: ((`"Hyperledger Fabric" AND "smart contract"`) OR `chaincode`) AND (`vulnerabilit*` OR `attack*`).

The latest major version, Hyperledger Fabric v2, was released in January 2020. Therefore, only results from 2020 onwards were initially considered to ensure that the vulnerabilities, attacks or countermeasures discussed in the literature are still relevant. Due to the lack of research on this topic, however, the time window for relevant literature was expanded to 2019. In addition to the query terms, the snowball and citation searching methods [28] were used to find related literature not captured by the initial search query.

A subset of the reported smart contract vulnerabilities was chosen based on their perceived implementation feasibility as well as the method of exploitation. This selection procedure allowed avoiding vulnerabilities with similar methods of exploitation and mitigation to cover a broader spectrum of vulnerabilities and countermeasures within a limited time frame.

3.2 Implementation

Contract deployment. The chosen vulnerabilities were studied and subsequently implemented in smart contracts deployed on a local Hyperledger Fabric (v2.2.3) test network. The test network was installed and set up using Docker (v20.10.6) following the official tutorial [29], consisting of a single channel, two organizations with one peer each, and one orderer. Interaction with the network was done via the command-line interface (CLI) using the provided `network.sh` script and `peer` [30] binary. Contract deployment was done using the `./network.sh deployCC` command. Unless stated otherwise, all contracts were deployed with the default endorsement policy, i.e. all peers are required to endorse transactions.

Fabric supports smart contracts written in several general-purpose languages, and some vulnerabilities may depend on the language used. To keep the results consistent, this paper focuses on smart contracts written in Go (v1.16.3), which was the language used in most of the literature reporting on vulnerabilities. The full code, including a setup and exploitation guide, is available on Github ¹.

Invoking transactions. The vulnerable smart contracts were invoked to achieve unexpected behavior or output, effectively simulating an attack on the contract. To invoke query transactions, the `peer chaincode query` command was used. These transactions are only executed by the peer whose address is stored in the `CORE_PEER_ADDRESS` environment variable. Similarly, the `peer chaincode invoke` command was used to invoke update transactions. After a successful malicious invocation, the vulnerable features of the contract that allowed the exploitation were identified, and the impact severity was calculated following the Common Vulnerability Scoring System (CVSS) [31]. Countermeasures that target the vulnerable features were then explored, and it was evaluated whether they could lower the impact severity.

4 Vulnerabilities

The vulnerabilities that were implemented and exploited are presented in this section. Each vulnerability is illustrated by code snippets, followed by a description of how the vulnerable features are exploitable. Finally, there is a discussion on possible countermeasures and whether they can solve the vulnerable features without introducing other problems. Tools that detect the vulnerabilities are discussed in section 4.4, and the vulnerabilities' impact severity is analyzed in section 4.5.

4.1 Updates using rich queries

The "updates using rich queries" vulnerability is also referred to as "range query risk" in some papers [14, 16, 32]. The vulnerability was implemented in a smart contract based on the "asset-transfer-ledger-queries" contract provided with the test network. The smart contract can perform rich queries on assets stored in the ledger by using CouchDB as the state database.

The seemingly innocuous `ChangeColorByOwner` function seen in listing 1 introduces the vulnerability into the contract. The `color` attribute is changed on line 10, and the asset

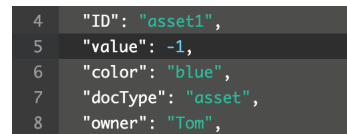
is written back to the ledger on line 12. It is not checked whether the asset obtained from CouchDB is consistent with the latest committed ledger value (or readset), which is exploitable.

```
1 func (t *SimpleChaincode) ChangeColorByOwner(ctx, owner string,
2   color string) error {
3   queryString := fmt.Sprintf(`{
4     "selector":{"docType":"asset","owner":"%s"}`, owner)
5
6   // Call wrapper method for getQueryResult()
7   result, _ := getQueryResultForQueryString(ctx, queryString)
8
9   // Change color of all assets and write back the change
10  for _, asset := range result {
11    asset.Color = color
12    assetBytes, _ := json.Marshal(asset)
13    ctx.GetStub().PutState(asset.ID, assetBytes)
14  }
15  return nil
16 }
```

Listing 1: `ChangeColorByOwner` changes the color of all assets owned by `owner` to the specified color.

