Detection of Vulnerabilities in Smart Contracts Specifications in Ethereum Platforms

Mauro C. Argañaraz
Departamento de Informática, Facultad de Ciencias Física Matemáticas y Naturales (FCFMyN), Universidad Nacional de San Luis, Argentina
marganaraz@gmail.com

Mario M. Berón
Departamento de Informática, Facultad de Ciencias Física Matemáticas y Naturales (FCFMyN), Universidad Nacional de San Luis, Argentina
mberon@unsl.edu.ar

Maria J. Varanda Pereira
Research Centre in Digitalization and Intelligent Robotics (CeDRI), Instituto Politécnico de Bragança, Portugal
mjoao@ipb.pt

Pedro Rangel Henriques
Centro Algoritmi (CAlg-CTC), Department of Informatics, University of Minho, Braga, Portugal
pedrorangelhenriques@gmail.com

Abstract

Ethereum is the principal ecosystem based on blockchain that provides a suitable environment for coding and executing smart contracts, which have been receiving great attention due to the commercial apps and among the scientific community. The process of writing secure and well performing contracts in the Ethereum platform is a major challenge for developers. It consists of the application of non-conventional programming paradigms due to the inherent characteristics of the execution of distributed computing programs. Furthermore, the errors in the deployed contracts could have serious consequences because of the immediate linkage between the contract code and the financial transactions. The direct handling of the assets means that the errors can be more relevant for security and have greater economic consequences than a mistake in the conventional apps. In this paper, we propose a tool for the detection of vulnerabilities in high-level languages based on automatized static analysis.

2012 ACM Subject Classification Security and privacy → Vulnerability scanners

Keywords and phrases blockchain, ethereum, smart contract, solidity, static analysis, verification

Digital Object Identifier 10.4230/OASIcs.SLATE.2020.2

Funding This work has been supported by FCT – Fundação para a Ciência e Tecnologia within the Project Scope: UIDB/05757/2020.

1 Introduction

Blockchain is a technology based on the combination between cryptography, networks and incentive mechanisms designed to support the verification, execution and registration of transactions among different peers. In other words, blockchain platforms can be defined as decentralized databases that offer attractive properties, such as immutability of the stored transactions and the creation of a sense of confidence between peers without the participation of a third party. Hence, this architecture is suitable as an open and distributed ledger that can save transactions in a verifiable and permanent manner.

Cryptocurrency was the first blockchain technology application and it is a new form of digital asset based on a network that is distributed across a large number of computers. This decentralized structure allows them to exist outside the control of governments and central authorities. The most famous and used cryptocurrencies are Bitcoin [16] and Ethereum [5]. They offer, in addition to the exchange of digital assets, the execution of smart contracts.

Smart contracts are basically two combined concepts. One of them is software. Insensible and austere code, that does what is written and executes it for the world to see. The other one is the sense of agreement between the parts. They are computer programs that facilitate, verify and ensure the negotiation and execution of legal contracts. They are executed through blockchain transactions, they interact with the cryptocurrencies and have interfaces to handle the information of the contract's participants. When a smart contract is executed on the blockchain, it transforms into an autonomous entity that automatically carries on specific actions under certain conditions.

Ethereum is the primary public distributed computing platform based on blockchain that provides an environment that enables the execution of smart contracts in a decentralized virtual machine, known as Ethereum Virtual Machine (EVM) [5, 25].

The virtual machine handles the compute and the state of the contracts and uses a language based on a stack structure with a predefined group of instructions (opcode) [25]. Essentially, a contract is simply a series of statements of opcodes that are sequentially executed by the virtual machine.

In this article, a tool is proposed that reinforces the security aspects of the specifications of Ethereum Platform's smart contracts. This has its basis on a solid foundation of design, established and tried code patterns that facilitate the functional code writing process with no errors, to provide a process that carries on a static program analysis and detect flaws automatically.

This document is structured in this way: Firstly, a summary about related work in Section 2 is provided. Furthermore, in Section 3 security aspects of the smart contracts are discussed, before presenting the vulnerabilities analysis and the detection rules in Section 4. In Section 5 the tool is proposed and in Section 6 a use case is being described. Finally, the conclusions and future works are being outlined.

2 Related Work

In the search for Ethereum's smart contracts safety, different approaches have been adopted, focusing on formal semantics, security patterns and verification tools. According to the analysis, a distinction can be made between verification and design.

The goal of the verification approaches is to check that the existent smart contracts, which are written in high level languages (such as Solidity) or in EVM bytecodes, meet a policy or a security specification. Some examples of works, techniques and tools in that direction are:

- **Static program analysis tools for the automatic search of bugs.** Oyente [14] is a static analysis tool for EVM bytecode that relies on symbolic execution. Oyente supports a variety of pre-defined security properties that can be checked automatically. Zhou et al. proposed SASC [27] that extends Oyente by additional patterns and provides a visualization of detected risks. Majan [17] extends the approach taken in Oyente to trace properties that consider multiple invocations of one smart contract.

