Information Flow Analysis for Detecting Non-Determinism in Blockchain

Luca Olivieri
University of Verona, Italy
Corvallis Srl, Padova, Italy

Luca Negrini
Corvallis Srl, Padova, Italy

Vincenzo Arceri
University of Parma, Italy

Fabio Tagliaferro
CYS4 Srl, Florence, Italy

Pietro Ferrara
Ca' Foscari University of Venice, Italy

Agostino Cortesi
Ca' Foscari University of Venice, Italy

Fausto Spoto
University of Verona, Italy

Abstract

A mandatory feature for blockchain software, such as smart contracts and decentralized applications, is determinism. In fact, non-deterministic behaviors do not allow blockchain nodes to reach one common consensual state or a deterministic response, which causes the blockchain to be forked, stopped, or to deny services. While domain-specific languages are deterministic by design, general-purpose languages widely used for the development of smart contracts such as Go, provide many sources of non-determinism. However, not all non-deterministic behaviours are critical. In fact, only those that affect the state or the response of the blockchain can cause problems, as other uses (for example, logging) are only observable by the node that executes the application and not by others. Therefore, some frameworks for blockchains, such as Hyperledger Fabric or Cosmos SDK, do not prohibit the use of non-deterministic constructs but leave the programmer the burden of ensuring that the blockchain application is deterministic. In this paper, we present a flow-based approach to detect non-deterministic vulnerabilities which could compromise the blockchain. The analysis is implemented in GoLiSA, a semantics-based static analyzer for Go applications. Our experimental results show that GoLiSA is able to detect all vulnerabilities related to non-determinism on a significant set of applications, with better results than other open-source analyzers for blockchain software written in Go.

2012 ACM Subject Classification Security and privacy → Distributed systems security; Theory of computation → Program analysis; Theory of computation → Program verification; Software and its engineering → Software notations and tools

Keywords and phrases Static Analysis, Program Verification, Non-determinism, Blockchain, Smart contracts, DApps, Go language

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.23

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.9.2.23`

Funding *Vincenzo Arceri*: Bando di Ateneo per la ricerca 2022, founded by University of Parma, project number: MUR_DM737_2022_FIL_PROGETTI_B_ARCERI_COFIN, *Formal verification of GPLs blockchain smart contracts*
Pietro Ferrara: SERICS (PE00000014) under the NRRP MUR program funded by the EU – NGEU, iNEST-Interconnected NordEst Innovation Ecosystem funded by PNRR (Mission 4.2, Investment 1.5) NextGeneration EU – Project ID: ECS 00000043, and SPIN-2021 "Static Analysis for Data Scientists" funded by Ca' Foscari University
Agostino Cortesi: SERICS (PE00000014) under the NRRP MUR program funded by the EU – NGEU, iNEST-Interconnected NordEst Innovation Ecosystem funded by PNRR (Mission 4.2, Investment 1.5) NextGeneration EU – Project ID: ECS 00000043, and SPIN-2021 "Ressa-Rob" funded by Ca' Foscari University

1 Introduction

In the last decade, blockchain software has undergone a notable evolution. In 2008, Bitcoin [33] introduced a Turing-incomplete low-level language to specify locking conditions that must hold for a transaction to be accepted by the network [3]. In 2013, Ethereum [8, 4] provided a Turing-complete bytecode where smart contract rules are enforced by the blockchain consensus. The execution of the code takes place on the Ethereum Virtual Machine (EVM), resulting in software identified as *decentralized applications* (DApps). EVM bytecode is supported by high-level domain-specific languages (DSLs), such as Solidity and Vyper, that have been designed from scratch for the purpose of being executed in the restricted environment of blockchain. Subsequently, thanks to frameworks such as Hyperledger Fabric [2], Tendermint [6, 29], and Cosmos SDK [30], general-purpose programming languages (GPLs) such as Go, Java and JavaScript can also be used to develop smart contracts and DApps, with Go being the most popular in industrial blockchains.

The popularity of GPLs for writing smart contracts and DApps is steadily increasing. Their success is mostly due to the maturity of the languages themselves, directly resulting in wide communities, consolidated tools (such as IDEs and debuggers), and most importantly a pool of expert and knowledgeable developers that can write highly efficient smart contracts. Yet, GPLs were not conceived solely for blockchain ecosystems: code that is harmless and bug-free in other contexts may result in vulnerabilities and errors. Among these, one of the most insidious is non-determinism. When the result of an operation on a blockchain is non-deterministic, there is no guarantee that a common state can be reached by the network's nodes, possibly preventing it from reaching consensus. This can manifest, among other possibilities, as transaction failures or denial of service. Nevertheless, not all instances of non-determinism are intrinsically dangerous: logging the time of a transaction can result in different timestamps appearing in each node's logs, but it does not endanger consensus as it is not *observable* by other nodes. In fact, non-deterministic instructions are problematic only if they can affect the shared blockchain state.

As an example, consider the code in Figure 1, reporting an excerpt of the `ValidateBasic` method from module `x/authz` (part of the Cosmos SDK versions 0.43.x and 0.44.{0,1}) and affected by the vulnerability reported in CVE-2021-41135[1]. The code is meant to fail the validation of expired grants. Note that the guard at line 2 involves the local clock of nodes (`time.Now()`) rather than leveraging the timestamp included in the Block header provided by the Byzantine Fault Tolerant clock, that is agreed upon by the consensus. As reported in the official Cosmos forum [12]:

> *Local clock times are subjective and thus non-deterministic. An attacker could craft many Grants, with different but close expiration times (e.g., separated by a few seconds), and try to exercise the granted functionality for all of them close to their expiration time. It is likely in such a scenario that some nodes would consider a grant to have expired while others would not, leading to a consensus halt.*

The code was then fixed in version 0.44.2, but is still a clear example of a vulnerability arising from non-deterministic constructs.

The problem of non-determinism in blockchain software is clearly felt by the communities of the blockchain frameworks treated in this paper. As a representative example, the Tendermint Core documentation [27], while discussing non-determinism, reports:

[1] `https://nvd.nist.gov/vuln/detail/CVE-2021-41135`.

```go
func (g Grant) ValidateBasic() error {
  if g.Expiration.Unix() < time.Now().Unix() {
    return sdkerrors.Wrap(ErrInvalidExpirationTime, "Time can't be in the past")
  }
  // [...]
}
```

Figure 1 Cosmos SDK code affected by CVE-2021-41135.

> *While programmers can avoid non-determinism by being careful, it is also possible to create a special linter or static analyzer for each language to check for determinism. In the future we may work with partners to create such tools.*

Paper contribution

This paper presents a software verification approach based on static analysis for the detection of non-deterministic vulnerabilities in blockchain ecosystems, covering the most popular frameworks for developing this kind of software, such as Hyperledger Fabric, Tendermint Core and Cosmos SDK. We shift the classical focus that has been applied in this context beyond the mere syntactic absence of non-deterministic constructs. In fact, we aim at distinguishing *harmful* usages of non-determinism, that is, constructs affecting the blockchain state and response, from *harmless* ones. As a consequence, the set of alarms issued to the user sensibly shrinks, as shifting from a syntactic approach towards a semantic one leads to a sensible reduction in false positives. We propose a semantic flow-based static analysis for detecting flows from non-deterministic constructs to blockchain state modifiers and response builders. The choice of a flow-based analysis seems natural when the problem is phrased as *"is there execution where a non-deterministic value affects the blockchain state or the contract's response?"*. We thus exploit the well-consolidated literature in this area to adopt scalable solutions that soundly over-approximate all program executions.

We provide a static analyzer implementing our approach: GoLiSA[2], a sound static analyzer based on abstract interpretation [10] for Go applications. Intuitively, we use our analyzer's fixpoint engine to mark *all* program variables (local variables, objects' fields, ...) that can contain values affected, directly or indirectly, by a non-deterministic construct or computation. Specifically, we can perform a shallower analysis detecting only explicit flows using *Taint* analysis [43, 14], where non-deterministic constructs and blockchain state modifiers are modeled as sources and sinks, respectively. Alternatively, we can perform a deeper analysis able to also detect implicit flows by means of the *Non-interference* analysis [24, 25], where non-deterministic constructs and blockchain state modifiers are instead modeled as low and high variables, respectively. Both solutions are implemented in GoLiSA, whose analysis starts by syntactically visiting the input application to annotate all sources and sinks. The annotations are dynamically generated depending on the kind of application of interest (i.e., Hyperledger Fabric, Cosmos SDK, or Tendermint Core). Since there is no predefined set of sources in the target program, both *Taint* analysis and *Non-interference* are parametric: they consider as *harmful* (i.e., tainted or low integrity, depending on the analysis that is to be executed) only variables that are annotated as sources. The fixpoint engine then takes care of propagating values coming from sources on the entirety of the program, exploiting our analyses implementations. After the fixpoint converges, a mapping stating if each program variable is the result of a non-deterministic computation is available at each program

[2] `https://github.com/lisa-analyzer/go-lisa`

point. These are then used by our non-deterministic semantic checkers, that visit the whole application searching for statements annotated with the sink annotation. Whenever one is found, the mappings are used to determine if the values used as parameters of the call are critical or, in the case of *Non-interference*, if the call happens on a critical state.

