Detection of Read-Write Issues in Hyperledger Fabric Smart Contracts

Luca Olivieri
Ca' Foscari University of Venice
Venice, Italy
luca.olivieri@unive.it

Luca Negrini
Ca' Foscari University of Venice
Venice, Italy
luca.negrini@unive.it

Vincenzo Arceri
University of Parma
Parma, Italy
vincenzo.arceri@unipr.it

Pietro Ferrara
Ca' Foscari University of Venice
Venice, Italy
pietro.ferrara@unive.it

Agostino Cortesi
Ca' Foscari University of Venice
Venice, Italy
cortesi@unive.it

ABSTRACT

Hyperledger Fabric is a well-known framework for developing enterprise blockchain solutions. Developers of these blockchains must ensure the correct execution of read and write operations so that the smart contracts' application logic is consistent with the business logic. In this paper, we present a static analysis approach based on abstract interpretation to detect read-write set issues in Hyperledger Fabric smart contracts and avoid bugs and critical errors that could compromise blockchain applications. The analysis is implemented in GoLiSA, a semantics-based static analyzer for Go applications. Our experimental results show that the proposed analysis can detect read-write set issues on a significant benchmark of existing applications. Moreover, it achieves better results in detecting read-after-write issues than other well-known open-source analyzers for Hyperledger Fabric smart contracts.

CCS CONCEPTS

• **Theory of computation** → **Program verification**; • **Software and its engineering** → **Formal software verification**; **Automated static analysis**; **Software verification**;

KEYWORDS

Smart contracts, Chaincode, Blockchain, Hyperledger Fabric, Distributed Ledger Technology, Static analysis, Abstract interpretation, Read after write, Read-write conflict, Read your write consistency.

ACM Reference Format:
Luca Olivieri, Luca Negrini, Vincenzo Arceri, Pietro Ferrara, and Agostino Cortesi. 2025. Detection of Read-Write Issues, in Hyperledger Fabric Smart Contracts. In *Proceedings of ACM SAC Conference (SAC'25).* ACM, New York, NY, USA, Article 4, 9 pages. https://doi.org/xx.xxx/xxx_x

SAC'25, March 31 – April 4, 2025, Sicily, Italy

ACM ISBN 978-1-4503-9517-5/23/03…$15.00
https://doi.org/xx.xxx/xxx_x

1 INTRODUCTION

Hyperledger Fabric (HF) [2] is a popular open-source platform for developing custom *permissioned* blockchain ledgers, hosted by the Linux Foundation. HF supports the development and execution of *chaincodes* [13] (i.e., smart contracts), which are programs, primarily written in the Go programming language, that are deployed and executed within the blockchain ledger. Chaincodes can interact with and modify the world state (key-value data structure) through explicit read and write operations. In this way, they can query the ledger to retrieve existing information and store data, maintaining an immutable record of the changes in the ledger (blockchain data structure).

However, write operations are committed only after the result of a chaincode execution is simulated and validated by blockchain peers. Consequently, within the same execution, *read-your-writes* semantics are not supported [15] leading two non-trivial behaviors: (i) if multiple writes occur for a given key, only the last one executed will be committed to the ledger state; (ii) if the value for a given key is read, it is retrieved from the committed state, even if a write was previously executed to update the value of that key.

Hence, naive programming by developers attempting to achieve such semantics can lead to critical errors and unexpected behavior. Therefore, it is crucial to detect potential issues arising from these read-write semantics.

According to Ren et al. [34], the state-of-the-art for read-write risk detection in HF has shortcomings related to a low detection accuracy and coverage of a large number of read and write logic in actual projects. In addition, they do not follow a *sound* approach [7], leading these tools to produce several false negatives, which results in the deployment of immutable issues in the blockchain. In this paper, we make the following contributions:

- a more comprehensive summary of the issues related to read-write set semantics;
- the design and implementation of an analysis for detecting *read-after-write* issues using static analysis via abstract interpretation [7]. To the best of our knowledge, this is the first sound analysis based on formal methods for this type of issue;
- the design and implementation of an analysis for detecting *over-write* issues. As far as we know, this is the first analysis to address this problem.

The analysis was implemented by extending GoLiSA [31], an open-source static analyzer based on abstract interpretation that supports the analysis of several blockchain frameworks written in Go. The evaluation is conducted on a benchmark suite of existing smart contracts retrieved from public GitHub repositories, and it empirically shows that our approach can successfully identify read-write set issues.

Paper structure. Sect. 2 introduces the blockchain-based ledger of HF. Sect. 3 provides an overview of read-write set issues in HF. Section 4 and Sect. 5 present the design of our core contribution for detecting read-write set issues and its implementation in GoLiSA. Sect. 6 experimentally evaluates the proposed analysis implemented in GoLiSA on a benchmark suite of existing smart contracts, while also benchmarking other tools in detecting the same issues. Sect. 7 presents related work and a comparison with the state of the art. Sect. 8 concludes the paper.

2 BLOCKCHAIN-BASED LEDGER

In HF, the blockchain-based ledger [12] consists of two distinct, though related, parts: (i) the **World State**, i.e., a *mutable* data structure by default expressed typically as versioned key-value pairs, that can change over time; (ii) the **Blockchain**, an *immutable* data structure that records all the changes that determine the world state, as a transaction log. The blockchain data structure is different from the world state because once written, it cannot be modified. It is an immutable sequence of blocks, each containing a set of ordered transactions.

Furthermore, smart contracts never directly modify the blockchain; they only interact with the world state through specific read and write instructions (Tab. 1). These interactions will be collected in read and write sets ,which are subsequently stored in the blockchain by the HF protocol.

3 READ-WRITE SET ISSUES

This section describes the read and write issues of HF exploring their causes and impacts on the application logic of smart contracts and the data stored in the blockchain-based ledger.

