Ensuring Determinism in Blockchain Software with GoLiSA: An Industrial Experience Report

Luca Olivieri
luca.olivieri@univr.it
University of Verona
Corvallis S.r.l.
Italy

Fabio Tagliaferro
fabio.tagliaferro@univr.it
University of Verona
Commercio.network S.p.A.
Italy

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

Marco Ruaro
marco.ruaro@gmail.com
Commercio.network S.p.A.
Italy

Luca Negrini
luca.negrini@unive.it
Ca' Foscari University of Venice
Corvallis S.r.l.
Italy

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

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

Fausto Spoto
fausto.spoto@univr.it
University of Verona
Italy

Enrico Talin
enrico.talin@commerc.io
Commercio.network S.p.A.
Italy

Abstract

Ensuring determinism is mandatory when writing blockchain software. When determinism is not met it can lead to serious implications in the blockchain network while compromising the software development, release, and patching processes. In the industrial context, it is widespread to adopt general-purpose languages, such as Go, for developing blockchain solutions. However, it is not surprising that non-deterministic behaviors may arise, being these programming languages not originally designed for blockchain purposes. In this paper, we present an experience report on ensuring determinism in blockchain software with GoLiSA, a static analyzer based on abstract interpretation for Go applications, in an industrial context. In particular, we ran GoLiSA on Commercio.network, a blockchain-based solution for exchanging electronic documents in a legally binding way. Thanks to GoLiSA, non-trivial bugs got detected and the analysis performed made it possible to identify the critical points where to apply the fixes.

CCS Concepts: • **Security and privacy** → **Distributed systems security**; • **Theory of computation** → **Program analysis**; • **Software and its engineering** → *General programming languages.*

Keywords: Static Analysis, Blockchain, Non-determinism, Cosmos SDK, Go language, Software testing

ACM Reference Format:
Luca Olivieri, Fabio Tagliaferro, Vincenzo Arceri, Marco Ruaro, Luca Negrini, Agostino Cortesi, Pietro Ferrara, Fausto Spoto, and Enrico Talin. 2022. Ensuring Determinism in Blockchain Software with GoLiSA: An Industrial Experience Report. In *Proceedings of the 11th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '22), June 14, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3520313.3534658

SOAP '22, June 14, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9274-7/22/06…$15.00
https://doi.org/10.1145/3520313.3534658

1 Introduction

A blockchain can be described as a deterministic state machine replicated into a distributed and decentralized network, preventing data tampering through a consensus mechanism. In simple terms, a deterministic state machine is just a program that holds a state and changes deterministically based on the inputs it receives. However, the blockchain is a distributed and decentralized entity on multiple nodes of a network. Therefore, it requires software running on the majority of nodes to compute and commit the same result, in order to achieve a common consensus state among peers and update the blockchain global state.

In the industrial context, general-purpose programming languages are increasingly used for writing blockchain software. Their success is mostly due to their wide diffusion and popularity, which implies that large pools of expert developers can be re-employed. Moreover, these languages are typically well supported in terms of libraries and tools

(IDEs, test suites, monitoring and profiling tools, etc.). However, general-purpose languages were not conceived for blockchain software and may introduce non-deterministic behavior inside the blockchain software.

In joint work with the company Commercio.network, we studied the impact of non-deterministic behaviors in a blockchain and how they can compromise the software development, release, and patching of blockchain solutions.

In this paper, we present the results about issues of non-determinism detected by formal verification, with the GoLiSA analyzer, of the open-source software provided by Commercio.network. Besides identifying these issues, we also discuss the implications during the development phase, such as mitigating risks before release and problems related to a possible patch.

2 Ensuring Determinism in Blockchain

The software deployed into a blockchain is distributed and decentralized in different peers of the blockchain network. The consensus mechanism is the component that checks the results of blockchain software, the state of peers involved in the consensus and allows or not to update the global state of the blockchain. If a certain threshold of commonly committed states among peers is reached, the common state is used to update the global state (i.e., consensus reached), otherwise it is discarded (i.e., consensus not reached) avoiding updating the state differently among the blockchain nodes.

