Don't Panic: Error Handling Patterns in Go Smart Contracts and Blockchain Software

Luca Olivieri
Ca'Foscari University of Venice
Venice, Italy
luca.olivieri@unive.it
0000-0001-8074-8980

Luca Negrini
Ca'Foscari University of Venice
Venice, Italy
luca.negrini@unive.it
0000-0001-9930-8854

***Abstract*—Issues in error handling may have critical consequences in blockchain software, ranging from silent execution with invalid states to denial of services due to unexpected crashes. This paper discusses the pitfalls of errors handling within blockchain frameworks written in Go such as Hyperledger Fabric, Tendermint Core (including its derivatives, e.g. CometBFT, Ignite), and other frameworks (e.g. Cosmos SDK), as well as the Ethereum implementation. Then, it explores how a static analysis approach can be applied for the automatic detection of such of issues, allowing to fix buggy code before deployment, i.e., when the code becomes difficult to patch being blockchain a trustless, distributed, and decentralized environment. Finally, we evaluate our analysis implementation within GoLiSA on a set of existing smart contracts and blockchain applications, empirically demonstrating the feasibility of the proposed approach.**

***Index Terms*—Blockchain, Smart Contracts, Error Handling Exception Handling, Program Verification, Static Analysis, Golang, Hyperledger Fabric, Chaincode, Cosmos SDK, Tendermint Core, Ignite, CometBFT, Ethereum**

I. Introduction

Error handling is a fundamental aspect of software development, and should be considered as important as any other core component of the codebase. Depending on the programming language, errors can manifest in a number of ways: exceptions (handled through try-catch blocks), error codes (that must be manually checked), and many more. Go's approach to error handling is distinct from that of many other languages and may lead to subtle issues if not managed carefully. Specifically, Go functions might return a special value of type `error` which indicates that an error has occurred. Go developers thus need to manually check the non-nullity of error values returned by functions, making error-checking explicit. Furthermore, Go provides also the primitive `panic`, which halt the normal control flow when typically an unexpected condition occurs. In conjunction, the `recover` primitive can be employed within deferred functions to intercept the panic, check and handle the exceptional state, and subsequently restore normal execution. However, such checking are optional: no compile-time check is performed on the control flow of the function, and the choice of whether to handle the error or not is left to the developer.

Blockchains are systems where correctness, reliability, and security are crucial. Several blockchain frameworks rely on the Go programming language, even though it was not specifically designed for blockchain purposes [43]. This makes it essential to adopt automatic verification techniques, such as static analysis, to detect potential issues during the early stages of development. In this paper, we focus on the detection of error handling issues in blockchain software written in Go, as (i) functions interacting with the blockchain often return an error value that must be taken into account when developing smart contracts, and (ii) both Go and blockchain software have guidelines on how to handle errors, which must be enforced to avoid critical issues. This paper makes the following contributions:

- the identification of critical vulnerabilities due to error handling issues in the blockchain software written in Go;
- the design and implementation of a static analysis tailored to detecting error handling issues in Go smart contracts;
- the design and implementation of a static analysis to detect panic executions within critical components, also considering *defer-and-recover* mechanisms, in blockchain software written in Go;
- an experimental evaluation on Hyperledger Fabric (HF) smart contracts and Cosmos's applications to benchmark the analysis and to understand the impact of error handling issues on existing code.

The analysis has been implemented by extending GoLiSA, a semantic-based static analyzer for Go programs, based on the library LiSA [13], [41], [42] , supporting also the verification of blockchain software [44]–[47], [49].

Paper Structure: Section II investigates issues related to error handling in blockchain software written in Go. Section III deals with the design and implementation details for the detection of error handling issues. Section IV proposes an experimental evaluation of the detection of errors handling issues in existing blockchain software written in Go. Section V discusses related work. Section VI concludes the paper.

II. Errors Handling Issues in Go Blockchains

In blockchain, error handling issues are not simply minor problems or code smells to be fixed. They may give rise to exploitable vulnerabilities that lead to critical consequences:

- *State inconsistencies and data corruption*: Blockchain can store inconsistent ledger state data, when errors of write/commit operations in blockchain are not properly

checked. Over time, this can result in unexpected behaviors, where malicious users can exploit them to gain an unfair advantage and performs attacks. For instance, if an error is not checked during a token exchange, the outcome could be a double-spending attack [6].

- *Consensus disruption*: Improper and missing error handling during the consensus phase (e.g., block validation, transaction verification and execution) can cause blockchain peers to diverge in the result, leading to a lack of majority and other non-trivial consequences (e.g., forks and liveness failures). Furthermore, malicious actors could exploit such weaknesses by sending ad-hoc transactions to trigger unhandled errors and then force honest peers to be out of sync, or be subject to penalties from the consensus algorithm.
- *Smart contract misbehavior*: Correct error management within smart contracts is of fundamental importance. Indeed, if not handled properly, it may have several consequences ranging from lost funds, locked contracts, unmet preconditions or security vulnerabilities (e.g., failed external calls, unexpected reentrancy, numerical overflows) to disrupt the expected application/business logic of the contract, causing it to behave in ways that contradict intended contract purpose [58].
- *Reduced auditability and incident response*: If errors are not properly handled, logs may fail to capture critical events and relevant information. This not only affects the search for the causes of failures, but also has security implications because it does not allow forensic analysis after an attack, leaving systems more vulnerable to repeated exploitation.

In Go, the philosophy of error handling is to encourage explicit and consistent checks, in favor of clarity over abstraction, and control over convenience. Unlike other programming languages that rely on exceptions and intricate try-catch constructs, errors in Go are *values* [51]. This design allows errors to be explicitly inspected and handled, placing the responsibility for error management on the developers. Furthermore, in situations where an error is deemed irremediable such that program execution cannot reasonably continue, Go provides the built-in `panic` function [18], that triggers a run-time error that causes the halting of the program execution, unless explicitly recovered with a *defer-and-recover* mechanism [15].