Exploit: Illegal value propagation

The default access controls provided with CouchDB are poor [15]. If these are not changed, a malicious actor on the network could access a peer's CouchDB instance with little difficulty. The attacker could then directly modify and control the world state perceived by that peer without invoking any ledger transactions. Thus, it is difficult for the peer to know whether its state has been changed since there is no record of the changes [33].



```
4  "ID": "asset1",
5  "value": -1,
6  "color": "blue",
7  "docType": "asset",
8  "owner": "Tom",
```

Figure 3: The value has been modified from 300 to -1 through the CouchDB GUI accessible on `localhost:5984/_utils`.

When combined with the vulnerability presented in listing 1, illegal changes made in the state database can propagate to the ledger. For the propagation to succeed, the attacker must modify the state database of a sufficient number of endorsing peers to pass consensus; to simplify implementation, the vulnerable contract had only one endorsing peer. To reproduce this exploit, the ledger was first initialized with assets. An asset to exploit was then chosen, see figure 3. The original value of this asset was modified from 300 to -1 in the peer's state database. Although the smart contract contains no functions to modify the value attribute, invoking the vulnerable `ChangeColorByOwner` function allows the illegal change to propagate to the ledger. As a result, `ChangeColorByOwner` not only changed the color of the asset, but it also inadvertently changed the value, see listing 2.

```
1 // 1. Query asset1
2 $ peer chaincode query ... -c '{"Args":["ReadAsset","asset1"]}'
3   -> [{"ID":"asset1","color":"blue","owner":"Tom",
4     "value":300}]
5
6 // 2. Change the value of asset1 to -1 in state database
```

¹<https://github.com/CathrinePaulsen/rp-chaincode>

```

7
8 // 3. Change color of all assets owned by Tom
9 $ peer chaincode invoke ... -c
  '{"Args":["ChangeColorByOwner","Tom","black"]}'
10 -> Chaincode invoke successful. result: status:200
11
12 // 4. ReadAsset verifies that the value of asset1 has changed on
  the ledger
13 $ peer chaincode query ... -c '{"Args":["ReadAsset","asset1"]}'
14 -> [{"ID":"asset1","color":"black","owner":"Tom",
15   "value":-1}]

```

Listing 2: Step-by-step results of the illegal value propagation exploit.

Countermeasures

Literature discussing the vulnerability suggest rich query methods like `GetQueryResult` only be used for query transactions [14, 32], including Fabric’s own documentation [22], because the query results are not verified in the validate phase. Besides avoiding rich queries within update transactions altogether, the following **design pattern** can be adopted:

1. Use rich query to retrieve the appropriate keys from the state database.
2. Use a safe key-based query (e.g. `GetState`) to retrieve the latest committed values from the ledger.

See listing 3 for an implementation of this pattern. This pattern was based on community discussions on the Hyperledger Fabric forums [34]. Redoing the steps in listing 2 with the safer update function could not reproduce the illegal value change, so the design pattern is indeed effective at preventing this exploit.

```

1 func (t *SimpleChaincode) ChangeColorByOwnerFIXED(ctx, owner
  string, color string) error {
2   queryString := fmt.Sprintf(`{
3     "selector":{"docType":"asset","owner":"%s"}`, owner)
4   // Call wrapper method for getQueryResult()
5   result, _ := getQueryResultForQueryString(ctx, queryString)
6
7   // Change color of all assets and write back the change
8   for _, asset := range result {
9     key := asset.ID
10    // Safe key-based query
11    assetBytes, _ := ctx.GetStub().GetState(key)
12    json.Unmarshal(assetBytes, asset)
13    asset.Color = color
14    assetBytes, _ := json.Marshal(asset)
15    ctx.GetStub().PutState(asset.ID, assetBytes)
16  }
17  return nil
18 }

```

Listing 3: Safer implementation of the `ChangeColorByOwner` function following the proposed design pattern.

However, it is important to note that even though the values are retrieved from the ledger, the keys are still retrieved from the rich query result. If an attacker can modify the state database, he can also control the outcome of the rich query. Although no illegal value changes are propagated, the attacker could exclude or include specific assets in the query results. While a simple if-statement can be added to the smart contract to check if an asset passes the query, it is much more difficult to check if certain assets have been excluded without doing an additional iteration over all assets. However, this

defeats the purpose of using rich queries as it significantly impacts performance. Instead, adopting a stronger endorsement policy and more secure access controls could alleviate this problem, but these changes are not related to the smart contract itself.