Our approach, as highlighted by our evaluation, shows a significant decrease of false positives on real-world blockchain applications compared to other analyzers for blockchain non-determinism. The solution has been experimented on a benchmark of more than 600 real-world blockchain programs written in Go. These show that GoLiSA is able to perform the analysis on the totality of smart contracts in this significant benchmark, and to successfully report their non-determinism vulnerabilities.

The analyses are then evaluated in terms of precision of the results (true positive, false positive, and false negative alarms). Based on these criteria, GoLiSA outperforms existing open-source static analyzers for Go blockchain software. Moreover, the evaluation shows that the execution time of the analyses is not impractical for real use cases.

To the best of our knowledge, GoLiSA is the first sound semantic-based static analyzer for blockchain software able to precisely detect critical non-determinism behaviors while scaling to real-world programs.

Summarizing, our contribution is threefold, as we provide:

- a detailed investigation on the sources and the sinks that lead to non-determinism issues in the most popular blockchain frameworks;
- a flow-based static analysis for the detection of critical non-determinism behaviors, with two instantiations exploiting different formalizations;
- an open-source sound static analyzer for detecting critical non-deterministic behaviors in blockchain software written in Go.

Paper structure

Sect. 2 reports an overview about blockchain software using Go and the most popular frameworks to develop it. Sect. 3 discusses the problem of non-deterministic behavior in blockchain context. After reporting an overview on information flow analyses, Sect. 4 presents our core contribution for detecting non-deterministic behavior in blockchain software, that relies on GoLiSA. Sect. 5 reports our experimental results. Sect. 6 discusses the related work. Finally, Sect. 7 concludes the paper.

2 Preliminaries: Go and Blockchain

Go (`https://golang.org`) is a statically typed, compiled, open-source, and general-purpose high-level programming language designed by Google to speed up software development, and that is appreciated for its cross-compilation feature. Its versatility and performance contributed to its diffusion in the blockchain environment: popular frameworks such as the Hyperledger Fabric[3], Tendermint[4] and the Cosmos SDK[5] are written in Go. These rely on Go to develop efficient smart contracts and DApps, exploiting its high performances.

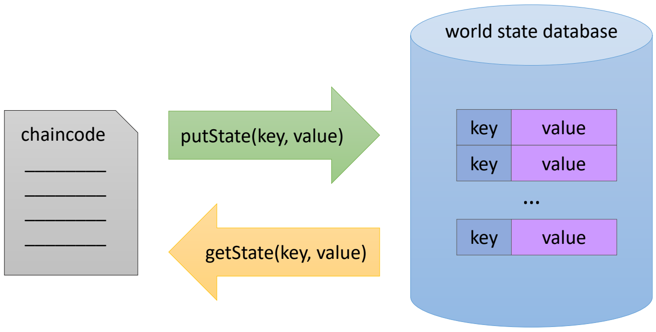

Figure 2 The world state database of Hyperledger Fabric.

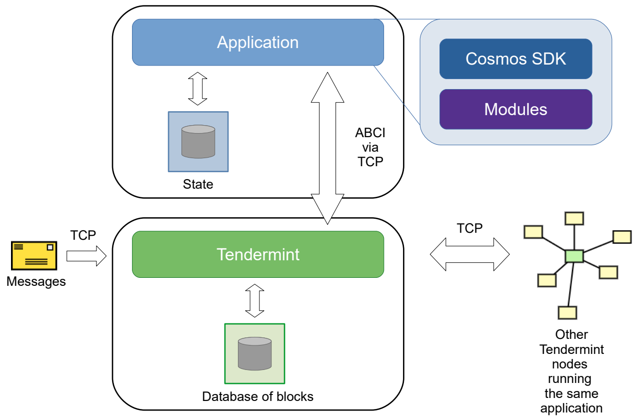

Figure 3 Cosmos SDK architecture.

2.1 Blockchain Environments

Hyperledger Fabric (HF) is a permissioned blockchain framework designed to be adopted in enterprise contexts, supported by the Linux Foundation and other contributors such as IBM, Cisco, and Intel. In HF, smart contracts and DApps are written in *chaincode* that can be implemented in several GPLs such as Go, JavaScript, and Java. In most cases, the chaincode interacts only with the world state database component of the ledger, and not with the transaction log [26]. Go is currently the most popular language on GitHub related to *chaincode*[6], as Go smart contracts are the best performing ones [23].

Tendermint Core, recently rebranded as *Ignite*, is a platform for building blockchain nodes, supporting both public and permissioned *proof-of-stake* (PoS) networks. It is a Byzantine Fault Tolerant (BFT) middleware that separates the application logic from the consensus and networking layers, allowing one to develop blockchain applications written in any programming language, including Go, and replicate them on many machines [7].

Cosmos SDK is an open-source Go framework that eases the development of blockchain applications while optimizing their execution by running them on Tendermint Core. As shown in Figure 3, Cosmos SDK abstracts all the boilerplate code needed to set up a Tendermint

[3] `https://www.hyperledger.org/use/fabric`
[4] `https://tendermint.com/`
[5] `https://v1.cosmos.network/sdk`
[6] Querying the keyword `chaincode` on GitHub (`https://github.com/search?q=chaincode`) results in more than 2100 repositories, about half of which are written in Go. Accessed: 01-12-2022.

Core node, allowing for customized protocol configurations. The programming style follows the object-capability model, where the security of subcomponents is imperative, especially those belonging to the core library. Cosmos SDK is a framework for DApps, supporting different functionalities through highly customizable modules (that can also manage smart contracts).

2.2 Blockchain Consensus

Consensus protocols ensure the validity and authenticity of transactions performed in the blockchain, as they check results of smart contracts or DApps computations through the state of the network's nodes. If a given number of nodes agree on the final state, consensus is reached and the transaction is validated. Otherwise, it is discarded and the nodes proposing spurious states are excluded from the network. When consensus cannot be reached, the blockchain either forks or halts. Deterministic execution is thus required for software that runs in a blockchain, as it guarantees that, when starting from a common state, the same result is reached in any distinct blockchain node, avoiding inconsistencies among peers and consensus failures. Nevertheless, GPLs provide several components that can explicitly lead to non-determinism, such as (pseudo-)random number generators or external computations. Furthermore, even methods that are explicitly sequential and deterministic pose a threat when executed on different nodes, such as the `time.Now()` call from Figure 1. Despite these threats, popular blockchain frameworks such as HF and Cosmos SDK do not enforce particular restrictions on the usage of non-deterministic methods and components.

3 Non-Deterministic Behaviors in Blockchain Software: Sources and Sinks

When trying to prevent non-deterministic vulnerabilities, a first solution is to limit the expressiveness of the GPL by either black- or white-listing APIs and constructs. Consider the Go snippets reported in Figure 4. Both fragments rely on the `time` API to retrieve a timestamp from the host system. In general, the results of calls to the `time` API are subjective to the node executing them, and they might lead to blockchain non-determinism due to different system settings (e.g., time, date, time zones, ...) or due to nodes executing the code at slightly different times. Specifically, Figure 4a shows a safe usage of the `time` API: the timestamp is only used for logging with no observable consequences on the blockchain state or the execution result. Instead, Figure 4b reports a problematic usage of the API, as the timestamp is stored in the blockchain using `PutState`, an HF-specific function that updates the shared network state. Since timestamps could differ on each node, this potentially leads to inconsistent executions (i.e., different blockchain states or execution results), causing transaction failure[7].

It should thus be evident that identifying sources of non-determinism and preventing their usage is not enough when we aim at discerning between harmful and harmless non-deterministic constructs. In fact, one should also recognize how these are used, determining if they can influence the shared blockchain state. In the rest of this section we discuss, for each blockchain framework presented in Section 2, (i) the constructs that generate potentially harmful non-determinism (that is, *sources* of non-deterministic values), and (ii) the blockchain

[7] In this case, the `GetTxTimestamp` method from the HF API should have been used instead of `time.Now`.

```go
func transfer(from, to Address,  value int64, stub *shim.ChaincodeStub) {
  start := time.Now()
  //... transfer operations that takes some milliseconds ...
  elapsed := time.Now().Sub(start)
  log.Println("Time elapsed for the transfer operations: ", elapsed)
}
```

(a) Example of safe use of the time API.

```go
func transfer(from, to Address, value int64, stub *shim.ChaincodeStub) {
  t := time.Now()
  //... transfer operations ...
  err := shim.PutState("transaction-time", t)
  //... other operations ...
}
```

(b) Example of issue of non-determinism with the time API.

Figure 4 Examples of harmless and harmful non-determinism in blockchain.

state modifiers and response builders (i.e., statements that make a transaction succeed or fail), namely *sinks* that are sensitive to non-determinism[8]. This will prepare the ground for the core contribution of this paper: a static approach to detect critical usages of non-determinism in blockchain software, reported in Section 4.

