IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 3 May 2024, accepted 1 June 2024, date of publication 5 June 2024, date of current version 14 June 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3410019

RESEARCH ARTICLE

Detection of Phantom Reads in Hyperledger Fabric

LUCA OLIVIERI[1], LUCA NEGRINI[1], VINCENZO ARCERI[2], BADARUDDIN CHACHAR[1], PIETRO FERRARA[1], AND AGOSTINO CORTESI[1]

[1]Department of Environmental Sciences, Informatics and Statistics, Foscari University of Venice, 30123 Venice, Italy
[2]Department of Mathematical, Physical and Computer Sciences, University of Parma, 43121 Parma, Italy

Corresponding author: Vincenzo Arceri (vincenzo.arceri@unipr.it)

This work was supported in part by Bando di Ateneo per la Ricerca 2022, funded by the University of Parma, under Grant MUR_DM737_2022_FIL_PROGETTI_B_ARCERI_COFIN and Grant CUP: D91B210 05370003; in part by the Formal Verification of General-Purpose Languages (GPLs) Blockchain Smart Contracts; in part by SERICS under Grant PE00000014-CUP H73C2200089001; and in part by the Interconnected Nord-Est Innovation Ecosystem (iNEST) projects funded by the Piano Nazionale Ripresa Resilienza (PNRR) NextGeneration European Union (EU) under Grant ECS00000043-CUP H43C22000540006.

ABSTRACT In concurrent transactional systems, a *phantom read* occurs when a transaction retrieves a set of data, and simultaneously, new data is inserted, updated, or removed from that set by one or more other transactions, leading to unexpected data being read. In Hyperledger Fabric (HF), a popular enterprise-grade framework for developing permissioned blockchain platforms, phantom reads are detected during the transaction validation phase. It inspects the values from read operations and checks their consistency, also re-executing some domain-specific read operations called *range queries*. However, being HF based on an optimistic concurrency control model, managing an excessive number of conflicts related to phantom reads could result in sudden system slowdowns. Additionally, some kind of range queries are not considered in the validation and verification process. For the latter, the re-execution is not performed and checks are not provided leading to undetected phantom reads when the values returned from them are written to the ledger. Hence, the burden of implementing phantom read-free applications (i.e., *smart contracts*) is on the developers, who need to correctly manage the read instructions in the code and use automatic verification tools to detect any unsafe implementations leading to system slowdowns and undetectable phantom reads. In this paper, we explore the phantom reads detection problem at the smart contract level and demonstrate how a verification approach through formal methods can identify possible bottlenecks caused by phantom reads and mitigate range query risks, outperforming the current state-of-the-art and state-of-the-practice for their detection. Our approach is implemented with GoLiSA, a semantic static analyzer based on abstract interpretation for Go applications.

INDEX TERMS Smart contracts, blockchain, hyperledger fabric, static analysis, formal verification, phantom reads detection.

I. INTRODUCTION

Over the last decade, there has been a growing interest in blockchain-based ledgers among businesses. However, popular *permissionless* blockchains such as Ethereum [1], [2] cannot meet strict and solid industrial requirements for confidentiality, scalability, and flexibility. In this context, HF [3], [4] stands out as one of the most widely adopted frameworks, even by leading cloud providers such as AWS, Azure, IBM, Google, and Oracle [5]. HF is an open-source platform hosted by the Linux Foundation. It is designed for developing custom *permissioned* blockchain ledgers and tailored for enterprise applications offering a modular architecture and features that align with the unique requirements of business-oriented applications.

Among its several features, HF introduces a new transaction architecture based on a *simulate-order-validate-commit* model (also known as the *execute-order-validate* model)

The associate editor coordinating the review of this manuscript and approving it for publication was Muhammad Aleem.

[6], [7]. This models employs an optimistic concurrency control strategy for parallel transaction processing, increasing throughput and scalability. This approach assumes that transactions are likely to be conflict-free, allowing for the concurrent execution of transactions during the execution/simulation and ordering phases, under the assumption that they do not read and write to common states, thus avoiding conflicts such as phantom reads [8], [9], [10]. This assumption is optimistic, since conflicts may occur but are expected to be not frequent. Indeed, conflicts are handled in the subsequent validation phase, where performance bottlenecks can arise if too many conflicts are detected [10]. In this phase, all detected conflicts are resolved by rejecting one or more of the conflicting transactions. The rejected transactions become invalid and their effects do not persist, while and non-conflicting transactions are committed to the blockchain. Clients are then responsible for submitting new transactions incorporating the necessary changes from previously rejected transactions. Moreover, developers must be aware of potential race conditions and design their smart contracts to minimize conflicts, ensuring transactions execute correctly and avoiding potential bottlenecks [10].

Although HF detects most conflicts, some data query instructions are not taken into account during the validation phase, leading to hidden phantom reads (also known as *range query risk*). Developers must take measures to detect these occurrences, enhancing their development pipeline with tools or architectures capable of identifying such phantom reads.

The novel contributions of this paper can be summarized as follows:

- a comprehensive summary of the issues related to phantom reads in HF;
- an investigation of the state-of-the-art in detecting phantom reads and *range query risk* in HF;
- the design and implementation of an analysis for detecting phantom reads and range query risk using static analysis through abstract interpretation [11], [12], i.e. providing formal guarantees and ensuring the detection of phantom reads for all possible program execution paths.

To the best of our knowledge, this is the first analysis based on formal methods for detecting phantom reads and range query risk at the smart contract level in HF.

Paper Structure: Section II introduces background concepts related to HF. Section III discusses phantom reads in HF. Section IV presents related work. Section V describes the adopted methodology to address phantom read detection and a comparison of our approach with the state-of-art tools. Section VI, Section VII, and Section VIII provide our core contribution's design and implementation details for detecting phantom reads, and discuss the proposed approach also with a practical example. Section IX concludes the paper.

II. HYPERLEDGER FABRIC FEATURES

This section provides a detailed and comprehensive overview of the key concepts of HF, essential for understanding the content and terminology used throughout this paper.

A. BLOCKCHAIN-BASED LEDGER OF HF

In HF, the blockchain-based ledger [13] consists of two distinct, though related, components (see Figure 1):

- **World state**, a database that maintains the current values of a set of ledger states, typically implemented through LevelDB or CouchDB [14]. The world state enables programs to directly access the current state values without needing to calculate them by traversing the entire transaction log. By default, ledger states are represented as versioned key-value pairs. The world state can change over the time, as states can be added, updated, and deleted.
- **Blockchain**, a transaction log collection that records all the changes that define the world state. Transactions are grouped into blocks, which are then appended to the blockchain. This allows for the history of state changes leading to the current world state to be accurately recorded. Unlike the world state, the blockchain structure is immutable; it is a permanent sequence of blocks, each containing a set of ordered transactions.

In HF's distributed ledger technology, the ledger is distributed across a decentralized network, where redundant copies of the ledger are maintained by the peers. Its distribution is managed through a consensus mechanism, enabling peers to reach an agreement on the complete transaction process, from proposal and endorsement to ordering, validation, and finally, the commitment of transactions.

B. SMART CONTRACTS IN HF

Smart contracts are computer programs stored on a blockchain-based ledger that can be executed through transactions. In HF, smart contracts are implemented through a type of logic called *chaincode* [15], which use specific APIs to read and write the ledger and are mainly written in Go for efficiency reasons [16]. In this paper, we use the more specific term *chaincode*, but can be interpreted simply as smart contract.