4.2 Pseudorandom number generator

A secure random number generator (RNG) should be unpredictable [35], whereas the outcome of invoking a smart contract must be deterministic. These conflicting properties make implementing secure RNGs in smart contracts a challenging problem [36]. A truly unpredictable RNG generates a new random number every time it is called. As a result, unpredictable RNG cannot be used in smart contracts; every endorsing peer will calculate a different random number and the contract will be non-deterministic. To prevent non-determinism, the outcome of the RNG must be predictable across all peers, for example by using a pseudorandom number generator (PRNG). However, this makes the contract vulnerable to exploitation.

PRNGs generate number sequences by following a deterministic algorithm: Given the same input, or seed, the sequence output by the PRNG is the same every time it is run [37]. Since all peers need access to the same seed, it must be available on the blockchain. However, then it is also available to a potential attacker. If an attacker knows the seed used by the PRNG, he can predict the outcome and exploit the contract.

This PRNG vulnerability was implemented into a simple lottery smart contract, see listing 4. The smart contract generates a random number using the `math/rand` PRNG seeded with the transaction timestamp on line 5, which is the vulnerable feature of this contract. The number is hashed and stored on the ledger for future reference. The participants can then invoke the contract to try and guess the random number to win the lottery.

```

1 func (s *SmartContract) generateNewWin(ctx) error {
2   // Generate and hash random number
3   timestamp, _ := ctx.GetStub().GetTxTimestamp()
4   rand.Seed(timestamp.GetSeconds())
5   randomNumber := rand.Int()
6   hashedRandomNumber := ...
7   // Create and store new winning number
8   asset := Asset{
9     ID: "current_win",
10    Number: hashedRandomNumber,
11    Won: false,
12  }
13  assetBytes, _ := json.Marshal(asset)
14  ctx.GetStub().PutState(asset.ID, assetBytes)
15  return nil
16 }

```

Listing 4: Function used by the vulnerable lottery contract to generate the new winning number using a PRNG.

Exploit: Predicting the outcome

Both the transaction timestamp and the smart contract are available to everyone on the channel. A malicious participant is therefore able to re-calculate the random number and predict the lottery with the information available on the ledger. To reproduce this scenario, the ledger was inspected using

Hyperledger Explorer (v1.1.5), see figure 4. This can also be done via the CLI, but it may be difficult to navigate to the relevant information. Although the winning number itself is hidden, the timestamp used to calculate the number is plainly visible. The attacker can then use the same PRNG method to re-calculate the winning number and invoke the contract to win the lottery.

```
Time: 2021-06-06T13:24:29.288Z
Writes:
  root: [] 2 items
    set: [] 1 item
      0: {} 3 keys
        key: "current_win"
        is_delete: false
        value: "{\"id\":\"current_win\",\"number\":\"158aa5e022a47c1750\",\"won\":false}"
```

Figure 4: Although the winning number is hidden, inspecting the ledger reveals the transaction timestamp used to generate it.

Countermeasures

Because randomness on blockchains is its own complex research topic, this section will only present a few of the main ideas used for blockchain RNG.

Passing seed as input. Transient data is not recorded on the ledger, so passing the seed as transient input data allows using a PRNG while keeping the seed secret from the blockchain [25]. However, this only moves the security issue from the smart contract to the invoking client as the client becomes a single point of failure. If the client does not generate and store the seed securely or becomes compromised, the smart contract is also compromised.

Centralized oracles. Calculating the random number off-chain by a centralized oracle using a high entropy function is another option. The smart contract can then request the random number from the oracle. The oracle is not restricted by the same non-deterministic constraints as the smart contract, and can therefore generate truly random numbers. Possible external oracles compatible with Fabric include Provable and Gardener. The downside to this solution is that control over the random number calculation is lost, and the solution relies on the third-party oracle being trusted and secure since it can fully control the outcome of the RNG. This defeats some of the purpose of using smart contracts since blockchain technology was created partly to avoid relying on (centralized) trusted third-parties [38]. As with the previous countermeasure, if the oracle is compromised, then so is the smart contract.

Decentralized oracles. Distributed RNGs (or decentralized oracles) like RANDAO, Bitcoin Beacon [35] or the commit-reveal design scheme [39] address the compromised oracle problem above. Random numbers are generated based on random input from several participants. The input is then combined in some deterministic way to create the final random number. The advantage of this approach is that although all participants can influence the outcome, no single participant can fully control it. While RANDAO and the commit-reveal scheme are implemented in Ethereum context, the underlying ideas could still be implemented in Fabric smart contracts. For example, every endorsing peer of the RNG con-

tract can use a high entropy function to generate a random number stored in private data. Endorsing peers of the lottery contract can then retrieve and combine the numbers to create the winning lottery number. The issues related to RANDAO and the commit-reveal scheme due to incentivization [39] and anonymity in Ethereum are not present in Fabric since all participants are known and there is no native currency or concept of gas.