3.1 Sources of Non-Determinism

The sources of non-determinism can be logically split in two families, the first being related to the combination of framework and GPL adopted to develop the software. This family comprises a set of constructs and APIs allowed by the framework that may break the consensus during the execution of smart contracts or DApps. In Go, these are:

- *iteration over maps* that, being the iteration order unspecified[9], is not guaranteed to be deterministic;
- *parallelization and concurrency*, that can lead to race conditions on shared resources, thus creating non-determinism on the computed values;
- *global variables*, that may change innately and cause inconsistencies to the results, since they depend on the application state of a peer and not on that of the blockchain [32, 5].
- *random value generators*, that can potentially be allowed in smart contracts [9] to employ custom logic while being non-deterministic by-definition.

The second family instead involves statements related to the underlying environment, such as file systems, operating systems, databases, and Internet connections. While these are not intrinsically non-deterministic, they become dangerous when their result is expected to be consistent on different environments. These comprise APIs handling:

- *file systems*, as the program might rely on files that are not present on all nodes, as they might have been deleted, edited, moved, or there might be insufficient disk space causing any operation to fail;
- *operating systems* (OS), since the blockchain might operate on various hosts and language APIs could return different results on each OS (e.g., time and date methods could return different values if nodes are not synchronized);

[8] The complete list of sources and sinks of non-determinism is available at `https://github.com/lisa-analyzer/go-lisa/blob/master/go-lisa/sources-sinks.md`.

[9] `https://golang.org/ref/spec#For_statements`

Table 1 Potential non-deterministic behaviors related to Go.

Level	*Category*	*Package*	*Statements/Methods*
Framework/Language	Map iteration	-	`range` on map
	Parallelization/concurrency	-	`go` (Go routine), `<-` (channel)
	Random number generation APIs	`math/rand`, `crypto/rand`	*
	Global variables	-	-
Environment	File system APIs	`io`, `embed`, `archive`, `compress`	*
	OS APIs	`os`, `syscall`, `internal`, `time`	*
	Database APIs	`database`	*
	Internet APIs	`net`	*

```
func (s *SmartContract) transaction(APIstub shim.ChaincodeStubInterface) sc.Response {

  if rand.Int() % 2 == 0 {
    return shim.Error("Fail")
  } else {
    return shim.Success(nil)
  }
}
```

Figure 5 Example of issue of non-determinism related to the blockchain response.

- *databases*, where records might be deleted, edited, or contain different data;
- *Internet connections*, as networking setup or errors could cause some addresses to be unreachable on few nodes of the network.

Table 1 summarizes the instructions and libraries of Go[10] that we considered as cases of non-determinism, where $*$ represents the entirety of the package. For the sake of simplicity, the table reports instructions and packages omitting the full signatures of each method. Note that only few methods within those packages lead to non-deterministic behaviors: for instance, most methods from package `time` handling dates and times do not pose a threat in smart contracts and DApps, and are in fact quite common. However, operations such as retrieving the current time of the OS (i.e., methods `Since`, `Now`, `Until`) are potentially dangerous.

3.2 Sinks of Non-Determinism

Sinks of non-determinism comprise constructs and APIs with the ability of both modifying the common state of the blockchain or having an impact on the response of blockchain networks. While the former is inherently involved in consensus protocols, the execution of code within the blockchain does not necessarily change the shared state (e.g., functions that simply read a value). However, the execution may lead to non-deterministic responses, compromising the consensus of the network, as in the simple example reported in Figure 5. Table 2, where the **Critical point** column identifies what part of the API should not receive non-deterministic values, summarizes the main instructions and components that we consider as sinks for non-determinism.

3.2.1 Hyperledger Fabric APIs for Go

In HF, chaincode executes transaction proposals against world state data that may change its state. Programmatically, interface `ChaincodeStubInterface` from the HF Go APIs enables access and modification of the blockchain state. Table 2 reports the current components (as

[10] The full list of Go APIs sources considered in our analyses is available at `https://github.com/lisa-analyzer/go-lisa/blob/master/go-lisa/src/main/resources/for-analysis/nondeterm_sources.txt`. The list consider API until Go version 1.17.

Table 2 Main sinks for blockchain software written in Go.

Framework	Package	Type/Interface	Statements/Methods	Critical point
HyperLedger Fabric	`shim`	`ChaincodeStubInterface`	`PutState`	parameters
			`DelState`	parameters
			`PutPrivateData`	parameters
			`DelPrivateData`	parameters
			`Success`	statement
			`Error`	statement
Tendermint Core	`abci/types`	`Application`	`ResponseBeginBlock`	instance returned
			`ResponseDeliverTx`	instance returned
			`ResponseEndBlock`	instance returned
			`ResponseCommit`	instance returned
			`ResponseCheckTx`	instance returned
Cosmos SDK	`types`	`KVStore`	`Set`	parameters
			`Delete`	parameters
	`kv`, `dbadapter`, `gaskv`, `iavl`, `listenkv`, `prefix`, `tracekv`,	`Store`	`Set`	parameters
			`Delete`	parameters
	`types/errors`		`ABCIError`	statement
			`Redact`	statement
			`ResponseDeliverTx`	statement
			`ResponseCheckTx`	statement
			`WithType`	statement
			`Wrap`	statement
			`Wrapf`	statement

of version 2.4) involved in the data-write proposal. The semantics of these components does not affect the blockchain state until the transaction is validated and successfully committed. Hence, if these components lead to different results (i.e., changes to the shared state) due to non-determinism, consensus will not validate the transaction and no new state will be committed. Regarding the response statements, HF provides the `Success` and `Error` methods to yield successful and failed transaction responses, respectively.

3.2.2 Tendermint Core APIs for Go

Tendermint Core is a middleware with no explicit access to application state by design, enabling communication through the *Application BlockChain Interface* (ABCI[11]). Figure 6 depicts the consensus process used to validate and store a transaction using the ABCI methods. As reported in the official documentation [27] of Tendermint Core v. 0.35.1, only `BeginBlock`, `DeliverTx`, `EndBlock`, and `Commit` must be strictly deterministic to ensure consensus. Although the logic of these methods is different, they possess similar structure: they all accept a request and return a response (`ResponseBeginBlock`, `ResponseDeliverTx`, `ResponseEndBlock`, `ResponseCommit`), with the latter that must be deterministic.

3.2.3 Cosmos SDK APIs

Cosmos SDK handles both the application and the blockchain state through the *store*[12]. At a high level, the store is a set of key-value pairs used to store and retrieve data, implemented by default as a *multistore* (i.e., a store of stores), as shown in Figure 7. The multistore encapsulation enables modularity of the Cosmos SDK, as each module declares and manages

[11] `https://github.com/tendermint/tendermint/blob/7983f9cc36c31e140e46ae5cb00ed39f637ef283/docs/introduction/what-is-tendermint.md#abci-overview`.
[12] `https://github.com/cosmos/cosmos-sdk/blob/2b24afad075894dd1727d057f87e2be24238016f/docs/core/store.md`.

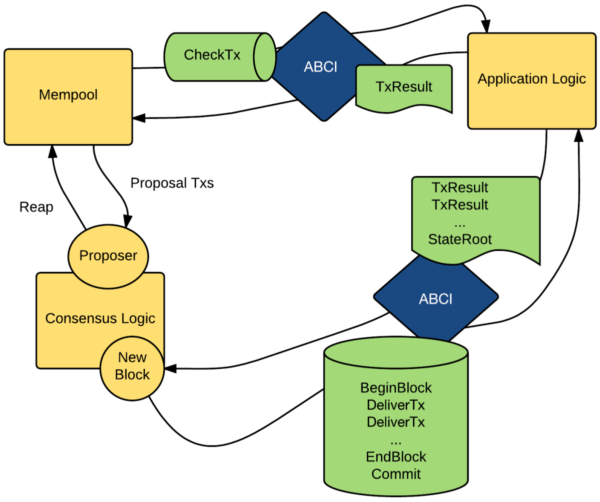

Figure 6 ABCI methods and consensus flow.

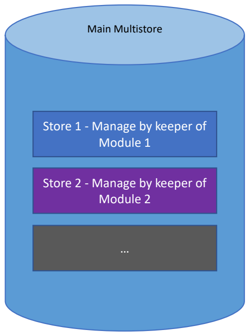

Figure 7 Main store of Cosmos SDK.

its own subset of the state using specific keys. Keys are typically held by *keepers*, a Cosmos SDK abstraction with the role of managing access to the multistore's subset defined by each module. The `Store` type is declared in several packages (e.g., `kv`, `tracekv`, `gaskv`, `ival`), with all definitions implementing the `KVStore` interface. The latter provides common APIs to access and modify the state of the blockchain using methods such as `Set` and `Del`. As for responses, Cosmos provides several methods (such as `ABCIError`, `Wrap`, `ResponseDeliverTx`) in package `types/errors` to return failed transaction responses.