Therefore, ensuring determinism in blockchain software becomes of crucial importance. In particular, starting from a common state, it guarantees that the same result is reached with the same response in any distinct blockchain peer, avoiding inconsistency among blockchain participants and mitigating consensus failures.

Domain-specific languages for blockchain software, such as Solidity for developing *smart contracts* in Ethereum [17], are deterministic by design. Instead, general-purpose programming languages, such as Go, may not guarantee it because they might lead to non-deterministic behaviors due to the semantics of some standard operations or methods, such as map iteration[1], random API, system API, etc. (Tab. 1 shows the potential sources of non-determinism in the Go language). For this reason, on the latter, it is necessary to check that the code does not present this type of problem before using it to launch a blockchain network.

3 Commercio.network and Cosmos SDK

Commercio.network [4] is an open-source decentralized application provided by the homonymous company[2]. As a blockchain, it can be described as a *permissioned* Proof-Of-Stake network, where a validator must join a consortium for being able to participate in the consensus. It can be described also as *public* since anyone can set up a node[3] and synchronize it with the Commercio.network main-net.The main purpose of this blockchain is to exchange electronic documents in a legally binding way thanks to the eIDAS Compliance[4], while following the principles of Self-Sovereign Identity [1].

Table 1. Overview of non-deterministic behaviors in Go. Most of the APIs contained in these packages lead to non-deterministic behaviors, but some can be considered safe for determinism.

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

As shown in Fig. 1, the architecture of Commercio.network is based on the *Cosmos SDK* [12], an open-source framework written in the Go programming language allowing the creation of *modules* for blockchain software. More precisely, the Cosmos SDK is not a smart contract framework, i.e., the framework is not directly involved to implement smart contracts. It is a real framework for *decentralised applications* (DApps) that can support different and peculiar functionalities[5], running in a distributed set of nodes. DApp developers need to worry only about the application logic, which is separated from the consensus and networking layers. In fact, the Cosmos SDK abstracts the machinery needed to set up a network running through the consensus mechanism *Tendermint Core* [3], recently rebranded as *Ignite*[6]. The architecture of a module conventionally revolves around the *keeper*, a package and entity implementing its core functionalities. The Cosmos SDK can also be seen as a collection of modules, that can be used to build custom ones, following the object-capability model [7]. For example, the Commercio.network module `commerciokyc` uses the keeper of another custom module, `commerciomint`, along with other modules coming from the library of Cosmos SDK.

4 An Overview of GoLiSA

GoLiSA[7] is a parametric static analyzer based on abstract interpretation [5, 6] for Go applications. GoLiSA exploits LiSA [9] (Library for Static Analysis), a library that provides a complete infrastructure for the development of static

[1] In Go, the iteration over maps is not guaranteed to be deterministic: https://golang.org/ref/spec#For_statements

[2] https://commercio.network/

[3] https://github.com/commercionetwork/commercionetwork

[4] https://digital-strategy.ec.europa.eu/en/policies/eidas-regulation

[5] Potentially, also the management of smart contracts is feasible. The *CosmWasm* (https://cosmwasm.com) module, for example, allows to develop WebAssembly contracts with high level languages such as Rust. This paper does not focus on such kind of smart contracts.

[6] https://ignite.com/

[7] Available at https://github.com/UniVE-SSV/go-lisa/tree/soap22

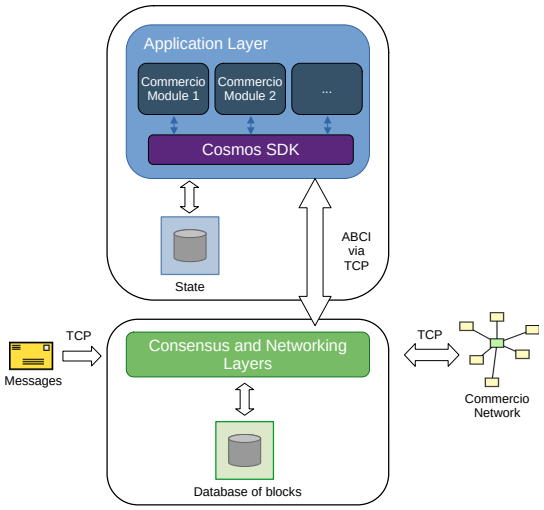

Figure 1. Commercio.network architecture.