However, as reported in the official Go Developer Survey 2024 H1 [17], error handling is the biggest challenge faced by Go developers, due to tedious and repetitive implementation of error checks, especially when deeply-nested structures are involved. Not surprisingly, missing or improper error handling are also common software vulnerabilities, frequently highlighted in leading security projects and standards (see, for instance, like OWASP [11], [50] and CWE [34]–[38]).

In Go, the verbosity required for explicit error handling not only increases cognitive overhead but also makes the code more error-prone. As an example, consider the code in Figure 1, which reports the typical Go error handling pattern. Function calls can return two values: the "normal" output,

```
1 data, err := ioutil.ReadFile("file.txt")
2 if err != nil {
3   panic("Error opening file!")
4 }
```

Fig. 1. Example of error handling in Go.

```
1 func (s *SmartContract) incorrectReadWrite(stub
      shim.ChaincodeStubInterface) peer.Response {
2   value, _ = stub.GetState("Alice")
3   stub.PutState("Bob", value)
4   return shim.Success([]byte("OK"))
5 }
```

Fig. 2. Unhandled errors in Hyperledger Fabric.

which is only valid if no error happened, and a possibly-`nil` error. The burden of ensuring that no error happens and that the normal output is valid is left to the developer, who must check the nullity of the error value and react accordingly. This however introduces new paths in the control-flow, which are often very similar and verbose. Developers may thus be inclined to deprioritize or altogether skip proper error handling in order to focus on core functionality, which further exacerbates the risk of undetected failures and unstable software behavior. Go is adopted by several popular blockchain frameworks such as Hyperledger Fabric [24], Ethereum (e.g. the most popular execution client *Geth* — Go-Ethereum — [3] and consensus client *Prysm* [29]), and Tendermint Core [64], including its derivatives (e.g. *CometBFT* [8], *Ignite* [25]) and frameworks (e.g. *Cosmos SDK* [63]).

Ensuring errors are correctly handled is thus pivotal. In the following, we partition error handling errors in two families.

A. Silent Running Despite Incorrect State or Data

As Go requires developers to explicitly handle errors, these do not automatically block or stop program execution. If an error is ignored, the program continues running silently, as if nothing went wrong, possibly with incorrect state or data. To gain a deeper understanding of the implications of silent running issue within blockchain systems, we examine a representative example of HF smart contracts. In HF, smart contracts can be developed without any restriction, fully using the features of Go (including standard APIs and third-party libraries) [43]. While this provides developers with a high degree of freedom, it also requires them to explicitly manage ledger interactions, such as invoking the functions `GetState` [1] and `PutState`[2] for reading from and writing to the ledger world state, respectively. However, these operations can return errors and fail to perform, leading to unexpected behaviors. Figure 2 illustrates a code snippet with a possible issue scenario. At line 2, `GetState` reads the value of `"Alice"` from the ledger world state and stores it into variable `value`, but

[1]Documentation of `GetState` available https://pkg.go.dev/github.com/hyperledger/fabric-chaincode-go/shim#ChaincodeStub.GetState.

[2]Documentation of `PutState` available https://pkg.go.dev/github.com/hyperledger/fabric-chaincode-go/shim#ChaincodeStub.PutState.

```
1  func (s *SmartContract) correctReadWrite(stub shim.
       ChaincodeStubInterface) peer.Response {
2    value, err := stub.GetState("Alice")
3    if err != nil {
4      return shim.Error(fmt.Sprintf("Failed to read
           state for Alice: %s", err.Error()))
5    }
6
7    if value == nil {
8      return shim.Error("No value found for Alice")
9    }
10
11   err = stub.PutState("Bob", value)
12   if err != nil {
13     return shim.Error(fmt.Sprintf("Failed to write
           state for Bob: %s", err.Error()))
14   }
15
16   return shim.Success([]byte("OK"))
17 }
```

Fig. 3. Possible solutions to the error handling issues of Figure 2.

discards any error thanks to the blank identifier _.[3] This can lead to unintended behavior when the key `"Alice"` is missing or an internal error occurs during the read operation. In such cases, variable `value` may be `nil` or contain an unexpected default value. However, since there is no check over errors, the contract is not halted: the execution continues, potentially with incorrect or unexpected values. At line 3, `PutState` writes this potentially invalid value to `"Bob"`. The silent failure of `GetState` might corrupt the ledger state by overwriting valid data with an empty or incorrect value. Furthermore, `PutState` might also fail to perform and return an error, but its value is not collected in any variable: it is thus not possible to ensure that the operation actually took place. Finally, at line 4, the function always returns a successful transaction response `shim.Success([]byte("OK"))` to the user client after the code execution. This masks failures, making it difficult to detect and recover from issues after deployment. Figure 3 provides a possible fix for the code snippet of Figure 2. Specifically, error values returned by `GetState` and `PutState` are collected in the `err` variable (lines 2 and 11), which is later checked for non-nullness (lines 3 and 12). If it is indeed non-`nil`, the execution terminates with an error (lines 4 and 13). Moreover, the non-erroneous value produced by `GetState` is checked for nullness (line 7), and if it is `nil`, the execution is halted with an error (line 8). Finally, the function returns a success response only if all operations are successful (line 16). Note that silently ignoring errors and continuing the execution with incorrect state or data may also affect critical components in Ethereum, Tendermint Core, and its derivatives, as real-world examples demonstrate [19]–[21].

