Automated smart contract vulnerability threat modelling with STRIDE

Nika Jeršič[1,*], dr. Muhamed Turkanović[1]

[1]University of Maribor, The Faculty of Electrical Engineering and Computer Science, Koroška cesta 46, 2000 Maribor, Slovenia

Abstract

Smart contracts are subject to critical vulnerabilities that existing tools often detect without a structured classification of threats. This article presents a heuristic methodology for classifying smart contract vulnerabilities according to the STRIDE model. By combining Solidity-specific semantic patterns with manually defined heuristics, the system enables transparent and understandable detection of threats in all STRIDE categories. Our approach, implemented as an API service, processes contract code, applies heuristics, and returns structured threat classifications without relying on machine learning or training data. The approach bridges the gap between code-level vulnerability detection and high-level security modelling by providing a reproducible framework for secure smart contract development.

Keywords

Smart Contracts, Web 3.0, Threat Modelling, STRIDE, Automation

1. Introduction

Smart contracts are mechanisms designed to automatically implement agreed rules without the need for intermediaries. Their integration into blockchain allows digital transactions to be carried out in a reliable, transparent and secure way. A key advantage of smart contracts is that they reduce the possibility of fraud and error, as the terms are predefined and independently verified [1]. Participants in smart contracts can deposit funds into a smart contract, which are then automatically reallocated based on the fulfilment of predefined conditions. These conditions are not necessarily known at the start of the contract, but can be created dynamically during the execution of the transaction [2]. Smart contracts deployed directly on the blockchain — referred to as on-chain smart contracts — must be executed and validated by every participating node, with all associated transactions remaining transparent and accessible across the entire blockchain network [3]. However, no changes can be made during processing - even if an error occurs, for example an incorrect amount or an incorrect product. On completion, a new block of transaction records is created and written to the chain. Once a block is validated and added to the blockchain, it becomes immutable and permanently accessible to all participants in the network [2].

SQAMIA 2025: Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications, September 10–12, 2025, Maribor, Slovenia

*Corresponding author.

nika.jersic@um.si (N. Jeršič); muhamed.turkanovic@um.si (dr. M. Turkanović)

0009-0008-4644-878X (N. Jeršič); 0000-0002-5079-5468 (dr. M. Turkanović)

Despite the many advantages, smart contracts reveal certain shortcomings that can reduce user reliability or lead to unexpected complications. The most common are Reentrancy, Requirement Violation, Function Default Visibility, Integer Overflow and Underflow, and State Variable Default Visibility [4].

In this context, threat modelling plays an essential role in protecting smart contracts. Threat modelling is a process for identifying, analysing and managing security risks. Its main purpose is to identify vulnerabilities and systematically define threats, their likelihood and impact on the system [5]. Each threat model, such as STRIDE, DREAD, PASTA, OCTAVE, LINDDUN and Attack trees, provides its own unique perspective on the vulnerabilities and threats. When implementing them into the development process, organizations can manage the security of their products on a higher lever, providing more focused testing in the developing and testing phase of the project.

However, these models are designed to systematically search for threats in classical Web2-based information systems and not in decentralized and Web3-based information systems. Additionally, there are limited resources on how to identify vulnerabilities in smart contracts. The most known smart contract vulnerability registry is swcregistry [6], which only includes thirty-six examples of vulnerabilities and is not actively maintained.

Our contribution includes an automated tool, that includes multiple vulnerabilities, their descriptions and use cases from multiple sources. In addition to these, our tool automatically detects patterns in the code and classifies them according to the STRIDE classification. This tool provides a practical solution to bring forward security threats in smart contracts.

The rest of this paper is organised as follows: In Section 2, we present a comprehensive review of the related work, highlighting the gaps and similarities in the existing approaches in automation of detecting threats in smart contracts. Section 3 presents related software tools that are already used for analysing smart contracts. In Section 4 we provide the explained methodology. Section 5 holds the description of the proposed software tool. Section 6 discusses the proposed heuristic method for classifying STRIDE threats in smart contracts and Section 7 concludes the paper.

2. Related work

In this Section we review existing research on smart contract vulnerabilities, automated threat modelling software tools, that primarily focus on STRIDE and existing solutions that combine both domains. At the end of this chapter we discuss the gaps in the existing research.