- **Static program analysis tools for the automatic verification of generic properties.** ZEUS [13] is a static analysis tool that analyzes smart contracts written in Solidity using symbolic model checking. Other tools proposed in the realm of automated static analysis for generic properties are Securify [24], Mythril [7] and Manticore [18] for analysing bytecode and SmartCheck [23] for analyzing Solidity code.
- **Frameworks for semi-automated testing of specific contract properties.** Hirai [12] formalizes the EVM semantics in the proof assistant Isabelle/HOL and uses it for manually proving safety properties for concrete contracts. Hildebrandt et al. [11] define the EVM semantics in the K framework. Bhargavan et al. [4] introduce a framework to analyze Ethereum contracts by translation into F*.
- **Dynamic monitoring of predefined security properties.** Grossman et al. [10] propose the notion of effectively callback free executions and identify the absence of this property in smart contract executions as the source of common bugs. A solution compatible with Ethereum is offered by the tool DappGuard [8].

On the other hand, the design approaches are aimed at the creation of secure smart contracts by providing frameworks for the development: they take into account new languages that are more verifiable, they provide a clearer and simpler semantics that is understandable for the smart contracts' developers or that allows a direct codification of the security policies. The examples of design propositions can be classified into:

- **High level languages.** Coblenz [6] propose Obsidian, an object-oriented programming language that makes states explicit and uses a linear type system for quantities of money. Flint [21] is a type-safe, capabilities-secure, contract-oriented programming language for smart contracts that gets compiled to EVM bytecode. Flint allows for defining caller capabilities restricting the access to security sensitive functions. These capabilities shall be enforced by the EVM bytecode created during compilation.
- **Intermediate languages.** Scilla [22] comes with a semantics formalized in the proof assistant Coq and therefore allows for a mechanized verification of Scilla contracts.
- **Security patterns.** Wöhrer [26] describes programming patterns in Solidity that should be adapted by smart contract programmers for avoiding common bugs.
- **Tools.** Mavridou and Laszka [15] introduce a framework for designing smart contracts in terms of finite state machines.

3 Security and Smart Contracts

The smart contracts on Ethereum are generally written in high level language and then are compiled in EVM bytecodes. The most prominent and most widely adopted is Solidity [9], it is used even in other blockchain platforms. Solidity is a contract oriented high level programming language whose syntax is similar to Javascript.

A smart contract analysis carried out by Bartoletti and Pompianu [3] shows that Bitcoin and Ethereum primarily focus on financial contracts. The direct handling of the assets means that the flaws are more likely to be relevant to the security and have greater financial consequences that the errors on typical applications, as evidenced by the DAO attack on Ethereum.

According to Alharby and van Moorsel [1], the current investigation on smart contracts has its focus on identifying and addressing the smart contract's issues and they classify them in the following four categories: codification, security, privacy and problems of performance. The technology behind Ethereum's smart contracts is still in the early stages, thus, codification and security are the most discussed topics.

3.1 Security Challenges in Ethereum

Security is the main concern when talking about Ethereum's programming owing to the following factors:

- Unknown runtime environment: Ethereum is different to the centrally administered runtime environments, either mobile, desktop or in the cloud. Developers are not used to their code being executed in a global network of anonymous nodes, without a secure relationship and with a profit reason.
- New software stack: The Ethereum stack (the Solidity compiler, the EVM, the consensus layer) is still in the developing stages, and security vulnerabilities are still being discovered.
- Highly limited ability to correct contracts: A deployed contract cannot be corrected, hence, it has to be correct before the deployment. This opposes the traditional software development process that promotes iterative techniques.
- Financially motivated anonymous attackers: In comparison with several cibernetic crimes, exploiting smart contracts offers greater incomings (cryptocurrencies' price has rapidly risen), facility for the charging (the ether and the tokens can be instantly commercialized) and a minor risk of punishment due to the anonymity and the lack of legislation on the subject matter.
- Rapid pace of development: Blockchain companies make an effort to rapidly launch their products, usually at the expense of the security.
- Sub-optimal high level language: Some investigations claim that Solidity as itself leads the developers to unsecure development techniques [26, 1].