C. QUERY IN HF

HF defines a query as a chaincode function that reads from the current world state, potentially targeting specific keys or a range of keys on the ledger. HF also accommodates *rich* queries, which can return data values in addition to keys, provided the underlying database (e.g., CouchDB) supporting such functionality.

III. THE PHANTOM READ PROBLEM

Phantom reads are a common problem in data collection systems concerning transaction contexts and simultaneous

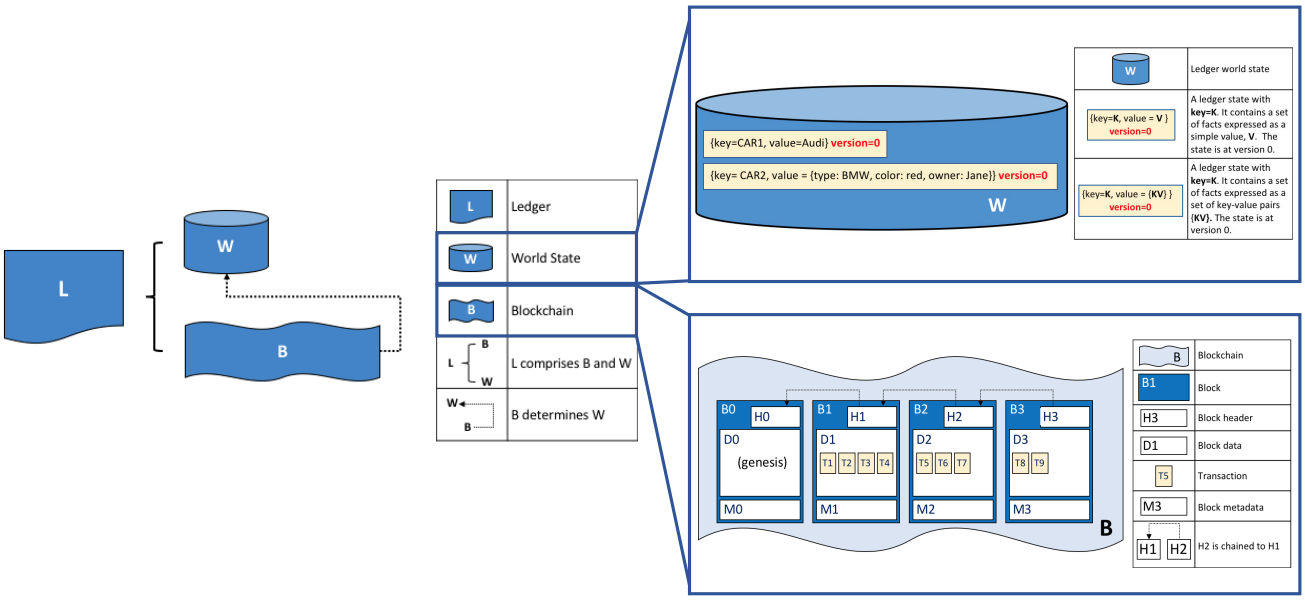

FIGURE 1. Blockchain ledger architecture of HF [13].

data access, which also affects HF. Generally, a phantom read can occur when one or more transactions write a state (such as insertion, updating, or removal) that another transaction, or more than one transaction, reads. Consequently, the transaction performing the read might encounter *"phantom"* data that was either missing at the beginning of the transaction or present at the beginning and then absent, leading to possible inconsistencies.

A. PHANTOM READS IN HF

In HF, when a transaction for a chaincode execution is received by the network, peers simulate the code execution, collecting all the read and write operations acting on the blockchain into read-write sets [17]. At the end of the simulation, these sets are inspected and verified against phantom reads during the validation phase of the consensus mechanism. As reported by the official HF documentation [17], a transaction is considered valid if the version of each key present in the read set of the transaction matches the version for the same key in the world state, assuming all the preceding valid transactions are committed (including the preceding transactions in the same block); in simple terms, a valid transaction does not contain any inconsistencies in the data it reads. An additional check is performed if the read-write sets contain one or more information from queries to ensure that no key has been added, deleted, or updated in the super range (i.e., the union of the ranges) of the results captured in the information from queries. In other words, the queries are re-executed during the validation phase to ensure that the result set has not changed since transaction endorsement. If the check detects phantom reads, the transaction fails with the status code `PHANTOM_READ_CONFLICT`.

However, this re-execution does not cover all query methods during the validation phase. From version `1.x` to the current `2.x` of HF, some of these methods are deliberately omitted by HF, as reported in the API documentation [18]. Table 1 provides the complete list of query functions, as of the latest public version `v2.5.5`, showing which are re-executed or not in relation to phantom reads. Moreover, as reported by the documentation [18], for the queries not re-executed during the validation phase, phantom reads are not detected. This means that other committed transactions could have added, updated, or removed keys affecting the result set, and these changes would not be detected at validation/commit time. Therefore, chaincodes using these query functions should not use the returned values of queries to update the ledger but should only perform read-only operations. These types of phantom reads are also known as *range query risk* [8], [9].

Summarizing the issues, phantom reads in HF can lead to:

- a degradation of the performance and scalability of the system based on HF [10], due to the optimistic concurrency model's inefficiency in handling runtime conflicts from excessive phantom reads;
- inconsistencies, especially due to range queries that are not re-executed.

This implies that the burden of preventing phantom reads is left to chaincode developers, who need to employ proper program verification to avoid such vulnerabilities.

IV. RELATED WORK

In this section, we investigate the current state-of-art and state-of-practice to highlight the main shortcomings of various approaches. Then, we explain in detail how our novel solution addresses them in Section V.

TABLE 1. List of query functions in HF. The superscript '+' denotes rich queries.

Package	Type/Interface	Query Function	Re-executed
shim	ChaincodeStubInterface	GetStateByRange	✓
		GetStateByPartialCompositeKey	✓
		GetQueryResult+	✗
		GetHistoryForKey	✗
		GetPrivateDataByRange	✓
		GetPrivateDataByPartialCompositeKey	✓
		GetPrivateDataQueryResult+	✗

A. LITERATURE REVIEW

In the literature, the problem of phantom read detection for HF is addressed on two different levels: (i) system level and (ii) smart contract level.

System-level solutions can offer several benefits, not only impacting phantom reads decreasing the number of invalid transactions but also mitigating other conflicts, such as multi-version concurrency control read conflicts (indicated by the status code `MVCC_READ_CONFLICT`). A taxonomy of state-of-the-art approaches for preventing transaction conflicts at the system-level is provided by Debreczeni et al. [7]. For instance, Sharma et al. [6] and Ruan et al. [19] describe how to combine transaction reordering and early abort of transaction. In a nutshell, during the ordering phase, their architectures inspect transaction semantics and arrange the transactions in a way such that the number of serialization conflicts is drastically reduced; then, it removes transactions that have no chance to commit anymore, as early as possible from the pipeline. However, to the best of our knowledge, we have not found any system-level solution dealing with range query risks, which currently still allows hidden and unhandled phantom reads. Moreover, applying these solutions in enterprise environments poses significant challenges. Indeed, they often require an extensive rework involving the entire or a portion of the HF architecture, which may discourage IT companies from adopting these system-level solutions. Such companies typically require high degrees of software maturity, including stable and long-term supported versions.