B. Unexpected Blockchain Application Crashes

Go developers should be familiar with the proverb:

"Don't panic." — [16], [52]

It is a general guideline for writing robust software, that suggests avoiding the use of `panic` or using it only as a last resort

[3]Documentation of blank identifier available https://go.dev/ref/spec#Blank_identifier.

```
1  func BeginBlocker(ctx sdk.Context, k keeper.Keeper)
       {
2    panic("Oh, dear. There is a reason to panic.")
3  }
```

Fig. 4. Panic crash in Cosmos SDK.

in the event of fatal and irremediable errors. Furthermore, this guideline becomes a fundamental mandate in blockchain systems, where stability, reliability and security are essential. In Go smart contracts, `panic` is typically managed and does not affect other components of the node. This is acceptable for smart contracts since most underlying blockchains revert all state changes when execution fails [26]. For example, in HF, a `panic` in a smart contract execution will lead to halt and cause the transaction to be invalidated, but it will not compromise the overall stability of the system. It is however generally preferable to properly log[4] the issue and return an appropriate error response, ensuring better debuggability and more graceful error handling. However, according to several security guidelines provided by pratictioners and security assessments companies (e.g. [1], [4], [26], [27], [30], [53], [56]), the instruction `panic` can lead to very harmful consequences when incorrectly used for the implementation of the blockchain node and its application logic, such as for blockchain implementations like Tendermint Core, Ethereum and their derivatives. Furthermore, the official guidelines state that `panic`s should not be allowed to propagate beyond the package boundary [23].

To gain a clearer understanding of the consequences of panic execution at the application layer of a blockchain node, we examine a representative application module built with the Cosmos SDK and designed to run on Tendermint Core and derivatives. Figure 4 proposes a code snippet with a possible issue scenario, where a `panic` is called at line 2 within the method `BeginBlocker`. As reported in the Cosmos's documentation:

> *"`BeginBlocker` and `EndBlocker` are a way for module developers to add automatic execution of logic to their module. This is a powerful tool that should be used carefully, as complex automatic functions can slow down or* ***even halt the chain****."*
> *— [57]*

For this reason, we considered these methods as critical components. Furthermore, in the example, `panic` is not recovered by any *defer-and-recover* mechanism [15]: it will thus lead to the crash of the entire application layer of the blockchain node, causing a denial-of-service (DoS). In addition, according to [56], a `panic` that depends on data manipulated by the user (e.g., the payload of a transaction) can create other serious vulnerabilities. For instance, an attacker might crash only a subset of nodes, reducing the number of active participants to perform other kind of

[4]Logging functions that may result in a `panic` execution such as `log.Panic` should be avoided.

```
1  func panicRecovery(ctx sdk.Context) {
2    // detect if panic occurs or not
3    r := recover()
4
5    if r != nil {
6      // Do recovery actions
7      // ...
8      ctx.Logger().Error("Panic Recovered", "error",
           r)
9    }
10 }
11
12 func BeginBlocker(ctx sdk.Context, k keeper.Keeper)
       {
13   defer panicRecovery(ctx)
14   panic("Oh, dear. There is a reason to panic.")
15 }
```

Fig. 5. Recovery of panic crash in Cosmos SDK.

attacks (e.g., double spending, etc.) or to achieve the majority consensus to approve malicious governance proposals and manage the entire blockchain network. An attacker could also target individual validators, causing economic losses through *stake slashing* as well as reputational harm due to continuous attacks and long downtime. Figure 5 provides a possible fix for the code snippet of Figure 4. At line 13, the function `panicRecovery` is deferred, i.e., it is executed at the end of the `BeginBlocker` function, after the `panic` at line 14. The function `panicRecovery` contains a `recover()` call at line 3, that it is used to detect possible `panic`s. If one is detected, it is handled by performing custom recovery actions at lines 6 and 7, and the error is logged at line 8 instead of leading to a blockchain node crash. Note that unexpected panics may also affect critical components in Ethereum, Hyperledger Fabric (framework implementation), as real-world examples demonstrate [10], [40], [55], [59], [66].

III. Static Detection of Error Handling Issues

Static analysis allows one to verify programs without running them [54]. A key advantage of static analysis in blockchain development is its ability to detect critical issues *before* the code is deployed to a distributed and decentralized network, and *before* it becomes immutable and changes are costly or impossible. Furthermore, it can be automated and integrated into the development workflow, making it a fundamental step to proactively detect and address issues, such as those outlined in Section II.

A. Detection of Unhandled Errors

The first step in detecting unhandled errors is to identify blockchain functions that return critical errors. Typically, these functions are known a priori (e.g. `PutState`, `GetState` mentioned in SectionII-A) and can be easily detected (e.g., by matching the signature of the function or through annotations [12]): it is thus possible to traverse the program structure in search for calls to critical functions, and issue warnings if:

1) the returned values are not assigned to variables;
2) the returned error value is discarded;
3) the returned error value is not checked for nullness.

In the first two cases, the check is merely syntactic. If the function call is not inside on the right-hand side of an assignment, the analysis issues an *unassigned error* warning. If it is, but a blank identifier is used to ignore the error value, then a *discarded error* warning is built. Instead, the third case requires reasoning about the execution flow. To determine whether an error variable is properly checked, it is necessary to reason about both the program's semantics and its control flow, as this involves an understanding of how the value is propagated and used. Hence, by traversing the code starting from the assignment of an error variable, a warning for an *unchecked error* is triggered if no conditional checks are found, or if the error variable is overwritten before it is used. Note that, according to Go best practices, error checking should be performed within the function itself, so we limited our analysis to intra-procedural reasoning. However, if desired, it can be extended to an inter-procedural level by considering the information of a sound call graph during the analysis.