3.2 Design Challenges and Patterns Usage

Understanding how smart contracts are used and how they are implemented could help smart contracts platforms' designers to create new domain-specific languages, which, with their designs, avoid vulnerabilities such as the ones that are being outlined posteriorly. In addition, this knowledge could help improve the analysis techniques for smart contracts, by promoting the usage of contracts with specific programming patterns. To this day, little efforts have been made in the collection and categorization of patterns and the toolbox they use in an organized way [26]. In the following bullet points, a general description of the typical design patterns that are inherently frequent or practical when talking about the codification of smart contracts.

- Authorization: This pattern is used for restricting the code in accordance with the invoker's direction. The vast majority of analysed contracts verify if the invoker's direction is the same as the direction of the owner of the contract, before carrying out critical operations (for instance, sending ether, calling the method suicide or selfdestruct).
- Oracle: Is possible that some contracts have to acquire data outside the blockchain. The Ethereum platform does not allow the contracts to consult external sites: otherwise, the determinism of the calculations would break, due to the fact that different nodes could receive different results for the same consultation. The oracles are the interface between the contracts and the outside.
- Randomisation: Since the execution of the contract needs to be deterministic, all the nodes have to obtain the same numerical value when requesting a random number: this conflicts with the desired randomisation requirements.
- Time limitations: Many contracts require the implementation of time restrictions, for instance, for specifying when an action is allowed.

- Termination: Due to the fact that the blockchain is immutable, a contract cannot be eliminated when it is no longer being used. In view of this, developers have to foresee a way of disable it, in a way that it still exists, but without responding. Generally, only the contract's owner is authorized to disable a contract.

The presented patterns address typical issues and vulnerabilities related with the execution of smart contracts. Wöhrer and Zdun [26] worked particularly with the security patterns: Check-Effects-Interaction, Emergency Stop (Circuit Breaker), Speed Bump, Frequency, Mutex and Balance Limit.

The most important in the security level is the Check-Effects-Interaction pattern that describes how a function's code should be structured in order to avoid secondary effects and undesired execution behaviours. It defines a certain order of actions: first, verifying all the previous conditions, then, the changes on the contract's state should be done and, finally, interacting with other contracts. In accordance with this principle, the interactions with other contracts should be, whenever possible, the last step in every function. This is because the moment a contract interacts with another contract, including the ether transactions, it gives control to the other contract. This gives the called contract the possibility of executing potentially damaging actions.

For instance, an attack known as re-entrancy, where an invoked contract returns the call to the current contract, before the completion of the first invocation of the function that contains the call. This can lead to an undesired execution behaviour of the functions, modifying the state variables to unexpected values or repeating the operations (such as the sending of funds).

4 Analysis of Vulnerabilities and Rules for its Detection

In this section a summary of the security vulnerabilities in the Ethereum platform and its high level language Solidity is being provided. The second part has its focus on security from a contract developer point of view, and some code patterns which were implemented in the tool are being described.

4.1 Vulnerabilities

The causes of the vulnerabilities are organized in the taxonomy proposed by the author Atzei et al. [2], classifying them depending on the context into: programming high level languages, virtual machine (EVM) and the particularities of blockchain.

- High level programming languages: This category assesses the weaknesses or flaws of the programming languages in addition to the misuse of the language made by the developers. In this piece of work, Solidity language vulnerabilities are being presented.
- Virtual Machine: In this category the vulnerabilities related to the virtual machine where the smart contracts are executed are being grouped.
- Blockchain: It involves the vulnerabilities of blockchain's infrastructure.

The vulnerabilities related to the virtual machine and to blockchain's infrastructure are common to every programming language. Nevertheless, the vulnerabilities related to the programming languages are particular to each one of them, therefore, it surges the necessity of counting with an extensible mechanism to define static program analysis rules that depend on the language which is being analysed.

All the vulnerabilities listed in Table 1 can be exploited to carry out attacks which, for example, steal money from the contracts. It is the case of the vulnerability known as re-entrancy, that allowed the attackers to steal 50 million dollars from the DAO organization.

Table 1 Vulnerabilities classification.

High level languages (Solidity)	Call to the unknown Exceptions disorder Send without gas Types conversion Re-entrancy Keeping Secrets
EVM	Immutable errors Lost ether in the transactions Stack size limited
Blockchain	Unpredictable state Random generation Time restrictions

4.2 Code Patterns and Static Program Analysis Rules

This section focuses on security from a developer's point of view. The aforementioned high level programming languages classification, the topics related to the Solidity code are subdivided into:

- Security: issues that lead to attacks from an user or malicious contract.
- Functional: they cause the violation of a scheduled functionality.

This distinction between the functional problems and the security ones is presented due to the fact that the first pose a problem even without an adversary (even though a malicious external actor can aggravate the situation), while the last ones do not. In Table 2 a list of analysis rules and its severity is being shown.

Table 2 Static program analysis rules.