4.3 Global variables

Global variables are easy to implement and their global scope can be useful; they can be accessed from any function or used to keep track of some global state. However, they become problematic when used in smart contracts and are a commonly reported source of non-determinism [13, 14, 16, 32]. A malicious actor can exploit global variables to make the contract non-deterministic or behave unexpectedly.

See listing 5 for the implementation of the vulnerability. The global variable `totalAssets` keeps track of the number of assets added by the function `CreateAsset`. This variable is then used to calculate the value of the asset. Finally, the asset is added to the ledger if it does not already exist.

```
1 var totalAssets = 0 // Global variable
2 func (s *SmartContract) CreateAsset(ctx, id string, owner string)
   error {
3     totalAssets++
4     asset := Asset{
5         ID: id,
6         Owner: owner,
7         Value: totalAssets / maxAssets * 100,
8     }
9     assetJSON, _ := json.Marshal(asset)
10    if exists, _ := s.AssetExists(ctx, id); exists {
11        return fmt.Errorf("asset already exists with id: %v", id)
12    } else {
13        return ctx.GetStub().PutState(asset.ID, assetJSON)
14    }
15 }
```

Listing 5: The `totalAssets` variable is updated every time an asset is created by `CreateAsset`. The value of an asset is based on this variable.

Exploit: Causing non-determinism

As explained in section 2, it is the client's responsibility to send the invocation requests to the peers; the client is not required to send the request to all peers. This fact was exploited to make the contract non-deterministic, see listing 6.

First, the client sends a transaction proposal to invoke the `CreateAsset` function to only one peer. Since the endorsement policy requires the endorsement of all peers, the transaction fails with an endorsement policy error. Although the transaction was rejected, the values stored in the global variables are now different across the peers. The peer that received the transaction proposal has updated its global variables, causing the contract to be non-deterministic. All future update transactions will fail to pass consensus and the malicious caller has successfully put the contract out of service.

```
1 // 1. Invoke update transaction on only one peer
2 $ peer chaincode invoke <peerA> -c
   '{"Args":["CreateAsset","4","Alice"]}'
3 $ docker logs -f <peerA>
4   -> err validation of endorsement policy
```

```

5
6 // 2. Invoke update transaction on all peers
7 $ peer chaincode invoke ... -c
  '{"Args":["CreateAsset","4","Alice"]}'
8 -> Error: could not assemble transaction

```

Listing 6: Step-by-step results of the non-determinism exploit. The first invocation is only sent to one peer, which causes all subsequent invocations to fail.

Exploit: Causing unexpected behavior

The values stored in global variables persist across transactions, and there is no distinction between valid and invalid transactions. Any changes made in an invalid transaction are persisted in the global variable, which can be exploited to make the contract behave unexpectedly as shown in listing 7.

The malicious caller sends a transaction proposal to all peers to create an asset with an id that already exists. This transaction fails and returns an error. However, the `totalAssets` global variable was changed before exiting the `CreateAsset` function, and this new value will persist to the next invocation. The caller can then send another transaction proposal, this time with a valid id. As can be seen in the output of the query, the value is 400 instead of 300. Thus, the malicious caller has successfully corrupted the next transaction, causing the contract to behave unexpectedly.

```

1 // 1. Query ledger after initialization
2 $ peer chaincode query ... -c '{"Args":["GetAllAssets"]}'
3 -> [{"ID":"1","owner":"Tomoko","value":100},
4     {"ID":"2","owner":"Brad","value":200},
5
6 // 2. Invoke invalid transaction
7 $ peer chaincode invoke ... -c
  '{"function":"CreateAsset","Args":["1","Alice"]}'
8 -> Error: asset already exists with id: 1
9
10 // 3. Invoke valid transaction and query the result
11 $ peer chaincode invoke ... -c
  '{"function":"CreateAsset","Args":["3","Alice"]}'
12 $ peer chaincode query ... -c '{"Args":["GetAllAssets"]}'
13 -> [{"ID":"1","owner":"Tomoko","value":100},
14     {"ID":"2","owner":"Brad","value":200},
15     {"ID":"3","owner":"Alice","value":400}]

```

Listing 7: Step-by-step results of the unexpected behavior exploit. A rejected transaction caused the next accepted transaction to have a higher value than expected.

