Evaluation of Smart Contract Vulnerability Analysis Tools: A Domain-Specific Perspective

Bahareh Lashkari and Petr Musilek *

Electrical and Computer Engineering, University of Alberta, Edmonton, AB T6G 1H9, Canada; bahareh1@ualberta.ca
* Correspondence: pmusilek@ualberta.ca

Abstract: With the widespread adoption of blockchain platforms across various decentralized applications, the smart contract’s vulnerabilities are continuously growing and evolving. Consequently, a failure to optimize conventional vulnerability analysis methods results in unforeseen effects caused by overlooked classes of vulnerabilities. Current methods have difficulty dealing with multifaceted intrusions, which calls for more robust approaches. Therefore, overdependence on environment-defined parameters in the contract execution logic binds the contract to the manipulation of such parameters and is perceived as a security vulnerability. Several vulnerability analysis tools have been identified as insufficient to effectively identify certain types of vulnerability. In this paper, we perform a domain-specific evaluation of state-of-the-art vulnerability detection tools on smart contracts. A domain can be defined as a particular area of knowledge, expertise, or industry. We use a perspective specific to the area of energy contracts to draw logical and language-dependent features to advance the structural and procedural comprehension of these contracts. The goal is to reach a greater degree of abstraction and navigate the complexities of decentralized applications by determining their domains. In particular, we analyze code embedding of energy smart contracts and characterize their vulnerabilities in transactive energy systems. We conclude that energy contracts can be affected by a relatively large number of defects. It also appears that the detection accuracy of the tools varies depending on the domain. This suggests that security flaws may be domain-specific. As a result, in some domains, many vulnerabilities can be overlooked by existing analytical tools. Additionally, the overall impact of a specific vulnerability can differ significantly between domains, making its mitigation a priority subject to business logic. As a result, more effort should be directed towards the reliable and accurate detection of existing and new types of vulnerability from a domain-specific point of view.

Keywords: blockchain; distributed ledger technology; security analysis; vulnerability detection

1. Introduction

In essence, smart contracts are self-executing collections of codified agreements deployed on decentralized networks of blockchain. Smart contracts support decentralization through automated execution, blockchain integration, and immutable protocols. They can also contribute to trust through transparency, automated execution, immutable history, and cryptography. As event-driven programs, smart contracts facilitate trusted transactions, allowing anonymous parties to exchange digital assets or data [1]. The smart contract is the main pillar of Ethereum and is widely adopted in various business domains. As a proclaimed framework for integrating the execution of smart contracts, Ethereum offers great capacity in the development of decentralized applications (DApps). Most Ethereum smart contracts are written in Solidity, a high-level object-oriented programming language, and then compiled into bytecode for execution using the Ethereum Virtual Machine (EVM). The EVM creates a level of abstraction between the executing code and the machine on which it runs, isolating the DApps and their corresponding hosts [2].
Solidity smart contracts, like any other program, are susceptible to bugs, vulnerabilities, and security flaws caused by a lack of security patches [3]. However, smart contracts are associated with immutability features that preclude any modification after deployment, thereby enforcing the “code is law” principle [4]. Consequently, it is necessary to verify that a smart contract does not contain hidden programming defects that could have severe security implications. These defects could be maliciously exploited by an attacker to initiate unintended processes the smart contract was not set up to perform. As more blockchain-based services are created, there is greater emphasis on the reliability of smart contracts. In an analysis of approximately one million Ethereum smart contracts, 34,200 have been identified as vulnerable [5]. Therefore, the early detection of vulnerabilities prior to the deployment of smart contracts is extremely important. Therefore, automated methods and tools have been developed to analyze smart contracts to detect vulnerabilities and bad coding practices [6,7].

The performance of comparable vulnerability analysis tools varies substantially among different smart contract analysis studies. This is explicable through different experimental settings and performance metrics. Consequently, evaluating the detection competencies of these tools is very difficult.

