Potential Risks of Hyperledger Fabric Smart Contracts

Kazuhiro Yamashita*, Yoshihide Nomura*, Ence Zhou†, Bingfeng Pi†, and Sun Jun†
* FUJITSU LABORATORIES LTD., Japan; {y-kazuhiro, y.nomura}@fujitsu.com
† FUJITSU Research & Development Center, China; {zhouence, winter.pi, sunjun}@cn.fujitsu.com

***Abstract*—Blockchain is a decentralized ledger technology, and it is the technology underlying Bitcoin and Ethereum. The interest in blockchain has been increasing since its emergence. Hyperledger Fabric is one of the permissioned blockchain frameworks. One of the characteristics of Hyperledger Fabric is it utilizes general-purpose programming languages, e.g., Go, Node.js, and Java, to implement smart contracts (called chaincode in Hyperledger Fabric). The advantages of utilizing these languages are already known to potential developers, and development tools might already exist. However, one of the disadvantages is that these languages were not originally designed for writing smart contracts. Hence, there may be risks that developers do not need to consider when using specific languages such as Solidity of Ethereum. Furthermore, even though development tools exist, how many risks are covered by the tools is an open question. In this paper, we focus on Go language and the tools. First, we surveyed what kind of risks are associated with chaincodes are developed using Go language and observed there are 14 potential risks. Then, we investigated how many risks can be covered by Go tools, e.g., *golint* and *gosec*, and a vulnerability detection tool for chaincodes called *Chaincode Scanner*. From our results, we observed that some risks are not covered by the existing tools. Hence, we develop a detection tool to cover risks by static analysis. Finally, in this paper, we describe how to find the risks with our tool and evaluate the usefulness.**

***Index Terms*—Smart Contract, Validation Tool, Blockchain, Hyperledger Fabric**

I. Introduction

Blockchain technology (also referred as distributed ledger technology) is a decentralized ledger technology, and it is the technology underlying Bitcoin and Ethereum. The interest in blockchain has been increasing since its emergence because of characteristics such as immutability, transparency, and privacy. There are two types of blockchains: public and permissioned blockchains. In public blockchain systems, such as Bitcoin and Ethereum, any party can participate and open any information stored in the blockchain network. From the point of view of enterprises, the features of public blockchains are not suitable for enterprise usages, for instance, dealing with customers' sensitive data. Therefore, some parties have developed blockchain frameworks such that owners of the blockchain network can control access (e.g., Hyperledger Fabric[1] by the Linux Foundation and Corda[2] by the R3 CEV).

One prominent feature of blockchains is the exchange of digital assets such as Bitcoin [27]. Some blockchains provide an environment to execute smart contracts. Smart contracts are computer programs that enforce the execution of a contract based on consensus between untrusted parties. Smart contracts are triggered through transactions on a blockchain network.

As smart contracts deal with digital assets or cryptocurrencies, defects in smart contracts lead to serious problems or attacks by malicious users such as The DAO[3] and Parity wallet[4] cases. In the DAO case, the attacker exploited a vulnerability, which is called *Reentrancy*, and stole about 60 million US dollars from the DAO. The Parity case caused the freezing of about 150 million US dollars.

To prevent such a situation, some researchers [9, 20] have pointed out the requirement of blockchain-oriented software engineering. In particular, they presented testing [6], tools for smart contracts [13, 16, 17, 24], best practices and development methodology [14], metrics [23], and design patterns [3, 15] as research directions for blockchain-oriented software engineering.

A permissioned blockchain controls access to the blockchain networks. This suggests that permissioned blockchains are safer than public ones. Even in the safer situation, developing smart contracts without vulnerabilities is one of the major concern for developers. However, there are only few papers [7, 26] focused on the risks associated with permissioned blockchain smart contracts, although there are many studies on the risks associated with public blockchain smart contracts [2, 16, 17].

In this paper, we focus on *Hyperledger Fabric* [1], which is one of the most popular permissioned blockchain frameworks and was developed by the Linux foundation. Blockchain developers can develop smart contracts (called as *chaincode* in Hyperledger Fabric community) using general-purpose programming language, e.g., Go, Node.js, and Java. Typically, smart contracts are developed using domain-specific languages (DSLs), which have specific restrictions and features for blockchain, but Hyperledger Fabric applies general-purpose programming languages to reduce learning costs for developers. Therefore, because of the absence of specific restrictions and features, the types of risks associated with Hyperledger Fabric may be different from these associated with the smart contracts written by DSLs. Furthermore, for Ethereum and

[1]https://www.hyperledger.org/projects/fabric
[2]https://www.corda.net/

[3]https://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413\#code
[4]https://github.com/paritytech/parity-ethereum/issues/6995\#issuecomment-342409816

other frameworks, some tools have been developed to detect smart contract risks. On the other hand, for Hyperledger Fabric, to the best of our knowledge, there is no such tool although there are some tools for development.

In this study, we consider the case of implementing smart contracts using Go because Hyperledger Fabric has provided a software development kit (SDK) for Go from the early stage of the project, and the Go community has many tools (e.g., *golint* and *govet*) to improve code qualities. First, we surveyed and summarized the potential risks associated with Hyperledger Fabric smart contracts. Second, we considered the risks that can be solved or prevented using tools.

From the result of our study, we found that some risks cannot be solved by the existing tools. Hence, we implemented a prototype of static analysis tool for detecting such risks from smart contracts. Furthermore, we applied our tool to two applications developed in our company to evaluate the usefulness of our tool.

Paper organization. The rest of the paper is organized as follows. Section II discusses related work. Section III introduces Hyperledger Fabric. Section IV describes risks that developers need to consider. Section V shows how many risks can be covered by using the conventional tools in Go development. Section VI presents our tool to cover risks. Section VII discusses the results and our tool. Finally, in Section VIII, was draw our conclusions.