Mi et al. [7] present a framework for automatically detecting vulnerabilities in smart contracts based on disassembled bytecode and the use of control flow graphs (CFG). The authors then create n-gram features and feed them into a deep neural network with criteria, which improves the separation between cases. The system is effective on large datasets, but it does not offer an explanation of the results and does not allow direct connection to high-quality models such as STRIDE. Furthermore, it does not allow manual rule expansion or control over classification.

Ottati et al.[8] conducted an experimental comparison of three widely used vulnerability detection tools: Oyente, Osiris, and Slither. Each of these tools uses different methods: Oyente is based on symbolic execution, Osiris includes pollution analysis, and Slither uses static source

code analysis. The results show that no tool is comprehensive—each successfully detects only certain types of vulnerabilities (e.g., Slither successfully detects reentrancy and low-level calls, while Oyente detects overflow errors). The authors conclude that multiple tools must be used for reliable security analysis. Unlike these systems, our model does not combine existing tools, but creates domain-specific heuristics that enable coverage of key STRIDE categories with more precise control over detection.

Iuliano and Di Nucci [9] did a thorough systematic review of the literature, analysed 222 scientific studies, and identified 192 unique vulnerabilities and 219 automated detection tools. They also developed a hierarchical taxonomy of vulnerabilities and a mapping between tools and vulnerabilities. Although their analysis is comprehensive and up-to-date, the authors note that existing tools often cover only a limited subset of vulnerabilities, leaving many unprotected. In addition, many approaches are designed without a clear framework such as STRIDE. Our contribution directly addresses this gap with a methodology that incorporates STRIDE and enables a modular, heuristic extension of threat classification.

Bahar [10] proposes a criteria-based feedback methodology (MBFM) that links the results of bug bounty programs with the upgrading of threat models based on STRIDE. The purpose of this methodology is to identify the root causes of vulnerabilities and apply criteria to improve existing threat models. This approach is especially interesting for organizations' security strategies and vulnerability management at the system level. However, it stays at the conceptual level and does not focus on concrete smart contract analysis or code-level integration. Our model fills this gap by performing a heuristic analysis of smart contracts that directly classifies threats according to STRIDE and enables connectivity with existing security processes (e.g., CI/CD or test environments).

Rouzbahani and Garakani [11] present a systematic review of threats in blockchain environments and analyse the use of various threat models, including STRIDE, DREAD, OCTAVE, and PASTA. They emphasize the importance of structured threat modelling and present the different layers at which threats occur (e.g., network, implementation, user). STRIDE is highlighted as a suitable framework for security analysis, including in the context of smart contracts. Nevertheless, their work remains at the level of a conceptual review of methods and does not offer an operational framework or implementation. In contrast, our system enables the direct integration of the STRIDE classification at the code level via a lightweight FastAPI interface and manually defined patterns specific to Solidity.

The reviewed literature demonstrates significant progress in the development of tools for automation of vulnerability scanning in smart contracts. Often automated tools offer identifying specific classes of vulnerabilities, while recent advances in machine learning aim to generalize detection. Methodological works have often explored the use of threat modelling tools, mostly using STRIDE, as a way yo contextualize and prioritize security risks. However, a clear gap persists at the intersection of formal threat classification and code-level analysis. Most automated tools are not aligned with any threat models. Furthermore, machine learning models often sacrifice transparency and auditability, limiting their applicability in high-assurance domains.

Our work addresses this gap by offering a methodology that combines manually defined heuristics, based on Solidity semantics, direct classification to threat according to STRIDE, full transparency and deployment-ready implementation via API.

3. Related software tools

The need for secure development and auditing of smart contracts has encouraged the embrace of numerous tools and frameworks that aim to detect vulnerabilities or verify security properties. Most vulnerability detection tools are tailored to Solidity, which is the predominant language for Ethereum smart contracts [12]. They often focus on identifying known coding patterns that lead to exploits such as reentrancy, unchecked calls, or arithmetic overflows. While they are effective in exposing security flaws, these tools typically lack structured classification of threats based on established threat models. This Section reviews some of the relevant tools and frameworks across both domains: automated tools for analysing solidity code and formal threat modelling methodologies. Our goal is to position our heuristic-based STRIDE classification approach within this landscape.