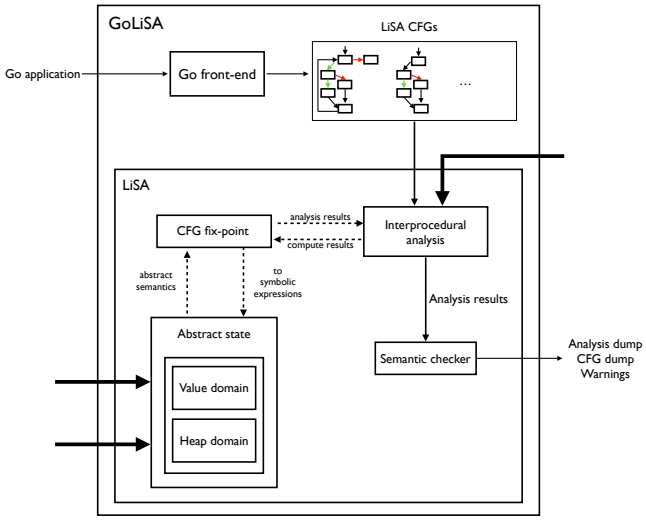

Figure 2. GoLiSA execution.

analyzers. 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. 2. The analysis starts with the Go front-end translating Go source code into LiSA CFGs that are then passed to LiSA, which analyzes them in a generic language-independent way. In particular, CFGs are passed to an *interprocedural analysis*, managing fixpoint computations. This component relies on a *call graph* to resolve calls and compute CFGs results. Each individual fixpoint relies on language-specific analysis-independent semantics for CFG nodes, which 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. Symbolic expressions are processed and analyzed by an *abstract state* [8], consisting of a heap domain (analyzing dynamic memory) and a value domain (analyzing program variables and memory locations). The abstract state computes a sound over-approximation of the symbolic 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.

4.1 How GoLiSA Ensures Determinism

In order to ensure determinism, GoLiSA provides two different checkers for Cosmos SDK: one based on a syntactic approach, and the other based on a semantic approach. In the former case, a syntax checker analyzes the program statements by basing its checks purely on the syntax of a program (e.g., if a certain Go statement appears in the source code, if a certain function/method is declared, etc.). Part of determinism is ensured by GoLiSA's syntactic checkers, performing a pattern matching on the program of interest to detect signatures of methods and statements that compromise determinism. In the second case, instead, a semantic checker analyzes the program statements inferring information and relationships between the components, then giving meaning (semantic) to the various parts of the program. In particular, it is necessary to specify an abstract state [8], that consist of a combination between an abstract domain that models the dynamic memory (the heap domain) and an abstract domain that tracks (abstract) values of program variables and dynamic memory locations (the value domain). In this paper, the analysis module of GoLiSA is instantiated as follows: non-interference analysis [10, 11] as value domain, tracking the integrity level (low or high) of each program variable or memory location, and monolithic heap analysis as heap domain, abstracting any concrete memory location allocated in a single abstract location.

The non-interference analysis implemented in GoLiSA treats sources of non-determinism with a low-integrity level and any blockchain state modifier or response to a transaction request with a high-integrity level. At this point, determinism is ensured by GoLiSA thanks to its semantic checkers that, *(i)* performs a syntactic analysis to extract the components which might bring non-deterministic behaviors, *(ii)* executes the aforementioned semantic analysis, and *(iii)* checks the absence of (implicit or explicit) flows from low-integrity level variables (i.e., sources of non-determinism) to high-integrity level components (i.e., blockchain state modifier or responses to transaction requests) within the program. Note that, a single non-deterministic component does not necessarily lead to non-determinism issues but it depends on whether it affects the global state of the blockchain or compromises the response to a transaction request. For this reason, it is necessary to track the flows of non-deterministic

data in the program, in order to ensure that they involve critical components.

5 Bug Fixing in Blockchain

Considering the software development workflow, the cost of detecting and fixing software defects increases exponentially with time. Fixing bugs and running code in a blockchain simulation is as easy as using the available tools for the underlying platform. Instead, when dealing with software that runs in the nodes of a live public blockchain, patching the bugs becomes critical and recovering the caused problems even more difficult. When it comes to blockchain, this factor gets more problematic by the lack of best practices and standard architectures [2, 13, 16].

