Static Detection of Untrusted Cross-Contract Invocations in Go Smart Contracts

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

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

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

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

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

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

ABSTRACT

A blockchain is a trustless system in an environment populated by untrusted peers. Code deployed in blockchain as a smart contract should be cautious when invoking contracts of other peers as they might introduce several risks and unexpected issues. This paper presents an information flow-based approach for detecting cross-contract invocations to untrusted contracts, written in general-purpose languages, that could lead to arbitrary code executions and store any results coming from them. The analysis is implemented in GoLiSA, a static analyzer for Go. Our experimental results show that GoLiSA is able to detect all vulnerabilities related to untrusted cross-contract invocations on a significant benchmark suite of smart contracts written in Go for Hyperledger Fabric, an enterprise framework for blockchain solutions.

CCS CONCEPTS

• **Software and its engineering** → **Automated static analysis**; **Software verification**; **Formal software verification**.

KEYWORDS

Cross-contract Invocation, Delegate Call, External Contract Call, Static Analysis, Abstract Interpretation, Blockchain, Distributed ledger technology, Smart Contracts, CWE-829, SWC-112

ACM Reference Format:
Luca Olivieri, Luca Negrini, Vincenzo Arceri, Pietro Ferrara, Agostino Cortesi, and Fausto Spoto. 2025. Static Detection of Untrusted Cross-Contract Invocations in Go Smart Contracts. In *The 40th ACM/SIGAPP Symposium on Applied Computing (SAC '25), March 31-April 4, 2025, Catania, Italy.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3672608.3707728

SAC '25, March 31-April 4, 2025, Catania, Italy

ACM ISBN 979-8-4007-0629-5/25/03.
https://doi.org/10.1145/3672608.3707728

1 INTRODUCTION

In a blockchain, smart contracts can interact with other deployed code by using *Cross-Contract Invocations* (CCIs for short), namely delegate calls or external contract calls. This mechanism promotes interoperability by allowing cooperation and exchange of information and services within the blockchain. Furthermore, achieving smart contract interoperability is also a crucial point for international regulations such as the European Data Act [37, 38]. However, a naive implementation of CCIs could lead to *untrusted* invocations (i.e., CCI where the callee or parameters can be controlled by users) exposing the contracts to critical issues such as code injection and execution of arbitrary code. This might have severe consequences, ranging from loss of assets, cryptocurrencies, or more generally fungible and non-fungible tokens, to denial of service [3, 5].

The novel contribution of this paper is the design of a two-phase analysis that detects *Untrusted Cross-Contract Invocations* (UCCIs) by using information-flow techniques: *(i)* to detect flows from untrusted user inputs to cross-contract invocations, and *(ii)* to detect flows from untrusted cross-contract execution to blockchain storage. To the best of our knowledge, there are currently no analyses covering these issues for general-purpose languages, such as Go. Furthermore, we implemented and evaluated our approach in GoLiSA, a static analyzer based on abstract interpretation that supports the analysis of several blockchain frameworks written in Go, such as Hyperledger Fabric[1] (from now on HF), Cosmos SDK[2], and Tendermint Core (recently rebranded as Ignite[3]). The evaluation is performed on a benchmark suite of existing smart contracts retrieved from public GitHub repositories, and shows, empirically, that our approach can successfully identify UCCIs.

Paper structure. Section 2 provides an overview of UCCIs in blockchain software. Section 3 and Section 4 present the design of our core contribution for detecting issues related to UCCIs and its implementation in GoLiSA. Section 5 experimentally evaluates the proposed analysis implemented in

[1] https://www.hyperledger.org/use/fabric
[2] https://v1.cosmos.network/sdk
[3] https://ignt.com/

GoLiSA on a data set of existing smart contracts. Section 6 presents related works. Section 7 concludes the paper.

2 UNTRUSTED CROSS-CONTRACT INVOCATIONS

CCIs allow smart contracts to execute the code of other contracts deployed in blockchain, by calling specific instructions. In general, these instructions require two distinct parameters that can be hard-coded or parameterized into the calling contract: the contract to call (e.g., contract name, its address within the blockchain) and optional data to process (e.g., the method to execute, its parameters, and tokens to transfer). CCIs are a powerful feature that can be involved in different use cases such as:

- *Contract interactions*: the primary purpose of CCIs is to communicate with other contracts for exchanging data, assets, and cryptocurrencies, as is often the case when communicating between different decentralized autonomous organizations (DAOs);
- *Libraries*: CCIs allow to build libraries of shared code that multiple contracts can access, hence promoting code modularity and reducing complexity; this can also help decrease deployment costs when those are related to the contract's size;
- *Contract Size Limit*: some blockchains, such as Ethereum, impose a limit on the size in bytes of each smart contract [4], and those exceeding the limits are not allowed to be deployed; CCIs can be used to split a large contract into smaller ones that interact with each other, therefore overcoming this limitation;
- *Proxy Upgrade Pattern*: as the code of deployed smart contracts is immutable, the proxy pattern [29, 41] can be used to circumvent this limit, by allowing smart contracts to be upgraded to include new features and patches, *after* they have been deployed: by partitioning the business logic across several contracts and by making them communicate through a *proxy* contract, the application logic can be updated by specifying a different target address in CCIs.

Despite the benefits derived from the adoption of CCIs, their naive use can introduce UCCIs that a malicious agent can exploit to inject arbitrary values that can lead to untrusted code execution by the blockchain (see CWE-829 [10] and SWC-112 [48]), such as extortionware attacks [3, 5]. An untrusted use of CCIs happens when the contract to call is parameterized and directly depends on the program input (i.e., data from outside the blockchain) that, in general, is untrusted: users can provide it anonymously.

Consider for instance the attack schema depicted in Fig. 1. A blockchain user might naively deploy a contract containing a UCCI and use it to handle assets. After contract deployment, its source code will remain exposed in the blockchain. An attacker could discover the vulnerability of the contract and exploit it to take over the assets managed by that contract. Specifically, the attacker could redirect the CCI to his own malicious contract, in order to demand a ransom or permanently take possession of the stolen assets.

2.1 Towards UCCI Detection for General Purpose Languages

According to Olivieri et al. [32], general-purpose languages (GPLs), such as Go, are supported by several blockchain frameworks for the development of smart contracts. Although they do not enjoy the same popularity as domain-specific languages (e.g. Solidity [2] for Ethereum), they are widely applied in industrial solutions offering greater flexibility, extensive libraries, and better tooling for scalability and integration with existing enterprise systems, as well as to a lesser extent by reducing the learning curve for developers.