VulnDetector [13] is an advanced tool for security analysis of smart contracts that detects more than 500 types of vulnerabilities. In addition, it offers suggestions for optimizing gas consumption, which is useful for developers and users. The tool is distinguished by its high accuracy and ability to detect complex security flaws, such as unlimited `delegatecall` calls, uninitialized variables, token transfer vulnerabilities, and reentrancy.

Slither [14] is one of the established tools for static analysis of smart contracts in Solidity. It was developed in Python and enables rapid vulnerability detection with a low false positive rate. The tool supports Solidity versions 0.4 and newer and can be easily integrated into CI/CD environments. It detects vulnerabilities such as unsafe use of `tx.origin`, uninitialized variables, and unsafe low-level calls.

SmartCheck [15] uses static analysis with XPath patterns to detect vulnerabilities in the source code of smart contracts. The tool converts the code into an intermediate XML representation and checks for the presence of known vulnerability patterns such as re-entrancy, unchecked external calls, timestamp dependencies, and unsafe use of `tx.origin`. Despite its effectiveness, it has limitations in detecting more complex vulnerabilities that require more advanced analysis.

Oyente [16] is one of the first tools for smart contract security analysis that uses symbolic execution to detect vulnerabilities. It focuses on vulnerabilities such as transaction order dependency, timestamp dependency, re-entry, and integer overflow. Despite its pioneering role, it has limited ability to detect a wider range of vulnerabilities.

Mythril [17] is a smart contract security analysis tool that uses symbolic execution, taint analysis, and other techniques to detect vulnerabilities. It supports the analysis of multiple blockchains using EVM and allows the analysis of already executed contracts via their addresses. It detects vulnerabilities such as transaction order dependency, re-entrancy, unchecked calls, and unsafe use of `tx.origin`.

ContractFuzzer [18] uses fuzzing techniques to detect vulnerabilities in smart contracts. Based on ABI specifications, it generates various input data for testing contracts and analyzes their behavior during execution. It detects vulnerabilities such as re-entrancy, timestamp dependency, dangerous `delegatecall` calls, and asset freezing. However, compared to other tools, it has a higher rate of false negatives.

Remix IDE [19] offers plugins for static analysis of smart contracts, helping developers detect vulnerabilities while writing code. Plugins such as Solidity Static Analysis and MythX detect vulnerabilities such as re-entrancy, unsafe use of `tx.origin`, use of the built-in assembler, and

timestamp dependencies. Despite the possibility of false positives, these plugins are useful for early bug detection.

Manticore [20] is a symbolic execution tool that enables the analysis of smart contracts and binary files. It supports languages such as Solidity, Vyper, and Bamboo, and allows integration with tools such as Truffle and Mythril. It detects vulnerabilities such as re-entrancy, unsafe external calls, uninitialized variables, and dependencies on transaction order. Despite its power, it has limitations in detecting business logic vulnerabilities.

sFuzz [21] is a smart contract fuzzing tool that uses a flexible strategy for generating test cases. It combines the AFL fuzzing method with a flexible multi-target strategy that targets hard-to-reach branches of code. It detects vulnerabilities such as re-entrancy, timestamp dependencies, dangerous `delegatecall` calls, and integer overflow. The tool is fast and covers a wide range of vulnerabilities, but it is recommended to use it in combination with other tools to improve code coverage.

MadMax [22] is a static analysis tool that focuses on detecting vulnerabilities related to gas consumption in smart contracts. It uses data flow and control flow analysis to detect vulnerabilities such as unlimited operations in large quantities, unisolated calls, and integer overflow. Despite its specific focus, it is recommended that MadMax be used in combination with other tools for a comprehensive security analysis of smart contracts.

Compared to existing smart contract auditing tools, our tool represents a conceptual and methodological improvement, as it focuses on threat classification based on the formal STRIDE model, which is not supported by most analysers. While tools such as Slither, Mythril, or Oyente effectively identify low-level vulnerabilities (e.g., reentrancy, unchecked calls, overflow), they generally do not provide a high level of threat categorization that would be directly useful in the context of system threat modelling or security reports. Our tool is based on transparent, manually defined heuristics that are directly linked to Solidity semantics and enable traceability, flexibility, and extensibility of threat classification. In addition, it allows for easy integration via API, enabling extension with external sources such as SmartBugs. This makes our tool an easy-to-use, understandable, and conceptually consistent solution that complements existing detection mechanisms with structured and formalized threat analysis.