II. Related Work

A. Smart Contract Validation and Verification

Since defects on smart contracts cause problems with large impact and smart contracts are difficult to modify after being deployed on blockchain networks, practitioners and researchers have developed ways to improve the quality of smart contracts.

There are tools focused on verification of smart contracts such as *Oyente* [16], *Mythril* [17], *Securify* [24], and *ZEUS* [13]. Oyente is based on symbolic execution and can detect four kinds of problematic codes, i.e., reentrancy, transaction order dependence, timestamp dependence, and mishandled exceptions. Similar to Oyente, Mythril [17] performed symbolic execution of Ethereum virtual machine (EVM) bytecode. Although useful in some settings, these approaches come with some drawbacks; for example, they can miss critical violations because of under-approximation [24]. To address these drawbacks, Tsankov *et al.* [24] developed an automated verifier, called *Securify*. The key ideas of Securify are to define two kinds of patterns: (i) compliance patterns, which imply the satisfaction of the property and (ii) violation patterns, which imply its negation. To check these patterns, Securify symbolical analyzes the dependency graph of the smart contract. Karla *et al.* [13] proposed ZEUS, a framework to verify the correctness and validate the fairness of smart contracts. ZEUS leverages both abstract interpretation and symbolic model checking. Although ZEUS is basically for Ethereum, Karla *et al.* conducted a case study to show the versatility with other blockchain frameworks (e.g., Hyperledger Fabric). From the case study, they found that it is easy to extend ZEUS to other blockchain frameworks with only minor changes.

Hirai [12] is the first person who applied formal verification to smart contracts in Ethereum. In his paper, he defined EVM in Lem, a language that can be compiled for a few interactive theorem provers. Bhargavan *et al.* [4] proposed a formal verification framework for Ethereum smart contracts using $F*$. $F*$ is a functional programming language for program verification. In the framework, Solidity source codes and EVM bytecodes are translated to $F*$ codes and then are verified them. However, the framework is not completely implemented.

As another approach to make safer smart contracts, various DSLs for smart contracts have been proposed. Harz and Knottenbelt [11] surveyed DSLs and categorized them into three levels (i.e., high, intermediary, and low). For example, considering high-level languages, Solidity[5] is the most used smart contract language for Ethereum. Vyper[6] restricts instructions, e.g., finite loops and no recursive calls, to prevent other features such as inheritance and overloading. Flint[7] introduces the definition of function access and creates an asset type. This approach tries to reduce defects by restricting instructions or adding specific features. Even when developers use such specific languages for smart contracts, vulnerabilities may still exist. Therefore, developers who write smart contracts using general-purpose languages need to take care of their codes.

B. Studies on Hyperledger Fabric

With regard to Hyperledger Fabric, there are a few papers [7, 26] related to potential risks of smart contracts; there are also papers related to performance evaluation [22], comparison to public blockchain frameworks [19], and proposing of applications [25, 28].

Christidis *et al.* [7] suggested that chaincode developers should avoid non-deterministic chaincodes. Vukolić [26] also pointed out that non-deterministic operations in chaincodes may make the endorsement results from different peers inconsistent.

In enterprise usage, one of the most critical challenges is performance, such as throughput (transactions per second/tps) and latency. Thakkar *et al.* [22] first identified bottlenecks in a use case system that is built on Hyperledger Fabric and then optimized the performance. They observed that endorsement policy, sequential policy validation, and state validation and commit are the main bottlenecks. By minimizing each bottleneck, the throughput of the system improved by 16 times (i.e., from 140 tps to 2,250 tps).

At the stage of planning a business using blockchain technology, selecting blockchain framework is one of the major considerations. To acquire further knowledge on blockchain frameworks, Pongnumkul *et al.* [19] conducted a performance analysis of two private blockchain frameworks, Hyperledger Fabric and Ethereum. From the comparison, Hyperledger

[5]https://github.com/ethereum/solidity
[6]https://github.com/ethereum/vyper
[7]https://github.com/flintlang/flint

Fabric consistently outperforms Ethereum across evaluation metrics such as execution time, latency, and throughput. However, it was also found that both frameworks are still not competitive with the current database systems.

Some papers explore use cases of Hyperledger Fabric. Zhang *et al.* [28] proposed a trust data sharing framework for AI-powered network framework. They implemented the prototype based on Hyperledger Fabric to utilize the distributed and tamper-proof attributes of blockchain. Uchibeke *et al.* [25] developed a blockchain access control ecosystem using Hyperledger Fabric. The system provides asset owners the sovereign right to effectively manage access control of large data sets and protect against data breaches.

Although some papers suggest to avoid non-deterministic chaincode, there is no paper that explores factors of non-determinism of chaincodes and other risks. Hence, in this paper, we survey some kinds of artifacts and try to summarize the potential risks related to chaincode.

III. Hyperledger Fabric

Hyperledger Fabric [1] is an open source blockchain framework implementation. Fabric is one of the Hyperledger projects under the auspices of the Linux Foundation. Hyperledger Fabric released version 1.0 in July 2017, and version 1.3, the latest version, was released in October 2018. Because the architectures of Hyperledger Fabric released before and after version 1.0 (i.e., version 0.6 and after 1.0) are different [18], in this study, we only focus on the architecture of version 1.0 or newer versions.