3.1 The Read-after-write Problem

As reported in the official HF documentation [15], it is important to note that during the execution of a read operation, the retrieved value corresponds to the one present in the world state of the blockchain ledger *before* the transaction's execution. Therefore, HF does not employ a *read-your-writes consistency* [39]. In simpler terms, the read value will not reflect the most recent update within the same transaction, even if the key's value is updated before reading it. This phenomenon is known as *read-after-write* (aka *read-write conflict*), and it is the cause of several bugs by HF developers due to its counterintuitive nature, leading to unexpected behaviors in the chaincode.

For instance, let us consider the code snippet in Fig. 1a, starting with an initial world state containing the data `{key: "myuser", value: "100"}`. At line 5, the `PutState` call updates the key-value entry with `{key: "myuser", value: "0"}`. However, at line 7, when the `GetState` call is executed to retrieve the value

```
1  func ReadAfterWriteIssue(stub *shim.ChaincodeStub) {
2    var user string := "myuser"
3    var value string := "0"
4
5    err = stub.PutState(key, []byte(value))
6    // ...
7    response, err = stub.GetState(key)
8    // ...
9  }
```

(a) ***read-after-write***

```
1  func OverWrite(stub *shim.ChaincodeStub) {
2    var value1 string := "0"
3    var value2 string := "1"
4
5    err = stub.PutState("key", []byte(value1))
6    // ...
7    err = stub.PutState("key", []byte(value2))
8  }
```

(b) ***over-write***

Figure 1: Go examples containing the issues.

associated with `"myuser"`, it still returns `"100"` (not `"0"`). As previously explained, this behavior occurs because write operations are only committed after the consensus mechanism's approval, meaning they take effect after the code execution.

3.2 The Over-write Problem

When a transaction writes a value multiple times to the same key, only the last written value is retained, implying that prior write operations on the same key are not preserved in the blockchain storage [15], thus missing the writing and therefore the tracking of data in the blockchain. Let us consider the code snippet in Fig. 1b, assuming an initial world state with no existing data. At line 5, the `PutState` operation writes the key-value pair `{key: "key", value: "0"}`. Also, line 7 writes a value associated with the key `"key"`, i.e., the key-value `{key: "key", value: "1"}` . As previously explained, after receiving consensus approval and completing code execution, only one write operation for a specific key is recorded in the blockchain. In our example, for the key `"key"`, only the value `"1"` is written in blockchain, overwriting the value of the previous write operation, then discarding `"0"`. This behavior could lead to unintended loss of data that should ideally be preserved into the blockchain (e.g., for supply-chain systems or tracking applications) or lead to redundant execution of write operations in the code.

3.3 Running Example

We propose a running example to better show the criticality of these two issues. Fig. 2 shows a code snippet of a smart contract in a token transfer scenario, containing both read-after-write and over-write issues. At lines 12-30, the method `transferTokenToFoo` transfers some tokens from one user to another, called *Foo*. The user information of sender and the amount to transfer are collected in the variables `from` and `valueToTransfer` at lines 14 and 16, respectively. These values are retrieved from the transaction input using `GetFunctionAndParameters()` at line 4. Since users can input arbitrary values, any value may be assigned. The *Foo* user is declared

Table 1: Read and Write Go APIs in HF.

Category	shim.ChaincodeStubInterface's methods
Write	PutState(key string, value []byte) DelState(key string) SetStateValidationParameter(key string, ep []byte) PutPrivateData(collection string, key string, value []byte) DelPrivateData(collection, key string) PurgePrivateData(collection, key string) SetPrivateDataValidationParameter(collection, key string, ep []byte)
Read	GetState(key string) GetStateValidationParameter(key string) GetStateByRange(startKey, endKey string) GetStateByRangeWithPagination(startKey, endKey string, pageSize int32, bookmark string) GetStateByPartialCompositeKey(objectType string, keys []string) GetHistoryForKey(key string) GetPrivateData(collection, key string) GetPrivateDataValidationParameter(collection, key string) GetPrivateDataByRange(collection, startKey, endKey string) GetPrivateDataHash(collection, key string) GetPrivateDataByPartialCompositeKey(collection, objectType string, keys []string)

```
1  // ...
2  func (s *SmartContract) Invoke(APIstub shim.ChaincodeStubInterface
       ) sc.Response {
3
4    function, args := APIstub.GetFunctionAndParameters()
5
6    if function == "transferTokenToFoo" {
7      return s.transferTokenToFoo(APIstub, args)
8    }
9    // ... other operations
10 }
11 // ...
12 func (s *SmartContract) transferTokenToFoo(APIstub shim.
       ChaincodeStubInterface, args []string) sc.Response {
13
14   from := args[0]
15   to := "Foo"
16   valueToTransfer, err := strconv.Atoi(args[1])
17   // ... error handling
18   value, err := APIstub.GetState(from)
19   // ... error handling
20   fromValue := int(value)
21   // ... error handling
22   err := APIstub.PutState(from, []byte(fromValue - valueToTransfer
         ))
23   // ... error handling
24   value, err = APIstub.GetState(to)
25   // ... error handling
26   toValue := int(value)
27   // ... error handling
28   err := APIstub.PutState(to, []byte(toValue + valueToTransfer))
29   // ... error handling and other operations
30 }
```

Figure 2: Read and write issues.

at line 15, where the string `"Foo"` is assigned to the variable `to`. The token transfer occurs at lines 18-28, where `GetState(from)` retrieves the current token value of the `from` user from the ledger, `PutState(from, []byte(fromValue- valueToTranfer))` removes the value `valueToTranfer` from the `from` user and writes the results on the ledger, `GetState(to)` retrieves the current token value of the *Foo* user from the ledger, and `PutState(to, []byte(toValue + valueToTranfer))` adds the value to the `to` user.

In this contract, if `from` and `to` are different users, the expected behavior is the transfer of tokens, moving the value from an user to a *Foo* user, i.e., the amount is subtracted from the specified user and added to the *Foo* user.