In the case of *permissionless* blockchain, when a bug gets found and faults happen because of it, the caused problems are generally *immutable* and cannot be fixed. This immutability is achieved through a consensus mechanism that makes it difficult, or impossible, to withdraw transactions from the blockchain. In practice, the network majority should agree on rolling back to the state before an incident, effectively rewriting the history of the blockchain. However, the more a blockchain network increases in popularity, the more it is tricky to undertake such a turnaround. Part of the network may be interested in ignoring to resolve the consequences of a fault and continue as if nothing happened, leading to an independent blockchain, also known as a *hard fork*. For instance, this happened currently only once since the birth of the Ethereum blockchain because of the DAO attack [15].

In the case of *permissioned* blockchains, such as Commercio.network, it is possible to patch a buggy code through network governance. Typically, there is a limited subset of peers, with the power to propose a plan for halting, modifying, and restarting the blockchain with updated software, carefully migrating the state of the previous version. This kind of blockchain solution is often adopted in the industrial field, especially for enterprise or consortium blockchains. However, enforcing an update leads to a stop of services and data management problems. To reduce these problems, the Cosmos SDK offers an automatic process to apply blockchain upgrades, improving the synergy between the on-chain module `upgrade`, responsible for halting the chain, and an off-chain daemon capable of installing a new binary of the node software at the right time and autonomously restarting the node. Commercio.network reports that it is going to adopt this strategy, improving maintenance costs and reducing downtime for the consortium of validators.

In any case, it is of substantial importance to detect any kind of problem as soon as possible and, above all, before the software is used to run nodes of a blockchain network.

6 Limits of the Cosmos SDK Toolbelt

A toolbelt is a set of applications useful for code development and software maintenance.

For the Cosmos SDK, there is a limited number of tools tailored to the framework. For instance, *Ignite CLI*[8] (formerly known as Starport) is the most popular platform to build, launch, and maintain blockchain applications based on Cosmos SDK. Although it facilitates software development, nevertheless it has limitations. One of its most used features is to start a blockchain node in development with live reloading, i.e., when Ignite CLI detects that the source code of a Cosmos application has changed, it restarts the build process and then it launches a network using the updated software. However, at the time of writing, developers using it cannot observe non-determinism problems while testing the application. In fact, the execution happens on a single node network and therefore the underlying consensus mechanism will never conflict as a cause of non-deterministic executions. Hence, some kinds of issues related to non-determinism are really difficult to be detected during the development phase, since there is only one participant. Still, it is possible to use the Go toolbelt for testing a Cosmos SDK application. But this leads to limitations in the development, since testing and verification phase have not been designed ad hoc for blockchain frameworks.

As reported by the company, determinism is hard to ensure also after a complete test cycle, without the help of formal verification. Moreover, the complete test cycle is expensive in terms of resources and requires tools able to simulate the blockchain consensus. First, testing happens on a local-net. That is, a lightweight network running in a sandbox environment, destroyed and relaunched before each test. Commercio.network relies on a containerized local-net solution. Then, at least one network exposed to the public is required. These are fully-fledged networks, mirroring the functionalities of the current main network version (test-net) or featuring experimental features (dev-net). Similarly to a main-net, these networks should be composed of a diversified ecosystem of devices with a significant amount of nodes, different operative systems, system settings, geolocations, and so forth. However, also this level of testing does not guarantee the detection of faults, because it is not a sound procedure. If it is based on a limited number of transactions, then the problems might exist but remain undetected, since the conditions required to spot them have not been reached.

About analysis and verification tools, it is possible to start verifying Cosmos SDK applications with tools for the Go language, to score the code quality and detect issues. For instance, the Commercio.network company performs different iterations with Go analyzers on its code and publicly shares the results with the community using a Go Report Card[9]. However, as reported by the Commercio.network company,

[8] https://github.com/ignite-hq/cli

[9] https://goreportcard.com/report/github.com/commercionetwork/commercionetwork

Table 2. Analysis evaluation

# Affected	# Unaffected	Exec. time	Avg time per file	# Bugs
2 files	246 files	54m 27s	13.17s	2

the code coverage reached by dynamic testing is limited because the standard Go test suite has not been designed for blockchain development. Nevertheless, the advancement of coverage gets publicly reported, too[10].