Hyperledger Fabric is the first blockchain framework to apply general-purpose programming language to implement smart contracts (referred as `chaincode` in Hyperledger Fabric). While smart contracts of other popular blockchain frameworks such as Ethereum are implemented using DSL, chaincodes are implemented using general-purpose programming languages such as Go, Node.js, and Java; this is advantageous as developers are already familiar with standard programming languages, and verification tools might already exist [11]. On the other hand, the disadvantage of using these languages is that they were not originally designed for smart contracts; that is, there are no features or restrictions to write safe smart contracts, unlike DSLs earlier mentioned in Section II.

A. Architecture

To better understand potential risks in Hyperledger Fabric, we briefly present an overview of the architecture of Hyperledger Fabric. Hyperledger Fabric is mainly composed of the following three components.

1) Membership Service: Hyperledger Fabric is a permissioned blockchain framework. This means that the participants know each other. Hyperledger Fabric provides Membership Service Provider (MSP) to manage the participants of the network.

The identities of participants are issued by Certificate Authority (CA) using Public Key Infrastructures (PKI). Digital certificates, which are generated by PKI are linked with organizations, network components, and end users or client applications.

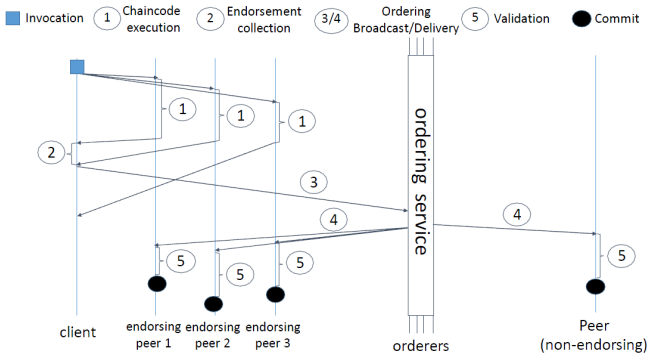

Fig. 1: Transaction Flow [1]

2) Ordering Service: In a Hyperledger Fabric blockchain network, there is a node called Orderer. Orderer sorts transactions that are passed from clients after endorsement in transaction flow. Then, Orderer broadcasts transactions to each peer on the network. Note that Orderer does not consider the content of transactions.

3) Peers: Peer is a fundamental element of a blockchain network; it hosts ledgers and chaincodes. In Hyperledger Fabric, there is an endorsement process in the transaction flow. Some peers work as endorsing peers; those peers simulate transaction proposals to validate the correctness of the proposals as a representative of peers. When a client application sends a transaction proposal, endorsing peers receive the proposal and independently simulate the chaincode of the proposal, then send the result back to the client. By limiting the number of peers to simulate proposals and endorse them, Hyperledger Fabric improves performance to produce blocks.

B. Transaction Flow

Here, we discuss the flow of processing a transaction (Fig. 1). Hyperledger Fabric introduces the *execute-order-validate* blockchain architecture. This architecture addresses the resiliency, flexibility, scalability, performance, and confidentiality challenges faced by the order-execute model.

1) Execution Phase: In the execution phase, clients sign and send the *transaction proposal* to one or more endorsing peers. The transaction proposal is a request to invoke a chaincode function. The endorsing peers verify four points: 1) the transaction proposal is well-formed, 2) it has not been submitted already in the past, 3) the signature is valid, and 4) the submitter (i.e., the client) is properly authorized to perform the proposed operation on the channel. If all the points are verified, endorsing peers take the transaction proposal inputs as arguments to the invoked chaincode's function. Then, the chaincode is executed against the current state database to return the results (*write set* and *read set*). Note that no updates are made to the ledger at this point. Finally, the endorsing peers send the results back to the client with the signatures of the endorsing peers (1st step in Fig. 1).

The client collects endorsements until the endorsement policy is satisfied (2nd step in Fig. 1).

2) Ordering Phase: When the client collects enough number of endorsements, the client assembles the proposal into a transaction and sends the transaction to the ordering service (3rd step in Fig. 1). Then, the ordering service orders the received transactions chronologically by channel, and creates blocks of transactions per channel. The blocks of transactions are delivered to all peers on the channel (4th step in Fig. 1).

3) Validation Phase: The blocks including transactions are delivered to all peers on the channel. In each peer, the transaction is validated to ensure the endorsement policy is satisfied and to ensure that there have been no changes to the current state of the ledger for read set variables, since the read set was generated by the transaction execution. Each peer appends the block to the chain, and write sets are committed to the current state database for each valid transaction (5th step in Fig. 1).

IV. Potential Risks in Hyperledger Fabric

In this section, we survey and summarize the risks involved when developers implement their business logic on Hyperledger Fabric. In the survey, we investigated some artifacts, e.g., research papers, discussion on GitHub issues[8], the official documents[9], and blogs[10], to identify the chaincode related potential risks that practitioners and researchers should be concerned. Table I summarizes the risks and the rationale. Furthermore, we show a brief example of source codes as well as the risks.

A. Non-determinism Arising From Language Instructions

Since chaincode is conducted in distributed and independent peers, determination of the business logic is a major concern [1, 7, 26]. Hence, some DSLs for smart contracts, such as Solidity, do not have instructions related to random number [7]. However, general-purpose programming languages generally have such instructions. If the chaincode is non-deterministic, it makes the endorsing result from different peers inconsistent [26]. A solution to filter non-deterministic operations on a blockchain has been proposed [5], but it appears costly in practice.

1) Global Variable: Not only in chaincode development but also in general development, developers need to consider global variables. Global variables can be changed inherently. Therefore, the usage of global variables might cause non-determinism of the chaincode.

2) Random Number Generation: Random number generation is one of the typical non-deterministic code. During the endorsing phase, endorsing peers independently simulate the chaincode as explained in Section III. Therefore, the results of simulation by endorsing peers may be inconsistent.