Security	Equality on the balance	Average
	Non-verified external call	High
	Use of send instead of transfer	Average
	Denial of a service because of an external contract	High
	Re-entrancy	High
	Malicious libraries	Low
	Use of tx.origin	Average
	Transfer of all the gas	High
Functional	Integer division	Low
	Blocked money	Average
	Non-verified maths	Low
	Dependence on the timestamp	Average
	Unsecure inference	Average

4.2.1 Security

Equality on the balance. Avoiding the verification of the strict equality of a balance, due to the fact that an adversary could send ether to any account through minery or through the selfdestruct call. The pattern detects comparison expressions with == and != that contain this.balance as right or left side. In Listing 1 example code is shown.

Listing 1 Avoid the verification of the balance equality.

```
if(this.balance == 1 ether) { ...} // Not Ok
if(this.balance >= 1 ether) { ...} // Ok
```

Non-verified external call. Wait until the calls to an external contract fail. When sending ether, verify the return's value and deal with the errors. The recommended way of carrying out the ether transactions is with the primitive transfer.

The pattern detects a call to an external function (call, delegate or send) that is not inside an if sentence. In Listing 2 some lines of vulnerable code are shown.

Listing 2 Fragment of vulnerable code.

```
addr.send(1 ether); // Not Ok
if(!addr.send(1 ether)) revert ; // Ok
addr.transfer(1 ether); // Recommended
```

Use of send instead of transfer. The recommended way of completing payments is addr.transfer(x),that automatically throws an exception if the transaction is unsuccessful, avoiding the aforementioned problem. The pattern detects the keyword send.

Denial of a service because of an external service. A conditional sentence (if, for, while) should not depend on an external call due to the fact that the invoked contract can fail (throw or revert) permanently, not allowing the invoker to complete the execution.

In Listing 3, the invoker expects that the external contract returns an integer, but the real implementation of the external contract can generate an exception in some cases or in all of them.

Listing 3 Fragment of vulnerable code.

```
function dos(address oracleAddr) public {
    badOracle = Oracle(oracleAddr);
    if(badOracle.answer() < 1) revert;
}
```

This rule contains multiple patterns:

- An if sentence with an external call in the condition and a throw or revert in the body.
- A for or if instruction with a call to an external function in the condition.

In Listing 4 a possible fraudulent implementation of the answer() method of the Oracle contract is presented.

Listing 4 Answer() method of the Oracle contract.

```
function answer() public returns int8 {
  throw;
}
```

Re-entrancy. In Section 3.2, it was stressed the importance of implementing the Check-Effects-Interaction pattern for mitigating the re-entrancy attacks. In Listing 5, a method exposed to the aforementioned vulnerability can be observed.

Listing 5 Method exposed to re-entrancy attack.

```
mapping (address => uint) balances;

function withdraw () public {
    uint balance = balances[msg.sender];
    if(msg.sender.call.value(balance)() {
        balances [msg.sender] = 0;
    }
}
```

The contract on msg.sender can obtain multiple refunds and recover all the ether of the contract put as an example through a recursive call to withdraw before its fee descends to 0. In Listing 6 it is showed the same functionality but taking into account the pattern: first the invariants are verified, then the internal state is updated and, finally, they communicate with external entities. The pattern detects a call to an external function that is followed by a call to an internal function.

Malicious libraries. Third parties' libraries can be malicious. Avoid the external dependencies or make sure that the third parties' code only implements the desired functionality. The pattern simply detects the keyword library.

Listing 6 Implementation of Check-Effects-Interaction.

```
function withdraw() public {
  uint balance = balances[msg.sender];
  balances[msg.sender] = 0;
  msg.sender.transfer(balance);
}
```

Use of tx.origin. The contracts can call the public functions of the rest. tx.origin is the first account in the call chain (not a contract); msg.sender is the immediate invoker. For example, in an A $\rightarrow$ B $\rightarrow$ C call chain, from C's point of view, tx.origin is A, and msg.sender is B. Use msg.sender instead of tx.origin for authentication. The pattern detects the environment variable tx.origin.

Transfer of all the gas. Solidity provides a myriad of ways of transferring ether. addr.call.value(x)() transfers x ether and redirects all the gas to addr, which could generate vulnerabilities such as re-entrancy. The recommended way of transfering ether is addr.call.value(x), that only provides an allowance of 2300 units of gas to the recipient. The pattern detects the functions whose name is call.value and whose argument list is empty.

4.2.2 Functional

Integer division. Solidity does not admit floating points types or decimals. For the integer division, the quotient is rounded down. Be aware of that, especially when calculating the ether or token quantities. The pattern detects the division (/) where the numerator and denominator are literal numbers.