B. Detection of `panic` Execution within Critical Components

For detecting panic executions within critical components through static analysis, we leverage a program representation based on a set of CFGs and the computation of a sound call graph. The first step is to detect panic statements, and it can be easily done by traversing the program structure and holding all occurrences of function calls with the `panic` function signature. Next, it is necessary to determine whether a critical component can execute the `panic`. This involves analyzing the signature of the function in which the `panic` is contained, and checking if it matches a critical component (e.g., `BeginBlocker` mentioned in Section II-A). If the function does not belong to a critical component, the call graph must be traversed backwards, starting from the CFG containing the `panic` and inspecting its callers transitively. During this phase, the analysis also keeps track of defer-and-recovery mechanisms, considering also possible ones (e.g., defers with unknown targets not part of the codebase under analysis). Once the execution paths leading to critical panic scenarios are identified, the analysis can classify different behavioral patterns:

1) if none of the paths that reach the `panic` invoke any (even potential) defer-and-recover mechanisms, the analysis reports a *fatal panic* warning, indicating that any panic will result in a program halt;
2) if at least one path includes a (possible) defer-and-recover mechanism and another does not, the analysis issues a *partial panic recovery* warning, as the program might recover depending on the execution path;
3) if all panic paths are guarded by defer-and-recover mechanisms, but some of them are possible, a *possible panic recovery* warning is raised to indicate that the recovery is uncertain and requires further inspection.

IV. Experimental Evaluation

This section presents the results of the application of GoLiSA's analyses for the detection of issues reported in

Section II on a benchmark of existing HF smart contracts and Cosmos's applications written in Go. First, the analysis is evaluated from a quantitative point of view, proposing statistics on performances and classifying the warnings triggered by the analysis on the benchmark. Subsequently, the quality of the analysis results is evaluated on a representative subset of the benchmark. Experiments have been performed on a machine with a 16-Core 3.50 GHz CPU, 128 GB of RAM, 20 TB SSD, running Ubuntu 22.04.3 LTS, Open JDK version 21, and with 8 GBs of RAM allocated to the JVM. The version of GoLiSA used for the evaluation is available at: https://github.com/lisa-analyzer/go-lisa/tree/brains2025

The experimental evaluation can be reproduced using the following artifact: https://doi.org/10.5281/zenodo.17220346

A. Experimental Datasets

For our experiments, we considered the dataset presented in [45], [47], that we refer to as HF, consisting of 651 files ($\sim$ 16 739 Lines of Code) retrieved from public GitHub repositories by querying the keyword "`chaincode`" (i.e., HF smart contracts). To also include Cosmos SDK applications, we extended the dataset by adding files retrieved from public GitHub repositories that match the signature of critical Cosmos SDK methods (i.e., querying the keywords "`BeginBlocker(ctx sdk.Context, k keeper.Keeper)`"[5] and "`EndBlocker(ctx context.Context)`"[6]). The results were filtered to exclude duplicate files (detected through SHA256 checksum equality) and selecting Go files with those methods declarations. No additional filters were applied based on this result: we considered all files found in the crawled repositories to avoid bias in the analysis outcome. This led to the addition of 907 files ($\sim$135 289 Lines of Code), which we refer to as CS. In total, HF and CS are composed by 1 558 Go files. Note that, as Ethereum and Hyperledger Fabric framework/protocol implementations leave close to no space for customization by the developer, we did not consider them in our experiments.

B. Quantitative Evaluation

Using GoLiSA, we ran the *Unhandled Errors* analysis on HF and the *Panic Executions* on CS. Out of the 1 558 total Go files, GoLiSA successfully analyzed 1 419 programs ($\sim$91,1%), with failures due to language constructs not supported by the analyzer (e.g., some corner cases related to structure with inline fields initialization and others related to anonymous functions) and analysis timeouts (set to 10 minutes). The total execution time for the analyses (single thread) is $\sim$1 hour, with an average execution time of $\sim$2.35 seconds per program.

Table I shows the results of analyses implemented in GoLiSA over the experimental dataset. We denote by #TP, #FP, #FN the number of true positives, false positives, and false negatives among the raised warnings, respectively. Their classification was carried out through an in-depth manual investigation performed on all the programs in the experimental dataset.

TABLE I
ANALYSIS RESULTS OF GOLISA.

Analysis	Warning	#TP	#FP	#FN
Unhandled Errors	Discarded Error	230	0	0
	Unassigned Error	410	0	0
	Unchecked Error	33	5	0
Panic Executions	Fatal Panic	165	2	0
	Partial Panic Recovery	0	0	0
	Possible Panic Recovery	0	0	0

GoLiSA detected 230 true *discarded errors* on 95 files from HF, 410 true *unassigned errors* on 189 files from HF, and 33 true *unchecked errors* on 22 files from HF. Since the check for *discarded* and *unassigned errors* is merely syntactic, the analysis does not produce false positives for these warnings. Instead, the 5 false positives on *unchecked errors* are due to the choice of an intra-procedural analysis. As stated in Section III-A, Go best practice ask to check error immediately after their collection; then, if the program follow the practice, it is sufficient to carry out the intra-procedural analysis, but in those false positives, the error values are returned without any check and then they are checked in callers. The analysis can be easily extended to be inter-procedural, as it happens for the panic analysis, to avoid false positive warnings and provide a more precise analysis.

As for *panic executions*, GoLiSA detected 165 true *fatal panic* warnings on 102 files from CS, 0 *partial panic recovery* and *possible panic recovery* warnings. In this case, the 2 false positives are due to legitimate panics that drop and avoid the execution of unupdated application versions, a common practice also widespread for Ethereum clients [59].

C. Qualitative Evaluation

In this section, we present a qualitative evaluation of the analysis results, discussing a program from HF and one from CS that showcase the reasoning of our analyses.