Countermeasures

In the unexpected behavior example, the solution is obvious: The exploit can be prevented by adding **proper error-handling** to ensure that the global variable is only modified in valid transactions. However, in more involved smart contracts, it is easy to lose track of when and where the global variable was last modified. This makes preventing unexpected behavior due to global variables increasingly difficult.

While proper error-handling can prevent global variables from causing unexpected behavior, the non-deterministic issue cannot be fixed without completely avoiding them. Unlike the previous vulnerabilities described in sections 4.1 and 4.2, safer alternatives can easily replace global variables. For example, state database queries can replace global variables used to persist values across transactions to minimize performance loss. Global variables can still be used to share

variables across function calls, but they must be properly reset and initialized on every transaction; a safer option is to use the function parameters to pass local variables between functions.

4.4 Analysis tools

To apply the suggested countermeasures to the discussed vulnerabilities, a developer would first need to know whether the vulnerability is present. While an alert developer may detect these vulnerabilities during development, they can be easy to miss. Automated tooling should therefore be used to detect vulnerabilities the developer did not catch. This section will discuss the various tools applicable to Fabric smart contracts.

Compared to Ethereum, this study found few analysis tools compatible with Fabric smart contracts. In total, four static code analysis tools (ReviveCC, Chaincode Scanner, Chaincode Analyzer [14] and Lv et al. [32]) and two formal verification tools (Zeus [40] and Beckert et al. [41]) were found. Only two of the six tools were open-source, ReviveCC and Chaincode Analyzer. Of these two, this study only managed to run ReviveCC. Chaincode Analyzer was last updated in February 2020 and is likely currently incompatible with the newer Fabric versions.

According to the literature reviewed on the static code analysis tools, all of them except ReviveCC can detect all three vulnerabilities, see table 1. However, only the claims of ReviveCC could be verified. Furthermore, it is unclear whether Chaincode Scanner can detect the rich queries vulnerability; it can according to [14], but not according to the ReviveCC repository.

Both formal verification tools are working prototypes, but no source code or product was available, nor were any other papers or applications using them found. They also cannot verify contract behavior as a result of multiple invocations [40,41]. Hence, it is unclear whether they can prevent the discussed vulnerabilities since the exploits do not occur within a single transaction.

Table 1: Overview of static analysis tools compatible with Hyperledger Fabric smart contracts and which of the three vulnerabilities they are able to detect. Entries marked with * are verified.

	Global variables	PRNG	Rich queries	Open source
ReviveCC	yes*	yes*	no*	yes
Chaincode Scanner	yes	yes	yes	no
Chaincode Analyzer	yes	yes	-	yes
Lv et al.	yes	yes	yes	no

4.5 Impact severity

The following section discusses and compares the vulnerabilities' impact severities, and how well the proposed countermeasures are able to lower them. This study uses impact severity to measure the impact the exploitation of a vulnerability would have on the blockchain. The impact severity is derived from the base CVSS score for each vulnerability and is listed in table 2. These scores are not official numbers,

but rather estimations made for this study indicate the impact severity of the vulnerabilities.

The base score is calculated by considering different aspects of the vulnerability distributed over two categories: *ease of exploitation* and *impact*. The ease of exploitation is reflected by the following metrics: *attack vector*, *attack complexity*, *privileges required*, *user interaction* and *scope*. Only the first two are relevant for the comparisons in this section. The impact is expressed through *confidentiality* (leaking sensitive data), *integrity* (illegal modifications) and *availability*. The exact definitions of these metrics are explained in [31].

Table 2: The base CVSS scores of the three vulnerabilities explored in this study. The base score translates into an impact severity of low, medium or high.

	Global variables	PRNG	Rich queries
Base score	8.2	4.3-6.5	5.3
Impact severity	high	medium	medium
Attack vector	network	network	adjacent
Attack complexity	low	low	high
Privileges required	low	low	none
User interaction	none	none	none
Scope	unchanged	unchanged	unchanged
Confidentiality	none	low-high	none
Integrity	low	none	high
Availability	high	none	none

Although global variables are perhaps the simplest to both introduce and avoid out of the three vulnerabilities, it also has the highest impact severity. This is largely due to the high ease of exploitation, and the exploits affecting both the integrity and availability of the contract. Comparatively, both the PRNG and rich query exploits only affect one of the impact metrics: confidentiality and integrity, respectively. The rich queries vulnerability can severely impact the integrity of a contract, however the overall impact severity is lower than the global variables because the attack complexity is much higher. The PRNG vulnerability has the same ease of exploitation as the global variables, but it only affects the contract’s confidentiality. The impact on confidentiality also depends on the specific contract and how sensitive the random numbers are. In the case of a lottery application, for example, the impact on confidentiality will be high.