Our research has shown that existing research on the effectiveness of benchmark vulnerability analysis tools has not considered the application domain or the purpose of the contracts under assessment. Although correlations between smart contract categories and underlying vulnerabilities were identified in a recent study [8], security defects in energy contracts were not factored into the evaluation. To fill this gap, the primary focus of this study is the analysis of smart contracts deployed in transactive energy systems. The proposed vulnerability analysis workflow allows the investigating of the associations between various contract types and corresponding defects. Consequently, a domain-specific evaluation of both static and dynamic analysis tools is carried out using curated contracts as the benchmark. We have used state-of-the-art vulnerability analysis tools on energy and non-energy smart contracts to answer the following questions:

- Which tool performs best in analyzing the vulnerability of smart energy contracts?
- Do energy contracts contain more vulnerabilities and poor coding practices compared to other classes of contracts?
- Are there domain-specific security flaws that existing tools fail to detect?
- Are certain state-of-the-art vulnerability analysis tools more effective in specific application domains?
- Is there any benefit to developing domain-specific vulnerability detection tools?

2. Background and Motivation

Given the global interest in blockchain, researchers are investigating the vulnerabilities of smart contracts as they form the cornerstone of blockchain development. Zeus [9] evaluated over 22,400 solidity smart contracts and found that approximately 94.6% of them had security defects. Another database containing blockchain security defects constructed by researchers denoting smart-contract-related instances accounts for 22% of all incidents [10,11]. Smart contracts have complex time and order dependencies. Hence, inconsistencies in the logic of a contract code contribute to vulnerabilities and the incorrect execution of the smart contract [3]. As a result, blockchain-oriented software engineering is necessary to avoid defects and ensure effective programming practices.

Smart contract defects encompass a combination of security-related problems as well as design deficiencies that could impede the implementation or increase the possibility of future vulnerabilities or failures. Defects often result in the smart contract producing an erroneous or undesirable outcome or causing it to operate in directions that were not intended. Hence, the identification and elimination of these defects improve the software reliability and development process [12].

The severity of defects can be categorized into three distinct types based on their implications and the likelihood of causing a financial loss. Direct monetary losses are classified
as higher-grade defects, whereas risks of monetary losses that arise from unauthorized or unintended contract operation are rated as medium-severity defects. Defects that cause only superficial problems (such as poor accessibility or resource depletion), do not interfere with regular operations, and do not result in financial losses, are classified as low-severity defects. However, the broad spectrum of defects makes it challenging to define them precisely. Hence, defect patterns are incorporated as a conceptual representation to depict defect characteristics and attributes [13].

A pattern is an abstraction from a concrete design that recurs in predetermined, nonarbitrary instances. Patterns encompass an overarching outline of a persistent issue and an applicable solution with explicit goals and limitations. Different assessment expressions for smart contracts can be implemented by pattern design. Each defect type could entail a distinct collection of code segments and impose a varying set of complications. Consequently, the development of defect patterns is crucial for accurately and effectively conveying these security defects. Code elements (CE), relationship restrictions (RR), and element restrictions (ER) make up the majority of the defect pattern. The pattern promotes the evaluations required to improve the reusability of the contract code. Consequently, the validation and updating process for smart contracts can be simplified by reusing the corresponding verification rules [14].

However, as smart contract structures becomes increasingly complex, a multitude of vulnerabilities are emerging and expert-defined rules cannot keep up with constant vulnerability updates. The resulting overlay of expert-defined rules leads to substantial false alarm rates. This makes the rule-based detection of general vulnerabilities impractical. Several predeployment smart contract security analysis frameworks have been developed in response to the substantial economic implications of defects in smart contracts [15]. However, the primary concern remains the efficient and timely detection of smart contract vulnerabilities. As the functionality of smart contracts expands, their new attributes contribute to new types of security flaws. These flaws, in turn, allow for complex attacks such as the new ERC777 reentrancy and cross-chain attacks [16].

Integration and acceptance tests developed for defect detection require extensive technical knowledge of the blockchain framework. Any reliance on environment-defined parameters in the execution logic of the contract binds the contract to the manipulation of these parameters and is considered a security flaw [17]. Addressing prevalent smart contract vulnerabilities requires a syntactic and semantic understanding about the compromised contract. The literature suggests that smart contract classification with respect to the application domain and transaction context provides additional information on the syntactic and semantic properties of the domain [18]. To identify new vulnerabilities, researchers must continue to expand their detection prospects for the methods encoded in smart contracts that contain their business logic. This study examines the benefits of incorporating business logic and domain knowledge in the design and development of vulnerability analysis tools. We anticipate that its results will help to establish the groundwork for innovative solutions for the domain-specific classification and vulnerability analysis of smart contracts.