On the other hand, smart contract level solutions focus on designing chaincode that inherently avoids phantom reads. This is made possible by verification tools that detect these issues during development, i.e., analyzing the chaincode application software. Compared to the system-level approaches, this method only requires developers to analyze and make minor changes to the chaincode upgrading it [20], thus without affecting any component of the HF architecture and keeping the official HF frameworks unchanged. Currently, only a few verification tools are available for HF [8], [9], [21], [22] and these tools typically do not offer formal assurances regarding their results and findings, due to the complexity of developing such tools from scratch or adapting existing formal verification libraries [23].

B. TOOL REVIEW

To prevent phantom reads at the smart contract level, it is required to detect phantom reads during development, i.e., analyzing the chaincode application software using verification tools. In general, commercial verification tools such as SonarQube [24], CodeSonar [25], and Julia [26] were widely adopted to detect security issues and program bugs of software. Smart contracts have been a focus of several works in this field [27], [28], [29]. However, although several studies investigate issues related to HF [8], [9], [10], [30], [31], there are currently only a few analyzers able to detect phantom reads, with even fewer being publicly available.

One initial solution is to limit the program's expressiveness by blacklisting the queries that lead to the phantom reads. Chaincode Analyzer [21] adopts this approach, inherently limiting API usage and significantly reducing the benefits of adopting HF queries, even when the code does not pose any harm (read-only) to the blockchain ledger. Chaincode Analyzer employs syntactic checks on the an abstract syntax tree (AST) representation to verify the function's signatures within the program. If they match the signatures of a blacklisted query, an alarm is issued. However, this check is limited to the signature and does not verify whether the return value of the query is involved in write operations (e.g., `PutState`, `PutPrivateData`). Regarding the blacklisted instructions, Chaincode Analyzer does not cover the full list of possible instructions leading to phantom read issues, missing `GetPrivateDataQueryResult` (see Table 3).

Another approach is the one implemented by Reviveˆ CC [22], that inspects the syntax tree of the program to perform a symbolic data flow analysis. It checks whether the same variable name is used to receive the value from a critical query and a state write operation. Reviveˆ CC only checks read operations related to range query risks and also misses `GetPrivateDataQueryResult`. Moreover, it only considers `PutState` as write state operation, omitting several others, as reported in Table 3. Moreover, Reviveˆ CC's analysis is intraprocedural, namely if the write operation is located in a function to which a value of a range query has been passed as a parameter, the analysis will not detect the potential phantom read. In addition, Reviveˆ CC merely tracks variables named identically. However, there is no assurance that variables named identically contain the same query value or a derivative of it, especially in the Go languages which allows developers to declare a variable with the same name in an inner block [32].

Other tools exist for verifying phantom reads, but they are not publicly accessible, making it challenging to fully understand their functioning from the scientific

literature only. However, these tools are significantly influenced by Chaincode Analyzer and Reviveˆ CC. In particular, Yamashita et al. [8] and Penghui et al. [9] claim to detect range query risks on `GetHistoryForKey`, `GetQueryResult`, and `GetPrivateDataQueryResult`. However, Yamashita et al. [8] do not provide information on how the check is performed, while Penghui et al. [9] describe a blacklisting approach on reachable statements considering the call paths from the `Invoke` function, which is a typical entry point for chaincodes.

V. METHODOLOGY

In this work, we focus on detecting potential conflicts at the smart contract level, i.e., without impacting the official HF architecture. Then, the developers can detect these issues and change their code [20] to avoid range query issues (hence, data inconsistencies) and conflicts (hence, invalid transactions and bottlenecks).

To propose a novel solution that improves the state of the art, we investigated in depth the existing tools proposed above and we highlighted three main critical shortcomings: (i) the incomplete instruction coverage, (ii) a partial detection of phantom reads written in the ledger, (iii) the lack of formal guarantees.

Our core contribution is the design and implementation of an analysis that fills these shortcomings. In particular, in Sections VI and VII, we describe our solution based on information flow analysis for detecting phantom reads and range query risk using static analysis through abstract interpretation [11], [12]. As far as we know, this is the first information flow analysis applied to HF for the detection of this type of issue.

In a nutshell, the idea is to have complete coverage for the detection of instructions that can lead to phantom reads and all instructions that allow one to write the ledger. Then, the information flow analysis, starting from the instructions that can generate phantom reads, checks if their values do end up in the ledger's write instructions leading to the phantom read issues. We also choose to adopt a program representation based on Control Flow Graphs [33] (CFGs) instead of ASTs, because ASTs focus on representing the structure of the code based on its syntax, while CFGs focus on representing the execution paths within the code. Moreover, basing the analysis on abstract interpretation, our approach can consider program semantics improving the precision compared with syntactic checks of existing tools and also cover all possible program execution paths, formally ensuring the detection of phantom reads.

Specifically, our approach does not suffer from the limitations of blacklisting, i.e. the complete removal of every instruction that can generate phantom reads, as it only alerts when there is the possibility that phantom read values may be written in the ledger. In addition, Reviveˆ CC-like approaches do not provide formal guarantees about their findings, which, consequently, can lead to several false positives, or, in worst-case scenarios, false negatives. Instead, our approach offers several mathematical guarantees based on formal methods, such as the absence of false negatives (discussed in Section VII-A) as well as being interprocedural, i.e., examines the behavior of a program across multiple methods and functions calls, analyzing how the different interactions between them affect the overall behavior of the program. Therefore, this allows one to achieve full program coverage during the analysis considering also the peculiarities of languages such as multiple variable declarations in different inner blocks.

Tables 2 and 3 propose a summary view of tool comparison and instruction coverage, respectively, comparing our approach with the tools mentioned above. Note that, the notation $'n.d.'$ means that no data are available for that entry, while '$-$' symbol indicates not applicable. Specifically, for tools that use blacklisting, only read operations are taken into account, and write operations are not included in the analysis. As shown in Table 2, our approach is the only one that performs analysis based on program statement semantics providing a comprehensive understanding of chaincode behaviors allowing it to be more precise in terms of analysis as well as is one of the few to perform an interprocedural analysis. Furthermore, our approach seems to be currently the only one to cover issues from both re-executed and not re-executed statements (see Section III), i.e., it can prevent the bottlenecks due to the optimistic concurrency model's inefficiencies and avoid data inconsistencies, respectively. Regarding Table 3, our approach currently covers all related instructions that can perform phantom reads and also instructions that can write values to the HF ledger.

VI. DETECTION OF PHANTOM READS BY INFORMATION FLOW ANALYSIS

Program verification can be applied from the very beginning of the chaincode implementation. In the blockchain context, this aspect is particularly relevant as it allows for code verification before deployment, i.e., before the code becomes difficult to patch. In particular, static analysis can automatically verify the properties of computer programs without executing the code [34], thereby reducing the burden on developers for bug fixing and giving them the chance to fix issues and code smells at an early stage [35]. In this section, we address the problem of detecting phantom reads using static analysis and formal methods, specifically through abstract interpretation [36] and information flow analysis.

In a nutshell, abstract interpretation allows one to reason about program semantics by computing its abstract semantics, sacrificing precision to gain computability. Properties proved in the abstract semantics are guaranteed to hold in the concrete (i.e., actual) semantics, enabling the verification of correctness and safety.

Information flow analysis [37], [38] examines how information moves through a program or system. Typically, it is used to identify if private input (such as sensitive data or untrusted data) flows explicitly (i.e., through assignments) or

TABLE 2. Smart contract's analysis tool comparison.