GPLs may involved in different ways in smart contract development and they can be classified in: *full*, *restricted*, and *meta-programming* [32] languages. In the first two cases, the smart contract code is written in a GPL and executed "as is" in the blockchain. The difference is that the *restricted* ones are limited to a subset of language functionalities or instructions of the GPL. In the third case, the GPL is used at a high-level but then compiled into a low-level domain-specific language for the target blockchain.

Currently, Go is mainly used as a full language leaving developers the freedom to use all its functionalities and instructions.

The adoption of full languages in smart contracts and blockchain framework represents a challenge for verification [32, 39]. In particular, they may not provide the same level of security constraints offered by domain-specific languages for blockchain (e.g. determinism, avoid wrap-around semantics, . . .). Moreover, the adoption of external libraries or frameworks may increase the complexity of the analysis and the resource consumption potentially affecting the precision of analysis results.

Moreover, blockchain frameworks that use such languages are often rely on specific APIs, such as for data storage in blockchain, for sending transaction responses, and for managing the blockchain components in a versatile way. CCIs fall in this category: invocations of functions provided by the framework lead to the execution of other contracts. Given the presence of such explicit calls, it is necessary to reason about two distinct program behaviors: *(i) cross-contract invocation from untrusted input* and *(ii) storage of data returned from untrusted cross-contract executions*.

Fig. 2 reports a snippet of a Go smart contract for HF, exemplifying an extortionware attack scenario. At line 2, the input to the transaction request is retrieved through the function `GetStringArgs`. Line 4 stores the first element of the input in variable `contract`, later used at line 8 as the receiver of a CCI with the arguments contained in `queryArgs`: the method to invoke and the asset to pass to the method. As the user controls variable `contract`, there is a security problem since the user can send execution requests to any deployed contract, including a contract whose implementation of `SetAssetOwnership` is not that expected by the

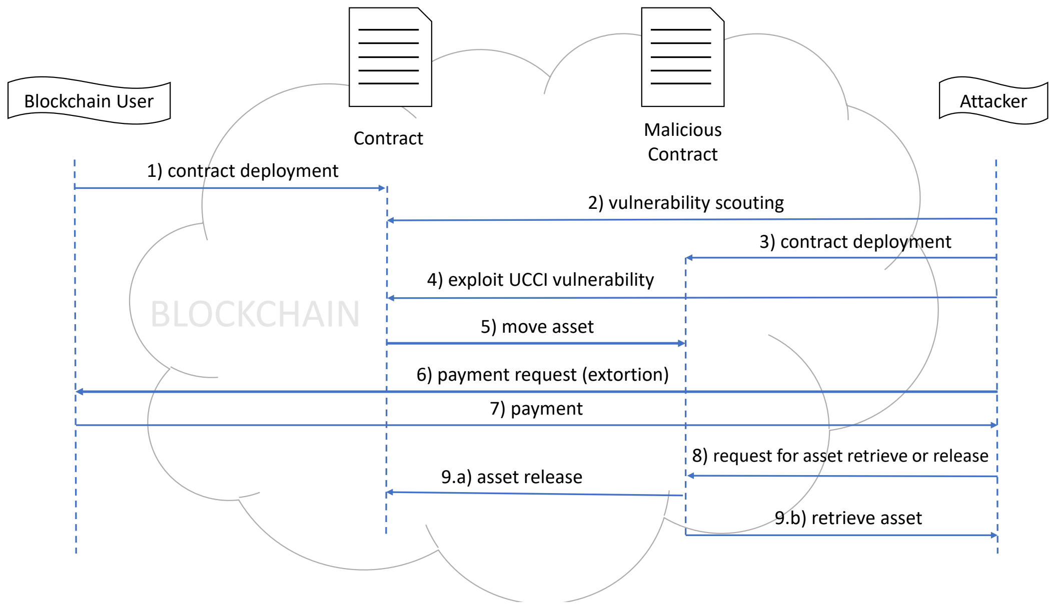

Figure 1: Extortionware attack model exploiting UCCIs [3, 5].

```
1 // Get the args from the transaction
2 args := stub.GetStringArgs()
3
4 contract := args[0]
5 // [...]
6 queryArgs[0] = "SetAssetOwnership"
7 queryArgs[1] = myasset
8 response := stub.InvokeChaincode(contract,
      queryArgs, "main-channel")
9 stub.PutState("owner", response.Payload)
```

Figure 2: Simplified smart contract for HF, featuring a UCCI.

developer of the snippet in Fig. 2. For instance, the injected `SetAssetOwnership` method could unexpectedly change the ownership of `myasset`. As the untrusted input can change the contract target of the CCI, this is an example of *cross-contract invocation from untrusted input*. Finally, at line 9, the execution result is retrieved from (`response.Payload`) and is stored in the blockchain through `PutState`, which allows one to perform a data-write proposal of blockchain global state, leading to a *blockchain data storage from untrusted cross-contract executions*. Note that, upon successful execution, the change in ownership that is stored through `PutState` becomes part of the blockchain global state.

3 UCCI DETECTION BY TAINT ANALYSIS

Taint analysis [8, Section 47.11.8] is an instance of information flow analysis that allows one to detect if untrusted information *explicitly* flows from some *source* to critical program points, called *sinks*. It means that one can logically split program variables into two sets: *tainted* variables $\mathbb{T}$, that is, those that an external attacker can tamper; and its dual set of *clean* variables $\mathbb{C}$. At the start of the analysis, $\mathbb{T}$ contains the *sources* only, that is, the variables that can be directly modified by the attacker. The analysis iteratively moves variables from $\mathbb{C}$ to $\mathbb{T}$ whenever one is assigned to a value computed by using at least a variable in $\mathbb{T}$, and from $\mathbb{T}$ to $\mathbb{C}$ whenever one is assigned to a value computed by using a sanitizer (that is, functions that vet the tainted values, therefore making sure that there is no potential influence in the result). Consequently, the analysis computes, for each program point, the set of variables containing values that can be controlled by the attacker. With such information, one can check if a *sink* receives a value computed by using at least a variable in $\mathbb{T}$, thus detecting potential security vulnerabilities. Taint analysis can be also applied with formal method frameworks to provide several guarantees. Among these, an important guarantee is *soundness*, i.e., the absence of *false negatives* for a given property, which can be achieved for instance by using abstract interpretation [7, 9] and over-approximating program semantics [8]. Moreover, this generic schema has been instantiated to detect many vulnerabilities in real-world software (e.g., SQL injection [11, 53], privacy issues [15, 17], IoT issues [28], non-determinism [36, 40], phantom reads [34]), achieving significant practical results (see [13] for an example).