3. Vulnerability Analysis Tools

The smart contract vulnerability analysis tools can be classified into two main categories, static and dynamic [19]. Static analysis is the process of analyzing a program without executing it. This method can be applied to both source code and bytecode representations of smart contracts. Dynamic analysis, on the other hand, is a run-time environment method that screens the contract’s behavior during execution for potential vulnerabilities or security breaches. Fuzzing is a form of dynamic analysis that passes defective data to the executed smart contract to examine its response to the malformed input. Using the Smartbugs [20] interface, this study targets a series of static and dynamic analysis tools including Mythril, Slither, Smartcheck, Honeybadger, Osiris, Solhint, Oyente and Conkas, and Confuzzius.
• **Honeybadger** [20] is an Oyente-based honeypot detection system that relies on symbolic execution and well-defined heuristics.

• **Osiris** [21] is another Oyente-based tool that claims to be capable of finding previously unknown critical vulnerabilities in some cases. Using symbolic execution coupled with taint analysis, Orisis offers the detection of a diverse range of defects with improved detection specificity.

• **Solhint** [22] was proposed as a linting tool for solidity smart contracts. Using pre-configured patterns and rulesets, it offers a good coverage of known security defects.

• **Smartcheck** [23] validates contracts against XPath queries using their XML representation. This intermediate representation facilitates the localization of detections across the source code to provide complete code coverage.

• **Oyente** [24] leverages operational semantics to search for execution traces in the code where the transaction sequence has affected the Ether flow or the result of computations is dependent on timestamps.

• **Conkas** [20] is another static analysis method that incorporates control flow graphs (CFGs) as intermediate representations for symbolic execution. If the user does not specify the dependency files, Conkas is not capable of tracing the vulnerabilities encapsulated in the library files.

• **Mythril** [25] is intended to uncover common security issues and cannot detect the concerns ingrained in business logic. It incorporates concolic, taint, and control flow analysis to search for attributes that cause vulnerabilities in smart contracts.

• **Slither** [26] employs its own internal representation language for an intermediate representation and performs data flow and taint analysis for information retrieval and refinement. Slither determines a set of predefined analyses and a static single assessment (SSA) in a multistage procedure. With the abstract syntax tree (AST) as input, it can provide enhanced information to the other components and simplify the computation of a diverse array of code analysis.

• **Confuzzius** [27] is the first hybrid fuzzer that integrates evolutionary fuzzing with constraint solving to explore both shallow and deep fragments of contracts. Using dynamic data dependency analysis, Confuzzius can derive transaction sequences that lead to states with implicit security flaws.

4. **Domain-Specific Perspective**

Several attempts have been made to investigate smart contract vulnerabilities; however, no research has examined the correlation between different types of contract and their corresponding security flaws. A recent study reported a positive association between smart contract categories and their vulnerabilities. Giacomo et al. [8] suggest that gambling contracts are consistently associated with bad randomness. However, the smart contract classification in this study fails to consider many other application domains, including transactive energy systems.

Smart contracts have a broad range of applications in the energy industry, including monitoring the production, distribution, and consumption of energy. These codified agreements can also be used for the monitoring of carbon credits and the administration of peer-to-peer energy markets. These markets represent an architectural transition from traditional centralized energy distribution models. They are well positioned to transform the energy landscape, with smart contracts enabling secure and transparent transactions between energy suppliers and end users. The inherent trust and transparency of blockchain technology, combined with the automated characteristics of smart contracts, fosters a decentralized energy exchange ecosystem that allows individuals to participate in direct energy trading, consumption, and reimbursement. In a comparable study, smart contracts used in transactive energy systems are examined to derive the business logic and disclose the features of the contracts and the verbal intuition of the energy contracts [28].
Energy smart contracts encapsulate the terms and conditions governing a contract among counterparties to regulate the transactions of electricity such as processing market bids, billing, and optimal pricing for double-auction.

The code snippet illustrated in Listing 1 is an energy contract instance called EnergyTrading that authorizes energy transactions by establishing the price and availability using setEnergyPrice and setEnergyAmount functions. This contract offers another function called SellEnergy that allows for keeping track of the energy sold and ensures that the balance does not exceed the available energy. EnergySold is another event used in this contract to monitor sales.