4 Information Flow Analysis for Non-Determinism Detection

In this section we introduce and discuss our approach for detecting non-deterministic behaviors in blockchain software. In particular, we consider non-determinism as *critical* only if a non-deterministic value can affect the blockchain state, either directly (i.e., being stored inside the state) or indirectly (e.g., guarding the execution of state updates). Any other usage of non-determinism is considered safe, as it does not affect the blockchain state or response. As such, when mentioning non-determinism in the remainder of the chapter, we implicitly refer

```
var l, h
h := l
```

(a)

```
var l, h
if l = true then
  h := 3
else
  h := 42
```

(b)

```
var l, h
if h = 1 then
  (* time-consuming work *)
l := 0
```

(c)

Figure 8 Example of (a) explicit, (b) implicit, and (c) side channel flows, where `h` and `l` represent secret and public variables respectively.

to its critical version. We rely on information flow analysis for detecting values originating from sources of non-determinism that can affect the state of the blockchain. We only focus on static analyses, since they soundly over-approximate all possible behaviors of target programs and can thus give guarantees about the absence of such behaviors. We instantiate two types of analyses: a *Taint* analysis, able to capture the so-called *explicit flows*, and a *Non-interference* analysis, that can also detect *implicit flows*.

4.1 An Overview on Information Flow

Information flow analyses [11, 38] address the problem of understanding how information *flows* from one variable to another during a program's execution. These analyses usually partition the space of program variables into *private* (or secret) and *public*, with the latter being accessible to – and in some cases also modifiable by – an external attacker. The goal of these analyses is then to find program executions where information *flows* from one partition to the other, that is, where values of variables from one partition can affect the values of variables from the other one. Figure 8 reports examples[13] of the three main types of flows, namely:

- *explicit flow*: when a secret variable is assigned to a value obtained starting from public variables;
- *implicit flow*: when an assignment to a secret variable is conditionally executed depending on values of public variables;
- *side channel*: where some observable properties of the execution, e.g., the amount of computational resources used, depends on the values of some secret variables.

In general, the term *source* is traditionally used for variables holding values that one wants to track along program executions, while *sink* is used to describe locations where values coming from sources should not flow. Using this terminology, when the property of interest ensures the *integrity* of secret variables, information flow analyses can be instantiated using public variables as sources and private ones as sinks, exactly as in Fig. 8 and in the list above. These are able to detect situations where (i) a possibly corrupted value provided by a malicious attacker could be stored into variables whose content is supposed to be reliable, or (ii) such a value governs the update to private variables. If, however, one wants to ensure the *confidentiality* of secret variables, the same analyses can be recast with private variables acting as sources and public ones as sinks, thus searching for flows in the opposite direction. The target of the analysis is then to find disclosures of private data to external entities.

In the context of non-deterministic behaviors in blockchain environments, information flow analyses can be used to detect when non-deterministic values end up or affect the blockchain's state, thus checking the *integrity* of that state w.r.t. non-deterministic values.

[13] `https://en.wikipedia.org/wiki/Information_flow_(information_theory)`.

As such, we are interested in information flowing from public to private variables, and we will use *sources* to identify ones that are initialized to non-deterministic values and *sinks* to identify all variables that have an effect on the blockchain's state. Moreover, we will focus on explicit and implicit flows. In fact, side channels are typically studied to detect secret information leaking through, for instance, execution time, thus violating the confidentiality of that information instead of its integrity. On the other hand, explicit and implicit flows identify non-deterministic values that are either used to update the blockchain's state or a transaction's result, or that govern their execution. As a concrete example, recall the code from Figure 1: the vulnerability presented there is an implicit flow since the blockchain's state is not directly updated with non-deterministic values, but the execution of the update (i.e., the `return` statement) is conditional to some non-deterministic value (i.e., `g.Expiration.Unix() < time.Now().Unix()`).

In the following, we introduce two well-established information flow analyses that we will use for non-determinism detection.

4.1.1 Non-Interference

Non-interference [24, 25] is a notion of security capturing the idea that if computations over private information are independent from public information, then no leakage of the former can happen. In simple terms, after partitioning the space of inputs of a program `P` into *low* (private or secret, denoted by $\mathbb{L}$), and *high* (public or available to anyone, denoted by $\mathbb{H}$), *Non-interference* is satisfied if changes in the high input do not affect the observable (i.e., low) output of the program:

$$\forall i_{\mathbb{L}} \in \mathbb{L}, \forall i_{\mathbb{H}}, i'_{\mathbb{H}} \in \mathbb{H} \;.\; \mathtt{P}(i_{\mathbb{L}}, i_{\mathbb{H}})_{\mathbb{L}} = \mathtt{P}(i_{\mathbb{L}}, i'_{\mathbb{H}})_{\mathbb{L}}$$

This notion is often instantiated in language-based security by partitioning the space of program variables between $\mathbb{L}$ and $\mathbb{H}$, and finding instances of explicit or implicit flows between these partitions. Such analysis computes, for each program point, a mapping from variables to the information level they hold (low or high), while also keeping track of an execution state depending on the information level of the Boolean conditions that guard the program point. Violations of *Non-interference* for integrity can then be detected whenever an assignment to a variable in $\mathbb{H}$ either *(i)* assigns a low value (that is, an expression involving variables in $\mathbb{L}$), or *(ii)* happens with a low execution state (that is, guarded by at least a Boolean condition that involves variables in $\mathbb{L}$), thus identifying both explicit and implicit flows. This can be formalized as a type system for security [38].

4.1.2 Taint Analysis

Taint analysis [43, 14] is an instance of information flow analysis that can be seen as simplification of *Non-interference* considering only explicit flows. In this context, variables are partitioned into *tainted* and *untainted* (or *clean*), with the former representing variables that can be tampered with by an attacker and the latter representing variables that should not contain tainted values across all possible program executions. Roughly, *Taint* analysis corresponds to the language-based *Non-interference* instantiation without the execution state, thus unable to detect implicit flows. *Taint* has been instantiated to detect many defects in real-world software, such as web-application vulnerabilities [16], privacy issues [22] (also related to GDPR compliance [20]), and vulnerabilities of IoT software [17].

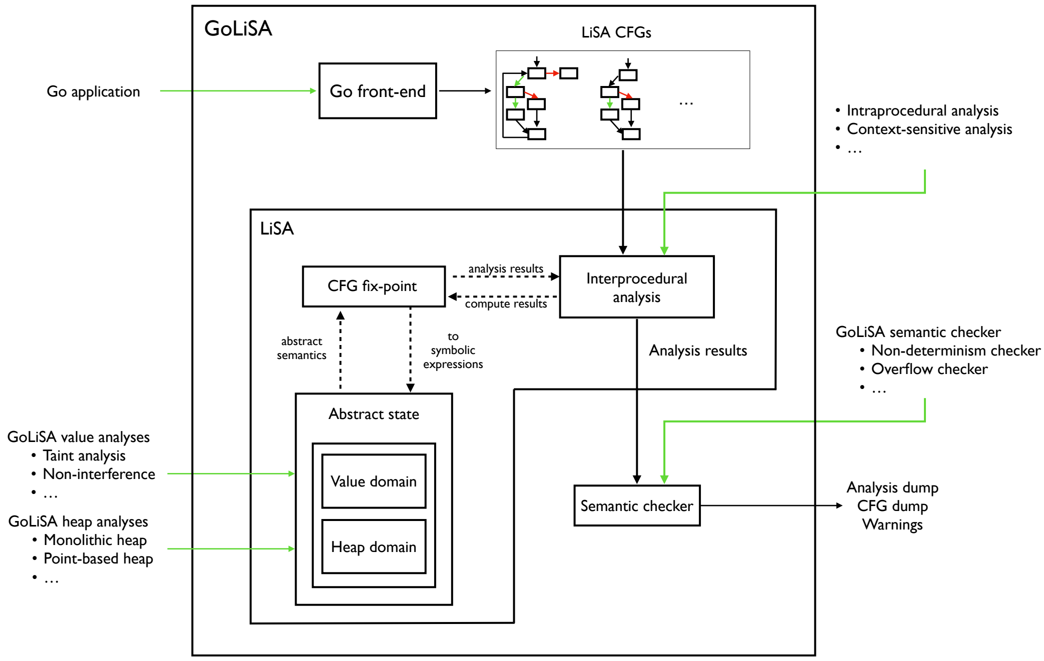

Figure 9 GoLiSA overall execution.

4.2 The GoLiSA Static Analyzer

GoLiSA[14] is an abstract interpretation [10] based static analyzer for Go applications, on which we will rely for the rest of the paper for reasoning about blockchain software written in Go. In this section, we present its architecture and its main feature. GoLiSA relies on LiSA [19, 34] (Library for Static Analysis[15]), a Java library that provides a complete infrastructure for the development of static analyzers based on abstract interpretation. In particular, LiSA implements several standard components of abstract interpretation-based analyzers, such as an extensible control-flow graph representation (CFG), a common analysis framework for the development of new static analyses, and fixpoint algorithms on LiSA CFGs.