4. Methodology

The objective of this study is to propose a domain-specific methodology for classification of security threats in smart contracts, with STRIDE. Our approach implements hand-crafted heuristics tailored to the semantics of Solidity code, aiming to improve the contextual accuracy and relevance of threat classification.

The proposed methodology is grounded in a conceptual model that defines the core entities and relationships involved in the classification process. This conceptual flow is visualized in Figure 1. Our proposed model consists of the following concepts:

- Smart contract: input component, represented as Solidity source code.
- Semantic patterns: code-level constructs that may indicate a potential vulnerability
- Heuristic: manually defines rule that maps semantic patterns to a specific class of a threat

- STRIDE: the classification threat model that includes Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service and Elevation of Privilege.
- Mitigation of threats: based on the threat classified, the mitigation of threats is also proposed.

These concepts are linked as follows:

- A smart contract contains one or more semantic patterns
- A semantic pattern triggers a corresponding heuristic
- A heuristic detects a STRIDE threat, which is then returned as the output classification
- Based on the STRIDE classification and the threat itself, a mitigation is proposed.

The classification operates in four stages: (1) lexical preprocessing of Solidity input, (2) extraction of semantic patterns, (3) heuristic evaluation, and (4) STRIDE threat mapping. At each stage, intermediate representations are stored to allow traceability and debugging.

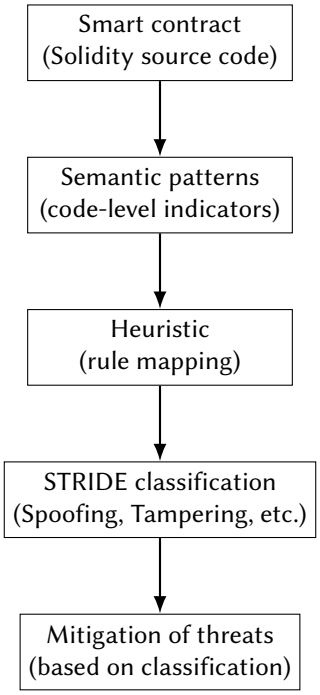

Figure 1: Heuristic-based STRIDE threat classification process

4.1. Heuristic-based analysis

In contrast to machine learning approach, this methodology relies entirely on already existing expert knowledge that is taken from the registries. This means that each heuristic is crafted based on known vulnerabilities and secure coding guidelines in the context of smart contracts. For example, the presence of ‘`tx.origin`‘ in the authorization check will trigger a heuristic

pattern that reports a Spoofing classification flag. This design ensures interoperability, expertise, transparency and precise control over security threats [23].

Each heuristic is formally defined as a tuple

$$H = (P, C, S)$$

where:

- P denotes a matching pattern (e.g., regular expression or abstract syntax tree fragment),
- C specifies the matching condition (e.g., control flow or data flow context),
- S is the STRIDE threat tag corresponding to one of the six categories.

Heuristics are implemented using a lightweight rule engine, which enables precise control and logging. This design ensures high transparency, auditability, and deterministic behaviour, facilitating validation by domain experts.

To ensure consistency and interoperability, each heuristic is annotated with a reference to a corresponding SWC-ID.

The proposed methodology thus enables domain-specific precision, where heuristics are crafted with explicit knowledge of Solidity's semantics and known exploit primitives. Additionally, classification decisions are fully explainable and traceable to individual code-level features, allowing fine-grained validation and refinement of heuristics.

4.2. Incorporation of STRIDE threat model in the methodology

The proposed methodology is designed to detect and classify threats with STRIDE in smart contracts. The API assumes that the smart contract, that is in the input, is deployed to a public blockchain (e.g. Ethereum) and is exposed to potentially malicious actors.

STRIDE [24] classifies detected threats into six STRIDE categories, as it is presented in Table 1.