**Listing 1.** Sample energy smart contract.

```solidity
pragma solidity ^0.8.0;

contract EnergyTrading {

    uint256 public energyPrice;
    uint256 public energyAmount;

    event EnergySold(address seller, uint256 amount, uint256 price);

    constructor(uint256 initialPrice) public {
        energyPrice = initialPrice;
    }

    function setEnergyPrice(uint256 newPrice) public {
        energyPrice = newPrice;
    }

    function setEnergyAmount(uint256 newAmount) public {
        energyAmount = newAmount;
    }

    function sellEnergy(uint256 amount) public {
        require(amount <= energyAmount, "Not enough energy available for sale ");
        energyAmount -= amount;
        msg.sender.transfer(amount * energyPrice);
        emit EnergySold(msg.sender, amount, energyPrice);
    }
}
```

The DOT-formatted control flow graph of the EnergyTrading contract is depicted in Figure 1 using Surya [25]. This is a common method intended to perform manual contract analysis, inspecting the contract’s complexity using control flow graphs and inheritance graphs. This approach to contract development minimizes complexity while avoiding security breaches. As shown in Figure 1, the contract functions are not interconnected and cannot be called by external contracts. However, research shows that this is not a recurring pattern for energy contracts. This is because these contracts feature various levels of inheritance that embed complex computations. Additionally, energy contracts encompass a plethora of external and internal calls that are related to the complexity of the contract [29]. Consequently, smart contract vulnerability detection research would benefit from an analysis of the effectiveness of vulnerability detection methods in connection with certain vulnerabilities and application domains.
As a result of different experimental setup and performance criteria, the performance of benchmark tools varies between different vulnerability analysis studies. Tailoring the experimental setup to a target domain enables the search for critical security breaches in contracts with comparable violation structures and behavioural patterns.

5. Analysis

As shown in Figure 2, the analysis in this study is carried out using three distinct data sets, each containing 20 smart contracts. The first data set, used to benchmark smart contracts, is the curated data set of SB [20]. The primary objective of this data set is to provide a collection of previously identified vulnerabilities that can be used to evaluate the performance of analysis tools. The appointed contracts encompass vulnerabilities such as reentrancy, access control, time manipulation, bad randomness, and arithmetic issues. The other two data sets encompass 20 manually labeled energy and 20 non-energy contracts from Etherscan. As stated earlier, energy contracts are used to automate energy trading activities, while non-energy contracts are formulated to regulate transactions in other application domains, such as gaming, gambling, finance, and so on. The vulnerability analysis is carried out on both energy and non-energy categories using all analysis tools. The analysis can be broken down into two segments: benchmark and energy, each of which is discussed in the sections that follow.

5.1. Benchmark

The first part of the experiment serves as a benchmark to evaluate cutting-edge smart contract analysis tools on most critical vulnerabilities using the curated data set. Accordingly, the detection rate of each tool can be evaluated using true positive, false
negative, average run-time, and accuracy metrics. As illustrated in Table 1, across all the tools, Slither identifies most defects and has the highest accuracy and the shortest runtime. It is widely regarded as a highly effective tool that incorporates code and constraint-solving mechanisms. Therefore, the obtained results are consistent with previous research on the evaluation of smart contract analysis tools [30]. Honeybadger, with symbolic execution and constraint solving, is the least accurate method, followed by Solhint with code instrumentation. Mythril is the second slowest method with mediocre accuracy, showing the incompetence of symbolic execution paired with constraint solving [19].