However, there are read-after-write and over-write issues that be can exploited by *Foo* user to increase its amount of value. Since `from` can have any value (as it comes from a transaction input), nothing prevents the *Foo* user from creating an input where the value of `from` is the same as `to` (i.e., a transaction to oneself). In this scenario, suppose the initial state of ledger is `{"Foo": 100}`, where the attacker has 100 units of value. If the attacker sends the transaction input `args = {"Foo", 99}`, we have that `from = "Foo"` at line 15, and `valueToTransfer = 99` at line 16, while `fromValue = 100` after lines 18-19. Then, `PutState("Foo", []byte(100 - 99))` occurs at line 27, but `value = 100` after lines 24-25 (read-after-write issue) as it is still read from the pre-execution state. This is followed by `PutState("Foo", []byte(100 + 99))` at line 28. Moreover, `PutState` instructions at lines 22 and 28 both involves the same user `"Foo"`: the only value written to the ledger is `100+99` (over-write issue). Therefore, at the end of the computation of the `transferTokenToFoo` method, no subtraction is performed or recorded in the blockchain for the attacker. Consequently, `99` units of value are added instead of keeping the value unchanged.

Algorithm 1 Detection of issues related to Read-Write Set.

```
1:  procedure Analysis(program)
2:      stringValues ← stringValueAnalysis(program)
3:      readers, writers ← extractReadAndWrite(program, stringValues)
4:      read-write, write-write ← extractPairsSameKey(readers, writers)
5:      alerts ← ∅
6:      for each r, w ∈ read-write do                ▷ Read-After-Write Detection
7:          if isAfter(r, w, program) then
8:              alerts ← alerts ∪ buildReadAfterWriteAlarm(r, w)
9:      for each w1, w2 ∈ write-write do             ▷ Over-Write Detection
10:         if isAfter(w1, w2, program) then
11:             alerts ← alerts ∪ buildOverWriteAlarm(w1, w2)
12:         else if isAfter(w2, w1, program) then
13:             alerts ← alerts ∪ buildOverWriteAlarm(w2, w1)
14:     return alerts
```

4 DETECTION OF READ-WRITE SET ISSUES

This section presents a high-level overview of the static analysis approach designed to identify issues related to read-after-write and over-write problems, as discussed in the previous section. Static

analysis [36] allows one to verify program properties before the program execution automatically. Moreover, static analysis via abstract interpretation [7] can provide guarantees about the analyses such as *soundness*, i.e., the absence of *false negatives* for a given property. In practice, this means that if a sound static analyzer proves that a program respects a property (e.g., absence of read-write issues), then all possible executions (with all possible inputs) of that program will respect that property (i.e., no execution will expose read-write issues).

Alg. 1 outlines the key components of the proposed analysis. The algorithm starts by computing the keys for each read/write operation. Note that keys are strings, as indicated in Tab. 1. At line 2, stringValueAnalysis performs an abstract interpretation-based static analysis to soundly over-approximate the possible string values of variables for each program point. It is worth noting that Alg. 1 is agnostic to the specific choice of the string abstract domain, allowing the approach to be instantiated with a variety of string abstractions, such as [3–6, 24]. Line 3 relies on the function extractReadAndWrite to determine the set of read and write instructions, `readers` and `writers`, respectively, by searching for the signatures of functions reported in Tab. 1. extractReadAndWrite also uses the previously computed `stringValues` to store abstractions of the keys used by extracted instructions. At line 4, the extractPairsSameKey function generates pairs of instructions, namely `read-write` and `write-write` sets, which may have common keys. These pairs are later used to detect read-after-write and over-write problems at lines 6-8 and 9-13, respectively. In particular, the isAfter function checks whether an instruction is possibly executed after another, i.e., if there exists at least one execution path in the program where the first instruction is executed after the second one. When isAfter returns true, the analysis triggers an alarm because a read-write or a write-write set problem was detected.

Inter-procedural Analysis. Consider the code snippet in Fig. 3a, which contains a read-after-write issue. Function `F` calls functions `A` and `B`, resulting in indirect calls to `PutState` (line 9) and `GetState` (line 13) methods, both using the same key `"key"`, thus causing the read-after-write problem. According to Ren et al. [34], the combination of multiple functions and methods makes detecting read-write set issues difficult without proper automatic tools.

To tackle scenarios such as the one in Fig. 3a, both stringValueAnalysis and isAfter need to be *inter-procedural*, that is, they should consider interactions across multiple functions and methods, tracking how values are passed from callers to callees. This enables our analysis to detect read-write set issues such as the one reported in Fig. 3a. Adopting an intra-procedural approach (as seen in several tools that will be discussed in Sect. 7) would weaken the analysis, making it incapable of detecting issues like the one reported in 3a and, in turn, leading to false negatives.

4.1 Critical Issues

For simplicity, Alg. 1, discussed in the previous section, provides a high-level overview of the analysis, omitting specific details on handling specific cases. In the following, we pinpoint these specific cases and discuss the behavior of the approach when encountering them.

```
1  func F(stub *shim.ChaincodeStub) {
2    var key string := "key"
3    var data string := "Hello World!"
4
5    A(stub, key, []byte(value))
6    B(stub, key)
7  }
8  func A(stub *shim.ChaincodeStub, key string, value []byte ) {
9    err = stub.PutState(key, []byte(value))
10   // ...
11 }
12 func B(stub *shim.ChaincodeStub, key string) {
13     response, err = stub.GetState(key)
14     // ...
15 }
```

(a) Requires interprocedural analysis.

```
1  func InputFoo(stub *shim.ChaincodeStub) {
2    fun, args = stub.GetFunctionAndParameters()
3
4    var key1 string = args[1]
5    var key2 string = args[2]
6    // ...
7
8    err = stub.PutState(key1, []byte(value))
9    // ...
10
11   response, err = stub.GetState(key2)
12   // ...
13 }
```

(b) Requires the approximation of user input.

```
1  func RangeExec(stub *shim.ChaincodeStub) {
2    err = stub.PutState("aa", []byte(value))
3    // ...
4    response, err = stub.GetStateByRange("a", "cc");
5  }
```

(c) Requires to handle ranges of strings.

```
1  func DExec(stub *shim.ChaincodeStub) {
2    var key string := "key"
3    var value string := "1"
4
5    defer err = stub.GetState(key)
6    // ...
7    err = stub.PutState(key, []byte(value))
8  }
```

(d) Requires to handle the behavior of `defer` instruction.