Blocked money. The contracts programmed for receiving ether should implement a way of withdrawing it, in other words, call a transfer (recommended), send or call.value at least once. The pattern detects contracts that contain a payment function (payable), but that contain none of the aforementioned withdrawal functions.

Non-verified maths. Solidity is prone to suffer integer overflows. The overflow produces unexpected effects and could provoke a loss of funds if it is exploited by a malicious account. Use the SafeMath library [19] that verifies the overflows. The pattern detects arithmetic operations +.-.*, that are not inside of a conditional declaration.

Dependence on the timestamp. The miners can manipulate the environment variables and they are likely to do so if they can benefit from it. Use the block number and the average time between blocks for estimating the current time or use secure randomisation sources. The pattern detects the environment variable now.

Unsecure inference. Solidity admits type inference: the type of i in var = 42; is the smallest integer type that is enough for storing the value of the right side (uint8). Consider for loop in Listing 7:

Listing 7 Example of unsecure type inference.

```
for (var i = 0; i < array.length; i++) { ... }
```

The type of i is inferred to uint8. If array.length is bigger than 256, a overflow will occur. Explicitly define the type when declaring integer variables. The pattern detects allocations where the left side is a var and the right side is an integer (that matches with ^[0-9]+$).

5 Tool

In this piece of work a new project called OpenBalthazar is being proposed. It aims to raise a strategy for the detection of vulnerabilities in high level programming languages (Solidity) based on automatized static analysis.

5.1 Architecture

The static code analysis ensures complete coverage without executing the program and fast enough in a reasonably sized code. It usually includes three stages:

- Construction of an intermediate representation like the abstract syntax tree (AST), for a more thorough analysis in comparison with the analysed text.
- Enrichment of the AST with additional information using algorithms and language processing techniques.
- Detection of vulnerabilities, a repository of patterns that define the criteria when talking about intermediate language terms.

As it is shown in Figure 1, the tool is composed of four components:

Language Selector. This component is responsible for detecting the programming language in which the specification is written (source code) and determines the component that will be instantiate, through dynamic mechanisms that framework .NET (reflection) presents, for processing a request.

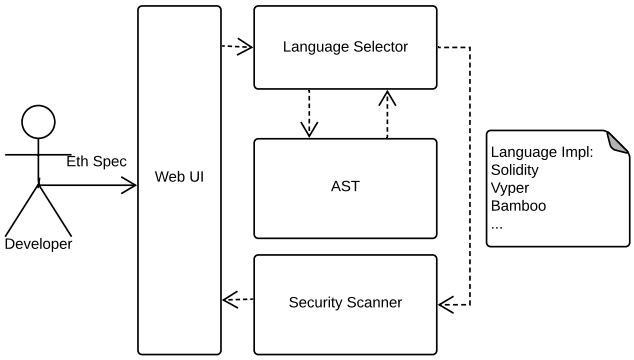

Figure 1 Principal components of the tool.

AST. This component builds an abstract syntax tree of a programming language, such as Solidity. Each tree's node has information of the language construction, for instance, sentences, condition structures, exception management, and applying different routes to the AST, the desired information can be extracted.

Security Scanner. This component takes as an input an AST of a specific language and executes the verification rules defined in a pattern repository for that language.

Web UI. This is the component that the developer interacts with. It provides tools for editing source code, such as file management, syntax coloring, line enumeration , etc. This frontend component was developed with the ReactJS javascript framework.

5.2 Implementation

OpenBalthazar is a static program analysis web tool for Ethereum platform smart contracts implemented with Microsoft.NET Core 3.1 in backend components. Security scanner component executes lexical and syntactic analysis in the Ethereum supported languages source code. Currently, Solidity, which is the *de facto* language of the industry, is implemented; still, the tool is extensible and the rest of the languages, such as Vyper or Bamboo, can be incorporated through the envisaged extensibility mechanisms.

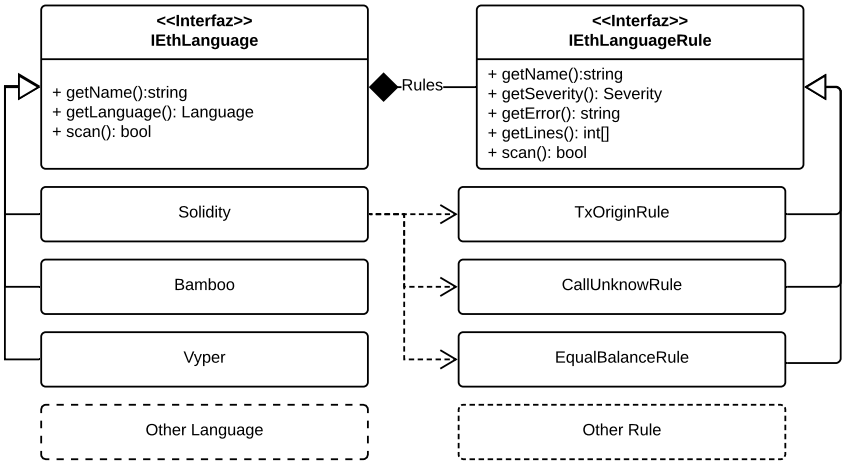

Figure 2 Extensibility mechanism.