[8]e.g., https://github.com/hyperledger-archives/fabric/issues/1273

[9]e.g., https://hyperledger-fabric.readthedocs.io/en/latest/index.html

[10]e.g., https://gist.github.com/arnabkaycee/d4c10a7f5c01f349632b42b67cee46db

Listing 1: Example of Non-determinism

```
1  // Something like a lottery application
2  // Users predict a number
3
4  // User's prediction
5  pred := arg[0]
6
7  // Answer
8  rand.Seed(seed)
9  sel := rand.Intn(10)
10
11 if pred == sel {
12    PayPrizeToUser(user, prize)
13 }
```

Listing 1 shows an example of the non-deterministic risk. Lottery application is one of the typical blockchain applications because of the transparency of the result. In this example, users predict a number that is randomly generated, and if the prediction is correct, the user receives prize. Since the chaincode is simulated in each of the endorsing peers during the endorsing phase, the generated numbers are different in each peer.

Note that the seed of generating random number is fixed as 1 in Go. Hence, developers have to set the seed such as `rand.Seed(time.Now().UnixNano())` or use `crypto/rand` package.

3) System Timestamp: As in the case of random number generation, there is no guarantee to call timestamp functions at the same time in each endorsing peer.

4) Map Structure Iteration: Because of the specification of Go, the order of the key values when developers use iteration with map structure is not unique. Therefore, the usage of map structure iteration may cause non-determinism. Different from random number generation and timestamp, the behavior is characterized by hidden implementation details of Go. Developers who know much about Go specification or implementation can avoid introducing the risk, but the others may introduce it.

5) Reified Object Addresses: Developers can handle the value of the variables through a pointer. A pointer is an address of memory, and the address depends on the environment. Therefore, the usage of reified object addresses may cause non-determinism.

6) Concurrency of Program: Go has rich support for concurrency using *goroutine* and *channel*. If a concurrent program is not appropriately dealt with, a race condition problem will easily occur.

B. Non-determinism Arising From Accessing Outside of Blockchain

The following risks are also related to non-determinism but arise from accessing outside of the blockchain.

TABLE I: POTENTIAL RISKS ASSOCIATED WITH HYPERLEDGER FABRIC SMART CONTRACTS

Category	Risks	Rationale	Source
Non-determinism Arising From Language Instructions	Global Variable	Global variable can be changed inherently.	GitHub
	KVS Structure Iteration	Key value is returned in a random order.	GitHub, [26]
	Reified Object Addresses	Addresses of memory depend on the environment.	GitHub
	Concurrency of Program	Concurrency can cause non-deterministic behavior such as race condition.	GitHub
	Generating Random Number	Each peer obtains different result of generating random number.	GitHub, Blog, [26]
	System Timestamp	It is difficult to ensure timestamp functions are executed at the same time for each peer.	GitHub, Blog, [26]
Non-determinism Caused From Accessing Outside of Blockchain	Web Service	Need to ensure that the results of calling the web service are not different among peers.	GitHub, Blog
	System Command Execution	Need to ensure that the results of calling the system command are not different among peers.	GitHub
	External File Accessing	Need to ensure that the results of accessing the file are not different among peers.	GitHub
	External Library Calling	Need to consider the behavior of the external library.	GitHub
State Database Specification	Range Query Risk	Two functions to call range query cause phantom read of the ledger.	Document, Blog
Fabric Specification	Field Declarations	It is not guaranteed that every transaction is executed on every peer. Hence, the value of a variable declared at structure field may not sustain same value among peers.	ChaincodeScanner
	Cross Channel Chaincode Invocation	No data will be committed in the other channel.	Blog
	Read Your Write	Fabric does not support Read-Your-Write semantic.	Document
Common Practices	Unhandled Errors	Should not ignore return values related to errors.	Document, ChaincodeScanner
	Unchecked Input Arguments	Should check input arguments.	Document, ChaincodeScanner

1) Web Service: In the context of smart contract, when a business logic needs information from the outside of the blockchain, developers need to access an entity called "oracle," which provides data about the outside of the blockchain to smart contract [8]. The obvious practical issues with such contracts is the trust required from the oracle.

Calling Application Programming Interface (API) to reuse functionalities developed by third parties is common. Furthermore, microservice, i.e., a cohesive, independent process interacting via messages [10], is a recent emerging trend. However, in the blockchain context, developers need to use web services with special attention. If the service returns different results to each peer, it causes inconsistency of the ledgers.

2) External Library Calling: Similar to web service, developers apply third party libraries to reduce the efforts of developing the software. Developers also need to be careful of the behaviors of the library and understand what happens in the library.

3) System Command Execution: Go can execute external commands through `os/exec` package. The feature is useful, but developers need to be careful when they use it in the context of chaincode development. There is no assurance of observing same results of the commands among endorsing peers.

4) External File Accessing: Similar to system command execution, developers also consider what will happen when the chaincode has access to external files in each independent environment (e.g., `os.Open()`). There is no assurance of observing same results of the commands among endorsing peers.

C. State Database Specification

The risk arises from the state database specification. In Hyperledger Fabric, since developers can choose a database for state database, the risk is not present when using another database.

1) Range Query Risk: Hyperledger Fabric provides methods to access the state databases. In these methods, some range query methods, i.e., `GetQueryResult()`, `GetHistoryForKey()` and `GetPrivateDataQueryResult()`, need to be used with attention, because these methods are not re-executed during the validation phase.[11] This means that phantom reads are not detected. Phantom read reads data that other transactions add or delete and changes the result of the process.

D. Fabric Specification

The risks depend on the specification or implementation of Hyperledger Fabric. Hence, these risks will be fixed along with the improvement of Hyperledger Fabric.