Tool	Range Query Risk (Not Re-Execution) Detection	Phantom Read (Re-Execution) Detection	Interprocedural	Available For Evaluation
Chaincode Analyzer [21]	syntatic check on AST (black-listing)	✗	✗	✓
Revive^CC [22]	syntatic check on AST (symbolic data flow)	✗	✗	✓
Yamashita et al. [8]	*n.d.* (presumably syntactic on AST)	*n.d.*	✗	✗
Lv et al. [9]	syntactic on AST + call paths (black-listing)	*n.d.*	✓	✗
Our Approach	semantic check on CFGs (taint analysis)	semantic check on CFG (taint analysis)	✓	✓

TABLE 3. Instruction coverage of smart contract's analysis tools for range query risk.

Instruction	Type	*Chaincode Analyzer [21]*	*Revive^CC [22]*	*Yamashita et al. [8]*	*Lv et al. [9]*	Our Approach
`GetHistoryForKey`	Read Operation	✓	✓	✓	✓	✓
`GetQueryResult`	Read Operation	✓	✓	✓	✓	✓
`GetPrivateDataQueryResult`	Read Operation	✗	✗	✓	✓	✓
`PutState`	Write Operation	-	✓	*n.d.*	-	✓
`DelState`	Write Operation	-	✗	*n.d.*	-	✓
`SetStateValidationParameter`	Write Operation	-	✗	*n.d.*	-	✓
`PutPrivateData`	Write Operation	-	✗	*n.d.*	-	✓
`DelPrivateData`	Write Operation	-	✗	*n.d.*	-	✓
`PurgePrivateData`	Write Operation	-	✗	*n.d.*	-	✓
`SetPrivateDataValidationParameter`	Write Operation	-	✗	*n.d.*	-	✓

implicitly (i.e., through control flow) to a public output (like unauthenticated web views or SQL query execution routine). Over the last forty years, this analysis and its generalizations have produced significant scientific and industrial outcomes. However, analyses that track both implicit and explicit information flows have seen limited industrial application, mostly due to the false positives associated with implicit flows and challenges with scalability [39]. For this reason, *taint analysis* [36, Section 47.11.8] is often preferred over traditional information flow analysis. Taint analysis is an instance of information flow analysis that focus on detecting if tainted information *explicitly* flows from specific *sources* to critical program points, traditionally called *sinks*, without any intermediate *sanitization*. Program variables, denoted by $\mathbb{V}$, are partitioned into *tainted* variables, denoted by $\mathbb{T}$, and *clean* variables, denoted by $\mathbb{C}$, where $\mathbb{V} = \mathbb{T} \cup \mathbb{C}$ and $\mathbb{T} \cap \mathbb{C} = \emptyset$. Tainted variables may contain values coming from sources, while clean variables are assured to be free from tainted values across all program executions. The analysis identifies information flows (in the form of value propagations) from $\mathbb{T}$ to $\mathbb{C}$ variables, focusing only on explicit flows to reduce false positives. In our approach, we use one level of taintedness (that is, data can be only *taint* or *clean*) to improve performance and scalability. Taint analysis has proven to be effective in identifying vulnerabilities across various real-world applications [39], [40], [41], [42], [43], [44], [45]. Additionally, taint analysis can also be applied with formal method frameworks to achieve *soundness*.

A. PHANTOM READ DETECTION AS TAINTNESS PROBLEM

The phantom reads problem can also be seen as a taintness problem. Specifically, since values returned from a query are intended to be read-only, detecting explicit flows of these values to a write state operation, can identify a phantom read conflict or a range query risk, depending on the query statement involved. We can consider the query functions (refer to Table 1) as sources, and the write state operations (refer to Table 4) as sinks. Consequently, variables in $\mathbb{T}$ are those that may contain phantom values coming from query statements, while variables in $\mathbb{C}$ are guaranteed not to contain phantom values across all possible program executions. For the sake of clarity, we have treated all range queries as sources. However, it is possible to conduct two distinct taint analyses while keeping the same sinks but splitting the sources into two different sets: one comprising the re-executed queries and the other including non-re-executed ones. This solution enables analyses to differentiate between phantom reads detected by HF, which lead to performance slowdowns, and those related to range query risks, which result in inconsistencies, respectively.

TABLE 4. List of write state functions in HF.

Package	Type/Interface	Write State Function
`shim`	`ChaincodeStubInterface`	`PutState`
		`DelState`
		`SetStateValidationParameter`
		`PutPrivateData`
		`DelPrivateData`
		`PurgePrivateData`
		`SetPrivateDataValidationParameter`

VII. IMPLEMENTATION IN GOLISA

We implemented a static taint analysis for detecting phantom reads in GoLiSA,[1] an open-source static analyzer for Go applications [43], [46]. It provides a frontend to parse and manage Go applications, and its analysis engine relies on LiSA [23], [47] (**Li**brary for **S**tatic **A**nalysis), a modular framework for developing static analyzers based on abstract interpretation. A high level overview of the GoLiSA architecture is reported in Figure 2.

After GoLiSA has successfully parsed a Go chaincode of interest and built a program representation based on CFGs, the taint analysis begins by identifying source statements within the program. This is achieved through matching the signatures of the program's query methods against those

[1] Available at https://github.com/lisa-analyzer/go-lisa

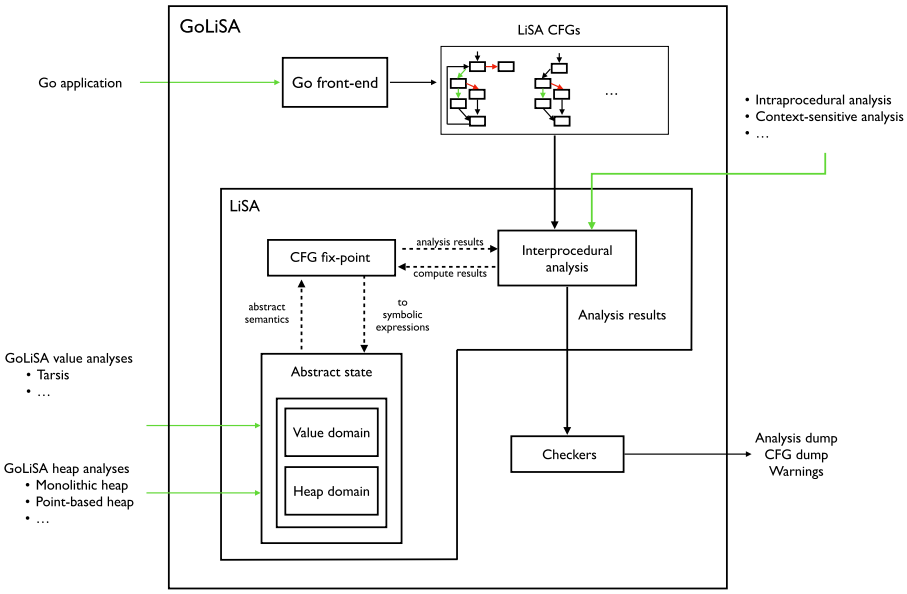

FIGURE 2. High-level overview of the GoLiSA architecture.

listed in Table 1. From these identified sources, the analysis propagates the information throughout the program. For each program point, GoLiSA determines whether each variable contains information originating from sources (i.e., whether it is tainted) or not (i.e., whether it is clean/untainted). The LiSA analysis engine then executes a program-wide fixpoint by computing each CFG's fixpoint and considering abstract semantics of each instruction.