1) Unexpected Blockchain State on FabCar: Figure 6 proposes a code snippet from *fabcar.go*, originating in a forked repository of *Hyperledger Fabric Samples*[7]. The FabCar application is a beginner-friendly example demonstrating how to build a basic blockchain application using the Fabric framework. It implements the logic of a car factory and serves as a tutorial for understanding key HF concepts such as smart contracts, peers, ordering service, and how applications interact with the ledger. However, FabCar lacked of several proper error handling when reading from and writing to the blockchain world state in official versions 1.2 through 1.4. This issue was addressed and resolved starting with release v2.2. While FabCar is a helpful learning resource, bugs or mismatches in an official example can lead to critical issues

[5]https://github.com/search?q=language%3AGo+Begin-Blocker%28ctx+sdk.Context%2C+k+keeper.Keeper%29&type=code. (Accessed: 27/05/2025)

[6]https://github.com/search?q=language%3AGo+End-Blocker%28ctx+context.Context%29&type=code. (Accessed: 27/05/2025)

[7]https://github.com/hyperledger/fabric-samples/blob/release-1.2/chaincode/fabcar/go/fabcar.go

```
1  // [...]
2  func (s *SmartContract) changeCarOwner(APIstub shim
       .ChaincodeStubInterface, args []string) sc.
       Response {
3    // [...]
4    carAsBytes, _ := APIstub.GetState(args[0])
5    // [...]
6    car.Owner = arg[1]
7    //[...]
8    APIstub.PutState(args[0], carAsBytes)
9    return shim.Success(nil)
10 }
11 // [...]
```

Fig. 6. The function `changeCarOwner` of *fabcar.go*

because typically used as a base without proper understanding or adaptation by practitioners.

Figure 6 presents a buggy portion of the FabCar contract. This code is shown for the sake of readability and compactness. Nevertheless, the full contract contains additional error handling issues, including problems in the car creation logic and other feature. Specifically, in the proposed scenario, the code allows to change the ownership of a car. However, the possible error produced by `GetState` is discarded: invalid data could be retrieved and may corrupt the blockchain state through `PutState`. Furthermore, the possible error returned by `PutState` itself is discareded: the write operation could silently fail without finalizing the change in ownership, while still returning a success transaction response.

Analyzing the code snippet of Figure 6, GoLiSA detects the execution of read and write operations at lines 4 and 8, respectively. Hence, in the first case, the analysis checks the assignment after the `GetState` and detects the blank identifier that discard the error value, issuing a *discarded error* warning at line 4. Then, the analysis checks the call to `PutState` and detects that the returned error value is not stored in a variable, producing a *unassigned error* warning at line 8.

2) Panic on Althea Cosmos Gravity Bridge: Figure 7 proposes the code of *abci.go* from *Althea Cosmos Gravity Bridge* [2], affected by a documented panic issue in the official Althea Cosmos Gravity Bridge's repository [28], leading to possible blockchain node crashes during *slashing* operations.

Analyzing *abci.go*, GoLiSA finds the panic execution at line 19. Then, it checks if the `panic(err)` statement is located in a critical component. Being `ValsetSlashing` not critical, GoLiSA performs a backward inspection of the function's callers. It first considers function `slashing`, that calls `ValsetSlashing` at line 8. It then follows the call chain transitively, reaching the call to `slashing` at line 4 of `EndBlocker`. Here, GoLiSA infers that `EndBlocker` is a critical component, thus determining that `panic(err)` is itself critical. Since the only `defer` encountered during traversal (line 14) does not call `recover()`, no *defer-and-recover* can happen. Thus, a warning is generated at line 19, indicating a *fatal panic* that can lead to a crash of the blockchain node.

```
1  // [...]
2  func EndBlocker(ctx sdk.Context, k keeper.Keeper) {
3    // [...]
4    slashing(ctx, k)
5  }
6  func slashing(ctx sdk.Context, k keeper.Keeper) {
7    // [...]
8    ValsetSlashing(ctx, k, params)
9    // [...]
10 }
11 func ValsetSlashing(ctx sdk.Context, k keeper.
       Keeper, params types.Params) {
12   // [...]
13   unbondingValIterator := k.StakingKeeper.
         ValidatorQueueIterator(ctx, blockTime,
         blockHeight)
14   defer unbondingValIterator.Close()
15   // [...]
16   for _, valAddr := range unbondingValidators.
         Addresses {
17     addr, err := sdk.ValAddressFromBech32(valAddr)
18     if err != nil {
19       panic(err)
20     }
21     // [...]
22   }
23   // [...]
24 }
25 // [...]
```

Fig. 7. The function `EndBlocker` of *abci.go*

V. Related Work

The design and implementation of verification tools is a non-trivial task [48]. For this reason, the Go language provides the `golang.org/x/tools` module[8], containing various basic APIs for static analysis of Go programs (e.g., code parser, abstract syntax trees builder). This module gave birth to several *linters*[9] such as *GoSec* [39] and *Staticcheck* [22] that provide mainly syntactic intra-procedural checks for the detection of generic issues and code smells, including simple error handling issues. However, these tools are generally not designed for blockchain frameworks and therefore require extensions to process them. For example, *GoSec* has been extended to support the Cosmos SDK by incorporating custom rules [5]. These SDK-specific rules help detect non-deterministic behaviours and the use of unsafe or insecure packages, which are critical concerns in blockchain-based applications where deterministic execution is essential for consensus. Similarly, Reviveˆ CC [7] is a specialized version of the static analyzer Revive [14], tailored specifically for analyzing HF smart contracts. Other tools are based on `golang.org/x/tools` module. Chaincode Analyzer [68], by Fujitsu from Hyperledger Labs, implements checks for HF smart contracts. Lv et al. [33] and Yamashita et al. [67] propose similar tools inspired by Chaincode Analyzer and Reviveˆ CC, but they cover more issues and more accurately. However, they are not publicly available. Compared with our approach, the main limitation of `golang.org/x/tools` (and of tools based on it) is that it does not support native components of formal methods (e.g., fixpoint algorithms, ab-

[8]Documentation available at https://cs.opensource.google/go/x/tools
[9]https://en.wikipedia.org/wiki/Lint_(software)

stract interpreters, satisfiability modulo theories (SMT) solver interfaces, etc.), making these tools inaccurate and unable to provide formal guarantees on their findings. Instead, the main technical differences are that these tools typically work on the *abstract syntax tree level* (AST), without computing the CFG, and also often using *static single-assignment* form (SSA) as intermediate representation. The available checks are mainly related to discarded and unassigned errors, and to panic detection without any identification of critical components and *defer-and-recover* mechanisms.