Figure 3: Go code examples that require specific checks and analysis settings.

String Value Over-approximation. We adopt a fully static approach, employing string static analysis to provide an over-approximation of the potential string values associated with keys of read-/write operations. Thus, when handling user inputs, our approach cannot precisely infer the possible string values of keys. Consider, for instance, the code fragment reported in Fig. 3b. At line 2, the user input is acquired through `GetFunctionAndParameters` and then propagated to `key1` and `key2` at lines 4–5. These are then used as keys at lines 8 and 11. In this case, a read-after-write issue arises if the `key1` and `key2` have the same value; however, since the value of both variables is statically unknown, they are modeled using the top element $\top$ of the chosen string abstract domain, modeling any possible string. In turn, our sound analysis (specifically

extractPairsSameKey at line 4 of Alg. 1) returns a potential match in the used keys when either one of the keys is abstracted to $\top$. A similar sound reasoning is applied to partial matches.

Range Key Instructions. Most of the read/write instructions reported in Tab. 1 require a single key as input, with some exceptions such as `GetStateByRange`.

The `GetStateByRange` instruction returns an iterator ranging all the keys between `startKey` (inclusive) and `endKey` (exclusive), w.r.t. the string lexicographical order. Note that `startKey` and `endKey` can be empty strings, implying an unbounded range query at the start or end. Following a fully static approach, it is impossible to consider only the key values stored in the blockchain between `startKey` and `endKey` because they can dynamically change over time. Therefore, in our analysis, we abstract the key range using the smallest interval enclosing the possible keys. The interval takes the form of $[\sigma_l, \sigma_u)$, where σ_l is the string lower bound and σ_u is the string upper bound; any key $\sigma \in [\sigma_l, \sigma_u)$ satisfies $\sigma_l \leq \sigma < \sigma_u$. Such an abstraction allows us to create an over-approximation of the range of keys. Consequently, when identifying key matches within the extractPairsSameKey function (at line 4 of Alg. 1), we can consider key strings belonging to the interval $[\sigma_l, \sigma_u)$.

For instance, consider the code snippet in Fig. 3c. Line 2 writes a value associated with the key `"aa"`, while the `GetStateByRange` instruction at line 4 reads a range of possible values spanning from the key `"a"` to `"cb"`. Intuitively, our analysis builds the interval [`"a"`, `"cc"`), abstracting the following set of values {`"a"`, `"b"`, …, `"z"`, `"aa"`, `"ab"`, …, `"cb"`}. Hence, during the execution of extractPairsSameKey, a potential key match is identified, as `"aa"` $\in$ [`"a"`, `"cc"`).

Partial Composite Key Instructions. Another instruction with a particular behavior is `GetStateByPartialCompositeKey`, which returns an iterator ranging over all composite keys whose prefix matches the given partial composite key. In such cases, extractPairsSameKey will return a potential key match for each key whose prefix corresponds to the key of the partial composite keys.

Private Data and Collections. In HF, chaincodes read and write private data using specific instructions such as `GetPrivateData` and `PutPrivateData`. In addition to the keys, they are required to specify private data collections [14], which allow a defined subset of organizations to endorse, commit, or query private data without creating different channels. During the analysis, the values of collections must be computed and compared before the keys comparison. If the instructions have the same keys but different collections, there are no read-write set issues. Since strings identify collections, their values are abstracted during the string value analysis and subsequently checked, considering over-approximations to preserve soundness.

Defer Execution. In Go,the `defer` statement allows to delay of the execution of a function call until the surrounding function returns. As shown in Fig. 3d, the read instruction at line 5 occurs before the write instruction at line 7. Nevertheless, at run-time, the execution of the read instruction is deferred until the return of `DExec`. Thus, the read instruction is executed after the write one, leading to a read-after-write issue. In our analysis, isAfter handles deferred reads and writes accordingly to this semantics, adapting the search heuristics (e.g., while isAfter normally checks for an execution path leading from a write to a read, it searches for the opposite path – from read to write – if the read instruction is deferred). Other approaches rely on separate instrumentations instead [34]. It is important to highlight that the use of deferring statements and concurrency-related constructs is highly discouraged within blockchain communities. Hence, it is rare to find real-world code that uses these statements on read and write operations of HF.

5 ANALYSIS IMPLEMENTATION IN GoLiSA

Our implementation is based on GoLiSA [30, 32, 33], an extension of LiSA (**Li**brary for **S**tatic **A**nalysis) [10, 25, 26] for the analysis of programs written in Go, also used in the blockchain context.

For the string value analysis, we rely on Tarsis [23, 24], an abstract domain that approximates string values through finite state automata, allowing us to perform sound analyses and obtaining precise results that are on par with state-of-the-art approaches, while also achieving significantly improved performance.

5.1 Running the Analysis

Analyzing the chaincode reported in Fig. 2, GoLiSA successfully identifies both the read-after-write and the over-write issues. Specifically, by performing the string analysis with Tarsis, it infers that the value states containing of variable `from` can abstract the variable as an automaton that represents any string (i.e., top element $\top$ for Tarsis) because their values are statically unknown, coming from an user input. On the other hand, `to` can be abstracted as a finite automaton that recognizes the string `"Foo"`. Inspecting the CFG, GoLiSA identifies the read operations at lines 18 and 24, and write operations at lines 22 and 28. It then checks the key values and infers that the automaton representing `from`'s values can simulate the automaton representing `to`'s value, thus it is possible that the concrete values of `from` may be equal to `to`. At this point, GoLiSA collects the `read-write` and `write-write` sets of Alg. 1. The first contains the pair of `GetState` at line 24 and `PutState` at line 28, while the second contains the pair of `PutState` instructions at lines 22 and 28, respectively. For each pair, GoLiSA computes if an execution path exists between the two statements of the pair. Using a graph search, it verifies whether there exists at least one path in the CFG connecting the two statements. Finally, once it is ensured that the paths exist, two warnings are triggered, one for read-after-write and one for over-write.