At the end of the information propagation phase, GoLiSA checks if tainted variables flowed into a sink. If so, an alert is issued to indicate that tainted information has reached a critical program point. It is important to highlight that the analysis performed by GoLiSA is interprocedural, meaning it analyzes the program across different procedures or function calls, ensuring a thorough examination of the program's behavior.

A. APPROXIMATIONS, ABSTRACTIONS, AND SOUNDNESS

Abstract interpretation allows one to formalize a notion of soundness. An analyzer is sound concerning a property of interest if it examines all possible program executions of a target program, ensuring definite guarantees about the property. Consequently, if it does not raise any alarms, the property is ensured to be upheld in every execution. In other words, sound analyzers have no false negatives (i.e., the property holds in at least one concrete execution, but the analyzer does not detect it and no alarm is issued). To pursue soundness, we employ over-approximations of abstract semantics of instructions in GoLiSA. This ensures that our static taint analysis conservatively assumes that properties might hold, as it considers a broader spectrum of executions than concrete ones.

```
1  // ...
2  resultsIterator, err := stub.GetQueryResult(queryString)
3  if err != nil {
4    return nil, err
5  }
6
7  defer resultsIterator.Close()
8
9  sum = 0
10 for resultsIterator.HasNext() {
11   queryResult, err := resultsIterator.Next()
12   if err != nil {
13     return shim.Error([]byte("Error!"))
14   }
15   sum = sum + getValue(queryResult)
16 }
17
18 stub.PutState("sum", []byte(sum))
19 // ...
```

FIGURE 3. A fragment of chaincode leading to a range query risk.

Our proposed taint analysis follows a fully static approach. The main limitation lies in the inability to accurately capture runtime behaviors and context-specific information due to the lack of dynamic information. Furthermore, taint analysis tracks as abstract values only binary information (tainted/clean) and not all the possible values readable from the sources. Consequentially, over-approximating program behaviors and values, this approach does not exclude the presence of false positives. However, it allows users to scales with different codebases, making it suitable for projects of various sizes and complexities.

VIII. ANALYSIS IN THE PRACTICE

In this section, we assess our analysis through a simple yet expressive example scenario, highlighting the effectiveness and the practical implications of our approach.

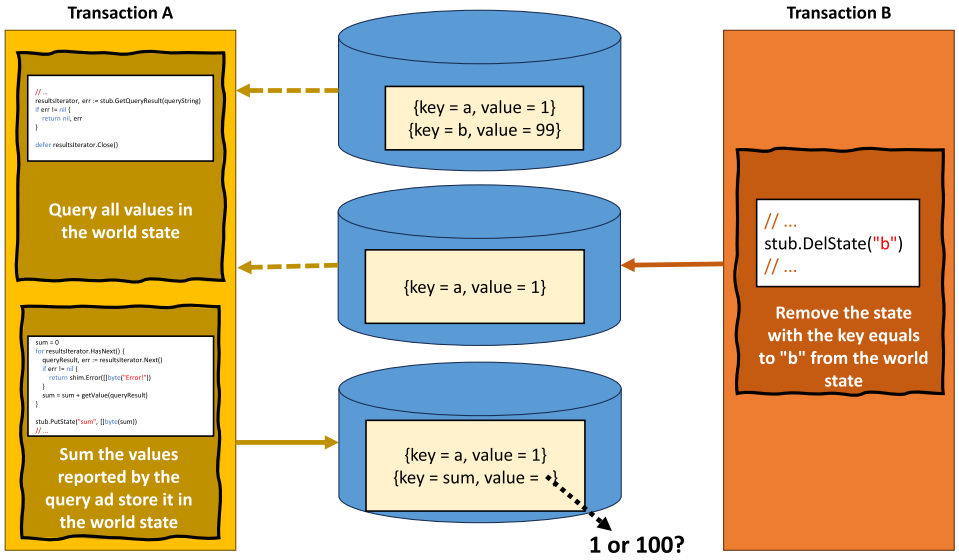

FIGURE 4. Example of phantom read in HF [13].

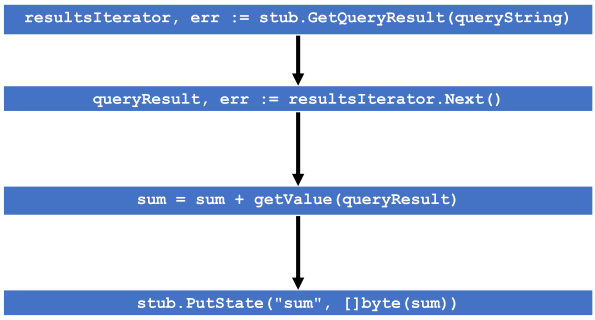

FIGURE 5. Explicit flow leading to range query risk in Figure 3.

A. EXAMPLE OF PHANTOM READ

Let us consider the fragment of chaincode reported in Figure 3. The main idea of this chaincode is to, given an arbitrary query `queryString` (written using the syntax of the underlying state database), retrieve the query results from the current world state using the `GetQueryResult` statement (line 2), iterate over the query result contained in `resultsIterator` (lines 10 − 11), extract the values of query records, and sum them in a variable `sum` (line 15), i.e., sum the values of keys matching that query in the current world state. Then, the computed `sum` value is stored using `PutState` in the world state at the end of the computation (line 18).

The code reported in Figure 3 includes the `GetQuery-Result` statement, which can lead to a potential phantom read or, more specifically, to a range query risk as it is not subject to re-execution, as reported by Table 3. High-throughput is a critical application requirement in industrial settings, meaning that chaincodes are required to handle transactions in a concurrent environment where the world state can change multiple times and quickly.

Let us now consider Figure 4. Transaction *A* proposes the execution of the chaincode described in Figure 3, while transaction *B* proposes the execution of a chaincode that removes a key-value pair from the world state. If the two transactions occur concurrently, and since there is no re-execution for transaction *A* during the validation phase, an inconsistency could arise, leading to different `sum` values if the transaction *B* removes a key-pair value in the world state simultaneously.

B. RUNNING THE ANALYSIS

Analyzing the chaincode reported in Figure 3, GoLiSA successfully identifies the flow leading to a range query risk, which originates from `GetQueryResult` and reaches the write state instruction `PutState`, as shown in Figure 5. Specifically, at line 2, `GetQueryResult` instruction may return a value containing phantom data. This data is propagated through `resultsIterator` to the variable `queryResult` at line 10, reaching the `PutState` instruction through `getValue(queryResult)` at line 15, where it is written in the world state.