**Table 1.** Analysis results on curated data set.

<table>
<thead>
<tr>
<th>Vulnerabilities</th>
<th>Metrics</th>
<th>Confuzzius</th>
<th>Conkas</th>
<th>Honeybadger</th>
<th>Mythril</th>
<th>Osiris</th>
<th>Oyente</th>
<th>Slither</th>
<th>Smartcheck</th>
<th>Solhint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bad randomness #1</td>
<td>Duration</td>
<td>8.737881184</td>
<td>43.17423201</td>
<td>42.6293509</td>
<td>117.7532399</td>
<td>35.85272484</td>
<td>12.72189796</td>
<td>2.460914135</td>
<td>7.229845047</td>
<td>2.348021984</td>
</tr>
<tr>
<td>Detection</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Bad randomness #2</td>
<td>Duration</td>
<td>5.140118837</td>
<td>4.026486026</td>
<td>77.663939</td>
<td>35.02412128</td>
<td>4.400139093</td>
<td>9.356572151</td>
<td>2.505437136</td>
<td>8.295716047</td>
<td>2.406748727</td>
</tr>
<tr>
<td>Detection</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Bad randomness #3</td>
<td>Duration</td>
<td>5.883339289</td>
<td>5.733039141</td>
<td>4.730108023</td>
<td>23.37864703</td>
<td>4.574569897</td>
<td>4.686722034</td>
<td>12.09117603</td>
<td>4.312527956</td>
<td></td>
</tr>
<tr>
<td>Detection</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Bad randomness #4</td>
<td>Duration</td>
<td>13.40132713</td>
<td>25.54713297</td>
<td>270.5662198</td>
<td>65.0775077</td>
<td>52.94841003</td>
<td>31.93285203</td>
<td>2.299463034</td>
<td>7.967276096</td>
<td>2.468337059</td>
</tr>
<tr>
<td>Detection</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Access Control #1</td>
<td>Duration</td>
<td>5.200572014</td>
<td>4.012099981</td>
<td>4.676046986</td>
<td>29.09435225</td>
<td>3.299620443</td>
<td>3.163719181</td>
<td>1.745530107</td>
<td>9.402600322</td>
<td>2.072764874</td>
</tr>
<tr>
<td>Detection</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Access Control #2</td>
<td>Duration</td>
<td>8.697600126</td>
<td>32.12349224</td>
<td>4.971230984</td>
<td>287.8219483</td>
<td>4.574535125</td>
<td>4.499389887</td>
<td>2.58026576</td>
<td>9.426314116</td>
<td>2.655647039</td>
</tr>
<tr>
<td>Detection</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Access Control #3</td>
<td>Duration</td>
<td>12.31246901</td>
<td>25.54713297</td>
<td>270.5662198</td>
<td>65.0775077</td>
<td>52.94841003</td>
<td>31.93285203</td>
<td>2.299463034</td>
<td>7.967276096</td>
<td>2.468337059</td>
</tr>
<tr>
<td>Detection</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Detection</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Reentrancy #1</td>
<td>Duration</td>
<td>26.17377186</td>
<td>556.63644987</td>
<td>22.8028943</td>
<td>64.55375109</td>
<td>106.8687789</td>
<td>56.45245993</td>
<td>3.139178038</td>
<td>103.8020959</td>
<td>7.275887251</td>
</tr>
<tr>
<td>Detection</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Reentrancy #2</td>
<td>Duration</td>
<td>8.124588966</td>
<td>3.576477051</td>
<td>2.981256692</td>
<td>66.90424418</td>
<td>3.465710878</td>
<td>4.16452982</td>
<td>2.385936795</td>
<td>8.54885602</td>
<td>2.344093084</td>
</tr>
<tr>
<td>Detection</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Reentrancy #3</td>
<td>Duration</td>
<td>10.48157692</td>
<td>3.779053289</td>
<td>2.671264648</td>
<td>18.282516</td>
<td>4.615363526</td>
<td>4.575246906</td>
<td>2.574928828</td>
<td>11.23968612</td>
<td>2.041010826</td>
</tr>
<tr>
<td>Detection</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Reentrancy #4</td>
<td>Duration</td>
<td>3.70638895</td>
<td>3.032967722</td>
<td>2.180539131</td>
<td>26.53004694</td>
<td>4.16880932</td>
<td>3.100974798</td>
<td>1.809810877</td>
<td>5.179394245</td>
<td>2.098110199</td>
</tr>
<tr>
<td>Detection</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Time Manipulation #1</td>
<td>Duration</td>
<td>4.84670186</td>
<td>3.274656057</td>
<td>5.65362005</td>
<td>21.299891</td>
<td>3.301398754</td>
<td>2.669829845</td>
<td>1.798388243</td>
<td>7.117391798</td>
<td>1.994307055</td>
</tr>
<tr>
<td>Detection</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Detection</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Time Manipulation #3</td>
<td>Duration</td>
<td>5.510846841</td>
<td>4.209907673</td>
<td>3.619573115</td>
<td>10.61389804</td>
<td>4.293622228</td>
<td>3.043540953</td>
<td>2.597222207</td>
<td>30.98754811</td>
<td>3.026561022</td>
</tr>
<tr>
<td>Detection</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Time Manipulation #4</td>
<td>Duration</td>
<td>4.280456066</td>
<td>5.245132685</td>
<td>6.81753301</td>
<td>47.40231466</td>
<td>3.229922056</td>
<td>9.511505842</td>
<td>2.836236</td>
<td>115.5346749</td>
<td>10.58052206</td>
</tr>
<tr>
<td>Detection</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
5.2. Energy