The high-level analysis process of GoLiSA is reported in Fig. 9. The analysis starts with the *Go front-end* (a sub-component of GoLiSA) that compiles Go source code into LiSA CFGs and defines the semantics, types and language-specific algorithms that implement the Go execution model, capturing the peculiarities of Go in order to make them understandable to LiSA (e.g., scoping and shadowing of variables[16]). These CFGs are then passed to LiSA, that analyzes them in a generic language-independent fashion. Roughly, CFGs are passed to an *interprocedural analysis*, a component that cooperates with a *call graph* to resolve calls and compute their results. The interprocedural analysis computes fixpoints over CFGs according to some implementation-specific logic (e.g., modularly, relying on call chains, ...). Each individual fixpoint relies on language-specific analysis-independent semantics for CFG nodes, that is directly provided by GoLiSA: each node is rewritten into a sequence of *symbolic expressions*, modelling the effects that executing a high-level instruction has on the program state through low-level atomic semantic operations. Each of these symbolic expressions is fed

[14] Available at `https://github.com/lisa-analyzer/go-lisa`
[15] LiSA project and documentation available at `https://github.com/lisa-analyzer/lisa`
[16] `https://go.dev/ref/spec#Declarations_and_scope`

to an *abstract state* [15], a combination of an abstract domain modelling the dynamic memory of the program (*heap domain*, e.g., point-based heap analysis [1]) and one for tracking values of program variables and memory locations (*value domain*, e.g., intervals [10]). The abstract state and its underlying domains compute a sound over-approximation of the expression's effects according to their specific logic, and this can later be exploited by *semantic checks* to issue warnings that are of interest for the user. All analysis components (interprocedural analysis, call graph, abstract state, heap domain, value domain and semantic checks) are part of LiSA's configuration, enabling modular composition and implementation of each component.

4.3 GoLiSA for Non-Deterministic Behaviors Detection

At this point, we are in position to instantiate GoLiSA for the static detection of non-deterministic behaviors in blockchain software. The core idea of our solution is to track the values generated by the hotspots identified in Section 3.1 during the execution of a program using either *Taint* analysis or *Non-interference*. Similarly, after the analysis completes, we can use a semantic checker to exploit the abstract information provided by the domain of choice, checking if any of the sinks specified in Section 3.2 receives one such non-deterministic value as parameter or, in the case of *Non-interference*, if the sink is found in a *low* execution state.

GoLiSA's analysis is instantiated as follows:

- *Taint* analysis and *Non-interference* are implemented as value domains, both of them being non-relational domains (i.e., mapping from variables to abstract values – taintedness and integrity level respectively – with no relations between different variables), with *Non-interference* keeping track of the abstractions for each guard;
- field-insensitive program point-based heap domain (Section 8.3.4 of [37]), where any concrete heap location allocated at a specific program point is abstracted to a single abstract heap identifier;
- context-sensitive [39, 28] interprocedural analysis, abstracting full call-chain results until a recursion is found;
- runtime types-based call graph, using the runtime types of call receivers to determine their targets;
- two semantic checkers, for *Taint* analysis and *Non-interference*, that scan the code in search for sinks, checking the taintedness or integrity level of each sink.

The analysis begins by visiting the input program to detect the statements annotated as sources and propagating the information from them. The analyses produce, for each program point, a mapping stating if each variable is the result of a non-deterministic computation. These mappings are then used by our semantic checkers, that visit the program in search for statements annotated as sinks. When one is found, the mappings are used to determine if values used as parameters of the call are critical or, in the case of *Non-interference*, if the call happens on a critical state. The choice of the analysis to run (and thus of the checker to execute) is left to the user.

For instance, let us consider the fragment reported in Figure 4a. At line 5, despite variable `elapsed` being marked as tainted, no warning is raised by GoLiSA regardless of the chosen analysis, as it does not reach any sensitive sink. Instead, analyzing the fragment from Figure 4b results in the following alarm:

```
The value passed for the 2nd parameter of this call is tainted,
and it reaches the sink at parameter 'value'
```

The warning is issued with both analyses, since variable `t` is marked as tainted and reaches a blockchain state modifier through an explicit flow.

Consider now the example reported in Figure 1. Here, no explicit flow happens at line 3, that contains the blockchain state modifier `Wrap`, but its execution depends on the non-deterministic value used in the condition at line 2, that is, `time.Now().Unix()`. As this is an implicit flow, the *Taint* analysis is not able the detect it. GoLiSA will however discover it with *Non-interference*, raising the following alarm:

```
The execution of this call is guarded by a tainted condition,
resulting in an implicit flow
```

4.4 Detection of Sources and Sinks in GoLiSA

To exploit information flow analyses, the analyzer must know which are the sources and sinks of the program. In this regard, GoLiSA provides a solution based on annotations, marking the corresponding statements as sources and sinks. In the following, we describe how GoLiSA annotates sources (Table 1) and sinks (Table 2) depending on their types.

Methods and functions

As shown in Tables 1 and 2, all sinks and several sources correspond to functions and methods of APIs from either the Go runtime or the blockchain frameworks. GoLiSA contains a list of the signature of these functions and methods and it automatically annotates the corresponding calls in the program by syntactically matching them. While we rely on manual annotations, they can also be generated using automated tools (e.g., SARL [18]). For instance, when GoLiSA iterates over the following snippet, it is able to discover the call to `time.Now`, that gets annotated as source, and the one to `PutState`, whose parameters get annotated as sinks:

```
1 key := "key"
2 tm := time.Now()
3 stub.PutState(key, []byte(tm))
```

Then, the information flow analysis propagates taintedness from the return value of `time.Now` to the second parameter of `PutState`, thus issuing an alarm at line 3.

Map iterations

To detect iterations over maps, one needs to reason about typing. GoLiSA exploits runtime types inferred by the analysis to identify `range` statements happening over maps. If a map iteration occurs, that is, if the object in a `range` statement is inferred to be a map, then GoLiSA marks as sources the variables used to store keys and values of the map. Consider as an example the following code snippet:

```
1 s := ""
2 kvs := map[string]string{"a": "hello", "b": "world!"}
3 for k, v := range kvs {
4   s += v
5 }
6 stub.PutState("key", []byte(s))
```

While analyzing the code, `range` statements are checked for the types of their parameter. GoLiSA annotates as sources both `k` and `v`, as `kvs` is inferred to be a map, while the sink at line 6 is detected through already discussed annotations. Information flow analyses can then propagate the taintedness from `v` to `s`, that in turn flows to the second parameter of `PutState`, issuing an alarm at line 6.

Global variables

GoLiSA syntactically annotates every global variable appearing in the program as a source of non-determinism, as their value could be modified independently on each peer. For instance, in the following code, the value of global variable `glob` could differ from peer to peer depending on the number of times function `inc` has been executed. This can happen as not all peers simulate the same transaction, for instance due to differences in the endorsement policy of each peer [32].

```go
1 var glob string
2 func inc() {
3   glob += "a"
4 }
5 func (s *SmartContract) transaction(stub shim.ChaincodeStubInterface) sc.Response {
6   stub.PutState("key", []byte(glob))
7 }
```

Before the analysis, GoLiSA iterates over all program components, annotating `glob` as a source. The sink at line 6 is annotated as sink as previously discussed. Then, the information flow analysis propagates taintedness from `glob` to the second parameter of the call to `PutState`, raising an alarm at line 6.

Go routines

GoLiSA inspects the code of Go routines, checking the scope of variables they use. If these are defined outside the routine using them, they are effectively shared among threads, potentially leading to race conditions or non-deterministic behaviors. Hence, GoLiSA annotates the such variables as sources. As an example, the following snippet defines and invokes a simple Go routine that modifies a variable defined in an enclosing scope:

```go
1 s:= ""
2 go func(){
3   for i := 1; i <= 10000; i++ {
4     s += "0"
5   }
6 }
7 stub.PutState("key", []byte(s))
```

When GoLiSA finds the Go routine, it checks the scopes of each variable, inferring that `s` is declared outside the routine itself. Hence, GoLiSA annotates `s` at line 1 as source, while the sink at line 7 is annotated as previously discussed. Then, the information flow analysis propagates taintedness from `s` to the second parameter of `PutState`, issuing an alarm at line 7 since the value of `s` depends how many times the Go routine has executed the loop body.

Go channels

Channels are pipes that connect concurrent Go routines. Operator `<-` allows interaction with channels to retrieve a value from them, blocking until one is available. GoLiSA annotates as sources the instructions reading values from channels, as the order in which these are written is intrinsically non-deterministic. Consider the following example:

```go
1 c := make(chan int)
2 go myroutine1(c)
3 go myroutine2(c)
4 x, y := <- c, <- c
5 stub.PutState("key", []byte(x))
```

GoLiSA iterates over the program searching for occurrences of the operator `<-`. It then annotates variables `x` and `y` as sources, as they receive a value from channel `c`. The sink at line 5 is detected as previously discussed. The information flow analysis can then propagate taintedness from `x` to the second parameter of `PutState`, resulting in an alarm at line 5.