Countermeasures. For the global variables vulnerability, only the unexpected behavior exploit can be prevented by implementing proper error-handling. Although this countermeasure would lower the overall impact severity by eliminating the impact on integrity, the impact on availability remains high since the non-deterministic exploit cannot be removed. Because the impact severity cannot be lowered to an acceptable level, global variables should always be avoided.

A two-step design pattern was proposed as a countermeasure for the rich query vulnerability. This countermeasure would lower the impact severity by lowering the impact on integrity. Although the risk to integrity is not removed, the attacker can no longer control the outcome of the exploit.

Because the attack complexity of the exploit is so high, the remaining impact severity is relatively low.

All of the proposed countermeasures for the PRNG vulnerability involve moving the random number generation outside of the contract: either into a centralized or decentralized oracle, or by passing the seed as input. Unfortunately, the latter countermeasure has the same issue as the centralized oracle: The impact severity of the exploit is then dependent on the security of a single point of failure and is therefore hard to estimate. It is, however, safe to assume that the general impact severity goes down because the attack complexity goes up; originally, any user with access to the blockchain could exploit the vulnerability.

5 Responsible Research

Both the ethical aspects and the reproducibility of the research methods were taken into consideration during the research for this paper. This section will discuss those aspects to ensure that the ethical impact of the paper poses no significant risks to individuals or society as a whole, and that the results are reproducible for future research.

5.1 Ethical aspects

The purpose of this study is solely to report on known exploits and vulnerabilities to facilitate secure smart contract development. The author of this paper has no personal affiliations with or stake in any blockchain or smart contract technology that could result in a conflict of interest. Furthermore, the vulnerabilities were implemented in smart contracts deployed in a private, local test environment. Therefore, no exploitation was done with real-world implications or interference with existing systems. This study also did not make use of any personal data, and no human test subjects were involved in any way.

The paper provides code snippets and descriptions of vulnerabilities and their exploitations to increase the reproducibility of the research; however, this information could potentially be used by a malicious actor to attempt exploitation of similar vulnerabilities in real-world systems or work around known countermeasures. Nevertheless, the vulnerabilities, exploits and countermeasures described in this study are already known and reported in other literature. A determined attacker could therefore find similar information elsewhere; hence, this study poses minimal additional risk to individuals or systems affected by similar technology. In fact, this survey could help smart contract developers become aware of how their systems can be made less vulnerable to exploitation.

5.2 Reproducibility

To ensure that the results of this research is reproducible for future experiments and research, code snippets of relevant code sections are included to show the relevant aspects of each vulnerability and exploit. The snippets are relatively small and specific, making them straightforward to implement in terms of software engineering. With basic knowledge of the Hyperledger Fabric platform and Go language, these code snippets can in theory be introduced into any compatible contract. Some superfluous code such as error handling

has been left out of the snippets for brevity, but the full code is available on Github². The countermeasures may be more difficult to reproduce since they are mostly theoretical suggestions and not all of them are fully implemented in this research. However, the countermeasures are based on existing implementations and research, which can be verified by checking the cited sources.

6 Conclusions and future work

This study managed to find and reproduce three security vulnerabilities in Hyperledger Fabric-based smart contracts: global variables, updates using rich queries and pseudorandom number generators. A number of countermeasures were explored and proposed for each vulnerability. The unexpected behavior exploit of global variables can be mitigated by proper error-handling, but this countermeasure does not scale well for larger and more complex contracts. The non-determinism exploit cannot be prevented by the smart contract. It is therefore better to avoid global variables altogether by replacing them with other suitable alternatives. Luckily, global variables are relatively easy to replace. The rich query and PRNG vulnerabilities, on the other hand, provide functionality that is difficult to replace. The proposed countermeasures can effectively reduce the impact severity, but there is still some risk involved. For the rich queries vulnerability, the risk can be removed but at the cost of significant loss of performance. A developer should therefore carefully consider whether the risk is acceptable for their use case and whether the functionality is necessary for the correct functioning of the contract.