Specifically, at line 2, GoLiSA identifies `GetQuery Result` as a tainted source because it yields a value from a query function that is not re-executed. Then, this tainted data propagates through the program, according to the abstract semantics of the instructions. At the end of the propagation phase, GoLiSA returns the analysis results as a set of CFGs enriched with the abstract post-state computed for each instruction, as reported in Figure 6. Note that `sum` $\in \mathbb{C}$ after instruction $n + 5$, while `sum` $\in \mathbb{T}$ after instruction $n + 6$; this is due to the fixpoint computation of the `for` loop taking into account both incoming values (i.e., $n+5$ and $n + 10$ instructions) to ensure over-approximation of the set of variables that *may* be tainted. These results are processed by a checker that inspects them to detect program points containing sinks and whose values come from variables contained in the taint set. Finally, an alert is issued on line 15, because the second parameter of `PutState` is flagged as

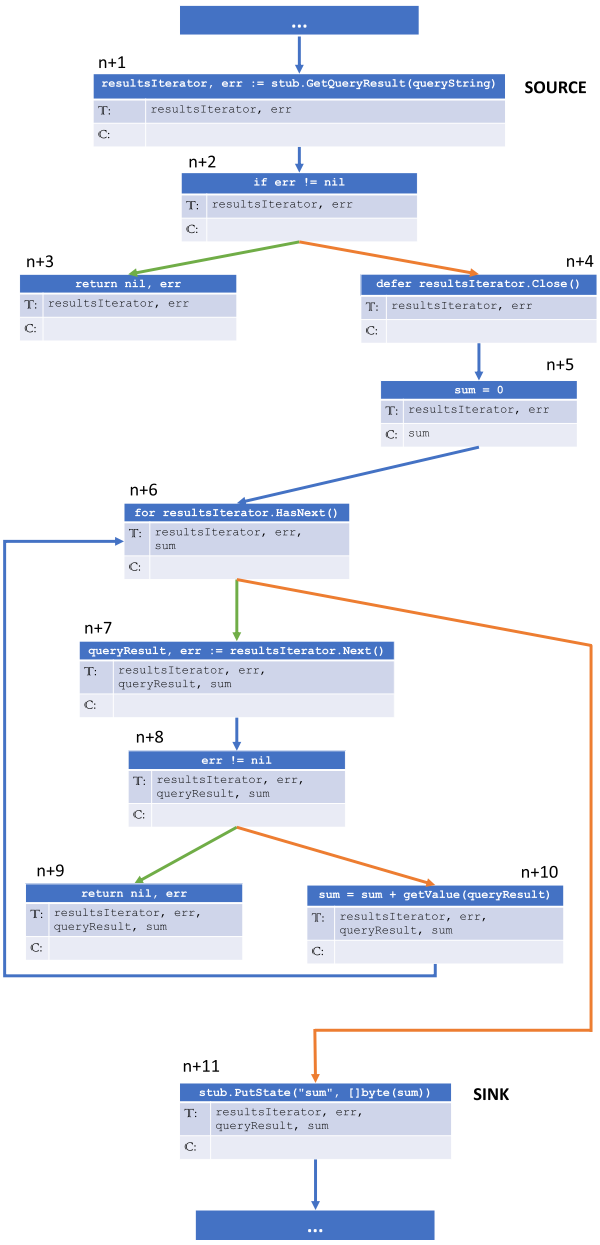

FIGURE 6. CFG containing information of taint analysis.

a sink by the analysis, the statement `[]byte{sum}` yields the value from `sum`, and `sum` $\in \mathbb{T}$ in that program point (instruction $n + 11$).

An additional graphical representation of the analysis related to the example is depicted in Figure7. The black text box indicates the sources, while the red text box indicates the sinks for the taint analysis. The complete propagation of the flow is highlighted in yellow.

IX. CONCLUSION

In blockchain-based contexts, software verification plays a key role, as when the software runs in a distributed and decentralized network where patching and fixing code is not always easy to do, and data is stored in the blockchain,

```
1   // ...
2   resultsIterator, err := stub.GetQueryResult(queryString)
3   if err != nil {
4     return nil, err
5   }
6
7   defer resultsIterator.Close()
8
9   sum = 0
10  for resultsIterator.HasNext() {
11    queryResult, err := resultsIterator.Next()
12    if err != nil {
13      return shim.Error([]byte("Error!"))
14    }
15    sum = sum + getValue(queryResult)
16  }
17
18  stub.PutState("sum", []byte(sum))
19  // ...
```

FIGURE 7. Graphical representation of taint analysis.

it becomes immutable. Therefore, it is necessary to detect, investigate, and fix issues early in software development to avoid network slowdown and the unexpected or wrong storage of immutable data. Moreover, particular attention must be paid to data management within smart contracts to remain compliant with regulations such as the recent European Data Act [48].

Regarding HF, verification tools are few and, although formal software verification has a long history in computer science, these tools cannot provide guarantees about their findings, as they mostly apply syntactic checks. However, in recent years, the scientific community has become very aware of these issues, and with the advent of new frameworks, such as LiSA [23], Mopsa [49], or Goblint [50], it has made these technologies more accessible to developers and practitioners.

Regarding phantom reads, we provide a solution based on taint analysis for their detection advances the current state-of-the-art for HF, considering the semantics of the instructions and providing formal guarantees, thanks to the abstract interpretation framework.

Future work will investigate techniques to improve the results and help developers during the bug-fixing phase to understand the root cause of issues, such as introducing a backflow analysis to reconstruction taint graphs [51], as well as investigating techniques for making the analysis configurable by automatically discovering sources and sinks [52].

Although increasing precision means degrading analysis performance, smart contracts are typically minimal programs with low complexity. Then, it could increase the precision without excessively impacting the analysis performance.

Moreover, given the multi-language nature of LiSA [47], once the frontends of interest are supported, we will be in the position to support the detection of phantom reads and range query risks for other chaincode languages (such as Java and JavaScript) without changing the taint analysis engine of LiSA.

REFERENCES

[1] G. Wood, "ETHEREUM: A secure decentralised generalised transaction ledger," Ethereum, Zug, Switzerland, Yellow Paper, pp. 1–32, 2014.