5 Experimental Evaluation

In this section, we discuss the experimental evaluation of the information flow analyses implemented in GoLiSA to detect non-determinism issues in real-world blockchain software. First, we study them from a quantitative point of view, on a set of 651 real-world HF smart contracts retrieved from public GitHub repositories. The evaluation focuses on the HF framework since, to the best of our knowledge, it is the only framework supported by several static analyzers detecting non-determinism issues. This will allow us to compare GoLiSA against state-of-the-art static analyzers in this domain. Furthermore, HF is currently the most popular and widespread blockchain framework among public GitHub repositories, with most smart contracts written in Go. Nevertheless, GoLiSA provides support also for detecting non-determinism behaviors for Cosmos SDK and Tendermint Core smart contracts and DApps.[17]

We compare GoLiSA with two open-source static analyzers for chaincodes, namely Revive^CC and ChainCode Analyzer. The experiments show that GoLiSA produces more precise results in detecting non-deterministic behaviors, outperforming existing static analyzers.

Then, we evaluate the quality of our results on two specific real-world applications, to show how the static analyses discussed in Section 4 work and how the information is propagated in smart contracts. In particular, we selected the first application from the HF benchmark, while the second one is a Cosmos SDK application.[18]

All the experiments was performed on a HP EliteBook 850 G4 equipped with an Intel Core i7-7500U at 2,70/2,90 GHz and 16 GB of RAM running Windows 10 Pro 64bit, Oracle JDK version 13, and Go version 1.17.

5.1 Quantitative Evaluation

The experimental artifact set has been retrieved from 954 GitHub repositories, by querying for the *chaincode* keyword, as smart contracts are called in HF, and selecting chaincodes from unforked Go repositories only[19], that include the `Invoke` and `Init` methods: these are the transaction requests' entry points for chaincodes.[20] Then, we filtered out files unrelated to smart contracts and removed chaincodes not analyzable due of failures either GoLiSA or the tools discussed in Sect. 5.1.1. In particular, GoLiSA failures on such chaincodes are due to current missing support of high-order functions, recursion, and C code invocation via the built-in Go `cmd/go` package.[21] This resulted in a benchmark consists of 651 chaincodes only ($\sim$167391 LoCs), that, from here on, we refer to as $\mathbb{HF}$. Then, each chaincode has been manually inspected before applying GoLiSA to search for critical non-deterministic behavior. In particular, for each chaincode, we manually searched for sources of non-determinism (if present) and checked if the result of the corresponding instructions could have an impact (i.e., an update) on the blockchain global state or on the response. If so, we classified this behaviour as critical/harmful. On the selected benchmark, we have found a total of 124 critical/harmful non-deterministic behaviours. In our evaluation, a warning raised by an

[17] An industrial application of GoLiSA for detecting non-determinism in Cosmos SDK can be found here [36].
[18] The example reported in Figure 1 contains a snippet of code of this application
[19] `https://api.github.com/search/repositories?q=chaincode+fork:false+language:Go+archived:false&sort=stars&order=desc`. Accessed: 17-10-2022.
[20] See `https://pkg.go.dev/github.com/hyperledger/fabric-chaincode-go/shim`.
[21] We decided not to implement those standard features since this would have required a relevant effort to support only a few more chaincodes.

Table 3 Analysis evaluation.

Analysis	**#A**	**#U**	**ET**	**AT**	**#W**	**#TP**	**#FP**	**#FN**
Taint	68	583	2h:15m:03s	12.45s	173	118	55	7
Non-interference	69	582	2h:25m:18s	13.39s	195	124	71	0

analyzer has been classified as true positive (TP) if it was part of the 124 critical behaviours mentioned above, and as false positive (FP) if not. All the critical behaviours, part of the 124 manually detected, for which there was no warning, have been marked as false negative (FN).

Table 3 reports the results of the experimental evaluation of GoLiSA over the benchmark $\mathbb{HF}$, where **#A** is the number of affected chaincodes (i.e., chaincodes where at least a warning was issued), **#U** is the number of unaffected chaincodes (i.e., chaincodes where no warning was raised), **ET** is the total execution time, **AT** is the average execution time, **#W** is the total number of warnings issued, **#TP** is the number of true positives among the raised warnings, **#FP** is the number of false positives among the raised warnings, and **#FN** is the number of false negatives. In terms of execution time, the analyses performed averagely in around 15 seconds per chaincode. The experiments shows that *Non-interference* performs better than *Taint* in terms of precision, being able to detect all the true positives contained in $\mathbb{HF}$, with a ratio of false positives less than 40%. This was expected since, as we have already discussed in Section 4 and unlike *Non-interference*, *Taint* is only able to track explicit information flows. In fact, the 7 false negatives (column **#FN** of Table 3) produced by *Taint* correspond to implicit non-deterministic behaviors.

5.1.1 Comparison

We compared GoLiSA with the open-source static analyzers for Go chaincode described in Section 6, namely ChainCode Analyzer and Revive^CC. Table 4 reports the comparison between GoLiSA and these tools over the same benchmark $\mathbb{HF}$ discussed in Section 5.1.

The comparison shows that GoLiSA - *Non-interference* finds all the true issues contained in the benchmark, achieving the best and most accurate result in terms of precision with a 36.41% false positives ratio. Instead, although it has some false negatives, GoLiSA - *Taint* is the analysis with the lowest percentage of false positives with the 31.79% .

Revive^CC triggers 351 warnings out of which 77.49% are false positives. The only non-deterministic behaviour not detected by Revive^CC (last column) is due to the fact that it considers the `ioutil.ReadFile` API as safe, although reading a file should be considered non-deterministic in the blockchain context. Finally, ChainCode Analyzer is more precise w.r.t. Revive^CC, with 66.50% of false positives, but it has also the greatest number of false negatives, failing to detect a huge number of critical non-deterministic behaviors. This can be attributed to the fact that ChainCode Analyzer does not consider several APIs leading to non-determinism as critical and it fails to soundly detect iteration over maps.

Note that the amount of true positives discovered by GoLiSA analyses differs from the ones of other tools. In fact, GoLiSA is the only tool involved in our comparison that issues warnings on sinks rather than sources. This translates to fewer alarms being issued whenever values of multiple sources flow to the same sink (here, GoLiSA issues a single warning, while other tools issue one for each source), and to more alarms being raised whenever the value of a single source flows to multiple sinks (here instead, other tools issue a single warning, while GoLiSA issues one for each sink).

Table 4 Warnings triggered by the analyzers on HF.

Tools	# W	# TP	# FP	# FN
GoLiSA - *Taint*	173	118	55	7
GoLiSA - *Non-interference*	195	124	71	0
ChainCode Analyzer	203	68	135	53
Revive^CC	351	79	272	1

```
1  func (s *SmartContract) registrarBoleto(APIstub shim.ChaincodeStubInterface, args []
       string) sc.Response {
2    // [...]
3    objBoleto.CodigoBarra = strconv.Itoa((rand.Intn(5) + 10000000 + // [...]
4    var notExpiredDate = time.Now()
5    objBoleto.DataVencimento = notExpiredDate.Format("02/01/2006")
6    // [...]
7    boletoAsBytes, _ := json.Marshal(objBoleto)
8    APIstub.PutState(args[0], boletoAsBytes)
9    // [...]
10 }
```

Figure 10 Method `registrarBoleto` of *boleto* contract.

5.2 Qualitative Evaluation

5.2.1 Explicit Flow: the Boleto Contract

The *boleto* contract[22], taken from HF, comes with a real non-determinism issue that can be found with explicit flows, and that was also detected by other tools during the comparison of Section 5.1.1. The *boleto* contract (Figure 10) seems to be a proof of concept application handling tickets in an e-commerce store, with the method `registrarBoleto` used to register a ticket.

Analyzing *boleto*, GoLiSA detects the explicit flow leading to a non-deterministic behavior with both *Taint* and *Non-interference*. Method `registrarBoleto` contains two different sources of non-determinism that directly flow into the same sink. The first source detected by GoLiSA is the usage of the *Random API* to generate a barcode at line 3. Instead, the second source is the usage of the *OS API* that retrieves the local machine's time to set a date at line 4. As values from both sources are used to update fields of `objBoleto`, the latter is marked as tainted by the analysis, resulting in `boletoAsBytes` being tainted as well. As reported in Table 2, `PutState`'s parameters are considered as sinks by GoLiSA's analyses. According to the official documentation of HF[23], the `PutState` method does not affect the ledger until the transaction is validated and successfully committed. However, a transaction needs to produce the same results among different peers to be validated. Hence, as passing non-deterministic values to `PutState` will cause the transaction to fail, GoLiSA raises a warning on line 8.

5.2.2 Implicit Flow: Cosmos SDK v.43

Analyzing the code in Figure 1, GoLiSA is able to detect an implicit flow that leads to a non-deterministic behavior, that can only be detected using *Non-interference*. The `ValidateBasic` method of Cosmos SDK v. 0.43.x and v. 0.44.{0,1} was designed to validate

[22] `https://github.com/arthurmsouza/boleto/blob/master/boleto-chaincode/boleto.go`
[23] `https://github.com/hyperledger/fabric-chaincode-go/blob/1476cf1d3206f620db7eea12312c98669d39fa22/shim/interfaces.go`.