Likewise, taint analysis can be applied to the detection of UCCIs. In this paper, we design an analysis composed of two phases to deal with the following taintedness problems:

- **Phase 1**: *detection of untrusted cross-contract invocations*. The analysis models input parameters given by users through transactions as sources, and the parameters of cross-contract calls specifying a contract as sinks; in this way, it is possible to trace arbitrary input values within a smart contract and check if there are flows that lead to cross-contract calls, possibly executing an arbitrary contract;

- **Phase 2**: *detection of untrusted blockchain storage from untrusted cross-contract executions.* The analysis models the cross-contract calls detected during phase 1 that received untrusted input arguments as sources, and the parameters of blockchain data-write proposal calls as sinks; in this way, it is possible to trace the results of untrusted executions within a smart contract and check if there are flows that lead to the immutable storage of this information through blockchain data-writes and transaction response proposals.

Algorithm 1 Detection of issues related to UCCIs.

```
1: procedure UCCIAnalysis(program, framework)
2:     alerts ← ∅
                                                          ▷ Phase 1
3:     sourcesP1 ← retrieveSourcesP1FromSignatures(program, framework)
4:     sinksP1 ← retrieveSinksP1FromSignatures(program, framework)
5:     if | sourcesP1 | > 0 ∧ | sinksP1 | > 0 then
6:         resP1 ← taint(program, sourcesP1, sinksP1)
7:         alerts ← getAlerts(resP1)
                                                          ▷ Phase 2
8:         sourcesP2 ← retrieveSourceP2FromTaintResultsP1(sinksP1, resP1)
9:         sinksP2 ← retrieveSinksP2FromSignatures(program, framework)
10:        if | sourcesP2 | > 0 ∧ | sinksP2 | > 0 then
11:            resP2 ← taint(program, sourcesP2, sinksP2)
12:            alerts ← alerts ∪ getAlerts(resP2)
13:    return alerts
```

Algorithm 1 shows the high-level structure of the proposed analysis. It requires only the `program` (i.e., the smart contract) to analyze and specify the blockchain `framework` (e.g., HF, Cosmos SDK, Tendermint Core, ...) on which it is based. The algorithm starts from phase 1 (lines $3-7$), by computing sources and sinks. Typically, the full list of signatures related to the methods for introducing arbitrary inputs (i.e., sources of phase 1) and of the CCIs (i.e., sinks of phase 1) are always known a priori and depend on the framework (e.g., see Table 1 for HF). Hence, at lines $3-4$, retrieveSourcesP1FromSignatures and retrieveSinksP1FromSignatures perform a signature matching on the program statements to retrieve those matching the signature list specific to the framework (they select only sources and sinks that do appear in the given program) and collect them in `sourceP1` and `sinksP1`, respectively. At this point, at lines $5-7$, if there is at least a source and a sink in the program, the algorithm runs a taint analysis to detect UCCIs and generates alerts from the taint analysis result `resP1`. Phase 2 (lines $8-12$) can start only after phase 1, because it requires its taint analysis information. Indeed, the sources (i.e., UCCIs) for phase 2 are not known a priori as they are computed at the end of phase 1 (line 6), i.e., they are the sinks of phase 1 into which a tainted value has flowed. Hence, at line 8, retrieveSourceP2FromTaintAnalysisResultP1 checks this and the interested statements are collected in `sourceP2`. Regarding sinks for phase 2 (i.e., blockchain data-write and transaction response proposals), they are always known a priori and depend on the framework (e.g., see Table 1 for HF). Then, they are also computed via signature matching through the function retrieveSinksP2FromSignatures

```go
1 // Get the args from the transaction
2 args := stub.GetStringArgs()
3
4 contract := args[0]
5 // [...]
6 queryArgs[0] = "SetAssetOwnership"
7 queryArgs[1] = myasset
8 response := stub.InvokeChaincode(contract, queryArgs,
     "main-channel")
9 stub.PutState("owner", response.Payload)
```

(a) Phase 1

```go
1 // Get the args from the transaction
2 args := stub.GetStringArgs()
3
4 contract := args[0]
5 // [...]
6 queryArgs[0] = "SetAssetOwnership"
7 queryArgs[1] = myasset
8 response := stub.InvokeChaincode(contract, queryArgs, "
     main-channel")
9 stub.PutState("owner", response.Payload)
```

(b) Phase 2

Figure 3: Taint analysis results on the code snippet of Figure 2.

and the program statements collected in `sinksP2` at line 9. At lines $10-12$, if there is at least a source and a sink in the program for this phase, Algorithm 1 performs another taint analysis to detect the storage and transaction response where there is untrusted information coming from UCCIs and collects the analysis alerts. Finally, at line 13, Algorithm 1 returns collected alerts to fill the analysis report, containing the following information: *(i)* the potential flows of untrusted data to a cross-contract invocation, and *(ii)* the potential storage in the blockchain of data coming from an untrusted cross-contract execution.

3.1 Running Example

Consider the code snippet of a HF smart contracts (also known as *chaincodes*, the term used for HF's code) in Figure 2.

In phase 1, Algorithm 1 detects `GetStringArgs` at line 2 and the parameters of `InvokeChaincode` at line 8 as source and sink by signature matching, respectively. Subsequentially, it performs taint analysis and propagates tainted values from the source (see Figure 3a). At the end of the computation, Algorithm 1 detects that variable `contract` is tainted when it is used in the sink at line 8. Then, it issues an alarm because an *untrusted cross-contract invocation* is detected and sets the `InvokeChaincode` at line 8 as a source for the phase 2. Note that, although `response.Payload` is tainted, it is not possible to trigger an alarm *blockchain data storage from untrusted cross-contract execution* because this phase of the analysis tracks untrusted input propagation and not untrusted execution results.

Table 1: HF methods of interest for the detection of UCCIs.

`shim.ChaincodeStubInterface`'s ***Method***	***Target***	***Category***
`GetArgs`	return value	Arbitrary Input
`GetStringArgs`	return value	Arbitrary Input
`GetFunctionAndParameters`	return value	Arbitrary Input
`GetArgsSlice`	return value	Arbitrary Input
`GetTransient`	return value	Arbitrary Input
`InvokeChaincode`	parameters, return value	CCIs
`PutState`	parameters	Data Storage
`DelState`	parameters	Data Storage
`SetStateValidationParameter`	parameters	Data Storage
`PutPrivateData`	parameters	Data Storage
`DelPrivateData`	parameters	Data Storage
`PurgePrivateData`	parameters	Data Storage
`SetPrivateDataValidationParameter`	parameters	Data Storage
`Success`	parameters	Transaction Response
`Error`	parameters	Transaction Response

In phase 2, the analysis detects `InvokeChaincode` as a source at line 8 because `contract` was tainted at the end of phase 1. Then, at line 9, it detects `PutState` as a sink by signature matching. Hence, it performs the second round of taint analysis (see Figure 3b). At the end of the computation, the analysis detects that the variable `response.Payload` is tainted and it issues an alarm because a *blockchain data storage from untrusted cross-contract execution* is found.