6 EXPERIMENTAL EVALUATION

In this section, we discuss the results of our experimental evaluation of the analysis implemented in GoLiSA for detecting read-write set issues in existing blockchain software.

We evaluated our approach on a set of 651 existing chaincodes retrieved from public GitHub repositories. This will allow us to understand the performance and accuracy of the analysis implemented in GoLiSA. Moreover, we compared the quality of results with two well-known open-source static analyzers for chaincodes, namely ChainCode Analyzer [18] and Revive^CC [38]. Since these tools do not provide analysis for over-write issues, our comparison focuses exclusively on the detection of read-after-write issues. The

Table 2: Read-write analysis results of GoLiSA.

Issues	Affected files	Unaffected files	#TP	#FP	#FN
Read-after-write	22	628	38	2	0
Over-write	245	405	481	201	0

evaluation shows that GoLiSA not only outperforms existing static analyzers in identifying these issues bu also guarantees soundness.

The experiments were conducted on a machine equipped with an AMD Ryzen 5 5600X 6-Core at 3.70 GHz, 16 GB of RAM DDR4, 1 TB SSD (read 540MB/s, write 500MB/s), running Windows 11 Pro 22H2, Open JDK version 13. During the analysis, 8 GBs of RAM were allocated to the JVM.

The experimental evaluation can be reproduced using the following artifact: `<url blinded>`

6.1 Benchmark Composition

The selected benchmark, henceforth referred to as $\mathbb{RW}$, is the one presented in [32]. $\mathbb{RW}$ consists of 651 chaincodes (~167391 Lines of Code) retrieved from public GitHub repositories by querying the keyword *"chaincode"*. To ensure uniqueness, only chaincodes from unforked repositories were selected, thereby avoiding duplication. No additional filters were applied based on this result: we considered all chaincodes found in the crawled repositories to avoid bias in the analysis outcomes.

6.2 Quantitative Evaluation

GoLiSA properly analyzed 650 chaincodes up on 651, with only 1 analysis failure due to an out-of-memory error. The total execution time for the detection of both issues is 1 hour, 5 minutes, and 18 seconds, with an average execution time of 6.02 seconds per chaincode. Tab. 2 shows the results of the read-after-write and over-write detection of GoLiSA over $\mathbb{RW}$. We denote by **#TP**, **#FP**, and **#FN** the number of true positives, false positives, and false negatives among the raised warnings, respectively.

Their classification was carried out through an in-depth manual investigation performed on all the chaincodes in $\mathbb{RW}$.

6.3 Qualitative Evaluation

To assess the effectiveness of the proposed analysis, we measure the number of true positives, false positives, and false negatives raised by the considered tools on $\mathbb{RW}$ for detecting the read-after-write problem[1]. Our benchmark includes 38 known read-after-write issues that we expect the analyzer to detect and raise warnings for.

Reviveˆ CC raises 17 warnings, split into 5 true positives and 12 false positives, but misses 33 read-after-write problems out of 38, corresponding to false negatives. ChainCode Analyzer issues 2 true and no false positives but misses 36 read-after-write problems out of 38. Instead, GoLiSA can detect all read-after-write problems, i.e., it issues 38 true positives, together with only 2 false positives and no false negatives. Tab. 3 summarizes the findings in terms of precision ($\frac{\#TP}{\#TP+\#FP}$), recall ($\frac{\#TP}{\#TP+\#FN}$) and F1 score ($\frac{\#TP}{\#TP+\frac{\#FN+\#FP}{2}}$).

[1]Both Reviveˆ CC and ChainCode Analyzer, when they meet a read-after-write problem, trigger two distinct warnings, one for the read and one for the write operations. Here, we count them as a single warning.

Table 3: Tool comparison on $\mathbb{RW}$.

Tool	Recall	Precision	F1
Reviveˆ CC	13,16%	29,41%	22,22%
ChainCode Analyzer	5,26%	**100,00%**	10,00%
GoLiSA	**100,00%**	95,00%	**97,44%**

GoLiSA achieves the best trade-off between recall and precision on $\mathbb{RW}$.

7 RELATED WORK

The consistency of the data read and written by different programs has been largely studied in the scientific literature during the last few decades. This topic spans many different contexts.

Multithreaded programs interact through shared memory, and poorly synchronized interactions can lead to data races [27]. The behavior of threads when accessing shared memory has been a topic largely debated and led to the definition of several memory models and many static analyses targeting the detection of data races [9, 22, 29] and the approximation of shared values [1, 8, 21] have been proposed. Instead, in the field of distributed systems, a problem that recently arose due to the popularity of microservices has been how to duplicate data (usually stored in databases) to improve the scalability of the software architecture. The so-called space-based architectures [35] target exactly such a scenario, where the same data is duplicated and maintained consistent across several distributed nodes.

The main difference w.r.t. our context is that we target the values read and written by a single sequential program to a persistent memory (i.e., the blockchain) instead of several programs (such as threads or services). Outside blockchain systems, such a problem is usually managed through databases guaranteeing ACID (Atomicity, Consistency, Isolation, and Durability) properties [11].

7.1 System-level Solutions

As far as we know, we have yet to find any system-level solution, although they have been proposed for other issues such as phantom reads [30]. System-level solutions may offer several benefits, not only impacting the handling of read-write and overwrite issues in the HF architecture but directly providing support for read-your-write consistency. However, as noted in [30], applying these solutions in enterprise environments poses significant challenges because they require an extensive rework involving the entire or a portion of the HF architecture, which may discourage IT companies. Indeed, companies typically require high degrees of software maturity, including stable, long-term supported versions and quick security updates. Therefore, it would require an official change of HF framework rather than unofficial custom solutions.