To add a new language to the tool or add a new analysis rule to an existing language, the IEthLanguageRule and IEthLanguage interfaces ought to be implemented, respectively. Figure 2 shows how the Solidity language was implemented and how to proceed for incorporating the rest of the languages. In the same way, each language has a set of analysis rules that implement the IEthLanguageRule interface. In the `scan()` method the algorithm for the detection of a particular vulnerability is specified, indicating the error message and the specific severity.

In AST component, ANTLR 4 is used in addition to a Solidity grammar for generating the XML analysis tree (AST). In release 4.2, Antlr introduced the *visitor* and *listener* mechanisms that lets you implement DOM visiting or SAX-analogous event processing of tree nodes. This feature improves group translation operations by patterns in the tree rather than spreading operations across listener event methods. Another important idea is that, since we are talking about parse trees not abstract syntax trees, we can use concrete patterns instead of tree syntax. For example, we can say `x = 0;` instead of `AST (= x 0)` where the ";" would probably stripped before it went into the AST.

The vulnerability patterns are detected through the usage of parse tree patterns and XPath queries in the IR. Thus, OpenBalthazar provides a full coverage: the analised code is fully translated to the IR, and all of its elements can be reached through the XPath's selection mechanism. The line numbers are stored as XML attributes and help locating the findings in the source code. The tool can be expanded for admitting other smart contracts languages adding the ANTLR grammar and a pattern database.

If the '`use of tx.origin`' vulnerability is considered, the aim of the rule is detecting constructions that prove the existence of these identifiers in a contract. The analysis tree of this construction is shown in Figure 3.

The corresponding XPath pattern is shown in Listing 8. In this case, false positives are not expected, due to the fact that the aimed construction can be precisely described with XPath. The more complex rules cannot be precisely described with XPath , which may leads to false positives. In some scenarios false positives are related with the limitation of de static code analysis. Nevertheless, if the re-entrancy rule is considered, OpenBalthazar informs about the violations to the `Checks-Effects-Interactions` pattern, that not always leads to a re-entrancy (false positives).

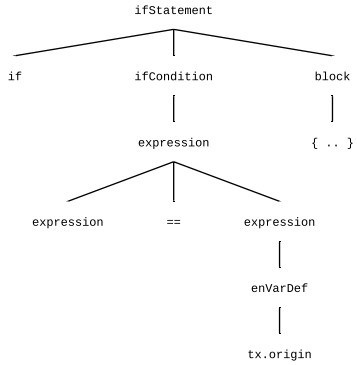

Figure 3 Analysis tree.

Listing 8 Fragment of the method Scan() corresponding to the class TxOriginRule.

```
IParseTree tree = solidityParser.sourceUnit();
ParseTreePattern pattern =
    solidityParser.CompileParseTreePattern("tx.origin",
    SolidityParser.RULE_expression);
IList<ParseTreeMatch> matches =
    pattern.FindAll(tree, "//expression");
foreach (ParseTreeMatch match in matches) {...}
```

In Listing 9 is shown some parse tree patterns.

6 Use Case: Etherscan Scanning

One of OpenBalthazar tools's strengths is the possibility of regaining the source code of the verified contracts of the EtherScan platform[1]. EtherScan is a platform for Ethereum that provides block exploration, search, analysis, source code's verification and APIs services. The source code's verification provides transparency for the developers who interact with smart contracts. When loading the source code, EtherScan will compare the compiled code with the one deployed in the blockchain.

OpenBalthazar provides the possibility to scan all the smart contracts whose source code has been published in EtherScan[2]. To accomplish this task, OpenBalthazar connects to the EtherScan API, retrieves the source code for published contracts, and then applies the set of rules predefined in Section 4.2. The result of the analysis is presented in a dashboard that allows examining the degree of vulnerability of the set of analyzed contracts.

The results of the analysis carried out by OpenBalthazar are shown in Figure 4. As it can be seen, 7.4% of the contracts are vulnerable. In this context and with the aim of simplifying the analysis, the results of the vulnerability known as re-entrancy will be shown, since, as

[1] `https://etherscan.io`

[2] At the time of writing this article, there are over one million smart contracts deployed on Ethereum. Out of these contracts, over 49.000 have been verified on EtherScan and 5462 contracts are available for viewing source code.