[11] https://github.com/hyperledger/fabric/blob/release-1.3/core/chaincode/shim/interfaces.go

Listing 2: Example of Field Declarations

```
type BadChaincode struct {
    globalValue string // this is a risk
}

func (t *BadChaincode) Invoke (stub shim.
  ChaincodeStubInterface) peer.Response {
    t.globalValue = args[0]
    return shim.Success([]byte("success"))
}
```

Listing 3: Example of Read Your Write Risk

```
// At the initial point: {key: "key", value
  : 0}
val := 1
// Update the value from 0 to 1
err := stub.PutState("key", val)
if err != nil {
    fmt.Printf("Error is happened. %s", err)
}
// The method returns 0, not 1
ret, err := stub.GetState("key")
if err != nil {
    fmt.Printf("Error is happened. %s", err)
}
```

Listing 4: Unhandled Errors

```
key := "keystring"
// Unhandled error is happened
ret, _ := stub.GetState(key)

// Unhandled error is not happened
ret, err := stub.GetState(key)
if err != nil {
    fmt.Printf("Error is happened. %s", err)
}
```

Listing 5: Unchecked Input Arguments

```
// Unchecked input arguments is happened
ret, err := stub.GetState(args[0])

// Unchecked input arguments is not
  happened
if len(args) != 2 {
    return shim.Error("Incorrect number of
      arguments. Expecting 2")
}
ret, err := stub.GetState(args[0])
```

1) Field Declarations: This risk is from *Chaincode Scanner*. Developers need to implement two methods (`Init()` and `Invoke()`) to satisfy the chaincode interface when they implement chaincodes in Go. When the methods are implemented as methods of a structure such as Listing 2, developers can define fields of the structure. The field can be accessed in the methods. The field can be used with a global state in the program. However, since every peer does not execute every transaction, the state does not keep same value among peers. Listing 2 shows a detailed example of the risk.

2) Cross Channel Chaincode Invocation: If both chaincodes are on the same channel, it is okay to invoke a chaincode from another. If not, developers need to understand that no data will be committed in the other channel and developers only obtain what the chaincode function returns.

Specifically, developers use `InvokeChaincode()` method to invoke another chaincode. In the source code of implementing the method[12], there is a comment related to the specification: *"...that is, chaincode calling chaincode doesn't create a new transaction message. ...If the called chaincode is on a different channel, only the Response is returned to the calling chaincode; any PutState calls from the called chaincode will not have any effect on the ledger; ..."*

3) Read Your Write: In distributed systems field, Read-Your-Write consistency is one of the consistency models [21]. If a system follows Read-Your-Write consistency, a value written by a process on a data item X will be always available to a successive read operation performed by the same process on data item X.

In Hyperledger Fabric, Read-Your-Write semantics are not supported. Hence, if a transaction reads a value for a key, the value in the committed state is returned even if the transaction has updated the value for the key before issuing the read.

Listing 3 shows a simple example of the risk. At the initial point of the code, "0" is stored in the ledger as the value of "key." In the code, the value of "key" is changed to "1" at Line 4. Since Read-Your-Write semantics are not supported, the value that is read from the ledger at Line 9 is still "0." If the developer intends to read the value after updating, this code results in a different result from what the developer wants.

E. Common Practices

There are some practices that developers should consider during development, not only for chaincode but also other source codes. At first, we did not consider these practices, because they can be addressed using Go tools (e.g., *golint*). Since *Chaincode Scanner* picks up these practices as a part of risks, we have included them in this paper. In this study, we investigate such practices by referring to *Effective Go*[13] and *Go Code Review Comments*[14].

[12]https://github.com/hyperledger/fabric/blob/release-1.3/core/chaincode/shim/interfaces.go

[13]https://golang.org/doc/effective_go.html

[14]https://github.com/golang/go/wiki/CodeReviewComments

Chaincode Scanner checks two practices. One is *Unhandled Errors* and the other one is *Unchecked Input Arguments*. Listings 4 and 5 show brief examples of these two practices.

1) Unhandled Errors: In Go, developers can skip the index or value by assigning to "_." Even though the value is error type, developers can skip it and ignore the occurrence of an error.

2) Unchecked Input Arguments: This is a common case in any programming language. If the input arguments are not checked and the program grants accesses to a non-existent element, an error will occur.

F. Summary

From our survey, we found 14 risks and two common practices that we did not consider at first but Chaincode Scanner targeted. We found that most of the risks in discussions of developers in GitHub issues and other risks are mentioned in official documents and blogs.

These risks are intuitive, not like *Reentrancy risk* in Ethereum [16]. However, as these features are usually used in software development, developers need to be consciously aware of the risks when developing smart contracts.

V. Development Tools for Go

In this section, we discuss risks that can be prevented by using tools that Go programming language developers typically use. If these tools are available for chaincode development and cover these risks, developers can develop their chaincodes without the risks. If not, we should consider using a tool or method to avoid introducing these risks.

A. Go Tools

In this study, we consider the tools that are listed in *go-tools*[15] and *gometalinter*[16]. Go-tools is a collection of tools and libraries for working with Go code. The Hyperledger community recommends using the tools for conducting static analysis of codes used in Hyperledger projects.[17] Gometalinter is a tool to run multiple linters and static analysis tools concurrently. It calls more than 30 tools, such as *go vet* and *golint*.

We investigate whether the aforementioned risks can be detected using the tools. The following only shows the tools that can detect more than one risks.

1) gochecknoglobals, varcheck: Gochecknoglobals checks that no globals are present in Go code. *Varcheck* finds unused global variables and constants. Using these tools, developers can detect "Global Variable" risks.