[2] A. M. Antonopoulos and G. Wood, *Mastering Ethereum: Building Smart Contracts and Dapps*. Sebastopol, CA, USA: O'Reilly, 2018.
[3] E. Androulaki et al., "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proc. 13th EuroSys Conf.* New York, NY, USA: Association for Computing Machinery, Apr. 2018, pp. 1–15, doi: 10.1145/3190508.3190538.
[4] Hyperledger Fabric. *Hyperledger Fabric Documentation—What is Hyperledger Fabric?* Accessed: May 2024. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.5/blockchain.html#what-is-hyperledger-fabric
[5] IBM. (2023). *Hyperledger Achieves Huge Milestone: Introducing Hyperledger Fabric 2.0*. Accessed: Dec. 2023. [Online]. Available: https://www.ibm.com/blog/hyperledger-achieves-huge-milestone-introducing-hyperledger-fabric-2-0/
[6] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich, "Blurring the lines between blockchains and database systems: The case of hyperledger fabric," in *Proc. Int. Conf. Manage. Data*. New York, NY, USA: Association for Computing Machinery, Jun. 2019, pp. 105–122, doi: 10.1145/3299869.3319883.
[7] M. Debreczeni, A. Klenik, and I. Kocsis, "Transaction conflict control in hyperledger fabric: A taxonomy, gaps, and design for conflict prevention," *IEEE Access*, vol. 12, pp. 18987–19008, 2024, doi: 10.1109/ACCESS.2024.3361318.
[8] K. Yamashita, Y. Nomura, E. Zhou, B. Pi, and S. Jun, "Potential risks of hyperledger fabric smart contracts," in *Proc. IEEE Int. Workshop Blockchain Oriented Softw. Eng. (IWBOSE)*, Feb. 2019, pp. 1–10, doi: 10.1109/IWBOSE.2019.8666486.
[9] P. Lv, Y. Wang, Y. Wang, and Q. Zhou, "Potential risk detection system of hyperledger fabric smart contract based on static analysis," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Athens, Greece, Sep. 2021, pp. 1–7, doi: 10.1109/ISCC53001.2021.9631249.
[10] J. A. Chacko, R. Mayer, and H.-A. Jacobsen, "Why do my blockchain transactions fail? A study of hyperledger fabric," in *Proc. Int. Conf. Manage. Data*. New York, NY, USA: Association for Computing Machinery, Jun. 2021, pp. 221–234, doi: 10.1145/3448016.3452823.
[11] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proc. 4th ACM SIGACT-SIGPLAN Symp. Princ. Program. Lang.* New York, NY, USA: Association for Computing Machinery, 1977, pp. 238–252, doi: 10.1145/512950.512973.
[12] P. Cousot and R. Cousot, "Systematic design of program analysis frameworks," in *Proc. 6th ACM SIGACT-SIGPLAN Symp. Princ. Program. Lang.* New York, NY, USA: Association for Computing Machinery, 1979, pp. 269–282, doi: 10.1145/567752.567778.
[13] Hyperledger Fabric. (2023). *A Blockchain Ledger*. Accessed: Dec. 2023. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.5/ledger/ledger.html#a-blockchain-ledger
[14] Hyperledger Fabric. (2023). *State Database Options*. Accessed: Jan. 2024. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.5/couchdb_as_state_database.html#state-database-options
[15] Hyperledger Fabric. (2023). *Chaincode Terminology*. Accessed: Dec. 2023. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.5/peers/peers.html?highlight=validation#chaincode-terminology
[16] L. Foschini, A. Gavagna, G. Martuscelli, and R. Montanari, "Hyperledger fabric blockchain: Chaincode performance analysis," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Jun. 2020, pp. 1–6, doi: 10.1109/ICC40277.2020.9149080.
[17] Hyperledger Fabric. (2023). *Transaction Simulation and Read-Write Set*. Accessed: Dec. 2023. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.5/readwrite.html#transaction-simulation-and-read-write-set
[18] Hyperledger Fabric. (2023). *Hyperledger Fabric GitHub—Intefaces*. Accessed: Dec. 2023. [Online]. Available: https://github.com/hyperledger/fabric-chaincode-go/blob/9207360bbddd5952479c24154353b82c4c044677/shim/interfaces.go
[19] P. Ruan, D. Loghin, Q.-T. Ta, M. Zhang, G. Chen, and B. C. Ooi, "A transactional perspective on execute-order-validate blockchains," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*. New York, NY, USA: Association for Computing Machinery, Jun. 2020, pp. 543–557, doi: 10.1145/3318464.3389693.
[20] (2023). *Upgrade a Chaincode*. Accessed: Feb. 2023. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.5/chaincode_lifecycle.html#upgrade-a-chaincode
[21] K. Yamashita and J. Ry. (2020). *Chaincode Analyzer*. Accessed: Feb. 2024. [Online]. Available: https://github.com/hyperledger-labs/chaincode-analyzer
[22] C. Siva. (2021). *Revivecc*. Accessed: Feb. 2024. [Online]. Available: https://github.com/sivachokkapu/revive-cc
[23] P. Ferrara, L. Negrini, V. Arceri, and A. Cortesi, "Static analysis for dummies: Experiencing LiSA," in *Proc. 10th ACM SIGPLAN Int. Workshop State Art Program Anal.* New York, NY, USA: Association for Computing Machinery, Jun. 2021, pp. 1–6, doi: 10.1145/3460946.3464316.
[24] *SonarQube*. Accessed: May 2024. [Online]. Available: https://www.sonarsource.com/products/sonarqube
[25] GrammaTech. *CodeSonar*. Accessed: Feb. 2024. [Online]. Available: https://www.grammatech.com
[26] F. Spoto, "The Julia static analyzer for Java," in *Static Analysis*, X. Rival, Ed. Berlin, Germany: Springer, 2016, pp. 39–57, doi: 10.1007/978-3-662-53413-7_3.
[27] M. Almakhour, L. Sliman, A. E. Samhat, and A. Mellouk, "Verification of smart contracts: A survey," *Pervas. Mobile Comput.*, vol. 67, Sep. 2020, Art. no. 101227, doi: 10.1016/j.pmcj.2020.101227.
[28] S. Kim and S. Ryu, "Analysis of blockchain smart contracts: Techniques and insights," in *Proc. IEEE Secure Develop. (SecDev)*, Sep. 2020, pp. 65–73, doi: 10.1109/SecDev45635.2020.00026.
[29] P. Tolmach, Y. Li, S.-W. Lin, Y. Liu, and Z. Li, "A survey of smart contract formal specification and verification," *ACM Comput. Surv.*, vol. 54, no. 7, pp. 1–38, Sep. 2022, doi: 10.1145/3464421.
[30] S. Brotsis, N. Kolokotronis, K. Limniotis, G. Bendiab, and S. Shiaeles, "On the security and privacy of hyperledger fabric: Challenges and open issues," in *Proc. IEEE World Congr. Services (SERVICES)*, Oct. 2020, pp. 197–204, doi: 10.1109/SERVICES48979.2020.00049.
[31] C. Paulsen, "Revisiting smart contract vulnerabilities in hyperledger fabric," M.S. thesis, TU Delft Elect. Eng., Math. Comput. Sci., TU Delft Intell. Syst., Delft Univ. Technol., Delft, The Netherlands, 2021.
[32] Google. (2023). *The Go Programming Language Specification—Declarations and Scope*. Accessed: Dec. 2023. [Online]. Available: https://go.dev/ref/spec#Declarations_and_scope
[33] F. E. Allen, "Control flow analysis," in *Proc. Symp. Compiler Optim.* New York, NY, USA: Association for Computing Machinery, 1970, pp. 1–19, doi: 10.1145/800028.808479.
[34] X. Rival and K. Yi, *Introduction to Static Analysis: An Abstract Interpretation Perspective*. Cambridge, MA, USA: MIT Press, 2020.
[35] B. Chess and J. West, *Secure Programming With Static Analysis*, 1st ed. Reading, MA, USA: Addison-Wesley, 2007.
[36] R. Wilhelm, "Principles of abstract interpretation: By patrick cousot MIT Press, 2021, ISBN 9780262044905, pp. 1–819. reviewed by reinhard wilhelm," *Formal Aspects Comput.*, vol. 34, no. 2, pp. 1–3, Jun. 2022, doi: 10.1145/3546953.
[37] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, no. 5, pp. 236–243, May 1976, doi: 10.1145/360051.360056.
[38] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J. Sel. Areas Commun.*, vol. 21, no. 1, pp. 5–19, Jan. 2003, doi: 10.1109/JSAC.2002.806121.
[39] P. Ferrara, L. Olivieri, and F. Spoto, "Tailoring taint analysis to GDPR," in *Proc. 6th Annu. Privacy Forum*, in Lecture Notes in Computer Science, vol. 11079, Barcelona, Spain. Cham, Switzerland: Springer, 2018, pp. 63–76, doi: 10.1007/978-3-030-02547-2_4.
[40] M. D. Ernst, A. Lovato, D. Macedonio, C. Spiridon, and F. Spoto, "Boolean formulas for the static identification of injection attacks in Java," in *Proc. 20th Int. Conf. Logic Program., Artif. Intell., Reasoning*, Suva, Fiji, in Lecture Notes in Computer Science, vol. 9450. Berlin, Germany: Springer, 2015, pp. 130–145, doi: 10.1007/978-3-662-48899-7_10.
[41] P. Ferrara, L. Olivieri, and F. Spoto, "Static privacy analysis by flow reconstruction of tainted data," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 31, no. 7, pp. 973–1016, Jul. 2021, doi: 10.1142/s0218194021500303.
[42] A. K. Mandal, P. Ferrara, Y. Khlyebnikov, A. Cortesi, and F. Spoto, "Cross-program taint analysis for IoT systems," in *Proc. SAC*, Brno, Czech Republic, C. Hung, T. Cerný, D. Shin, and A. Bechini, Eds., 2020, pp. 1944–1952, doi: 10.1145/3341105.3373924.
[43] L. Olivieri, L. Negrini, V. Arceri, F. Tagliaferro, P. Ferrara, A. Cortesi, and F. Spoto, "Information flow analysis for detecting non-determinism in blockchain," in *Proc. 37th Eur. Conf. Object-Oriented Program.*, in Leibniz International Proceedings in Informatics, vol. 263, Dagstuhl, Germany, K. Ali and G. Salvaneschi, Eds., 2023, pp. 23:1–23:25, doi: 10.4230/LIPIcs.ECOOP.2023.23.
[44] L. Olivieri, T. Jensen, L. Negrini, and F. Spoto, "MichelsonLiSA: A static analyzer for tezos," in *Proc. IEEE Int. Conf. Pervasive Comput. Commun. Workshops Affiliated Events (PerCom Workshops)*, Mar. 2023, pp. 80–85, doi: 10.1109/PerComWorkshops56833.2023.10150247.