4 IMPLEMENTATION IN GOLISA

We implemented Algorithm 1 inside GoLiSA[4], an open-source static analyzer for Go supporting several blockchain frameworks. GoLiSA relies on LiSA [14, 30, 31] (**Li**brary for **S**tatic **A**nalysis), a modular framework for developing abstract interpretation-based static analyzers.

4.1 Detection of Sources and Sinks in GoLiSA

The first step for taint analysis is the identification of the sources and sinks of the target blockchain framework.

Currently, GoLiSA supports three different blockchain frameworks, i.e., HF, Tendermint Core, and Cosmos SDK. However, only HF natively provides smart contract APIs written in Go. Other frameworks do not provide official APIs for cross-contract invocations, although they may support smart contract frameworks with custom or third-party implementations. For the sake of simplicity, we cover only HF but the same approach can be applied to any other smart contract framework. Furthermore, as reported by the IBM company, HF has become the unofficial standard for enterprise blockchain platforms [21].

Table 1 summarizes the Go APIs that we considered critical for issues related to UCCIs: ***Method*** identify API functions; column ***Target*** defines which portion of the function's signature is being considered, i.e. return values for the sources and the parameters for the sinks; column ***Category*** specifies type of instruction. In particular, methods categorized as *Arbitrary input* return the arguments of a transaction request (i.e., user input) [18], i.e., they are considered as sources for *Phase 1*; the method `InvokeChaincode` is the only standard way to perform a CCIs in HF [18]; methods categorized as *Data Storage* allow one to perform blockchain data-write proposals [18]; methods categorized as *Transaction Response* are used for transaction response proposals [19, 20].

GoLiSA contains a full list of the signatures of these functions, and it automatically annotates them before the analysis begins. Annotations are used by the taint analysis to generate tainted values whenever a call to a source is encountered. Instead, the semantic checker that runs after the analysis searches the program for calls targeting functions with at least one parameter annotated as sink, and checks if the value passed for it is tainted or not.

5 EXPERIMENTAL EVALUATION

This section presents the results of the application of GoLiSA's analysis for the detection of UCCIs on a set of smart contracts written in Go and retrieved from public GitHub repositories. Experiments have been performed on a machine equipped with an AMD Ryzen 5 5600X 6-Core at 3.70 GHz, 16 GB of RAM DDR4, 1 TB SSD (read 540MB/s, write 500MB/s), running Windows 11 Pro 23H2, Open JDK version 20. During the analysis, 8 GBs of RAM were allocated to the JVM.

The experimental evaluation can be replicated with the materials contained in the following repository: https://github.com/lisa-analyzer/go-lisa/tree/sac2025.

5.1 Experimental Data Set ($\mathbb{CCI}$)

We refer to the experimental data set as $\mathbb{CCI}$. To collect the experimental data set, we started by looking at existing ones but, to the best of our knowledge, only the benchmark proposed in Olivieri et al. [36] exists. However, it only contains 24 contracts implementing 41 CCIs. Then, in addition to them, we retrieved other smart contracts from public GitHub repositories. Specifically, we looked for the *.InvokeChaincode(* keyword (i.e., call to a CCI in HF) and selected Go files using that call.[5] We considered all the 681 files from the query result and the 24 from the benchmark of Olivieri et al. [36]. Then we removed duplicates, that is, files with the same SHA256 checksum (code duplication is a widely adopted practice in the blockchain industry [45]) and files that do not call the `InvokeChaincode` function (in some cases it was mocked or the instructions commented). In the end, $\mathbb{CCI}$ consists of 420 files for a total of 106277 Lines of Code (LoCs), containing 897 CCIs.

5.2 Experimental Results

We performed the UCCI analysis for all the files in $\mathbb{CCI}$. The execution required a total of 24 minutes and 59 seconds ($\sim$ 3.56 seconds on average per file). The results report 157 files where at least a warning is issued, 227 files where no warning is raised, 36 files not analyzed due to failures (unsupported operations, parsing errors, . . .) during the execution of GoLiSA. The amount of reported warnings is 584. Table 2 shows details for each phase, where warnings are classified as:

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

[5] https://api.github.com/search/code?q=.InvokeChaincode(+language:Go&type=code&l=Go. Accessed: 03/04/2024.

Table 2: Warning details of the UCCI analysis results.

UCCI Analysis	#TP	#FP	#EF	#FN
Phase 1	277	0	4	0
Phase 2	301	0	2	0

- *true positives* (column **#TP**) if they refer to a detected vulnerability that happens in at least one possible contract execution (that is, there is an explicit source-to-sink flow of tainted information)
- *false positives* (column **#FP**) if they refer to a vulnerability that cannot happen in any contract execution, but that is being considered due to over-approximation in the analysis (that is, the explicit source-to-sink flow never happens in any possible execution)
- *external flows* (column **#EF**) if they refer to a vulnerability cannot happen in any contract execution, but that might manifest if a contract's function is invoked by another contract (that is, the warning is a false positive when you only consider explicit source-to-sink flows, but it becomes a true positive if the source is in another contract)

Additionally, the table reports *false negatives* (column **#FN**) as the number of vulnerabilities missed by the analysis.

5.2.1 Limits of the Evaluation. Although HF is largely used in the industrial sector, its uses are related to the development of permissioned and often private blockchains, meaning that the related software is not publicly available or released with open-source licenses. This greatly limits the creation of a data set in comparison to public and permissionless blockchains such as Ethereum, where it is typically possible to collect experimental artifact sets of large dimensions simply by querying the public code deployed and available in blockchain, such as in Wang et al. [55], where more than three thousand distinct smart contracts are collected from the Ethereum blockchain.

Regarding the limits of warning classification, another criterion to classify true and false positives is to evaluate the analysis results with the runtime environment, since the static analysis is performed without the real execution information [8, 49]. For instance, the interactions between contracts from different channels in HF can be denied or limited [51], thus some UCCIs could be mitigated or avoided at run time depending on where the contracts are deployed. However, in this specific case, we could not do this type of check for the manual investigation since ℂℂ𝕀 has been retrieved from public repositories where the source code is statically stored and not deployed in a running blockchain environment.