We should recall that bugs in the testing tools exist, too, along with incorrect test design and malformed testware. Incomplete or incorrect sets of test cases that do not fail, displaying the green *OK* flag, add a false sense of trust to the programmer, which could let their guard down. Therefore, it is important to apply formal verification to detect problems since the early stages of implementation. Indeed, these tools are not enough to guarantee the safety of blockchain software because, while analyzing generic properties for Go, they do not take into account the particularities of blockchain development nor specifically the problems related to the *Cosmos SDK* framework. Hence, the need to develop customized tools for the framework of interest, as in our case the analysis on determinism with GoLiSA.

7 Code Evaluation

All the evaluations have been 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 memory running Windows 10 Pro 64bit, Oracle JDK version 13, and Go version 1.17. Our target application is Commercio.network v2.2.0[11], and in particular the evaluation is performed on the 248 Go files (14961 LoCs) contained in the repository.

7.1 The Analysis Choice

When carrying out an analysis it is necessary to take into account not only the properties it investigates but also the quality of its results and how these can impact on a possible bug fixing phase. As described in Sect. 4.1, the GoLiSA analyzer provides two different kinds of checkers.

The syntactic checkers are generally much faster than semantic ones because they do not perform heavy computations and reasoning about the code. However, by not thinking about semantics, they cannot correctly distinguish one program behavior from another. Hence, in the case of determinism, they indiscriminately deny all the black-listed signatures, generating several false positives alerts. During the patching phase, this involves a considerable expenditure of human effort in terms of time and capacity to filter the true positive alerts over the bug fixing. Manually filtering the results of the analysis could take several hours for each single alert.

The semantic checkers are more time-consuming in terms of analysis, but their cost is amortized considerably in the patching phase. In fact, they reason about the semantics of the program, discarding the parts of code that semantically are not relevant for the properties of analysis. Hence, it is possible to automatically reduce the number of false positives, thus limiting human effort. For these reasons, in the next section, the bug investigation will present results only with a semantic checker.

7.2 Bug Discussion

The result of the analysis performed by GoLiSA highlighted two problems regarding the same insidious issue (Table 2). Something similar affected an older version of the Cosmos SDK, which was reported by the NIST database as *"vulnerable to a consensus halt due to non-deterministic behaviour"* [14] caused by the use of the local clock time, obtained with the Go library `time`.

Bug #1. The bug appears in the `keeper` package of the module `commerciokyc`, in the method `Membership`. It is located at line 89 of file `keeper.go`[12]. In a nutshell, the method allows assigning a Commercio.network *membership* of the given type to the specified user. As shown in Fig. 3, the issue involves two main components: the method `time.Now` and the return of an error wrapped error. The first is a standard API of the Go language providing the current time of the device on which it runs. The second returns an error (wrapped with the `Wrap` method of the Cosmos SDK library[13]) to the method caller, leading to a transaction failure. Transaction executions among nodes must return a common result to achieve an update of the global status of the blockchain through the consensus mechanism. However, the current time provided by `time.Now` could be different from device to device because of custom settings (e.g. unsynchronized time, different time zones, etc.). Then, the same code execution on the nodes of blockchain network could result in different values, breaking the consensus.

An invocation to the buggy method may return an error or not depending on the result of the following guard:

```
expited_at.Before(time.Now())
```

Inspecting the invocations of `AssignMembership`, it can be found that the input variable `expited_at` is a timestamp computed with a support function that adds one year to the block time. Nodes with the current local time set to a timestamp that makes the guard evaluate to true (a timestamp bigger than `expited_at` is enough) will mark transactions invoking this code as failing since it returns an error. If the majority of nodes behave in this way, a denial of service may occur and block the assignment of new memberships.