Category	**Description**	**Example heuristic**
Spoofing	Forging an identity or source of request	Use of `tx.origin` for authentication
Tampering	Unauthorized modification of contract state	Misuse of `delegatecall`
Repudiation	Lack of audit trails or transaction proof	Absence of logging in critical transactions
Information Disclosure	Unintended exposure of sensitive information	Use of public state for private values
Denial of Service	Disruption of contract availability or functionality	Loops on unbounded input, external call reentrancy
Elevation of Privilege	Unauthorized gain of higher-level access	Incorrect `msg.sender` checks or fallback misuse

Table 1
Explained STRIDE threat model with a short description and examples

The first phase involves tokenizing and parsing the source code using a Solidity-aware parser. Extracted tokens and syntax trees are searched for contextually relevant keywords. These keywords are mapped to heuristic rules stored in a structured dictionary.

The dictionary of heuristic rules were developed based on a review of the SWC Registry [6] and open-source vulnerability disclosure developed and maintained by Trail of Bits [25]. Rules were iteratively validated by applying them to real-world vulnerable contracts and refining them using feedback from classification results [25]. As presented in Figure 2, each identified heuristic is deterministically mapped to a STRIDE category, enabling an interpretable and modular threat modelling mechanism.

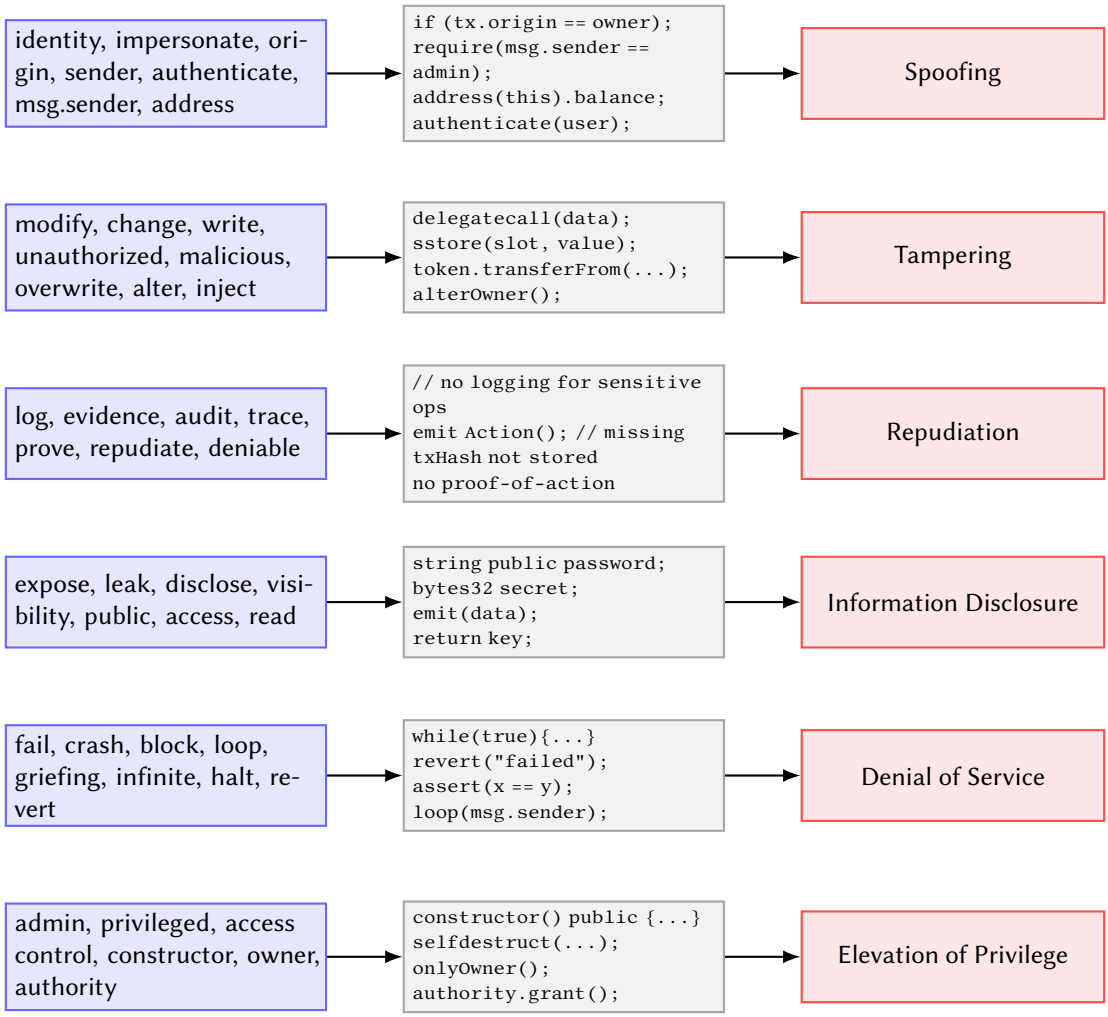

Figure 2: Graphical representation of the mapping of keywords and specific Solidity examples to STRIDE security categories.

4.3. Limitations

While the proposed heuristic STRIDE classification framework demonstrates high interoperability and accuracy for known semantic patterns, several limitations should be acknowledged.

First, the proposed system relies on a static and manually defined set of heuristics. As a result, it may fail to detect obfuscated or new vulnerabilities. Although heuristics can be updated, this requires ongoing expert input and domain knowledge, which limits scalability compared to machine-learning-based approaches.

Second, the current implementation only performs static code analysis, without executing contracts or observing runtime behaviour. This restricts the detection of threats that manifest dynamically and limits support for interacting with smart contracts.

Third, the dataset used for evaluation consists of logically stored, pre-labelled Solidity contracts, which may introduce biases or fail to capture the diversity of vulnerabilities observed in the wild. Furthermore, the majority of publicly available vulnerability databases for smart contracts are either no longer actively maintained or inherently limited in scope, rendering them increasingly outdated over time [25, 6].