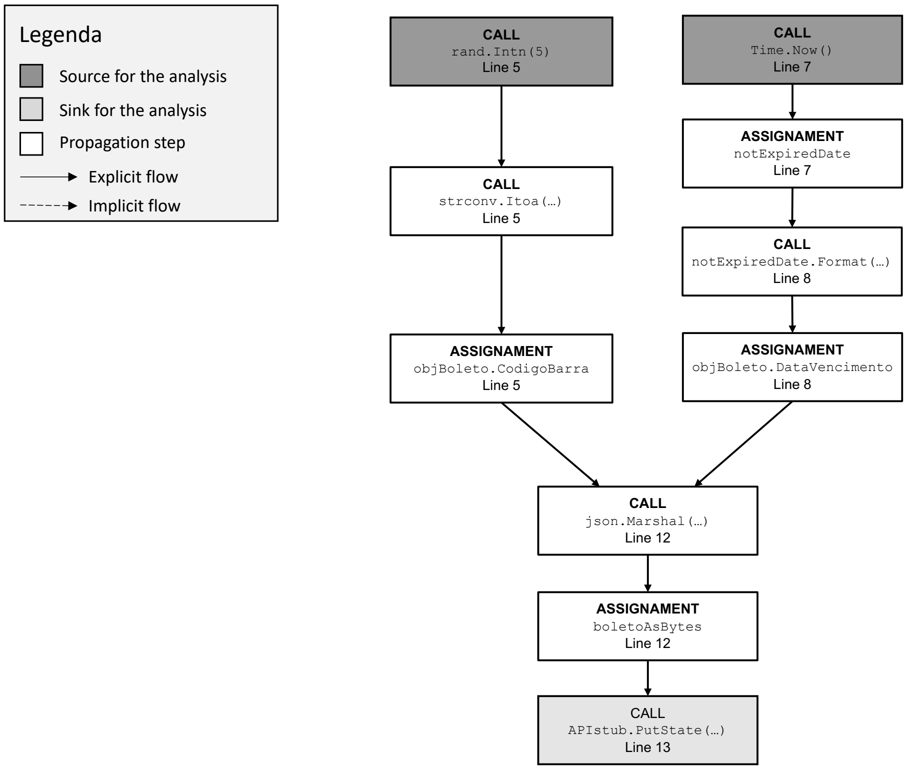

Figure 11 Simplified view of explicit flow computed by GoLiSA during the analysis of `registrarBoleto`.

a grant to ensure it has not yet expired. In this case, the source detected by GoLiSA is the *OS API* used to retrieve the local machine time involved in the expiration check of the grant time at line 2 of Figure 1. By propagating the information, GoLiSA detects that the expiration check governs the execution of return statement. Since the `Wrap` method is annotated as a sink, GoLiSA triggers an alarm at line 3 of Figure 1 as the sink is contained in a block whose guard depends on non-deterministic values.

5.3 Limits

Unlike some frameworks and GPLs used in other blockchains, frameworks targeted by this paper are used to develop *permissioned*, and often *private*, blockchains, meaning that the related software is not publicly available or released with open-source licenses. This is also the reason why the benchmark $\mathbb{HF}$ crawled from GitHub consists of 651 chaincodes, a number that is not comparable with smart contract benchmarks obtained investigating other (public and permissioned) blockchains. For instance, [44] collects 3075 distinct smart contracts from the Ethereum blockchain, resulting in a wider benchmark.

The proposed solution for detecting non-deterministic behaviors is fully static. It is well known that static analysis is intrinsically conservative and may produce false positives. Even if few have been raised by GoLiSA on the selected benchmark, one should expect more false positives when applying our approach to arbitrary DApps.

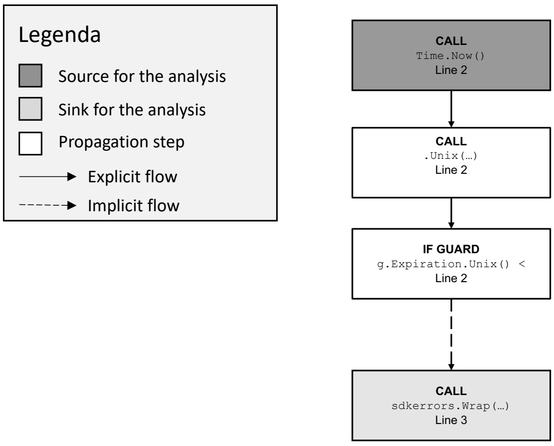

Figure 12 Simplified view of implicit flow computed by GoLiSA during the analysis of Figure 1.

6 Related Work

The non-determinism of smart contracts written in GPLs is a well-known issue [32, 45]. Frameworks such as Takamaka [41, 42] enforce determinism adopting a conservative approach that limits the set of instructions and APIs of the target language, avoiding unsafe statements that might lead to non-deterministic behaviors through white-listing fully deterministic APIs. This approach ensures safe development while preventing that API extensions coming with new language versions can bypass the check. However, it also severely limits the exploitable features of the GPL. On the other hand, black-listing undesired APIs is a much harder approach to maintain, but it seems the most widespread technique in Go analyzers. For instance, ChainCode Analyzer [31] and Revive^CC [40] detect mainly black-listed imports related non-deterministic APIs using a syntactical approach. Besides, they can detect non-deterministic map iterations by AST traversal with minimal syntactic reasoning. Signature of invoked functions can also be black-listed instead of imports [32]. These tools and frameworks inherently limit API usage, sensibly reducing the benefits of adopting a GPL even when the code poses no harm to the blockchain. The problem of detecting non-determinism has also been covered for parallel applications, suggesting that non-determinism is *"most often the result of a mistake on the part of the programmer"* [13].

7 Conclusion

In this paper, we proposed a flow-based approach for detecting critical non-deterministic behaviors, namely the ones affecting the blockchain state. Our proposal has been implemented in GoLiSA, a static analyzer for Go applications. To the best of our knowledge, GoLiSA is the first semantic-based static analyzer for blockchain software able to detect non-deterministic behaviors, with an extremely low false alarm prevision. In the context of smart contracts, the proposed approach is placed in an off-chain architecture, i.e., the analysis is done before smart contracts are deployed in the blockchain, and it is not mandatory. As future work, besides supporting the missing Go features discussed in Section 5 to enhance the analysis coverage, we plan to test GoLiSA in an on-chain architecture [35], making the non-determinism checker part of the consensus protocol, with the goal of keeping the code stored within the blockchain deterministic. The analysis could be enriched with a context-sensitive flow reconstructor, such as BackFlow [21], that starting from the results of a information flow engine, reconstructs how the information flows inside the program and builds paths connecting sources to sinks. Moreover, we have focused on the non-determinism problem, but our future research will address the problem of detecting other and equally critical vulnerabilities that can affect blockchain software written using general-purpose languages, such as numerical overflow.

Our proposal follows a fully static approach, justified by the fact that we aim at proving the determinism of blockchain software, regardless of the possible executions. However, even if the evaluation on the selected benchmark shows optimal results, the risk of getting false alarms analyzing other applications is still present, being our approach based on over-approximating possible executions via abstract interpretation. In future works, hybrid approaches between static and dynamic analyses will be investigated to get the benefits of both techniques.

Finally, in order to assess the effectiveness of our proposal, we have conducted our evaluation on Hyperledger Fabric blockchain software, mostly because it is the most popular framework among those cited in the paper. To give a larger coverage to GoLiSA of the blockchain software that can analyze, the next step will be to design significant benchmarks also for the other frameworks, such as Tendermint core and Cosmos SDK, on which we can experiment our static analyzer.

References

1 Lars Ole Andersen. Program Analysis and Specialization for the C Programming Language, 1994. Accessed: 01-12-2022. URL: `https://www.cs.cornell.edu/courses/cs711/2005fa/papers/andersen-thesis94.pdf`.

2 Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 30:1–30:15. ACM, 2018. `doi:10.1145/3190508.3190538`.

3 A. M. Antonopoulos. *Mastering Bitcoin: Programming the Open Blockchain*. O'Reilly, 2nd edition, 2017.

4 A. M. Antonopoulos and G. Wood. *Mastering Ethereum: Building Smart Contracts and Dapps*. O'Reilly, 2018.

5 Sotirios Brotsis, Nicholas Kolokotronis, Konstantinos Limniotis, Gueltoum Bendiab, and Stavros Shiaeles. On the security and privacy of hyperledger fabric: Challenges and open issues. In *2020 IEEE World Congress on Services (SERVICES)*, pages 197–204, 2020. `doi:10.1109/SERVICES48979.2020.00049`.

6 E. Buchman. *Tendermint: Byzantine Fault Tolerance in the Age of Blockchains*. PhD thesis, University of Guelph, 2016.

7 Ethan Buchman. Byzantine Fault Tolerant State Machine Replication in Any Programming Language. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 546, New York, NY, USA, 2019. Association for Computing Machinery.