2) errcheck: Errcheck is a program for checking for unchecked errors in Go program. Using this tool, developers can prevent "Unhandled Errors" risks.

3) gosec: Gosec inspects source code for security problems. In the security problems, four chaincode related risks can be detected, i.e., "Unhandled Errors," "Web Service," "System Command Execution," and "Generating Random Number."

4) golint: Golint makes suggestions for many of the mechanically checkable items listed in *Effective Go* and the *CodeReviewComments*. In the items, "Unhandled Errors" is related to chaincode potential risk.

In summary, developers can detect only 6 out of 16 potential risks. Even though these risks are easy to understand, developers need to detect and remove them by performing reviews.

5) go test -race: By adding `-race` option, developers can check for the existence of race condition in their source codes. The source code that includes goroutine should be checked with this option.

B. Chaincode Specific Tool

After we developed our tool, we found *Chaincode Scanner*[18] as a vulnerabilities detection tool for chaincodes. Therefore, we decide to compare the tool with ours. Chaincode Scanner was developed by *ChainSecurity* and was provided as a web application. Developers can use the tool by providing the URL to the Go project for non-commercial purposes.

In this study, we only consider the risks whose results and information we can find when we run the demo. In the demo, the tool checks five kinds of critical issues and four kinds of warnings. The following are the issues and warnings with brief descriptions.

As critical issues:

- *Goroutines:* The use of concurrency is discouraged in chaincode. This is "Concurrency of Program" in Table I.
- *Field Declarations:* The chaincode object should not declare any field.
- *Global State:* Operations on the ledger should not depend on global variables. This is "Global Variable" in Table I.
- *Blacklisted Imports:* The usage of certain libraries can lead to non-determinism (e.g., timestamp). This is "System Timestamp," "Random Number Generation," and some other risks in Table I.
- *Map Range Iterations:* Range iterations over map entries are not deterministic. This is "KVS Structure Iteration" in Table I.

As warnings:

- *Unchecked Input Arguments:* The number of arguments should be validated before their use.
- *Unhandled Errors:* Errors should not be ignored.
- *Read After Write:* Read after write operations to the same variable yields the old value. This is "Read Your Write" in Table I.
- *Phantom Read of Ledger:* Results of phantom reads should not be used to manipulate the ledger. This is "Range Query Risk" in Table I.

Note that "Blacklisted Imports" can detect multiple libraries, and the description is *"Libraries which allow communication with the outside world, grant file access or can introduce non-determinism can lead to inconsistencies in the execution among peers."* However, the blacklist is not released. Hence, the items that we show above (i.e., we put checkmarks at

[15] https://github.com/dominikh/go-tools

[16] https://github.com/alecthomas/gometalinter

[17] https://wiki.hyperledger.org/security/analysis-tools

[18] https://chaincode.chainsecurity.com/

TABLE II: COVERAGE OF EACH TOOL

Category	Risk	Our Tool	Chaincode Scanner	Go Tools
Non-determinism Arising From Language Instructions	Global Variable	✓	✓	✓(gochecknoglobals, varcheck)
	KVS Structure Iteration	✓	✓	
	Reified Object Addresses	✓		
	Concurrency of Program	✓	✓	✓(go test -race)
	Random Number Generation	✓	✓(Blacklist)	✓(gosec)
	System Timestamp	✓	✓(Blacklist)	
Non-determinism Arising From Accessing Outside of Blockchain	Web Service	✓	✓(Blacklist)	✓(gosec)
	System Command Execution	✓	✓(Blacklist)	✓(gosec)
	External Library Calling	✓		
	External File Accessing	✓		
State Database Specification	Range Query Risk	✓	✓	
Fabric Specification	Field Declarations		✓	
	Cross Channel Chaincode Invocation	✓		
	Read Your Write	✓	✓	
Common Practices	Unhandled Errors		✓	✓(golint, errcheck, gosec)
	Unchecked Input Arguments		✓	
Coverage		81.3% (13/16)	75% (12/16)	37.5% (6/16)

Random Number Generation, *System Timestamp*, *Web Service*, and *System Command Execution*) might be different.

VI. OUR TOOL

The result of our investigation shows that many risks are not covered by Go tools, since some risks are only in blockchain contexts. Therefore, we decide to design and implement a prototype tool to detect these risks.

Our tool only focuses on chaincodes written by Go language. Since Go language has a lot of tools to detect risks related to common practices as mentioned above, we only consider the risks related to chaincode development.

A. Design

These risks are easy to understand for developers, but these features are often used in other developments besides chaincode development. The fact suggests that developers might use these features without enough attention to their chaincodes in chaincode development. Furthermore, some risks are not inherently problematic and it depends on the usage. For example, we picked up "accessing web service" as one of the risks. However, if the developer knows that the return values from the web service are deterministic, the chaincode is deterministic. Hence, the biggest problem is using these operations without understanding or being aware of the potential risks.

The tool only notifies the developers of the existence of risks, and then developers can decide whether the detected risks are really problems or not. For such usage, it is important for developers to quickly obtain the result of the tool. Therefore, we implemented a static analysis tool that analyzes abstract syntax tree (AST) of chaincode.

We assume that our tool is used with other Go tools (e.g., linters and formatters) during development. Go language has a default environment to introduce libraries. Developers can obtain libraries or tools using `go get` command; for example, developers can introduce the tools mentioned in the previous section. To make it easy to use our tool in conjunction with other tools, we decide to implement our tool as a command line tool using Go language. Even if an environment cannot use `go get` command, we can create binaries for multiple environments (e.g., Windows, Linux, and Macintosh) because Go language supports cross compilation.

B. Implementation

Here, we show the steps to identify the risks in our static analysis.

1) Create AST from Chaincode: First, the tool transforms chaincodes contracts to AST.