Finally, the STRIDE threat model, was originally designed for traditional system architectures and may not fully capture blockchain-specific threat surfaces, or even business oriented threats. Our future work will be based to address these challenges. Despite these limitations, our proposed methodology provides a robust foundation for auditable security classification and can be extended through new heuristic modules and integration with external vulnerability databases.

5. Proposed software tool

We have developed a tool to automate vulnerability-based threat modelling for smart contracts using STRIDE. Additionally, we have developed an API to implement our methodology into the tool. The main use of this API is to connect the database of vulnerabilities, compare it to the code in the smart contract, search for patterns and categorize the threats according to STRIDE.

As presented in Figure 3, the analysis begins with the ingestion of Solidity source code, which is then preprocessed and transformed into an intermediate representation suitable for semantic matching.

To enable structured inspection, the input code is parsed into an Abstract Syntax Tree (AST) using the Solidity compiler. This AST serves as the primary intermediate representation (IR) for semantic pattern extraction, enabling precise identification of control structures, data declarations, function modifiers, and call semantics.

Semantic pattern matching is implemented using a hybrid mechanism that combines:

1. **AST-based structural matching**, where heuristics are applied over tree node patterns (e.g., call expressions, authorization checks, inheritance structures),
2. **Keyword scanning**, used as a preliminary filter to reduce the matching search space (e.g., detecting presence of sensitive keywords such as `tx.origin`, `delegatecall`, or `selfdestruct`),
3. **Contextual constraints**, which verify whether the matched constructs appear in semantically meaningful locations (e.g., function scope, access control logic, fallback functions).

While simple regex-based matching is used in limited preprocessing steps (such as detecting inline assembly or comments), all final heuristic decisions rely on syntactic structure extracted from the AST.

This design ensures that matching is not brittle to code formatting, comments, or variable renaming, and allows the system to detect patterns embedded in higher-order constructs. The AST nodes are traversed recursively with a set of matcher functions that evaluate whether a subtree satisfies the preconditions of a heuristic. If a match is found, the associated STRIDE category is returned together with metadata such as source location, matching rule identifier, and mitigation suggestion.

The analysis engine is implemented for AST inspection, supplemented with custom matchers and heuristic evaluators defined in a declarative rule format. This modular design allows easy extension of the rule set and integration with external vulnerability sources.