To the best of our knowledge, few tools employ formal methods on Go, and are mostly specialized on concurrency and parallelization issues. For instance, Veileborg et al. [65] describe a local abstract interpretation approach for detecting blocking errors on software with channel-based concurrency and multi-threading environments. Liu et al. [31] with GoCatch apply static analysis and SMT solvers during the computations. Instead, GFuzz [31] proposes a dynamic fuzzing approach. However, concurrency and parallelization are often avoided in blockchain software as they can also give rise to non-determinism problems and consequently consensus problems [47], [49]. Another notable tool is CodeQL, a semantic code analysis engine developed by GitHub. A dedicated query pack named `cosmos-sdk-codeql` [62] has been created for Cosmos-based projects. This pack includes seven targeted queries that identify common security vulnerabilities such as improper access control, unsafe data handling, and logic errors within Cosmos SDK applications. Another pack for CodeQL is also provided by Surmont et al. [61]. It provides queries for several common Cosmos SDK issues that are detected by investigating the AST and also a data flow graph (DFG). However, regarding error issues, the queries[10] currently support only the detection of panic executions without any additional checks for *defer-and-recover* mechanisms. To the best of our knowledge, GoLiSA is the first tool considering also *defer-and-recover* mechanisms during the analysis of blockchain software.

VI. Conclusion

In this study, we examined the pitfalls of error handling issues in Go related to blockchain context. Due to Go's explicit error handling style, developers often inadvertently ignore or mishandle errors, leading to bugs, data and state inconsistencies. Furthermore, we explore the issues related to panic executions within critical blockchain components that can compromise the reliability of the systems.

Our findings and proposed analyses highlight the importance of a consistent and proactive approach, such as static analysis, to detect these issues before code deployment and before that code becomes difficult to patch due to the distributed and decentralized nature of blockchain.

Future work may involve enhancing the proposed detection with deeper semantic understanding to verify more complex error handling issues, such as the detection of implicit panic executions that require advanced analysis. For instance, LiSA already implements numerical abstractions like Intervals [9] and Pentagons [32] that may be applied for the detection of division by zero and index out of bounds accesses, as well as the information flow analysis [46], [47] for the detection of panic statements dependent on external inputs (e.g., transaction payload). Finally, other analyses can be considered for implementation, such as those related to nullness verification [60].

[10]https://github.com/JasperSurmont/cosmos-sdk-codeql/blob/main/queries/beginendblockPanic.ql

Acknowledgement

Work partially supported by SERICS (PE00000014 - CUP H73C2200089001) and iNEST (ECS00000043 – CUP H43C22000540006) projects funded by PNRR NextGeneration EU.

References

[1] Akhtariev, R., Yukseloglu, A.: The Cosmos Security Handbook (2024), https://www.faulttolerant.xyz/2024-01-16-cosmos-security-1, Accessed: 06/2025

[2] Althea-net: Althea Cosmos Gravity Bridge - GitHub Repository (2024), https://github.com/althea-net/cosmos-gravity-bridge, Accessed: 06/2025

[3] go-ethereum Authors: go-ethereum (2025), https://geth.ethereum.org/, Accessed: 05/2025

[4] of Bits, T.: Building Secure Contracts - ABCI methods panic (2023), https://secure-contracts.com/not-so-smart-contracts/cosmos/abci_panic/index.html, Accessed: 05/2025

[5] Buchman, E.: gosec - Golang Security Checker for the CosmosSDK (2022), https://github.com/cosmos/gosec. Accessed 06/2025

[6] Chohan, U.W.: The double spending problem and cryptocurrencies. Available at SSRN 3090174 (2021). https://doi.org/10.2139/ssrn.3090174

[7] Chokkapu, S.: Revivecc (2021), https://github.com/sivachokkapu/revive-cc. Accessed 02/2025

[8] CometBFT: CometBFT Documentation (2025), https://docs.cometbft.com/v1.0/, Accessed: 05/2025

[9] Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. p. 238–252. POPL '77, Association for Computing Machinery, New York, NY, USA (1977). https://doi.org/10.1145/512950.512973, https://doi.org/10.1145/512950.512973

[10] Enyeart, D.: Hyperledger Fabric - Issue #5169: Panic in gossip/gossip/algo (2025), https://github.com/hyperledger/fabric/issues/5169, Accessed: 05/2025

[11] Ferragamo, J., et al.: Improper Error Handling (2025), OWASP Website. https://owasp.org/www-community/Improper_Error_Handling, Accessed 05/2025

[12] Ferrara, P., Negrini, L.: SARL: OO Framework Specification for Static Analysis. In: Christakis, M., Polikarpova, N., Duggirala, P.S., Schrammel, P. (eds.) Software Verification. pp. 3–20. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-63618-0_1

[13] Ferrara, P., Negrini, L., Arceri, V., Cortesi, A.: Static analysis for dummies: Experiencing lisa. In: Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis. p. 1–6. SOAP 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3460946.3464316, https://doi.org/10.1145/3460946.3464316

[14] Gechev, M.: Revive (2021), https://github.com/mgechev/revive. Accessed 02/2025