To investigate the performance of the analysis tools in different domains, they are applied to classified contracts. As shown in Figure 3, the results suggest that, among a comparable number of contracts in both categories, the energy contracts contain a relatively larger number of defects [31,32]. Consequently, energy contracts take longer to process, as presented in Figure 4.

The demographic profiles of the source code depicted in Figure 5 imply that the number of logical lines of code (LLOC), the depth of nesting levels (NL), and the number of attributes (NA) and statements (NOS) are higher among non-energy contracts. Similarly, coupling between object classes (CBO) appears more frequently among non-energy contracts. Energy contracts, on the other hand, have a deeper inheritance tree (DIT) and more lines of code (SLOC). Correspondingly, energy contracts are more time-consuming to process, with Mythril being the slowest tool across both categories, which is consistent with the results obtained from the curated data set. Furthermore, Osiris, Honeybadger, and Oyente endured evident instances of execution failures and timeouts in both categories. As demonstrated in Figure 6, an analysis of energy contracts resulted in far more timeouts, with Osiris having the most timeouts, followed by Oyente and Honeybadger.

![Figure 3. Vulnerabilities across different domains.](image-url)
The dominant vulnerabilities across both categories of contracts are comparable, with a more ubiquitous presence in energy contracts, as illustrated in Figure 7. To determine whether there are domain-specific vulnerabilities or security flaws that emerge in particular application domains, Figure 8 summarizes the findings across all dominant vulnerabilities discovered for each category.
There are comparable vulnerabilities among both categories of smart contracts. However, coding practices tend to be more detrimental among energy contracts due to the more prominent presence of each vulnerability in this domain. In general, any vulnerability pertaining to an energy smart contract’s logical statements could be exploited by malicious users to obtain unauthorized access, manipulate energy prices, execute unauthorized transactions, or disrupt energy trading operations. Except for `solidity-revert-requires` and `solidity-deprecated-construction`, which are uncommon in non-energy contracts, the dominant vulnerabilities in both subgroups of contracts are mostly the same. However, `pragma` vulnerability, which involves the use of various pragma directives, was frequently observed in energy contracts as opposed to non-energy contracts. Some vulnerabilities are prevalent in specific application domains, while others are extremely rare. Throughout the analysis, none of the designated tools detected a single instance of transaction ordering dependency in energy contracts. This observation is further reinforced by Figure 8, which shows the unidentified vulnerabilities in each category. However, energy smart contracts are built on complex economic models such as time-based conditions and real-time pricing and vulnerabilities affecting these models could have far-reaching implications. For instance, mishandling time-based conditions could result in incorrect pricing during peak or off-peak
periods, potentially leading to financial losses for traders. This suggests that security flaws may be domain specific and can, thus, be overlooked by existing analytical tools that are not optimized for specific domains. Several studies reported that specific bugs, such as bad randomness, were completely overlooked by the benchmarked tools [33].

6. Discussion

According to empirical analytical studies of smart contracts, only a few of the security flaws revealed by benchmark tools are verifiable [34,35]. Similar studies claim the evaluation results demonstrated numerous instances of vulnerabilities that fell within the tools’ detection span but were not flagged by them. Additionally, there is no single tool that can detect all known vulnerabilities [15,36,37].

With a substantial number of false positives and false negatives, most of the existing vulnerability analysis tools do not meet the requirements of practical scenarios that rely on extensive manual verification.

As shown in Section 5, Slither demonstrated the highest accuracy for curated contracts. As a result, it could serve as a benchmark to assess the efficacy of other analysis tools on classified contracts, as suggested by previous vulnerability analysis research [30]. The following segment of this study is directed at validating this proposition.