5.3 How to Classify Analysis Results

Below, we evaluate and classify the UCCI analysis results of a few snippets of code taken from ℂℂ𝕀. In the proposed examples (Figures 4, 5, 6), sources of Phase 1 and 2 are highlighted with blue and black boxes, respectively; the sinks of Phase 1 and 2 are highlighted with red and brown boxes, respectively; the tainted information propagated in the instruction sinks of Phase 1 and 2 is highlighted in gray and orange, respectively.

5.3.1 True Positive. Contract `chaincode_union_loan` in Fig. 4, a proof of concept implementation of bank loans in blockchain, is an example of true positive found in ℂℂ𝕀. Users call method `offer` to offer a loan. GoLiSA detects a flow that leads to an untrusted cross-contract invocation on tainted data about loan participants. Namely, at line 5 of method `Invoke`, GoLiSA considers `GetFunctionAndParameters` as a tainted source, since it yields a function name and arguments provided as part of the transaction request, hence under user control. This tainted data propagates through `args` to method `offer` at line 7, reaching `InvokeChaincode` through variable `chainCodeToCall` at line 15. GoLiSA issues a warning at line 16 during the first phase, since the first parameter of `InvokeChaincode` is tainted. Thanks to this warning, the return value of the call is considered a source for the second phase. The returned value, stored into variable `response`, is used to build the error message `errStr` for the `shim.Error` call at line 23, which GoLiSA considers as a sink for the second phase. Thus, a warning is also raised at this line because transactions with untrusted error responses should not be approved and should not be able to reach the ordering stage in HF.

```
1  func (t *UnionLoanChaincode) Invoke(
2  stub shim.ChaincodeStubInterface)
3  pb.Response {
4    function, args :=
         stub.GetFunctionAndParameters()
5    if function == "offer" {
6      return t.offer(stub, args)
7    }
8    // [...]
9  }
10 func (t *UnionLoanChaincode) offer(
11 stub shim.ChaincodeStubInterface,
12 args []string) pb.Response {
13   // [...]
14   var chainCodeToCall = args[0]
15   // [...]
16   response := stub.InvokeChaincode(
         chainCodeToCall, invokeArgs, "")
17   // [...]
18   errStr := fmt.Sprintf("Failed to invoke
         chaincode. Got error: %s", string(
         response.Payload))
19   return shim.Error(errStr)
20 }
```

Figure 4: Simplified code from *chaincode_union_loan*.

5.3.2 True Negative. The *sealtxnew* contract from ℂℂ𝕀, in Fig. 5, is a proof of concept implementation of seal transaction application for a trading blockchain. GoLiSA, correctly, does

```
1  func (s *SealTX) Invoke(
2  stub shim.ChaincodeStubInterface)
3  pb.Response {
4    function, args :=
           stub.GetFunctionAndParameters()
5    // [...]
6    switch function {
7      // [...]
8      case "querybykey":
9      return s.querybykey(stub, args)
10     // [...]
11   }
12 }
13 func (t *SealTX) querybykey(
14 stub shim.ChaincodeStubInterface,
15 args []string) pb.Response {
16   // [...]
17   return stub.InvokeChaincode("sealtx",
        args4old, "tradechannel")
18   // [...]
19 }
```

Figure 5: Simplified code from *sealtxnew*.

not raise any warning about untrusted cross-contract invocations. In fact, line 3 of method `Invoke` retrieves a tainted value through the source `GetFunctionAndParameters`. This tainted data propagates, through `args`, to method `querybykey` at line 9. Here, no warning is generated since the tainted information never reaches the first parameter of `InvokeChaincode`, i.e., the sink for the analysis. Indeed, this latter targets the hardcoded contract `sealtx`. It is thus only possible to query the `sealtx` contract, without risking an UCCI.

5.3.3 False Positive and External Flows. False positives are a consequence of excessive approximation. For instance, consider the code in Fig. 6, where GoLiSA issues a warning that can be classified as both external flow and false positive. In the file, the functions `GetFunctionAndParameters()` (lines 1-3) and `GetKYC()` (lines 11-25) are only declared and never used in the file. Nevertheless, since the file contains both source and sink (lines 2 and 23), the analysis is executed.

Although both functions are not explicitly called in the contract, the analysis soundly assumes that they might be called by functions of other chaincodes at run time. For this reason, during the propagation phase, `GetKYC` at line 11 is considered reachable and its the formal parameter `userId` is over-approximated as tainted as it is statically unknown. Such value is later propagated into `params`, `arg` and `queryArgs` at lines 13, 15 and 17, respectively. Finally, tainted value coming from `queryArgs` flows into the sink `ctx.GetStub().InvokeChaincode()` at line 23 and the analysis issues a warning. One might argue that this is a false positive, as there is no explicit source-to-sink flow happening. However, as `GetKYC` can be the target of a cross-contract call, we label the warning as an external flow since it might lead to a UCCI.

If one rules out the possibility of a cross-contract call, `queryArgs` at line 23 is clean. Nonetheless, a warning is still issued at the same line: `GetChannelID` at line 6 returns

```
1  func (ctx *TransactionContext)
       GetFunctionAndParameters() (string, []
       string) {
2    return ctx.GetStub().GetFunctionAndParameters()
3  }
4
5  func (ctx *TransactionContext)
       GetChannelName() (string, error) {
6    channelID := ctx.GetStub().GetChannelID()
7    // [...]
8    return channelID, nil
9  }
10
11 func (ctx *TransactionContext) GetKYC(
       userId string) (bool, error) {
12
13   // [...]
14   channelName, err := ctx.GetChannelName()
15   // [...]
16   params := []string{crossCCFunc, userId}
17   // [...]
18   queryArgs := make([][]byte, len(params))
19   for i, arg := range params {
20     queryArgs[i] = []byte(arg)
21   }
22   // [...]
23   response := ctx.GetStub().InvokeChaincode(
         crossCCName, queryArgs, channelName)
24   // [...]
25 }
26 // [...]
```

Figure 6: Simplified code from *read_transaction*.

a tainted value, that is then propagated into `channelID` and `channelName` at lines 8 and 14, respectively. Finally, `channelName` is used as parameter to the sink `ctx.GetStub().InvokeChaincode` at line 23, and the GoLiSA creates a warning at the end of the propagation phase. Such a warning is a false positive, and it is due to the over-approximation of the method `GetChannelID` at line 6. According to the documentation of HF, `GetChannelID` *returns the channel ID for the proposal for chaincode to process. This would be the 'channel_id' of the transaction proposal [...].* Such a value is thus statically unknown, and GoLiSA models it as an instruction that can return any possible string value. GoLiSA does not currently distinguish between *any possible value* and *any possible user-provided value*: in terms of taintedness, both are modeled as a statically unknown and possibly tainted value.