[15] Gerrand, A.: Defer, Panic, and Recover (2010), The Go Blog. https://go.dev/blog/defer-panic-and-recover, Accessed 05/2025

[16] Google: Go Wiki: Go Code Review Comments - Don't Panic, https://go.dev/wiki/CodeReviewComments#dont-panic, Accessed: 05/2025

[17] Google: Go developer survey 2024 h1 results (2021), The Go Blog. https://go.dev/blog/survey2024-h1-results, Accessed 05/2025

[18] Google: Effective Go - Panic (2025), https://go.dev/doc/effective_go#panic, Accessed 05/2025

[19] HelloBloc: Canine-chain - Issue #8: Need Error Handling for RNS's SendCoinsFromAccountToModule Function (2025), https://github.com/JackalLabs/canine-chain/issues/8, Accessed: 05/2025

[20] HelloBloc: Ignite CLI - issue #2828: Ignite Tutorial on Bankkeeper Use is Flawed (2025), https://github.com/ignite/cli/issues/2828, Accessed: 05/2025

[21] HelloBloc: Loan project - Issue #6: Need Error Handling for SendCoins Function, note=https://github.com/fadeev/loan/issues/6, accessed: 05/2025 (2025)

[22] Honnef, D.: Staticcheck website, https://staticcheck.io/, Accessed: 09/2022

[23] Hyperledger: General guidelines for error handling in Hyperledger Fabric (2023), https://hyperledger-fabric.readthedocs.io/en/release-2.5/error-handling.html#general-guidelines-for-error-handling-in-hyperledger-fabric, Accessed 05/2025

[24] Hyperledger: Hyperledger Fabric: A Blockchain Platform for the Enterprise (2025), https://hyperledger-fabric.readthedocs.io/en/release-2.5/#a-blockchain-platform-for-the-enterprise, Accessed: 05/2025

[25] Ignite: Ignite Documentation (2025), https://docs.ignite.com/welcome, Accessed: 05/2025

[26] James, W.: Cosmos Security: An Otter's Guide (2025), https://osec.io/blog/2025-06-10-cosmos-security, Accessed: 06/2025

[27] Jorgecastillot: Cosmos unmasked, a security guide to review cosmos application (2024), https://medium.com/@jorgecastillot2017/cosmos-unmasked-a-security-guide-to-review-cosmos-application-cfc9efbdd205, Accessed: 06/2025

[28] Kuprianov, A.: Endblocker panics may halt consensus engine #348 (2024), https://github.com/althea-net/cosmos-gravity-bridge/issues/348, Accessed: 06/2025

[29] Labs, O.: Prysm: An Ethereum Consensus Implementation Written in Go (2025), https://geth.ethereum.org/, Accessed: 05/2025

[30] Learn, G.: Cosmos Ecosystem Security Guide: Analyzing Security Challenges in Different Components (2024), https://www.zellic.io/blog/exploring-cosmos-a-security-primer/, Accessed: 06/2025

[31] Liu, Z., Xia, S., Liang, Y., Song, L., Hu, H.: Who goes first? detecting go concurrency bugs via message reordering. In: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. p. 888–902. ASPLOS '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3503222.3507753, https://doi.org/10.1145/3503222.3507753

[32] Logozzo, F., Fähndrich, M.: Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In: Proceedings of the 2008 ACM Symposium on Applied Computing. p. 184–188. SAC '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1363686.1363736, https://doi.org/10.1145/1363686.1363736

[33] Lv, P., Wang, Y., Wang, Y., Zhou, Q.: 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. pp. 1–7. IEEE (2021). https://doi.org/10.1109/ISCC53001.2021.9631249

[34] MITRE: CWE-248: Uncaught Exception (2025), https://cwe.mitre.org/data/definitions/248.html, Accessed 05/2025

[35] MITRE: CWE-252: Unchecked Return Value (2025), https://cwe.mitre.org/data/definitions/252.html, Accessed 05/2025

[36] MITRE: CWE-391: Unchecked Error Condition (2025), https://cwe.mitre.org/data/definitions/391.html, Accessed 05/2025

[37] MITRE: CWE-703: Improper Check or Handling of Exceptional Conditions (2025), https://cwe.mitre.org/data/definitions/703.html, Accessed 05/2025

[38] MITRE: CWE-755: Improper Handling of Exceptional Conditions (2025), https://cwe.mitre.org/data/definitions/755.html, Accessed 05/2025

[39] Murphy, G.: Secure go website, https://securego.io/, Accessed: 09/2022

[40] Nagai, T.: Hyperledger Fabric - Issue #5198: Creating a channel with the same name again causes orderer process to terminate abnormally (2025), https://github.com/hyperledger/fabric/issues/5198, Accessed: 05/2025

[41] Negrini, L., Arceri, V., Olivieri, L., Cortesi, A., Ferrara, P.: Teaching through practice: Advanced static analysis with lisa. In: Sekerinski, E., Ribeiro, L. (eds.) Formal Methods Teaching. pp. 43–57. Springer Nature Switzerland, Cham (2024). https://doi.org/10.1007/978-3-031-71379-8_3

[42] Negrini, L., Ferrara, P., Arceri, V., Cortesi, A.: LiSA: A Generic Framework for Multilanguage Static Analysis, pp. 19–42. Springer Nature Singapore, Singapore (2023). https://doi.org/10.1007/978-981-19-9601-6_2

[43] Olivieri, L., Arceri, V., Chachar, B., Negrini, L., Tagliaferro, F., Spoto, F., Ferrara, P., Cortesi, A.: General-purpose languages for blockchain smart contracts development: A comprehensive study. IEEE Access **12**, 166855–166869 (2024). https://doi.org/10.1109/ACCESS.2024.3495535