2) Check Imported Libraries: After creating AST, the tool traverses the AST and obtains various kinds of information. The tool checks the libraries that are imported in the chaincode by checking *ImportSpec* node at first. If there is no library related to potential risks (e.g., `time`), the tool skips the processes for detecting such risks.

3) Check Declaration of Global Variables: By checking *DeclStmt* node, the tool identifies global variables used in the chaincode. If the tool detects global variables in this step, the tool outputs notifications of existence of the global variables.

4) Detect Risks Related to Variables: According to the imported libraries, the tool detects risks related to variables (e.g., random value and timestamp). These types of risks cause problems when the variables that are assigned non-deterministic values are utilized at ledger operations such as `PutState()` and control flow statement such as if-statement. Hence, the tool detects such operations first and then traces the operations related to the variables.

5) Detect Risks Unrelated Variables: Part of risks are not related to variables. For instance, goroutine is called "`go func_name()`" format in source codes. The tool identifies these risks using a type of nodes such as *GoStmt*.

6) Output Results: Finally, the tool outputs all detected risks with information such as category, function, variable name, position, and source line (Fig. 2).

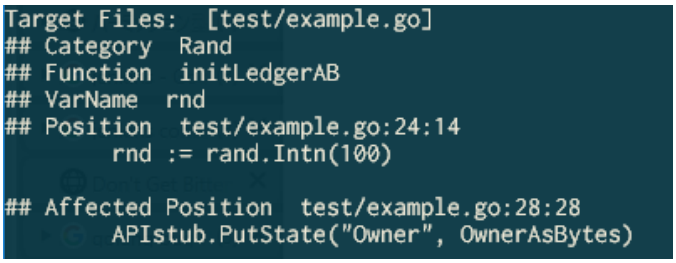

Fig. 2: Output of Our Tool (Random Value)

TABLE III: BRIEF DATA OF THE PRODUCTS

Application	Size (LOC)	Description
Wallet Application	9,557	Application to manage cryptocurrencies and coupons
Access Control Application	1,686	Application to manage rights to access data

C. Evaluation

To evaluate the usefulness of the tool, we apply our tool to two applications developed in our company. Table III shows a brief data of the products.

From the result of our tool application, we found the tool could analyze the chaincodes in a few seconds, but unfortunately could not find any risks. This is majorly due to the phase of these products. When we received these products, they were already released. This means that the products have been severally tested and reviewed. As we mention above, the tool can detect risks that are easy to understand. Hence, we assume that developers can detect these risks during review or modification. However, we believe that it is import to introduce tools to detect such risks automatically because releasing the products with risks cause serious problems such as The DAO and Parity cases.

VII. DISCUSSION

In this section, we discuss other perspectives of developing of chaincode and our results.

A. Other Languages and Tools for Hyperledger Fabric

In this paper, we only focus on development chaincodes using Go language because Go language is the first supported and the most popular language. However, in Hyperledger Fabric, developers can also choose Java and Node.js to write their chaincodes. We assume that some of the risks associated with using Go are not associated with other languages and there are risks that are not associated with Go language. For example, Java has three kinds of implementation of Map interface: *LinkedHashMap*, *TreeMap*, and *HashMap*. In these implementations, LinkedHashMap and TreeMap keep the order of keys, even though HashMap does not keep the order. This means that iteration using LinkedHashMap and TreeMap are deterministic and HashMap is non-dterministic.

Furthermore, Hyperledger Fabric provides *Hyperledger Composer*[19], which is an extensive, open development toolset and framework to make developing blockchain applications easier. Hyperledger Composer has a different architecture to develop blockchain applications. In the context of Hyperledger Composer, a business network is made up of assets, participants, transactions, access control rules, and optional events and queries. The definitions of components of a business network are written using *Hyperledger Composer Modelling Language* and JavaScript.

Therefore, as there might be specific risks for each of the languages and tools, developers need to consider risks according to their own development environment.

B. Validity of the Risks

In this paper, we summarize 16 risks in total. Since we observed these risks by investigating artifacts such as GitHub issues, documents, and blogs, these risks are known to Hyperledger Fabric development members and practitioners. Hyperledger Fabric is a newer product than other blockchain frameworks such as Ethereum and Bitcoin, and there are only a few applications. Therefore, experiences of chaincodes development have not yet been recorded. In such situation, there will be risks that have not yet been discovered. It is important to share new risks such as *Ethereum Smart Contract Best Practices*[20].

VIII. CONCLUSION

In this paper, we focus on possible risks associated with Hyperledger Fabric smart contracts. Unlike the Ethereum smart contracts, which uses Solidity, Hyperledger Fabric utilizes general-purpose programming languages (e.g., Go, Node.js, and Java). By using general-purpose programming languages, developers can save the effort of learning new languages. However, one of the largest demerits of using general-purpose programming languages such as Go is the absence of restrictions or specific features (which DSLs have) to safely write smart contracts. Despite the situation, there are only a few research papers and tools to support the development of smart contracts, according to our knowledge.

We first surveyed various artifacts to identify potential risks associated with Hyperledger Fabric. From the result, we identified 13 risks and added three risks based on Chaincode Scanner. Then, we investigated how many risks can be detected by Go tools. Since the investigation shows that only 6 risks can be detected using Go tools, we implemented a command line tool for chaincode that covers 13 risks. Finally, we evaluated our tool using two blockchain applications written by Go (about 10K and 1.5K LOC). Although the tool could not find new risks in the applications, the tool could analyze the applications in a few seconds. Therefore, we believe that developers can collaboratively use the tool with other Go tools.