[45] L. Olivieri, L. Negrini, V. Arceri, T. Jensen, and F. Spoto, "Design and implementation of static analyses for tezos smart contracts," *Distrib. Ledger Technol., Res. Pract.*, pp. 1–24, Jan. 2024, doi: 10.1145/3643567.
[46] L. Olivieri, F. Tagliaferro, V. Arceri, M. Ruaro, L. Negrini, A. Cortesi, P. Ferrara, F. Spoto, and E. Talin, "Ensuring determinism in blockchain software with GoLiSA: An industrial experience report," in *Proc. 11th ACM SIGPLAN Int. Workshop State Art Program Anal.*, San Diego, CA, USA, L. Gonnord and L. Titolo, Eds., Jun. 2022, pp. 23–29, doi: 10.1145/3520313.3534658.
[47] L. Negrini, P. Ferrara, V. Arceri, and A. Cortesi, *LiSA: A Generic Framework for Multilanguage Static Analysis*. Singapore: Springer, 2023, pp. 19–42, doi: 10.1007/978-981-19-9601-6_2.
[48] L. Olivieri and L. Pasetto, "Towards compliance of smart contracts with the European union data act," in *Proc. CEUR Workshop*, vol. 3629, 2024, pp. 61–66. [Online]. Available: https://ceur-ws.org/Vol-3629
[49] R. Monat, A. Ouadjaout, and A. Miné, "Mopsa-C: Modular domains and relational abstract interpretation for C programs (competition contribution)," in *Proc. 29th Int. Conf. Tools Algorithms Construct. Anal. Syst.*, in Lecture Notes in Computer Science, vol. 13994, Paris, France, S. Sankaranarayanan and N. Sharygina, Eds. Cham, Switzerland: Springer, 2023, pp. 565–570, doi: 10.1007/978-3-031-30820-8_37.
[50] S. Saan, M. Schwarz, J. Erhard, M. Pietsch, H. Seidl, S. Tilscher, and V. Vojdani, "GOBLINT: Autotuning thread-modular abstract interpretation—(Competition contribution)," in *Proc. 29th Int. Conf. Tools Algorithms Construct. Anal. Syst.*, in Lecture Notes in Computer Science, vol. 13994, Paris, France, S. Sankaranarayanan and N. Sharygina, Eds. Cham, Switzerland: Springer, 2023, pp. 547–552, doi: 10.1007/978-3-031-30820-8_34.
[51] P. Ferrara, L. Olivieri, and F. Spoto, "BackFlow: Backward context-sensitive flow reconstruction of taint analysis results," in *Proc. 21st Int. Conf. Verification, Model Checking, Abstract Interpretation*, New Orleans, LA, USA. Berlin, Germany: Springer, 2020, pp. 23–43, doi: 10.1007/978-3-030-39322-9_2.
[52] P. Ferrara and L. Negrini, "SARL: OO framework specification for static analysis," in *Software Verification*, M. Christakis, N. Polikarpova, P. S. Duggirala, and P. Schrammel, Eds. Cham, Switzerland: Springer, 2020, pp. 3–20, doi: 10.1007/978-3-030-63618-0_1.

LUCA OLIVIERI received the Ph.D. degree in computer science from the University of Verona, with a focus on verifying smart contracts and blockchain software. He was a Software Engineer and a Research Scientist for five years in the industrial field on static analysis based on abstract interpretation, mainly for Java and C# programs. Then, he joined the Software and System Verification (SSV) Group, Ca' Foscary University of Venice, where he is currently an Assistant Professor (non-tenure track) in computer science.

LUCA NEGRINI received the bachelor's and master's degrees from the University of Verona, Italy, and the Ph.D. degree in computer science from the Ca' Foscari University of Venice, in January 2023, with a focus on multi-language static analysis. After the master's degree, he received five years of industrial experience in the development and applications of static analysis and abstract interpretation with the Julia static analyzer. Then, he joined the Ca' Foscari University of Venice as a Research Fellow, where he is currently an Assistant Professor (non-tenure).

VINCENZO ARCERI received the Ph.D. degree in computer science from the University of Verona, in May 2020. From September 2019 to September 2021, he was a Postdoctoral Researcher with the Software and System Verification Research Group, Ca' Foscari University of Venice. In Summer 2016, he was a UROP Student with Imperial College London, U.K., under the supervision of Prof. Sergio Maffeis. He is currently an Assistant Professor (non-tenure track) with the Department of Mathematical, Physical, and Computer Sciences, University of Parma. His main research interests include static program analysis, abstract interpretation, string analysis and verification, blockchain software verification, and, more generally, formal methods for program security.

BADARUDDIN CHACHAR received the bachelor's degree in computer science from COMSATS University Lahore, Pakistan, in 2013, and the master's degree in software engineering from Sukkur IBA University, Pakistan, in 2018. He is currently pursuing the Ph.D. degree in computer science with Ca' Foscari University of Venice, Venice, Italy. His Ph.D. project focuses on smart contract vulnerability detection using static analysis under the supervision of Agostino Cortesi and Pietro Ferrara. He received a fully-funded national scholarship for science and engineering from the Ministry of IT, Pakistan.

PIETRO FERRARA is currently an Associate Professor with the Ca' Foscari University of Venice. Prior to academia, he gained industry experience bridging scientific research with software development and delivery. This included roles, such as the Head of Research and Development at JuliaSoft SRL and a Research Staff Member at IBM Research. He is also an expert in abstract interpretation-based static analysis, focusing on identifying security vulnerabilities and privacy breaches in object-oriented programs.

AGOSTINO CORTESI received the Ph.D. degree. He is currently a Full Professor of computer science with the Ca' Foscari University of Venice, Italy. He contributed to more than 200 articles in journals, such as *ACM Transactions on Programming Languages and Systems* and IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, and proceedings of international conferences, such as ACM POPL, ACM PLDI, and IEEE LICS. According to Scopus, his current H-index is 23, with more than 1800 citations. His main research interests include software engineering and software verification areas, combining theoretical and applicative approaches. He serves as the Co-Editor-in-Chief for the book series *Services and Business Process Reengineering* (SpringerNature).

Open Access funding provided by 'Università degli Studi di Parma' within the CRUI CARE Agreement