Since the implications of reentrancy threat are primarily determined by the smart contract’s business logic, the detection accuracy of the tools varies depending on the domain. In this section, we examine Slither’s performance on classified contracts, pertaining to its effectiveness on reentrancy detection for curated smart contracts. There are a total of five and seven reentrant contracts detected in the energy and non-energy group, respectively. According to the detection results in Table 2, Conkas detected reentrancy in all examined contracts, whereas Slither detected only one reentrancy in energy and three reentrancies in non-energy contracts. This can be interpreted as either FP (False Positive) for Conkas or FN (False Negative) for Slither. Therefore, we examined each contract independently to confirm the credibility of the results. The findings revealed Slither’s inability to detect reentrancy, notably in energy contracts.

Table 2. Slither’s reentrancy detection rate.

<table>
<thead>
<tr>
<th>Domain</th>
<th>Contract</th>
<th>Conkas</th>
<th>Slither</th>
<th>Osiris</th>
</tr>
</thead>
<tbody>
<tr>
<td>Energy</td>
<td>Xad417b.sol</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Xd9900.sol</td>
<td>✓</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X001cb.sol</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Xe69ba3.sol</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>X5f10fd.sol</td>
<td>✓</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Non-Energy</td>
<td>X99bf.sol</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>XE3bcd.sol</td>
<td>✓</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>X92658.sol</td>
<td>✓</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Xb0481.sol</td>
<td>✓</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>X32c72.sol</td>
<td>✓</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>X75d9e6.sol</td>
<td>✓</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>X091d3f.sol</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

The reentrancy vulnerability almost always results in the loss of smart contract money. However, in some cases, it may be the ability to create multiple instances of paid objects or to repeatedly orchestrate code that is only intended to be executed once per call [38,39]. Generally, when exploiting a reentrancy vulnerability, the attacker aims to invoke the same contract function every time it is called. In programming, this is widely recognized as the recursion principle. Hence, most reentrancy attacks involve the transfer, send or call function. The send and transfer functions are considered slightly safer because they are restricted to 2300 gas and the gas constraint precludes the corresponding contract from making costly
external function calls. Evidently, the `call` function is far more vulnerable [40,41]. Whenever an external function call is expected to carry out complex operations, the `call` function is commonly employed to forward the remaining gas. This allows the attacker to resume the original function or a specific function from the original contract using a cross-function reentrancy. For instance, looking at one of the energy contracts from Table 2, unlike Slither, Conkas reports potential reentrancy on line 117, as shown in Figure 9. However, a closer look at line 3 in Listing 2, which equates to line 117 of the solidity file, points to a possible reentrancy. It may occur when the function `receiveApproval` directly or indirectly calls `approveAndCall`.

Listing 2. Smart contract with reentrancy.

```solidity
function approveAndCall(address _spender, uint256 _value, bytes _extraData) returns (bool success) {
  allowed[msg.sender][_spender] = _value;
  Approval(msg.sender, _spender, _value);

  if (! _spender.call(bytes4(bytes32(sha3("receiveApproval(address,
        uint256,address,bytes)"))), msg.sender, _value, this, _extraData)) { throw; }
}
return true;
```

This can be avoided by updating the state before making external calls, ensuring that the contract retains the most recent balance in the event that the attacker calls `withdraw` again. Although `call` is a low-level function for interacting with other contracts, it is not the
preferred method for calling existing functions. It is strongly recommended to use *send* or *transfer over call* to reduce the attack surface [42,43].

Another preventative measure to further improve the safety of contracts is to refactor the code. The objective of refactoring is to minimize the attack surface by adding pre- and post-conditions, such as *require* and *assert*, to encompass the vulnerable *call* invocations and ensure secure code structures. It ensures that all state variables are updated before any external calls to prevent the attacker from recursively calling the functions that are intended to be called once. The malicious node attempts to leverage the *call* and take over the control flow of the system by transmitting it to an external contract. By employing pre- and post-conditions, refactoring makes it possible to completely terminate a transaction if any errors emerge, protecting genuine users from potential risks. Another form of refactoring is to safeguard the state of a contract with the inclusion of a state variable to manage mutual exclusion in functions. This approach is particularly beneficial when addressing cross-function reentrancy breaches. The primary objective is to protect code segments where common resources are accessed. Since mutual exclusion allows only one public function to be operating at a time, just one function will be able to modify the resource at once, completely eliminating cross-function reentrancy.