On the other hand, static analysis of chaincodes only requires developers to analyze and make minor fixes to the code, thus without affecting any component of the HF architecture and keeping the official HF framework unchanged.

7.2 Static Analysis of Chaincodes

In this section, we focus our attention on existing analyses for HF. In particular, read-after-write issues in HF are well-known [20, 40]. The most popular open-source tools for the verification of these issues are Chaincode Analyzer [18], by Fujitsu from Hyperledger Labs, and Reviveˆ CC [38], an extension of the Revive analyzer[2] for chaincodes. Both tools represent the program using abstract syntax trees (AST), applying intra-procedural analyses on them and syntactic checks to detect read and write statements. In particular, these checks match only the name signature of methods without performing additional checks on the full method signatures. Then, for each read-write tuple, they check the actual parameters in each key position and perform a syntactic check on them. These kinds of checks have limited expressiveness. They might consider the signatures of wrong methods with names equal to the Hyperledger APIs but different packages, leading to false positives. Moreover, regarding the keys, they do not infer the value passed by the actual parameter but perform checks on the token syntax of actual parameters. For instance, this means that two variables used as keys with an equal stored value but named differently will be considered different by the analyses, leading to false negatives. We experimentally compared these tools with our analysis, showing that our approach achieves significantly better results in terms of recall and precision.

Lv et al. [20] and Yamashita et al. [40] propose similar tools inspired by Chaincode Scanner and Reviveˆ CC, but they cover more issues and more accurately. However, they seem to be still performing mostly similar checks for read-after-write issues. Shah et al. [37] describe an additional solution based on ASTs. While promising, these three approaches report no information on key detection, and, unfortunately, their implementations are not publicly available, making it impossible to compare them with our approach. Li et al. [19] provide a tool based on ASTs and symbolic execution [17] for several HF issues. However, for the *read-after-write* detection, they perform only intra-procedural reasoning on ASTs without exploiting symbolic execution. Differently, Kim et al. [16] analyze call stacks to check if a `GetState` is called after a `PutState`, using symbolic execution with a SMT solver to match the keys of both calls. Another solution is proposed by Ren et al. [34]. They study the possible patterns of read-after-write issues, also considering `defer` instructions and inter-procedural functions, providing a static analysis-based detection scheme for *read-after-write* issues. Their tool uses pattern matching to extract features related to read and write statements from a pruned AST form of smart contracts. Subsequently, it builds one-way linked lists to represent the calling sequences of read and write methods, considering also inter-procedural bindings. Such lists are then used to detect read-after-write issues. As reported by the author, the tool can perform better than other tools like Reviveˆ CC, reducing the false negative rate and improving the accuracy rate. To the best of our knowledge, there are no tools other than GoLiSA able to detect over-write issues of HF.

7.2.1 Comparison with Our Approach. Tab. 4 and Tab. 5 propose a summary view of the tool comparison, summarizing the main differences between our approach w.r.t. other tools and coverage of instructions considered during the analysis for each tool, respectively. Unfortunately, most of the tools proposed (except Reviveˆ CC and Chaincode Analyzer) in the section above are not publicly available. For this reason, the comparison is based on a theoretical level, analyzing their available documentation and scientific literature. We indicate with *n.d.* when there is no data about that since tools are not publicly available and manuscripts do not specify if a method is covered. Regarding [37], the tool is dedicated to the analysis of Node.js chaincodes, but Node.js API names are similar and can be mapped to the Go ones.

As reported by Tab. 4 and Tab. 5, GoLiSA supports the complete list of write and read statements and exploits a semantic string analysis to detect the possible values of the keys rather than a syntactic pattern matching approach. In addition, GoLiSA does not require additional data structures (such as the linked lists used in [34]) to detect when a write is executed before a read statement since GoLiSA can perform search algorithms directly on the program. Moreover, GoLiSA pursues sound analysis, avoiding false negatives. Currently, the closest approach to ours is the one reported in [16]. The main difference is in the key detection technique, which involves a symbolic execution with an SMT solver. However, symbolic execution can suffer several drawbacks (such as path explosion, expensive constraint solving, and unsolvable paths). However, since the tool was not available, evaluating it and highlighting possible issues in this regard was impossible. Our string analysis, based on abstraction interpretation, can scale even in cases where the symbolic execution fails.

8 CONCLUSION

This paper proposes a semantics-based static analysis approach based on abstract interpretation for detecting read-write set issues. The experiments on real-world chaincodes, crawled from GitHub, empirically prove it useful and scalable in practice. Moreover, compared to the current state of the art, our approach is sound, leading to the absence of false negatives, and it can detect read-write set issues before chaincodes are deployed in a blockchain that would become immutable once they are stored in the blockchain. Future work will explore aliasing analysis [28] to reduce false positives and investigate alternative abstract string value domains for improved key detection. Additionally, we plan to extend the approach to propose fixes or patches for the detected issues, providing a more comprehensive solution.

ACKNOWLEDGMENTS

Work partially supported by SERICS (PE00000014 - CUP H73C2200089001) and iNEST (ECS00000043 – CUP H43C22000540006) projects funded by PNRR NextGeneration EU, and by Bando di Ateneo per la Ricerca 2022, funded by University of Parma, (MUR_DM737_2022_FIL_PROGETTI _B_ARCERI_COFIN, CUP: D91B210 05370003), "Formal verification of GPLs blockchain smart contracts".

REFERENCES

[1] Jade Alglave and Patrick Cousot. 2017. Ogre and Pythia: an invariance proof method for weak consistency models. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. ACM, 3–18. https://doi.org/10.1145/3009837.3009883

[2] Available at https://github.com/mgechev/revive

Table 4: Tool comparison.