Since the sink at line 23 is tied to two different flows, one of which is highlighting a real vulnerability, the warning referring to it was still classified as an external flow in Table 2 since our analysis aims at being as sound as possible.

6 RELATED WORK

Taint analysis is used extensively in smart contract verification tools to detect vulnerabilities, and can also be combined with graph reconstruction techniques to improve user experience [16]. For instance, it allows one to detect critical issues such as re-entrancy [1, 50, 52]. It is considered one of the most critical issues in smart contracts. It was also the root cause of the well-known DAO attack [46], which resulted in the loss of more than 50M of dollars for Ethereum users. The exploitation allows an attacker to execute a recursive callback of the main function, making an unintended loop that is repeated many times, leading to the fully destruction of a contract or stealing valuable economic assets and information. Typically, re-entrancy may be exploited by using CCIs and creating an inter-contract loop. This makes re-entrancy detection difficult. Furthermore, in the event of a UCCI, it would be even easier for an attacker to create an ad hoc malicious contract capable of exploiting the re-entrancy. In this case, our analysis can identify any UCCIs, but cannot detect the loop nor the re-entrancy in the case of trusted CCIs.

Another issue that can be dealt with taint analysis is the detection of Parity Wallet bug [42, 43]. It has become very popular because it has been exploited by an attacker to steal over 30M of dollars. The application implemented a proxy pattern/library to split the logic of a wallet into two separate smart contracts. The first smart contract calls the second (the library) with a CCI to execute wallet operations. Although this bug involves CCIs, the problem is different from what this paper studies. The library address of the CCI was hardcoded and the issue resided in the library containing an issue concerning the method visibilities, which resulted in the attacker directly taking control of the library. Instead, we considered only CCIs with flows from an untrusted input as UCCI, thus our analysis is not able to detect the issues related to the Parity Wallet bug because the address is hardcoded and not related to an untrusted input. Moreover, according to Sayeed et al. [50], there are several tools for the detection of Parity Wallet bugs but the coverage of this issue is still challenging (tools such as Oyente [25] detect only 20% of Parity Wallet hacks [54]).

Regarding UCCIs detection, to the best of our knowledge, currently, there are no tools for Go, except for GoLiSA, covering these issues. The Chaincode Analyzer [23], which does provide checks for *cross-channel invocation*, does not cover the UCCI cases. Indeed, cross-channel issues are similar but limited to a specific scenario regarding channels and do not lead to a code injection. In short words, channels [12] are private subnets of communication between two or more specific members of the blockchain network and where it is also possible to deploy chaincodes. However, a transaction failure happens when a chaincode calls another contract deployed in a different channel because the execution policies do not allow it [26]. In particular, Chaincode Analyzer [23] only checks that there are no CCIs with different hardcoded channel names, in order to report possible transaction failures.

Instead, concerning other smart contract languages, several techniques are applied to detect UCCIs. ContractFuzzer [22] generates fuzzing inputs and defines test oracles to detect security vulnerabilities, including problems related to UCCIs in Solidity. The tool contains an offline EVM instrumentation and an online fuzzing tool. The offline EVM instrumentation process is responsible for monitoring the execution of smart contracts to extract information for vulnerability analysis. The online fuzzer analyzes the smart contract under test with additional information, such as its *ABI* interface. Compared to our approach, fuzzing is a testing technique and can only spot the issues but not ensure their absence [49]. Wang et al. [56] propose a general platform for defect detection in smart contracts, including the UCCI issues. The platform generates the ASTs for each smart contract and obtains the semantic description of corresponding functions and variables. Hence, it generates assertions by knowledge of security model libraries and semantic descriptions of ASTs and expressions and then detects the defects of smart contracts. However, as also stated by the authors, there are still problems that need further research and improvement. In particular, they use manual assertions, which in case of implementation errors can lead to omissions. SolGuard [47] detects UCCIs at compile time in Ethereum and focuses mainly on smart contract-based multi-agent robotic systems. It implements the analyses using AST traversing and semantic flow checking. Mythril [44] bases the analyses on symbol execution and concrete execution techniques to discover vulnerabilities, including UCCIs. It combines static execution with dynamic execution to improve path coverage and detection accuracy. Note that the symbolic execution approach does not guarantee the exploration of all program paths, leading potentially to false negatives. SMARTSHIELD [57] dynamically highlights state changes and alterations after CCIs. It analyzes both the AST and the unrectified EVM bytecode of each contract to extract its bytecode-level semantic information. Then, the tool fixes insecure control flows and data operations through control flow transformation and the insertion of instruction sequences that perform certain data validity checks. Finally, in MichelsonLiSA [33, 35], Olivieri et al. provide a UCCI analysis prototype for smart contracts written in the Michelson language for the Tezos blockchain. However, cross-contract invocations are limited in Michelson language. Currently, they only allow one to transfer tokens and do not support the call of different contract methods. Moreover, MichelsonLiSA's implementation performs only the first taint analysis step because Michelson does not support explicit APIs such as for data storage in blockchain or for sending transaction responses, which instead are implicitly performed at the end of each smart contract execution.

About cross-contract analysis, tools like SmartDagger [24], CrossInspector [6], and Pluto[27] are specifically designed to perform analysis also considering the inter-connected components between different contracts. However, they can involve complex and dynamic interactions, making it difficult to predict the behavior of contracts in all scenarios, especially over time. Indeed, in the case of UCCIs, small changes in the input are often enough to create a new malicious contract. Therefore, inter-contract analysis may significantly burden the UCCI detection adding only marginal improvements.

7 CONCLUSION

This paper addresses the challenging issue of detecting untrusted cross-contract invocations in general-purpose languages, such as Go, and shows that the semantics-based static analysis approach based on information flow in two phases provides a precise, efficient, and scalable solution. Experiments on existing smart contracts written in Go, crawled from GitHub, empirically show that our approach is useful and scalable in practice. Moreover, they also confirm that, when targeting blockchain software, it is possible to adopt analysis techniques that would typically have performance and scalability problems over traditional industrial-size software. Future work will investigate cross-contract contexts and other challenges and issues such as re-entrancy and Parity Wallet bugs on general-purpose languages.