[10] https://app.codecov.io/gh/commercionetwork

[11] https://github.com/commercionetwork/commercionetwork/tree/v2.2.0

[12] Source code available at: https://github.com/commercionetwork/commercionetwork/blob/3e02d5e761eab3729ccf6f874d3c929342e4230c/x/commerciokyc/keeper/keeper.go#L89

[13] https://docs.cosmos.network/master/building-modules/errors.html

```
func (k Keeper) AssignMembership(ctx sdk.Context, /*[...]*/,
      expited_at time.Time) error {

  /*  [...]  */

  if expited_at.Before(time.Now()) {
    return sdkErr.Wrap(sdkErr.ErrUnknownRequest, fmt.Sprintf(
        "Invalid expiry date: %s", expited_at))
  }

  /*  [...]  */
}
```

Figure 3. Bug #1 A snippet of the `AssignMembership` method not ensuring determinism in the `commerciokyc` module.

However, in this case, the blockchain is not compromised only if malicious actors control the majority of the nodes, but the problem is due to a code bug during software development that do not allow to reach a majority on the blockchain with the same result. Because of these reasons, `AssignMembership` does not ensure determinism, and its invocation might break the consensus mechanism.

Bug #2. The bug appears in the `keeper` package of the `commerciomint` module, in the method `BurnCCC`. It is located at line 174 of file `keeper.go`[14]. In a nutshell, this method allows burning (i.e., removing) an amount of currency to the conversion rate stored in a *position*, retrievable from the keeper's store with a user account address and an id. If successful, `BurnCCC` gives back to the user the collateral amount, then updates or deletes the considered position but only if enough time, called *freeze period*, has passed since its creation. Similarly to Bug #1, as shown in Fig. 4, also this issue involves two main components: the non-deterministic method `time.Now` and the return of a wrapped error.

An invocation to the buggy method may return an error or not depending on the result of the following guard:

```
time.Now().Sub(pos.CreatedAt) <= freezePeriod
```

The `pos` variable represents a position stored in the module's keeper and it is used to read its self-documenting field `CreatedAt`. The timestamp `freezePeriod` is read from the store of the module, too. Nodes with the current local time set to a timestamp that makes the guard evaluate to true (a timestamp in the past is enough) will mark transactions invoking this code as failing since it returns an error. If the majority of nodes behave in this way, a denial of service may occur and block the redemption of funds in the positions. This is not caused by malicious actors but is due to a code bug during software development, similarly to the previous case. Because of these reasons, `BurnCCC` doesn't ensure determinism, and its invocation might break the consensus mechanism.

Bug patching. After a deep investigation, the company reports that no incidents or transaction failures happened because of these bugs during the live period of the release

[14]Source code available at: https://github.com/commercionetwork/commercionetwork/blob/3e02d5e761eab3729ccf6f874d3c929342e4230c/x/commerciomint/keeper/keeper.go#L174

```
func (k Keeper) BurnCCC(ctx sdk.Context, user sdk.AccAddress,
      id string, burnAmount sdk.Coin) error {

  pos, found := k.GetPosition(ctx, user, id)
  if !found { /*  [...]  */ }

  // Control if position is almost in freezing period
  freezePeriod := k.GetFreezePeriod(ctx)
  if time.Now().Sub(pos.CreatedAt) <= freezePeriod {
    return sdkErr.Wrap(sdkErr.ErrInvalidRequest, "cannot burn
        position yet in the freeze period")
  }

  /*  [...]  */
}
```

Figure 4. Bug #2 A snippet of the `BurnCCC` method not ensuring determinism in the `commerciomint` module.

V2.2.0. Both bugs were patched in the major release v3.0.0. Then, we performed the analysis on this release and we could not find problems. The issue has been resolved by getting the time directly from the current Tendermint block header, a source that is both deterministic and supported by consensus. More in detail, the Cosmos SDK context method `ctx.BlockTime()` has been used instead of `time.Now()` when the current time was needed.