Tool	Key Detection	Write-After-Read Check	Over-Write Check	Interprocedural
Revive`CC [38]	syntactic (pattern matching)	search in ASTs	Not supported	No
ChaincodeAnalyzer [18]	syntactic (pattern matching)	search in ASTs	Not supported	No
Lv et al. [20]	syntactic (pattern matching)	search in ASTs	Not supported	No
Yamashita et al. [40]	syntactic (pattern matching)	search in ASTs	Not supported	No
Shah et al. [37]	n.d. (presumably syntactic)	search in ASTs	Not supported	No
Li et al. [19]	n.d. (presumably syntactic)	search in ASTs	Not supported	No
Ren et al. [34]	syntactic (pattern matching)	search in ASTs + linked lists	Not supported	Yes
Kim et al. [16]	semantic (symbolic execution)	search in CFGs	Not supported	Yes
GoLiSA	semantic (abstract interpretation)	search in CFGs	search in CFGs	Yes

Table 5: Tool coverage of read and write instructions.

Instruction	[38]	[18]	[20]	[40]	[37]	[19]	[34]	[16]	GoLiSA
`PutState`	✓	✓	✓	✓	✓	✓	✓	✓	✓
`DelState`	✗	✗	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	✓
`SetStateValidationParameter`	✗	✗	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	✓
`PutPrivateData`	✗	✗	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	✓
`DelPrivateData`	✗	✗	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	✓
`PurgePrivateData`	✗	✗	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	✓
`SetPrivateDataValidationParameter`	✗	✗	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	✓
`GetState`	✓	✓	✓	✓	✓	✓	✓	✓	✓
`GetStateValidationParameter`	✗	✗	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	✓
`GetStateByRange`	✗	✗	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	✓
`GetStateByRangeWithPagination`	✗	✗	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	✓
`GetStateByPartialCompositeKey`	✗	✗	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	✓
`GetHistoryForKey`	✗	✗	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	✓
`GetPrivateData`	✗	✗	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	✓
`GetPrivateDataValidationParameter`	✗	✗	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	✓
`GetPrivateDataByRange`	✗	✗	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	✓
`GetPrivateDataHash`	✗	✗	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	✓
`GetPrivateDataByPartialCompositeKey`	✗	✗	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	✓

[2] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, et al. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the 13th EuroSys Conference (EuroSys '18)*. ACM, Article 30, 15 pages. https://doi.org/10.1145/3190508.3190538

[3] Vincenzo Arceri and Isabella Mastroeni. 2020. A Sound Abstract Interpreter for Dynamic Code. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing (SAC '20)*. Association for Computing Machinery, New York, NY, USA, 1979–1988. https://doi.org/10.1145/3341105.3373964

[4] Vincenzo Arceri, Isabella Mastroeni, and Sunyi Xu. 2020. Static analysis for ECMAScript string manipulation programs. *Applied Sciences* 10, 10 (2020), 3525.

[5] Agostino Cortesi and Martina Olliaro. 2018. M-String Segmentation: A Refined Abstract Domain for String Analysis in C Programs. In *2018 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. 1–8. https://doi.org/10.1109/TASE.2018.00009

[6] Giulia Costantini, Pietro Ferrara, and Agostino Cortesi. 2011. Static Analysis of String Values. In *Formal Methods and Software Engineering*, Shengchao Qin and Zongyan Qiu (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 505–521.

[7] Patrick Cousot. 2021. *Principles of Abstract Interpretation*. MIT Press.

[8] Pietro Ferrara. 2008. Static Analysis Via Abstract Interpretation of the Happens-Before Memory Model. In *Tests and Proofs - 2nd International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings (Lecture Notes in Computer Science)*, Bernhard Beckert and Reiner Hähnle (Eds.), Vol. 4966. Springer, 116–133. https://doi.org/10.1007/978-3-540-79124-9_9

[9] Pietro Ferrara. 2013. A generic static analyzer for multithreaded Java programs. *Softw. Pract. Exp.* 43, 6 (2013), 663–684. https://doi.org/10.1002/spe.2126

[10] Pietro Ferrara, Luca Negrini, Vincenzo Arceri, and Agostino Cortesi. 2021. Static Analysis for Dummies: Experiencing LiSA. In *Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP 2021)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3460946.3464316

[11] Jim Gray and Andreas Reuter. 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

[12] Hyperledger. 2024. A Blockchain Ledger. https://hyperledger-fabric.readthedocs.io/en/release-2.5/ledger/ledger.html#a-blockchain-ledger (Accessed 12/2023).

[13] Hyperledger. 2024. Chaincode Terminology. https://hyperledger-fabric.readthedocs.io/en/release-2.5/peers/peers.html?highlight=validation#chaincode-terminology (Accessed 01/2024).

[14] Hyperledger. 2024. Private Data - A use case to explain collections. BLOGCACM, https://hyperledger-fabric.readthedocs.io/en/lrelease-2.2/private-data/private-data.html#a-use-case-to-explain-collection Accessed: 09/2024.

[15] Hyperledger. 2024. Read-Write set semantics. https://hyperledger-fabric.readthedocs.io/en/release-2.2/readwrite.html#read-write-set-semantics. Accessed 09/2024.

[16] Sangsoo Kim, Yunsik Son, and Yangsun Lee. 2020. A Study on Chaincode Security Weakness Detector in Hyperledger Fabric Blockchain Framework for IT Development. *Journal of Green Engineering* 10 (2020), 7820–7844.

[17] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (jul 1976), 385–394. https://doi.org/10.1145/360248.360252

[18] kzhry. 2021. Chaincode Analyzer. https://github.com/hyperledger-labs/chaincode-analyzer. Accessed 07/2023.