As future work, we plan to expand our survey to other languages, e.g., Node.js and Java. As we mention in Section VII,

[19]https://hyperledger.github.io/composer/latest/

[20]https://consensys.github.io/smart-contract-best-practices/

other kinds of potential risks in may be associated with each language. Furthermore, we plan to continue surveying for new risks. As the number of use cases of Hyperledger Fabric application increases, practitioners will observe new risks. Therefore, we will need to update the information about risks and our tool.

REFERENCES

[1] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proc. EuroSys Conf.*, pages 30:1–30:15, 2018.

[2] K. Baqer, D. Y. Huang, D. McCoy, and N. Weaver. Stressing out: Bitcoin "stress testing". In *Proc. Int'l Conf. on Financial Cryptography and Data Secur.*, pages 3–18, 2016.

[3] M. Bartoletti and L. Pompianu. An empirical analysis of smart contracts: platforms, applications, and design patterns. *CoRR*, 2017.

[4] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin. Formal verification of smart contracts: Short paper. In *Proc. ACM Workshop on Programming Languages and Analysis for Secur.*, pages 91–96, 2016.

[5] C. Cachin, S. Schubert, and M. Vukolic. Non-determinism in byzantine fault-tolerant replication. In *Proc. Int'l Conf. on Principles of Distributed Systems*, pages 1–16, 2016.

[6] A. Chepurnoy and M. Rathee. Checking laws of the blockchain with property-based testing. In *Proc. Int'l Workshop on Blockchain Oriented Softw. Eng.*, pages 40–47, 2018.

[7] K. Christidis and M. Devetsikiotis. Blockchains and smart contracts for the internet of things. *IEEE Access*, 4:2292–2303, 2016.

[8] C. D. Clack, V. A. Bakshi, and L. Braine. Smart contract templates: foundations, design landscape and research directions. *CoRR*, 2016.

[9] G. Destefanis, M. Marchesi, M. Ortu, R. Tonelli, A. Bracciali, and R. Hierons. Smart contracts vulnerabilities: a call for blockchain software engineering? In *Proc. Int'l Workshop on Blockchain Oriented Softw. Eng.*, pages 19–25, 2018.

[10] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. chapter Microservices: Yesterday, Today, and Tomorrow, pages 195–216. 2017.

[11] D. Harz and W. J. Knottenbelt. Towards safer smart contracts: A survey of languages and verification methods. *CoRR*, pages 1–20, 2018.

[12] Y. Hirai. Defining the ethereum virtual machine for interactive theorem provers. In *Financial Cryptography and Data Secur.*, pages 520–535, 2017.

[13] S. Kalra, S. Goel, M. Dhawan, and S. Sharma. Zeus: Analyzing safety of smart contracts. In *Proc. ISOC Symp. on Network and Distributed System Secur.*, 2018.

[14] C. Liao, C. Cheng, K. Chen, C. Lai, T. Chiu, and C. Wu-Lee. Toward a service platform for developing smart contracts on blockchain in bdd and tdd styles. In *Proc. Int'l Conf. on Service-Oriented Computing and Applications*, pages 133–140, 2017.

[15] Y. Liu, Q. Lu, X. Xu, L. Zhu, and H. Yao. Applying design patterns in smart contracts. In *Proc. Int'l Conf. on Blockchain*, pages 92–106, 2018.

[16] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *Proc. SIGSAC Conf. on Computer and Comms. Secur.*, pages 254–269, 2016.

[17] B. Mueller. Smashing ethereum smart contracts for fun and real profit. In *HITBSecConf*, pages 1 – 54, 2018.

[18] Q. Nasir, I. A. Qasse, M. A. Talib, and A. B. Nassif. Performance analysis of hyperledger fabric platforms. *Secur. and Comm. Networks*, 2018:1–14, 2018.

[19] S. Pongnumkul, C. Siripanpornchana, and S. Thajchayapong. Performance Analysis of Private Blockchain Platforms in Varying Workloads. In *Proc. Int'l Conf. on Computer Comm. and Networks*, pages 1–6, 2017.

[20] S. Porru, A. Pinna, M. Marchesi, and R. Tonelli. Blockchain-oriented software engineering: Challenges and new directions. In *Proc. Int'l Conf. on Softw. Eng. Companion*, pages 169–171, 2017.

[21] A. S. Tanenbaum and M. v. Steen. *Distributed Systems: Principles and Paradigms*. 2006.

[22] P. Thakkar, S. Nathan, and B. Vishwanathan. Performance benchmarking and optimizing hyperledger fabric blockchain platform. In *Proc. Int'l Symp. on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 1–13, 2018.

[23] R. Tonelli, G. Destefanis, M. Marchesi, and M. Ortu. Smart contracts software metrics: a first study. *CoRR*, 2018.

[24] P. Tsankov, A. M. Dan, D. D. Cohen, A. Gervais, F. Buenzli, and M. T. Vechev. Securify: Practical security analysis of smart contracts. *CoRR*, 2018.

[25] U. U. Uchibeke, S. H. Kassani, K. A. Schneider, and R. Deters. Blockchain access control ecosystem for big data security. *CoRR*, 2018.

[26] M. Vukolić. Rethinking permissioned blockchains. In *Proc. ACM Workshop on Blockchain, Cryptocurrencies and Contracts*, pages 3–7, 2017.

[27] M. Wohrer and U. Zdun. Smart contracts: security patterns in the ethereum ecosystem and solidity. In *Proc. Int'l Workshop on Blockchain Oriented Softw. Eng.*, pages 2–8, 2018.

[28] G. Zhang, T. Li, Y. Li, P. Hui, and D. Jin. Blockchain-based data sharing system for ai-powered network operations. *J. Comms. and Info. Networks*, 3(3):1–8, 2018.