Listing 9 Fragments of parse tree patterns.

```
//EqualsBalanceRule
ParseTreePattern pattern =
    solidityParser.CompileParseTreePattern("this.balance (== | !=)
    <expression>", SolidityParser.RULE_expression);
IList<ParseTreeMatch> matches =
    pattern.FindAll(tree, "//expression");

//TimestampDependenceRule
ParseTreePattern pattern =
    solidityParser.CompileParseTreePattern("now",
    SolidityParser.RULE_expression);
IList<ParseTreeMatch> matches =
    pattern.FindAll(tree, "//expression");

// ReentrancyRule
ParseTreePattern pattern =
    solidityParser.CompileParseTreePattern("<expression>
        .<identifier>(<functionCallArguments>)",
        SolidityParser.RULE_expression);
IList<ParseTreeMatch> matches = pattern.FindAll(tree, "//expression");
foreach (ParseTreeMatch match in matches)
{
        if(match.Get("identifier").GetText().Equals("send") ||
                match.Get("identifier").GetText().Equals("value"))
        ...
}
```

mentioned in Section 3, it is considered the most important in terms of security. As of today, and with the improvements that have been incorporated into the language, it remains the responsibility of the programmers and auditors to mitigate their risk.

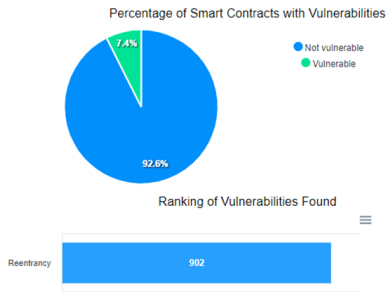

Figure 4 OpenBalthazar dashboard.

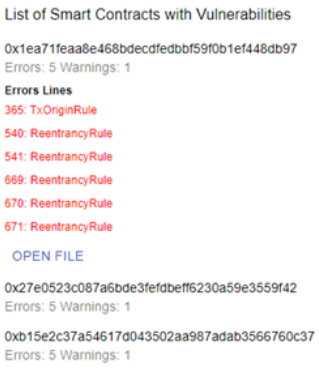

Figure 5 List of vulnerable contracts.

OpenBalthazar found 902 re-entrancy issues, it means contracts have calls to third-party contracts that could potentially be violated. In the right pane, shown in Figure 5, the addresses of vulnerable contracts are listed. Pressing on the address of the contract displays the list of vulnerabilities found and a button to open the source code of the contract in the editor of OpenBalthazar.

The facility for obtaining a deployed contract's source code in the principal network and the possibility of executing a complete analysis that detects the vulnerabilities transforms into a very powerful but hazardous characteristic, due to the fact that it enables the attacker to scan the network in the search of vulnerable contracts and prepare new contracts that exploit them. From the other side, a security specialist could use the vulnerabilities scan to activate the defense mechanisms that are usually implemented in the smart contracts for mitigating adverse situations.

7 Conclusions and Future Extensions

In the new programming paradigm and decentralized apps presented in the crypto ecosystem, where cryptography, distributed computing and the incentive mechanisms come together, the central issue that the high level programming languages face in the Ethereum platform is that the most complex language structures attempt against the security mechanisms and add more confusion to the contract developers, for example Solidity has 3 different way to send money to an account. Security methodologies such as SAMM [20] encourage the use of simple constructs to avoid introducing vulnerabilities. Hence, languages that are currently in the development stages has simpler structures and are focused on security concern. Until these new languages are available, it is necessary that tools are designed and constructed that helped in the chore of labour of programming, deploying and monitoring the smart contracts in terms of obtaining the best result possible.

The tool presented in this paper allows the contract developers to discover code vulnerabilities before deployment. In the case that the obtained result delivers high severity vulnerabilities, the professional will have to act consequently and try to stop the contract from providing services before it can be attacked. Another scenario where this tool aims to contribute is in Software Protection. In this context, we focus on `Man At The End` (MATE) attacks where the intruder may be a member of the development team or someone who was part of at some point.

EtherScan use case demonstrate how obtaining the source code of a group of deployed contracts in the principal network and how executing a complete analysis that detects yours vulnerabilities. However, only 0.5% of the source code is available in EtherScan to analyze, so it is essential to incorporate the EVM bytecode transformation, obtained from the principal network, into a high-level language (such as Solidity), in order to later carry out the aforementioned analyzes. Due to the novelty of this area we are still lacking good reverse-engineering tools to use, so we plan to build a Solidity decompiler for EVM bytecode.

Hereafter, some analysis included in the line of research are being succinctly described.