7. Conclusions

Smart contracts in the energy domain provide the necessary versatility to consolidate diverse processes according to the requirements of the application. Despite the positive association between smart contract categories and their vulnerabilities, vulnerability analysis tools do not consider violation structures and behavior patterns across different application domains. Unfortunately, there is no perfect contract analysis tool for any and all contracts and their underlying business logic. Furthermore, the results of vulnerability analysis tools cannot be replicated in the absence of the appointed data set. To examine this gap, the presented study evaluates the benchmark vulnerability analysis tools in classified and curated contracts.

This classification allows for an independent assessment of each domain’s vulnerability source so that developers can differentiate between domain-specific vulnerabilities when paired with different execution environments. Although the vulnerability analysis workflow used in this research is only applicable to smart contract source code, it is transferable to any application domain in the presence of contracts in the corresponding domain. Therefore, it is reasonable to infer that the detection accuracy of the tools varies depending on the domain, since the benchmark tools did not demonstrate the accuracy claimed in the curated contracts. Furthermore, the overall impact of a comparable vulnerability in one application domain may be more profound and detrimental than in another. This makes the mitigation priority of these defects subject to business logic. In addition, energy contracts demonstrate above-average security flaws and take longer to process, increasing the likelihood of failure or a timeout. The evaluation results revealed the competence of symbolic execution for the analysis of energy contracts. Accordingly, analysis tools that incorporate symbolic execution outperform code transformation coupled with constraint solving in detecting reentrancy in energy contracts.

Finally, the absence of granularity in governance regulation could lead to faulty smart contracts, sensitive endpoints, and ambiguous arbitration rules. One significant source of concern is the possibility of tampering, in which fraudulent parties exploit defects in smart contracts to gain unauthorized access or manipulate their intended functionality. One such attack vector is replay, where an attacker intercepts an authentic transaction and then replays it to illegitimately execute the same transaction. Therefore, a poor governance approach may develop a secondary risk surface that can be exploited to launch a targeted attack on smart contracts. As a result, novel forms of vulnerabilities are emerging, such as cross-chain attacks, flashload attacks, and other types of attacks. Because smart contracts operate automatically based on predetermined conditions, this form of attack can have severe repercussions. To reduce the risks associated with tampering, smart contract
developers must proactively adhere to best practices. Additionally, given the growing interest in the adoption of smart contracts in areas such as metaverse, further investigation is required to ensure secure decentralized governance. Therefore, more efforts should be directed towards improving the reliability and accuracy of detection and disclosing new forms of vulnerability from a domain-specific point of view.

**Author Contributions:** Conceptualization, B.L. and P.M.; methodology, B.L.; investigation, B.L.; resources, P.M.; writing—original draft preparation, B.L.; writing—review and editing, P.M.; supervision, P.M.; funding acquisition, P.M. All authors have read and agreed to the published version of the manuscript.

**Funding:** The support provided by the Government of Alberta under the Major Innovation Fund, project RCP-19-001-MIF, is gratefully acknowledged.

**Data Availability Statement:** The data presented in this study are available in Smartbugs: A Framework to Analyze Ethereum Smart Contracts [https://github.com/smartbugs/smartbugs](https://github.com/smartbugs/smartbugs).

**Conflicts of Interest:** The authors declare no conflict of interest.

**Abbreviations**

The following abbreviations are used in this manuscript:

- DApps Decentralized Applications
- EVM Ethereum Virtual Machine
- XML Extensible Markup Language
- XPath XML Path Language
- CE Code Elements
- RR Relationship Restrictions
- ER Element Restrictions
- CFG Control Flow Graphs
- SSA Static Single Assessment
- AST Abstract Syntax Tree
- SB Smart Bugs
- LLOC Logical Lines of Code
- NL Nesting Levels
- NA Number of Attributes
- NOS Number of Statements
- CBO Coupling Between Objects
- DIT Deeper Inheritance Tree
- SLOC Source Lines of Code
- SC Smart Contract
- FP False Positive
- FN False Negative

**References**


Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.