To implement countermeasures for a vulnerability, the developer first needs to be aware of the vulnerability. It is therefore recommended to always use some kind of analysis tool during development. For these particular vulnerabilities, static code analysis tools work well. Although few available tools were found, more general tools like Revive are highly customizable and a developer could create their own custom rules to warn against global variables, rich query methods and PRNGs relatively easily.

Future work. First, only a small subset of vulnerabilities could be explored due to time constraints. Future work could continue exploring other vulnerabilities in similar detail. Second, for the blockchain RNG problem, verifiable random functions (VRF) like Chainlink are available for Ethereum. Future work could explore how VRF can be used in Fabric. Third, because Fabric is a permissioned blockchain, it is unclear how likely the exploits presented in this study are to occur in practice. Future work could therefore investigate the exploits in a more realistic setting. Finally, there are currently few open-source analysis tools applicable for Fabric smart contracts, so future work on developing open-source analysis tools would greatly benefit the developer community.

References

[1] S. Nakamoto. (2008) Bitcoin: A peer-to-peer electronic cash system. (Retrieved 2021-05-05). [Online]. Available: <https://bitcoin.org/bitcoin.pdf>

²<https://github.com/CathrinePaulsen/rp-chaincode>

- [2] S. Haber and W. S. Stornetta, "How to time-stamp a digital document," in *Conference on the Theory and Application of Cryptography*. Springer, 1990, pp. 437–455.
- [3] N. Szabo, "Formalizing and securing relationships on public networks," *First monday*, 1997.
- [4] 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.
- [5] V. Gatteschi, F. Lamberti, C. Demartini, C. Pranteda, and V. Santamaria, "To blockchain or not to blockchain: That is the question," *IT Professional*, vol. 20, no. 2, pp. 62–74, 2018.
- [6] 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. 100179, 2021.
- [7] T. Hewa, M. Ylianttila, and M. Liyanage, "Survey on blockchain based smart contracts: Applications, opportunities and challenges," *Journal of Network and Computer Applications*, p. 102857, 2020.
- [8] B. K. Mohanta, S. S. Panda, and D. Jena, "An overview of smart contract and use cases in blockchain technology," in *2018 9th International Conference on Computing, Communication and Networking Technologies (IC-CCNT)*. IEEE, 2018, pp. 1–4.
- [9] 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.
- [10] Cryptopedia Staff. (2021) What was the DAO? (Retrieved 2021-04-23). [Online]. Available: <https://www.gemini.com/cryptopedia/the-dao-hack-makerdao>
- [11] 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.
- [12] C. Ferris. (2020) Hyperledger achieves huge milestone: Introducing Hyperledger Fabric 2.0. (Retrieved 2021-06-22). [Online]. Available: <https://www.ibm.com/blogs/blockchain/2020/01/hyperledger-achieves-huge-milestone-introducing-hyperledger-fabric-2-0/>
- [13] Y. Huang, Y. Bian, R. Li, J. L. Zhao, and P. Shi, "Smart contract security: A software lifecycle perspective," *IEEE Access*, vol. 7, pp. 150 184–150 202, 2019.
- [14] K. Yamashita, Y. Nomura, E. Zhou, B. Pi, and S. Jun, "Potential risks of Hyperledger Fabric smart contracts," in *2019 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2019, pp. 1–10.
- [15] A. Dabholkar and V. Saraswat, "Ripping the fabric: Attacks and mitigations on Hyperledger Fabric," in *International Conference on Applications and Techniques in Information Security*. Springer, 2019, pp. 300–311.