[19] Peiru Li, Shanshan Li, Mengjie Ding, Jiapeng Yu, He Zhang, Xin Zhou, and Jingyue Li. 2022. A Vulnerability Detection Framework for Hyperledger Fabric Smart Contracts Based on Dynamic and Static Analysis. In *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering (EASE '22)*. ACM, New York, NY, USA, 366–374. https://doi.org/10.1145/3530019.3531342

[20] Penghui Lv, Yu Wang, Yazhe Wang, and Qihui Zhou. 2021. Potential Risk Detection System of Hyperledger Fabric Smart Contract based on Static Analysis. In *IEEE Symposium on Computers and Communications, ISCC'21*. IEEE, 1–7. https://doi.org/10.1109/ISCC53001.2021.9631249

[21] Antoine Miné. 2011. Static Analysis of Run-Time Errors in Embedded Critical Parallel C Programs. In *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Saarbrücken, Germany (LNCS)*, Vol. 6602. Springer, 398–418. https://doi.org/10.1007/978-3-642-19718-5_21

[22] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*. ACM, 308–319. https://doi.org/10.1145/1133981.1134018

[23] Luca Negrini, Vincenzo Arceri, Agostino Cortesi, and Pietro Ferrara. 2024. Tarsis: An effective automata-based abstract domain for string analysis. *Journal of Software: Evolution and Process* (2024), e2647. https://doi.org/10.1002/smr.2647

[24] Luca Negrini, Vincenzo Arceri, Pietro Ferrara, and Agostino Cortesi. 2021. Twinning Automata and Regular Expressions for String Static Analysis. In *Verification, Model Checking, and Abstract Interpretation*, Fritz Henglein, Sharon Shoham, and Yakir Vizel (Eds.). Springer International Publishing, Cham, 267–290.

[25] Luca Negrini, Vincenzo Arceri, Luca Olivieri, Agostino Cortesi, and Pietro Ferrara. 2024. Teaching Through Practice: Advanced Static Analysis with LiSA. In *Formal Methods Teaching*, Emil Sekerinski and Leila Ribeiro (Eds.). Springer Nature Switzerland, Cham, 43–57. https://doi.org/10.1007/978-3-031-71379-8_3

[26] Luca Negrini, Pietro Ferrara, Vincenzo Arceri, and Agostino Cortesi. 2023. *LiSA: A Generic Framework for Multilanguage Static Analysis.* Springer Nature Singapore, Singapore, 19–42. https://doi.org/10.1007/978-981-19-9601-6_2

[27] Robert HB Netzer and Barton P Miller. 1992. What are race conditions? Some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)* 1, 1 (1992), 74–88.

[28] DJurica Nikolić and Fausto Spoto. 2012. Definite expression aliasing analysis for Java bytecode. In *International Colloquium on Theoretical Aspects of Computing*. Springer, 74–89.

[29] Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid dynamic data race detection. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming, PPOPP 2003, June 11-13, 2003, San Diego, CA, USA*. ACM, 167–178. https://doi.org/10.1145/781498.781528

[30] Luca Olivieri, Luca Negrini, Vincenzo Arceri, Badaruddin Chachar, Pietro Ferrara, and Agostino Cortesi. 2024. Detection of Phantom Reads in Hyperledger Fabric. *IEEE Access* 12 (2024), 80687–80697. https://doi.org/10.1109/ACCESS.2024.3410019

[31] Luca Olivieri, Luca Negrini, Vincenzo Arceri, Fabio Tagliaferro, Pietro Ferrara, Agostino Cortesi, and Fausto Spoto. 2023. Information Flow Analysis for Detecting Non-Determinism in Blockchain. In *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States (LIPIcs)*, Karim Ali and Guido Salvaneschi (Eds.), Vol. 263. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 23:1–23:25. https://doi.org/doi.org/10.4230/LIPIcs.ECOOP.2023.23

[32] Luca Olivieri, Luca Negrini, Vincenzo Arceri, Fabio Tagliaferro, Pietro Ferrara, Agostino Cortesi, and Fausto Spoto. 2023. Information Flow Analysis for Detecting Non-Determinism in Blockchain. In *37th European Conference on Object-Oriented Programming (ECOOP 2023) (Leibniz International Proceedings in Informatics (LIPIcs))*, Karim Ali and Guido Salvaneschi (Eds.), Vol. 263. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 23:1–23:25. https://doi.org/10.4230/LIPIcs.ECOOP.2023.23

[33] Luca Olivieri, Fabio Tagliaferro, Vincenzo Arceri, Marco Ruaro, Luca Negrini, Agostino Cortesi, Pietro Ferrara, Fausto Spoto, and Enrico Talin. 2022. Ensuring Determinism in Blockchain Software with GoLiSA: An Industrial Experience Report. In *Proceedings of the 11th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP 2022)*. Association for Computing Machinery, New York, NY, USA, 23–29. https://doi.org/10.1145/3520313.3534658

[34] Feixiang Ren and Sujuan Qin. 2023. Fabric Smart Contract Read-After-Write Risk Detection Method Based on Key Methods and Call Chains. In *Smart Computing and Communication*. Springer Nature Switzerland, Cham, 381–392. https://doi.org/10.1007/978-3-031-28124-2_36

[35] Mark Richards. 2015. *Software architecture patterns.*

[36] Xavier Rival and Kwangkeun Yi. 2020. *Introduction to static analysis: an abstract interpretation perspective.* Mit Press.

[37] Neelkumar K. Shah, Sachin Nandurkar, Mayur Bilapate, Mohammad Aziz Maalik, Niteshkumar Harne, Karimullah Shaik, and Ajai Kumar. 2023. Smart Contract Vulnerability Detection Techniques for Hyperledger Fabric. In *2023 IEEE 8th International Conference for Convergence in Technology (I2CT)*. 1–7. https://doi.org/10.1109/I2CT57861.2023.10126362

[38] sivachokkapu. 2021. ReviveCC. https://github.com/sivachokkapu/revive-cc. Accessed 07/2023.

[39] Andrew S Tanenbaum. 2007. *Distributed systems principles and paradigms.*

[40] Kazuhiro Yamashita, Yoshihide Nomura, Ence Zhou, Bingfeng Pi, and Sun Jun. 2019. Potential Risks of Hyperledger Fabric Smart Contracts. In *2019 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. 1–10. https://doi.org/10.1109/IWBOSE.2019.8666486