[44] Olivieri, L., Negrini, L., Arceri, V., Chachar, B., Ferrara, P., Cortesi, A.: Detection of phantom reads in hyperledger fabric. IEEE Access **12**, 80687–80697 (2024). https://doi.org/10.1109/ACCESS.2024.3410019

[45] Olivieri, L., Negrini, L., Arceri, V., Ferrara, P., Cortesi, A.: Detection of read-write issues in hyperledger fabric smart contracts. In: Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing. p. 329–337. SAC '25, Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3672608.3707721, https://doi.org/10.1145/3672608.3707721

[46] Olivieri, L., Negrini, L., Arceri, V., Ferrara, P., Cortesi, A., Spoto, F.: Static detection of untrusted cross-contract invocations in go smart contracts. In: Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing. p. 338–347. SAC '25, Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3672608.3707728, https://doi.org/10.1145/3672608.3707728

[47] Olivieri, L., Negrini, L., Arceri, V., Tagliaferro, F., Ferrara, P., Cortesi, A., Spoto, F.: Information flow analysis for detecting non-determinism in blockchain. In: Ali, K., Salvaneschi, G. (eds.) 37th European Conference on Object-Oriented Programming (ECOOP 2023). Leibniz International Proceedings in Informatics (LIPIcs), vol. 263, pp. 1–25. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2023). https://doi.org/10.4230/LIPIcs.ECOOP.2023.23, https://doi.org/10.4230/LIPIcs.ECOOP.2023.23

[48] Olivieri, L., Spoto, F.: Software verification challenges in the blockchain ecosystem. International Journal on Software Tools for Technology Transfer **26**(4), 431–444 (2024). https://doi.org/10.1007/s10009-024-00758-x

[49] Olivieri, L., Tagliaferro, F., Arceri, V., Ruaro, M., Negrini, L., Cortesi, A., Ferrara, P., Spoto, F., Talin, E.: 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. p. 23–29. SOAP 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3520313.3534658, https://doi.org/10.1145/3520313.3534658

[50] OWASP: Error Handling Cheat Sheet (2025), OWASP Website. https://cheatsheetseries.owasp.org/cheatsheets/Error_Handling_Cheat_Sheet.html, Accessed 05/2025

[51] Pike, R.: Errors are values (2015), The Go Blog. https://go.dev/blog/errors-are-values, Accessed 05/2025

[52] Proverbs, G.: Go Proverbs: Simple, Poetic, Pithy, https://go-proverbs.github.io, Accessed: 09/2022

[53] Rajvardhan: Exploring Cosmos: A Security Primer (2023), https://www.zellic.io/blog/exploring-cosmos-a-security-primer/, Accessed: 06/2025

[54] Rival, X., Yi, K.: Introduction to static analysis: an abstract interpretation perspective. Mit Press (2020)

[55] rjl493456442: Go-ethereum - Issue #30229: State corruption after unexpected terminations during snap sync (2024), https://github.com/ethereum/go-ethereum/issues/30229, Accessed: 05/2025

[56] Saigle, J.: Don't "Panic": How Improper Error-Handling Can Lead to Blockchain Hacks (2022), https://www.halborn.com/blog/post/dont-panic-how-improper-error-handling-can-lead-to-blockchain-hacks, Accessed: 05/2025

[57] SDK, C.: Cosmos SDK Documentation - BeginBlocker and EndBlocker (2025), https://docs.cosmos.network/v0.53/build/building-modules/beginblock-endblock#beginblocker-and-endblocker-1, Accessed: 05/2025

[58] Shabna Madathil Thattantavida, S.K.: Ibm developer - learn best practices for debugging and error handling in an enterprise-grade blockchain application (2023), https://developer.ibm.com/blogs/debugging-and-error-handling-best-practices-in-a-blockchain-application (Accessed 01/2024)

[59] Sikiric, D.M.: Go-ethereum - Issue #29425: Panic of geth (2024), https://github.com/ethereum/go-ethereum/issues/29425, Accessed: 05/2025

[60] Spoto, F.: Precise null-pointer analysis. Software & Systems Modeling **10**(2), 219–252 (2011), https://doi.org/10.1007/s10270-009-0132-5
[61] Surmont, J., Wang, W., Cutsem, T.V.: Static application security testing of consensus-critical code in the cosmos network. In: 2023 5th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS). pp. 1–8 (2023). https://doi.org/10.1109/BRAINS59668.2023.10316912
[62] Tauber, T.: cosmos-sdk-codeql - A query suite for common bug patterns in Cosmos SDK-based applications. (2023), https://github.com/crypto-com/cosmos-sdk-codeql. Accessed 06/2025
[63] Teams, I.C.: Cosmos SDK Documentation (2025), https://docs.cosmos.network/v0.53/learn/intro/overview, Accessed: 05/2025
[64] Tendermint: Tendermint Core Documentation (2025), https://docs.tendermint.com/v0.35/, Accessed: 05/2025
[65] Veileborg, O.H., Saioc, G.V., Møller, A.: Detecting blocking errors in go programs using localized abstract interpretation. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. ASE '22, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3551349.3561154, https://doi.org/10.1145/3551349.3561154
[66] Wood, S.: Hyperledger Fabric - Issue #5048: Panic on leveldb range query fabric peer (2024), https://github.com/hyperledger/fabric/issues/5048, Accessed: 05/2025
[67] Yamashita, K., Nomura, Y., Zhou, E., Pi, B., Jun, S.: Potential risks of hyperledger fabric smart contracts. In: 2019 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE). pp. 1–10 (2019). https://doi.org/10.1109/IWBOSE.2019.8666486
[68] Yamashita, K., Ry, J.: Chaincode Analyzer (2020), https://github.com/hyperledger-labs/chaincode-analyzer. Accessed 02/2025