8 V. Buterin. Ethereum Whitepaper, 2013. Available at `https://ethereum.org/en/whitepaper/`.

9 Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Arash Pourdamghani. Probabilistic Smart Contracts: Secure Randomness on the Blockchain. In *IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2019, Seoul, Korea (South), May 14-17, 2019*, pages 403–412. IEEE, 2019. `doi:10.1109/BLOC.2019.8751326`.

10 Patrick Cousot. *Principles of Abstract Interpretation*. MIT Press, 2021.

11 Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Commun. ACM*, 19(5):236–243, 1976. `doi:10.1145/360051.360056`.

12 ebuchman. Cosmos-SDK Vulnerability Retrospective: Security Advisory Jackfruit, October 12, 2021, 2021. Accessed: 01-12-2022. URL: `https://forum.cosmos.network/t/cosmos-sdk-vulnerability-retrospective-security-advisory-jackfruit-october-12-2021/5349`.

13 Perry A. Emrath and David A. Padua. Automatic Detection of Nondeterminacy in Parallel Programs. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, PADD '88, pages 89–99, New York, NY, USA, 1988. Association for Computing Machinery. `doi:10.1145/68210.69224`.

14 Michael D. Ernst, Alberto Lovato, Damiano Macedonio, Ciprian Spiridon, and Fausto Spoto. Boolean Formulas for the Static Identification of Injection Attacks in java. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, volume 9450 of *Lecture Notes in Computer Science*, pages 130–145. Springer, 2015. `doi:10.1007/978-3-662-48899-7_10`.

15 Pietro Ferrara. A generic framework for heap and value analyses of object-oriented programming languages. *Theor. Comput. Sci.*, 631:43–72, 2016. `doi:10.1016/j.tcs.2016.04.001`.

16 Pietro Ferrara, Elisa Burato, and Fausto Spoto. Security Analysis of the OWASP Benchmark with Julia. In *Proceedings of the First Italian Conference on Cybersecurity (ITASEC17), Venice, Italy, January 17-20, 2017*, volume 1816 of *CEUR Workshop Proceedings*, pages 242–247. CEUR-WS.org, 2017. Accessed: 01-12-2022. URL: `http://ceur-ws.org/Vol-1816/paper-24.pdf`.

17 Pietro Ferrara, Amit Kr Mandal, Agostino Cortesi, and Fausto Spoto. Static analysis for discovering IoT vulnerabilities. *Int. J. Softw. Tools Technol. Transf.*, 23(1):71–88, 2021. `doi:10.1007/s10009-020-00592-x`.

18 Pietro Ferrara and Luca Negrini. SARL: Oo framework specification for static analysis. In Maria Christakis, Nadia Polikarpova, Parasara Sridhar Duggirala, and Peter Schrammel, editors, *Software Verification*, pages 3–20, Cham, 2020. Springer International Publishing.

19 Pietro Ferrara, Luca Negrini, Vincenzo Arceri, and Agostino Cortesi. Static analysis for dummies: experiencing lisa. In Lisa Nguyen Quang Do and Caterina Urban, editors, *SOAP@PLDI 2021: Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, Virtual Event, Canada, 22 June, 2021*, pages 1–6. ACM, 2021. `doi:10.1145/3460946.3464316`.

20 Pietro Ferrara, Luca Olivieri, and Fausto Spoto. Tailoring Taint Analysis to GDPR. In *Privacy Technologies and Policy - 6th Annual Privacy Forum, APF 2018, Barcelona, Spain, June 13-14, 2018, Revised Selected Papers*, volume 11079 of *Lecture Notes in Computer Science*, pages 63–76. Springer, 2018. `doi:10.1007/978-3-030-02547-2_4`.

21 Pietro Ferrara, Luca Olivieri, and Fausto Spoto. Backflow: Backward context-sensitive flow reconstruction of taint analysis results. In *Verification, Model Checking, and Abstract Interpretation*, pages 23–43, Cham, 2020. Springer International Publishing.

22 Pietro Ferrara, Luca Olivieri, and Fausto Spoto. Static Privacy Analysis by Flow Reconstruction of Tainted data. *Int. J. Softw. Eng. Knowl. Eng.*, 31(7):973–1016, 2021. `doi:10.1142/S0218194021500303`.

23 Luca Foschini, Andrea Gavagna, Giuseppe Martuscelli, and Rebecca Montanari. Hyperledger Fabric Blockchain: Chaincode Performance Analysis. In *2020 IEEE International Conference on Communications, ICC 2020, Dublin, Ireland, June 7-11, 2020*, pages 1–6. IEEE, 2020. `doi:10.1109/ICC40277.2020.9149080`.

24 Joseph A. Goguen and José Meseguer. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, pages 11–20. IEEE Computer Society, 1982. `doi:10.1109/SP.1982.10014`.

25 Joseph A. Goguen and José Meseguer. Unwinding and Inference Control. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 29 - May 2, 1984*, pages 75–87. IEEE Computer Society, 1984. `doi:10.1109/SP.1984.10019`.

26 Hyperledger. Hyperledger fabric documentation. URL: `https://hyperledger-fabric.readthedocs.io/en/release-2.2/blockchain.html#what-is-hyperledger-fabric`.

27 Tendermint Inc. What is Tendermint: A Note on Determinism, 2022. Accessed: 01-12-2022. URL: `https://github.com/tendermint/tendermint/blob/7983f9cc36c31e140e46ae5cb00ed39f637ef283/docs/introduction/what-is-tendermint.md#a-note-on-determinism`.

28 Uday P. Khedker and Bageshri Karkare. Efficiency, precision, simplicity, and generality in interprocedural data flow analysis: Resurrecting the classical call strings method. In *Compiler Construction*, pages 213–228, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. doi:10.1007/978-3-540-78791-4_15.

29 J. Kwon. Tendermint: Consensus without mining, 2014.

30 J. Kwon and E. Buchman. Cosmos whitepaper, 2019.

31 kzhry. Chaincode Analyzer, 2021. Accessed: 01-12-2022. URL: `https://github.com/hyperledger-labs/chaincode-analyzer`.

32 Penghui Lv, Yu Wang, Yazhe Wang, and Qihui Zhou. Potential Risk Detection System of Hyperledger Fabric Smart Contract based on Static Analysis. In *IEEE Symposium on Computers and Communications, ISCC 2021, Athens, Greece, September 5-8, 2021*, pages 1–7. IEEE, 2021. doi:10.1109/ISCC53001.2021.9631249.

33 S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. Available at `https://bitcoin.org/bitcoin.pdf`, 2008.

34 Luca Negrini. *A generic framework for multilanguage analysis*. PhD thesis, Universitá Ca' Foscari Venezia, 2023.

35 Luca Olivieri, Fausto Spoto, and Fabio Tagliaferro. On-Chain Smart Contract Verification over Tendermint. In *Financial Cryptography and Data Security. FC 2021 International Workshops - CoDecFin, DeFi, VOTING, and WTSC, Virtual Event, March 5, 2021, Revised Selected Papers*, volume 12676 of *Lecture Notes in Computer Science*, pages 333–347. Springer, 2021. doi:10.1007/978-3-662-63958-0_28.

36 Luca Olivieri, Fabio Tagliaferro, Vincenzo Arceri, Marco Ruaro, Luca Negrini, Agostino Cortesi, Pietro Ferrara, Fausto Spoto, and Enrico Talin. Ensuring determinism in blockchain software with golisa: an industrial experience report. In Laure Gonnord and Laura Titolo, editors, *SOAP '22: 11th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, San Diego, CA, USA, 14 June 2022*, pages 23–29. ACM, 2022. doi:10.1145/3520313.3534658.

37 Xavier Rival and Kwangkeun Yi. *Introduction to static analysis: an abstract interpretation perspective*. Mit Press, 2020.

38 A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003. doi:10.1109/JSAC.2002.806121.

39 Micha Sharir, Amir Pnueli, et al. *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences, 1978.

40 sivachokkapu. Revivecc, 2021. Accessed: 01-12-2022. URL: `https://github.com/sivachokkapu/revive-cc`.

41 Fausto Spoto. A Java Framework for Smart Contracts. In *Financial Cryptography and Data Security - FC 2019 International Workshops, VOTING and WTSC, St. Kitts, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers*, volume 11599 of *Lecture Notes in Computer Science*, pages 122–137. Springer, 2019. doi:10.1007/978-3-030-43725-1_10.

42 Fausto Spoto. Enforcing Determinism of Java Smart Contracts. In *Financial Cryptography and Data Security - FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers*, volume 12063 of *Lecture Notes in Computer Science*, pages 568–583. Springer, 2020. doi:10.1007/978-3-030-54455-3_40.

43 Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 87–97. ACM, 2009. doi:10.1145/1542476.1542486.

44 Shuai Wang, Chengyu Zhang, and Zhendong Su. Detecting nondeterministic payment bugs in ethereum smart contracts. *Proc. ACM Program. Lang.*, 3(OOPSLA):189:1–189:29, 2019. doi:10.1145/3360615.

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