Dynamic Testing Considerations. The packages containing the bugs were tested with the standard Go testing framework and the libraries supported by Cosmos SDK, obtaining a satisfying level of code coverage. In particular, for the `keeper` packages of `commerciomint` and `commerciokyc` at version v2.2.0 test coverage is respectively 83.9% and 91.9%. However, both defects could not be detected by the test cases, due to incorrect initialization of testware. First, the Cosmos SDK blockchain `Context` passed to the keeper methods is set to the current time, with the invocation of the instructions `WithBlockTime(time.Now())`. Also, a different usage of `time.Now` has been leveraged for the initialization of testing variables and struct fields regarding time. These are definitely some testing anti-patterns, since not only blockchain code involved in consensus but also tests should be deterministic. Therefore, it is recommended to use a fixed timestamp in tests, wherever some logic depends on the use of time.

8 Conclusion

Blockchain technology has been considered mature enough to bring blockchain-oriented software even for enterprise realities. General-purpose languages, such as Go, increase productivity and reduce development costs. Companies adopt them for developing DApps. Still, blockchain code needs to be verified to improve its quality and reduce almost irreversible incidents. Available tools often lack controls to detect specific problems in blockchain frameworks, such as ensuring determinism. In this paper, we have shown through GoLiSA and the implementation of semantic checkers how it is possible to detect bugs of industrial code that break the deterministic execution of the blockchain, reducing the human effort in the phase of bug fixing and maintenance.

References

[1] Christopher Allen. 2016. The Path to Self-Sovereign Identity. http://www.lifewithalacrity.com/2016/04/the-path-to-self-soverereign-identity.html Accessed: March 2, 2022.

[2] Amiangshu Bosu, Anindya Iqbal, Rifat Shahriyar, and Partha Chakraborty. 2019. Understanding the motivations, challenges and needs of Blockchain software developers: a survey. *Empir. Softw. Eng.* 24, 4 (2019), 2636–2673. https://doi.org/10.1007/s10664-019-09708-7

[3] Ethan Buchman. 2016. *Tendermint: Byzantine fault tolerance in the age of blockchains.* Ph.D. Dissertation. University of Guelph.

[4] Commercio.network. 2022. Commercio.network - White Paper. https://commercio.network/project/ Accessed: March 9 2022.

[5] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. https://doi.org/10.1145/512950.512973

[6] Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen (Eds.). ACM Press, 269–282. https://doi.org/10.1145/567752.567778

[7] Jack B. Dennis and Earl C. Van Horn. 1966. Programming semantics for multiprogrammed computations. *Commun. ACM* 9, 3 (1966), 143–155. https://doi.org/10.1145/365230.365252

[8] Pietro Ferrara. 2016. A generic framework for heap and value analyses of object-oriented programming languages. *Theor. Comput. Sci.* 631 (2016), 43–72. https://doi.org/10.1016/j.tcs.2016.04.001

[9] Pietro Ferrara, Luca Negrini, Vincenzo Arceri, and Agostino Cortesi. 2021. Static analysis for dummies: experiencing LiSA. In *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*, Lisa Nguyen Quang Do and Caterina Urban (Eds.). ACM, 1–6. https://doi.org/10.1145/3460946.3464316

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

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

[12] J. Kwon and E. Buchman. 2019. Cosmos whitepaper. https://v1.cosmos.network/resources/whitepaper Accessed: February 18, 2022.

[13] Maaruf Ali Mahdi H. Miraz. 2020. Blockchain Enabled Smart Contract Based Applications: Deficiencies with the Software Development Life Cycle Models. *Baltica Journal* 33 (2020), 101–116.

[14] NIST. 2021. CVE-2021-41135 Detail. https://nvd.nist.gov/vuln/detail/CVE-2021-41135 Accessed: Febrauary 20, 2022.

[15] Nathaniel Popper. 2016. A Hacking of More Than $50 Million Dashes Hopes in the World of Virtual Currency. *The New York Times* (2016). https://www.nytimes.com/2016/06/18/business/dealbook/hacker-may-have-removed-more-than-50-million-from-experimental-cybercurrency-project.html Accessed: March 3, 2022.

[16] Simone Porru, Andrea Pinna, Michele Marchesi, and Roberto Tonelli. 2017. Blockchain-Oriented Software Engineering: Challenges and New Directions. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 169–171. https://doi.org/10.1109/ICSE-C.2017.142

[17] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. (2014). https://ethereum.github.io/yellowpaper/paper.pdf Accessed: March 7, 2022.