ACKNOWLEDGMENTS

Work partially supported by SERICS (PE00000014 - CUP H73C2200089001) and iNEST (ECS00000043 – CUP H43C22-000540006) projects funded by PNRR NextGeneration EU, and by Bando di Ateneo per la Ricerca 2022, funded by University of Parma, (MUR__DM737__2022__ FIL__PROGETTI __B__ARCERI__COFIN, CUP: D91B210 05370003), "Formal verification of GPLs blockchain smart contracts".

REFERENCES

[1] Mouhamad Almakhour, Layth Sliman, Abed Ellatif Samhat, and Abdelhamid Mellouk. 2020. Verification of smart contracts: A survey. *Pervasive and Mobile Computing* 67 (2020), 101227.

[2] A. M. Antonopoulos and G. Wood. 2018. *Mastering Ethereum: Building Smart Contracts and Dapps.* O'Reilly.

[3] Alessandro Brighente, Mauro Conti, and Sathish Kumar. 2022. Extorsionware: Exploiting Smart Contract Vulnerabilities for Fun and Profit. *ArXiv* abs/2203.09843 (2022). https://arxiv.org/abs/2203.09843

[4] Vitalik Buterin. 2016. EIP-170: Contract code size limit. https://eips.ethereum.org/EIPS/eip-170, Accessed: 02/2024.

[5] Christian Cattai. 2022. *Extorsionware: Bringing Ransomware Attacks to Blockchain Smart Contracts.* Master thesis. University of Padua, Italy.

[6] Xiao Chen. 2024. CrossInspector: A Static Analysis Approach for Cross-Contract Vulnerability Detection. *arXiv preprint arXiv:2408.15292* (2024).

[7] Patrick Cousot. 1997. Types as Abstract Interpretations. In *Proc. of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1997.* ACM Press, 316–331. https://doi.org/10.1145/263699.263744

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

[9] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of the 4th Symposium on Principles of Programming Languages, 1977.* ACM, 238–252. https://doi.org/10.1145/512950.512973

[10] CWE Content Team, MITRE. 2010. CWE-829: Inclusion of Functionality from Untrusted Control Sphere. https://cwe.mitre.org/data/definitions/829.html, Accessed: 12/2022.

[11] Michael D. Ernst, Alberto Lovato, Damiano Macedonio, Ciprian Spiridon, and Fausto Spoto. 2015. 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 (Lecture Notes in Computer Science, Vol. 9450).* Springer, 130–145. https://doi.org/10.1007/978-3-662-48899-7_10

[12] Hyperledger Fabric. 2024. Channels. https://hyperledger-fabric.readthedocs.io/en/release-2.5/channels.html#channels (Accessed 04/2024).

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

[14] Pietro Ferrara, Luca Negrini, Vincenzo Arceri, and Agostino Cortesi. 2021. Static analysis for dummies: experiencing LiSA. In *SOAP@PLDI 2021: Proc. of the 10th ACM SIGPLAN Int. Workshop on the State Of the Art in Program Analysis.* ACM, 1–6. https://doi.org/10.1145/3460946.3464316

[15] Pietro Ferrara, Luca Olivieri, and Fausto Spoto. 2018. 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 (Lecture Notes in Computer Science, Vol. 11079).* Springer, 63–76. https://doi.org/10.1007/978-3-030-02547-2_4

[16] Pietro Ferrara, Luca Olivieri, and Fausto Spoto. 2020. Backflow: Backward Context-Sensitive Flow Reconstruction of Taint Analysis Results. In *Verification, Model Checking, and Abstract Interpretation: 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16–21, 2020, Proceedings* (New Orleans, LA, USA). Springer-Verlag, Berlin, Heidelberg, 23–43. https://doi.org/10.1007/978-3-030-39322-9_2

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

[18] Hyperledger. 2024. Hyperledger Fabric Go API Documentation - Shim Interfaces. https://github.com/hyperledger/fabric-chaincode-go/blob/b84622ba6a7a9e543f3ca1994850c41423bc29a2/shim/interfaces.go, Accessed 04/2024.

[19] Hyperledger. 2024. Hyperledger Fabric Go API Documentation - Shim Response. https://github.com/hyperledger/fabric-chaincode-go/blob/b84622ba6a7a9e543f3ca1994850c41423bc29a2/shim/response.go, Accessed 04/2024.

[20] Hyperledger. 2024. Transaction Flow - Hyperledger Fabric Documentation. https://hyperledger-fabric.readthedocs.io/en/release-2.5/txflow.html, Accessed 04/2024.

[21] IBM. 2024. What is hyperledger fabric? https://www.ibm.com/topics/hyperledger (Accessed 03/2024).

[22] Bo Jiang, Ye Liu, and W.K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE).* 259–269. https://doi.org/10.1145/3238147.3238177

[23] kzhry. 2021. Chaincode Analyzer. https://github.com/hyperledger-labs/chaincode-analyzerc, Accessed: 09/2022.

[24] Zeqin Liao, Zibin Zheng, Xiao Chen, and Yuhong Nan. 2022. SmartDagger: a bytecode-based static analysis approach for detecting cross-contract vulnerability. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022).* Association for Computing Machinery, New York, NY, USA, 752–764. https://doi.org/10.1145/3533767.3534222

[25] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16).* Association for Computing Machinery, New York, NY, USA, 254–269. https://doi.org/10.1145/2976749.2978309

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

[27] Fuchen Ma, Zhenyang Xu, Meng Ren, Zijing Yin, Yuanliang Chen, Lei Qiao, Bin Gu, Huizhong Li, Yu Jiang, and Jiaguang Sun. 2022. Pluto: Exposing Vulnerabilities in Inter-Contract Scenarios. *IEEE Transactions on Software Engineering* 48, 11 (2022), 4380–4396. https://doi.org/10.1109/TSE.2021.3117966

[28] Amit Mandal, Pietro Ferrara, Yuliy Khlyebnikov, Agostino Cortesi, and Fausto Spoto. 2020. Cross-Program Taint Analysis for IoT Systems. In *Proceedings of the 35th Annual ACM*