- Construction of an intermediate representation, enriched by additional information using algorithms and language processing techniques to facilitate the static program analysis of the contracts.
- Static program analysis of the source code and decision of use of security patterns.
- Implementation of a tool that allows the automatization of the process of analysis and detection of security flaws.

Researchers will continue with studies in this field aiming at improving skills and carrying out a follow-up on the new threats, vulnerabilities and cyberattacks regarding the deployment and execution of smart contracts. It is also planned to continue with the following future work:

- Generalization for other blockchain platforms that support smart contracts.
- Development of a Solidity decompiler for EVM bytecode.
- Dynamic analysis of the source code.
- Analysis of the security aspects that stem from the interoperability with other platforms.

References

1 M. Alharby and A. van Moorsel. Blockchain-based smart contracts: A systematic mapping study. *Fourth International Conference on Computer Science and Information Technology (CSIT-2017)*, 2017. arXiv:1710.06372v1.

2 N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on ethereum smart contracts sok. In *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*, page 164–186, 2017. doi:10.1007/978-3-662-54455-6_8.

3 M. Bartoletti and L. Pompianu. An empirical analysis of smart contracts: Platforms, applications, and design patterns. In *Financial Cryptography and Data Security*, 2017.

4 K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, 2016. doi:10.1145/2993600.2993611.

5 V. Buterin. Ethereum: A next-generation smart contract and decentralized application platform, 2014. URL: https://github.com/ethereum/wiki/wiki/White-Paper.

6 M. Coblenz. Obsidian: A safer blockchain programming language. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, page 97–99, 2017.

7 ConsenSys. Mythril. https://github.com/ConsenSys/mythril, 2018.

8 T. Cook, A. Latham, and J.H. Lee. Dappguard: Active monitoring and defense for solidity smart contracts. https://courses.csail.mit.edu/6.857/2017/project/23.pdf, 2017.

9 Ethereum. Solidity. https://media.readthedocs.org/pdf/solidity/develop/solidity.pdf, 2018.

10 S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar. Online detection of effectively callback free objects with applications to smart contracts. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158136.

11 E. Hildenbrandt, E. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, and G. Rosu. Kevm: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217, 2018.

12 Y. Hirai. Defining the ethereum virtual machine for interactive theorem provers. In *Financial Cryptography and Data Security*, page 520–535. Springer International Publishing, 2017.

13 S. Kalra, S. Goel, M. Dhawan, and S. Sharma. Zeus: Analyzing safety of smart contracts. In *25th Annual Network and Distributed System Security Symposium*, 2018. doi:10.14722/ndss.2018.23092.

14 L. Luu, D.H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, page 254–269, 2016. doi:10.1145/2976749.2978309.

15 A. Mavridou and A. Laszka. Designing secure ethereum smart contracts: A finite state machine based approach, 2017. arXiv:1711.09327.

16 S. Nakamoto. Bitcoin: Un sistema de efectivo electrónico usuario-a-usuario, 2008. URL: http://bitcoin.org/bitcoin.pdf.

17 I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, page 653–663, 2018. doi:10.1145/3274694.3274743.

18 Trail of Bits. Manticore. https://github.com/trailofbits/manticore, 2018.

19 OpenZeppeling. Safemath, 2019. Accessed: 2019-05-03. URL: https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/math/SafeMath.sol.

20 OWASP. Software Assurance Maturity Model: A guide to building security into software development. Version 1.5, 2020. URL: https://owasp.org/www-pdf-archive/SAMM_Core_V1-5_FINAL.pdf.

21 F. Schrans, S. Eisenbach, and S. Drossopoulou. Writing safe smart contracts in flint. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*, page 218–219, 2018. doi:10.1145/3191697.3213790.

22 I. Sergey, V. Nagaraj, J. Johannsen, A. Kumar, A. Trunov, and K.C.G. Hao. Safer smart contract programming with scilla. *Proc. ACM Program. Lang.*, 2019. doi:10.1145/3360611.

23 SmartDec. Smartcheck: a static analysis tool that detects vulnerabilities and bugs in solidity programs, 2019. Accessed: 2019-05-03. URL: https://github.com/smartdec/smartcheck.

24 P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, page 67–82, 2018. doi:10.1145/3243734.3243780.

25 G. Wood. Ethereum: A secure decentralised generalised transaction ledger, 2017. URL: https://ethereum.github.io/yellowpaper/paper.pdf.

26 I. Wöhrer and U. Zdun. Smart contracts: Security patterns in the ethereum ecosystem and solidity. In *2018 International Workshop on Blockchain Oriented Software Engineering*, 2018.

27 E. Zhou, S. Hua, B. Pi, J. Sun, Y. Nomura, K. Yamashita, and H. Kurihara. Security assurance for smart contract. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5, 2018.