- [16] S. Brotsis, N. Kolokotronis, K. Limniotis, G. Bendiab, and S. Shiaeles, “On the security and privacy of Hyperledger Fabric: Challenges and open issues,” in *2020 IEEE World Congress on Services (SERVICES)*. IEEE, 2020, pp. 197–204.
- [17] B. Putz and G. Pernul, “Detecting blockchain security threats,” in *2020 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 2020, pp. 313–320.
- [18] Hyperledger. (2020) Introduction. (Retrieved 2021-04-29). [Online]. Available: <https://hyperledger-fabric.readthedocs.io/en/release-2.2/whatis.html>
- [19] ——. (2020) Key concepts: Ledger. (Retrieved 2021-06-10). [Online]. Available: <https://hyperledger-fabric.readthedocs.io/en/release-2.2/ledger/ledger.html>
- [20] ——. (2020) Key concepts: Introduction. (Retrieved 2021-06-10). [Online]. Available: <https://hyperledger-fabric.readthedocs.io/en/release-2.2/blockchain.html>
- [21] Hyperledger Fabric. (2020) Writing your first chaincode. (Retrieved 2021-06-23). [Online]. Available: <https://hyperledger-fabric.readthedocs.io/en/release-2.2/chaincode4ade.html>
- [22] Hyperledger. (2020) CouchDB as the state database. (Retrieved 2021-05-31). [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.2/couchdb_as_state_database.html
- [23] ——. (2020) Using CouchDB. (Retrieved 2021-06-01). [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.2/couchdb_tutorial.html
- [24] C. Olsson and M. Toorani, “A permissioned blockchain-based system for collaborative drug discovery,” in *ICISSP*, 2021, pp. 121–132.
- [25] Hyperledger. (2020) Key concepts: Private data. (Retrieved 2021-06-06). [Online]. Available: <https://hyperledger-fabric.readthedocs.io/en/release-2.2/private-data/private-data.html>
- [26] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich *et al.*, “Hyperledger Fabric: a distributed operating system for permissioned blockchains,” in *Proceedings of the thirteenth EuroSys conference*, 2018, pp. 1–15.
- [27] Hyperledger. (2020) Transaction flow. (Retrieved 2021-06-10). [Online]. Available: <https://hyperledger-fabric.readthedocs.io/en/release-2.2/txflow.html>
- [28] TU Delft Library. (2021) Making a search plan. (Retrieved 2021-05-24). [Online]. Available: <https://tulib.tudelft.nl/searching-resources/making-a-search-plan/>
- [29] Hyperledger. (2020) Using the Fabric test network. (Retrieved 2021-05-24). [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.2/test_network.html
- [30] ——. (2020) Peer (commands reference). (Retrieved 2021-06-07). [Online]. Available: <https://hyperledger-fabric.readthedocs.io/en/release-2.2/commands/peercommand.html>
- [31] Forum of Incident Response and Security Team. (2021) Common vulnerability scoring system v3.0: Specification document. (Retrieved 2021-06-15). [Online]. Available: <https://www.first.org/cvss/v3.0/specification-document>
- [32] P. Lv, Y. Wang, Y. Wang, H. Wang, and Q. Zhou, “Potential risk detection system of Hyperledger Fabric smart contract based on static analysis,” EasyChair, Tech. Rep., 2021.
- [33] S. Riedesel, P. Hakimian, K. Buyens, and T. Biehn. (2018) Tineola: Taking a bite out of enterprise blockchain. (Retrieved 2021-06-04). [Online]. Available: <https://github.com/tineola/tineola/raw/master/docs/TineolaWhitepaper.pdf>
- [34] Hyperledger. (2020) Validating transactions including rich queries (forum post). (Retrieved 2021-05-24). [Online]. Available: <https://lists.hyperledger.org/g/fabric/topic/validating-transactions/74932404?p=>
- [35] Y. Jo and C. Park, “BlockLot: Blockchain based verifiable lottery,” *arXiv preprint arXiv:1912.00642*, 2019.
- [36] N. Amiet, “Blockchain vulnerabilities in practice,” *Digital Threats: Research and Practice*, vol. 2, no. 2, pp. 1–7, 2021.
- [37] Golang.org. Package math/rand. (Retrieved 2021-06-05). [Online]. Available: <https://golang.org/pkg/math/rand/>
- [38] H. Al-Breiki, M. H. U. Rehman, K. Salah, and D. Svetinovic, “Trustworthy blockchain oracles: review, comparison, and open research challenges,” *IEEE Access*, vol. 8, pp. 85 675–85 685, 2020.
- [39] K. Chatterjee, A. K. Goharshady, and A. Pourdamghani, “Probabilistic smart contracts: Secure randomness on the blockchain,” in *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 2019, pp. 403–412.
- [40] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, “ZEUS: Analyzing safety of smart contracts.” in *Ndss*, 2018, pp. 1–12.
- [41] B. Beckert, M. Herda, M. Kirsten, and J. Schiffel, “Formal specification and verification of Hyperledger Fabric chaincode,” in *3rd Symposium on Distributed Ledger Technology (SDLT-2018) co-located with ICFEM*, 2018.

A Asset example

```

1 type Asset struct {
2     ID          string
3     Color       string
4     Owner       string
5     Value       int
6 }

```

Listing 8: The asset used by the smart contracts implementing the rich query and global variables vulnerabilities.