Symposium on Applied Computing (Brno, Czech Republic) *(SAC '20)*. 1944–1952. https://doi.org/10.1145/3341105.3373924
[29] Nick Mudge. 2020. ERC-2535: Diamonds, Multi-Facet Proxy. https://eips.ethereum.org/EIPS/eip-2535, Accessed: 02/2024.
[30] Luca Negrini, Vincenzo Arceri, Luca Olivieri, Agostino Cortesi, and Pietro Ferrara. 2024. Teaching Through Practice: Advanced Static Analysis with LiSA. In *Formal Methods Teaching*, Emil Sekerinski and Leila Ribeiro (Eds.). Springer Nature Switzerland, Cham, 43–57. https://doi.org/10.1007/978-3-031-71379-8_3
[31] Luca Negrini, Pietro Ferrara, Vincenzo Arceri, and Agostino Cortesi. 2023. *LiSA: A Generic Framework for Multilanguage Static Analysis*. Springer Nature Singapore, Singapore, 19–42. https://doi.org/10.1007/978-981-19-9601-6_2
[32] Luca Olivieri, Vincenzo Arceri, Badaruddin Chachar, Luca Negrini, Fabio Tagliaferro, Fausto Spoto, Pietro Ferrara, and Agostino Cortesi. 2024. General-Purpose Languages for Blockchain Smart Contracts Development: A Comprehensive Study. *IEEE Access* 12 (2024), 166855–166869. https://doi.org/10.1109/ACCESS.2024.3495535
[33] Luca Olivieri, Thomas Jensen, Luca Negrini, and Fausto Spoto. 2023. MichelsonLiSA: A Static Analyzer for Tezos. In *2023 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*. 80–85. https://doi.org/10.1109/PerComWorkshops56833.2023.10150247
[34] Luca Olivieri, Luca Negrini, Vincenzo Arceri, Badaruddin Chachar, Pietro Ferrara, and Agostino Cortesi. 2024. Detection of Phantom Reads in Hyperledger Fabric. *IEEE Access* 12 (2024), 80687–80697. https://doi.org/10.1109/ACCESS.2024.3410019
[35] Luca Olivieri, Luca Negrini, Vincenzo Arceri, Thomas Jensen, and Fausto Spoto. 2024. Design and Implementation of Static Analyses for Tezos Smart Contracts. *Distrib. Ledger Technol.* (jan 2024). https://doi.org/10.1145/3643567 Just Accepted.
[36] Luca Olivieri, Luca Negrini, Vincenzo Arceri, Fabio Tagliaferro, Pietro Ferrara, Agostino Cortesi, and Fausto Spoto. 2023. Information Flow Analysis for Detecting Non-Determinism in Blockchain. In *37th European Conference on Object-Oriented Programming (ECOOP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 23:1–23:25. https://doi.org/10.4230/LIPIcs.ECOOP.2023.23
[37] Luca Olivieri and Luca Pasetto. 2024. Towards Compliance of Smart Contracts with the European Union Data Act. In *CEUR Workshop Proceedings*, Vol. 3629. https://ceur-ws.org/Vol-3629
[38] Luca Olivieri, Luca Pasetto, Luca Negrini, and Pietro Ferrara. 2024. European Union Data Act and Blockchain Technology: Challenges and New Directions. *CEUR Workshop Proceedings* 3791. https://ceur-ws.org/Vol-3791
[39] Luca Olivieri and Fausto Spoto. 2024. Software verification challenges in the blockchain ecosystem. *International Journal on Software Tools for Technology Transfer* (2024). https://doi.org/10.1007/s10009-024-00758-x Published 2024/07/12.
[40] 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 *SOAP '22: 11th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, San Diego, CA, USA, 14 June 2022*, Laure Gonnord and Laura Titolo (Eds.). ACM, 23–29. https://doi.org/10.1145/3520313.3534658
[41] OpenZeppelin. 2018. Proxy Patterns. https://blog.openzeppelin.com/proxy-patterns/, Accessed: 02/2024.
[42] Santiago Palladino. 2017. The Parity Wallet Hack Explained. https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/, Accessed: 03/2024.
[43] Santiago Palladino. 2017. Parity Wallet Hack Reloaded. https://blog.openzeppelin.com/parity-wallet-hack-reloaded/, Accessed: 03/2024.
[44] Nikhil Parasaram. 2020. Mythril Wiki Page. https://github.com/ConsenSys/mythril/wiki Accessed: 10/2022.
[45] Giuseppe Antonio Pierro and Roberto Tonelli. 2021. Analysis of Source Code Duplication in Ethreum Smart Contracts. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 701–707. https://doi.org/10.1109/SANER50967.2021.00089
[46] Nathaniel Popper. 2016. A Hacking of More Than $50 Million Dashes Hopes in the World of Virtual Currency. *The New York Times* (2016). June 17th.
[47] Purathani Praitheeshan, Lei Pan, Xi Zheng, Alireza Jolfaei, and Robin Doss. 2021. SolGuard: Preventing external call issues in smart contract-based multi-agent robotic systems. *Information Sciences* 579 (2021), 150–166. https://doi.org/10.1016/j.ins.2021.08.007
[48] SWC Registry. 2020. SWC-112: Delegatecall to Untrusted Callee. https://swcregistry.io/docs/SWC-112/ (Accessed 04/2024).
[49] Xavier Rival and Kwangkeun Yi. 2020. *Introduction to static analysis: an abstract interpretation perspective*. Mit Press.
[50] Sarwar Sayeed, Hector Marco-Gisbert, and Tom Caira. 2020. Smart Contract: Attacks and Protections. *IEEE Access* 8 (2020), 24416–24427. https://doi.org/10.1109/ACCESS.2020.2970495
[51] KC Tam. 2020. Cross-Chaincode Invoking in Hyperledger Fabric. https://kctheservant.medium.com/cross-chaincode-invoking-in-hyperledger-fabric-8b8df1183c04, Accessed: 03/2024.
[52] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. 2021. A Survey of Smart Contract Formal Specification and Verification. *ACM Comput. Surv.* 54, 7, Article 148 (jul 2021), 38 pages. https://doi.org/10.1145/3464421
[53] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. 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*, Michael Hind and Amer Diwan (Eds.). Acm, 87–97. https://doi.org/10.1145/1542476.1542486
[54] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 67–82. https://doi.org/10.1145/3243734.3243780
[55] Shuai Wang, Chengyu Zhang, and Zhendong Su. 2019. Detecting nondeterministic payment bugs in Ethereum smart contracts. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 189:1–189:29. https://doi.org/10.1145/3360615
[56] Xiaoqiang Wang, Jianhua Li, and Xuesen Zhang. 2022. A Semantic-Based Smart Contract Defect Detection General Platform. In *2022 IEEE International Conference on Advances in Electrical Engineering and Computer Applications (AEECA)*. 34–37. https://doi.org/10.1109/AEECA55500.2022.9918903
[57] Yuyao Zhang, Siqi Ma, Juanru Li, Kailai Li, Surya Nepal, and Dawu Gu. 2020. SMARTSHIELD: Automatic Smart Contract Protection Made Easy. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 23–34. https://doi.org/10.1109/SANER48275.2020.9054825