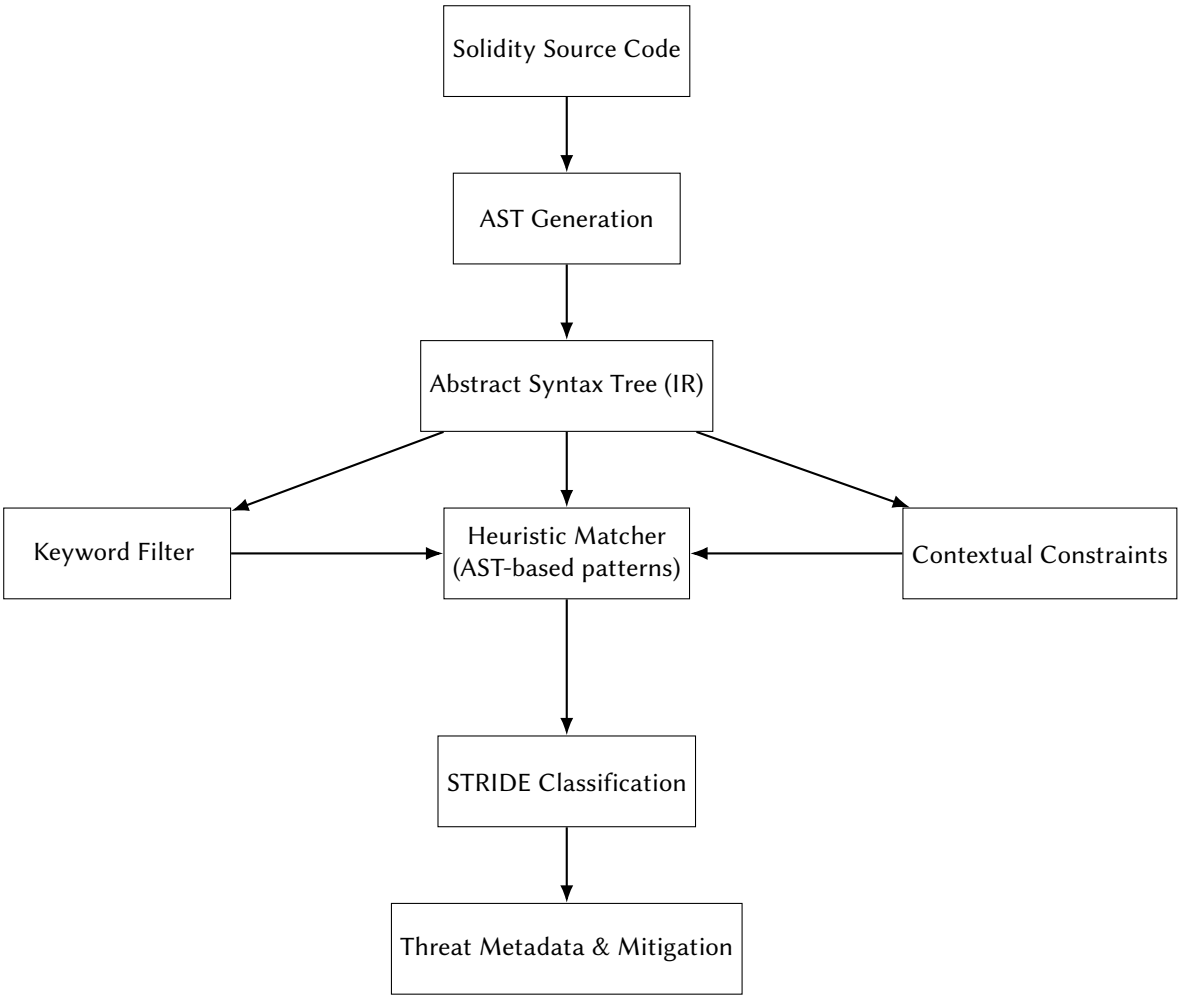

Figure 3: Processing pipeline for STRIDE classification based on AST and heuristic evaluation

5.1. Evaluation

To assess the effectiveness and functional capability of our methodology, we conducted a structured assessment based on a representative set of Solidity smart contracts known to contain vulnerabilities corresponding to each category of the STRIDE threat model. An example for the evaluation process is presented in Figure 4. The goal was to confirm that when applied to these contracts, the system correctly identifies the expected types of threats based on the associated heuristics.

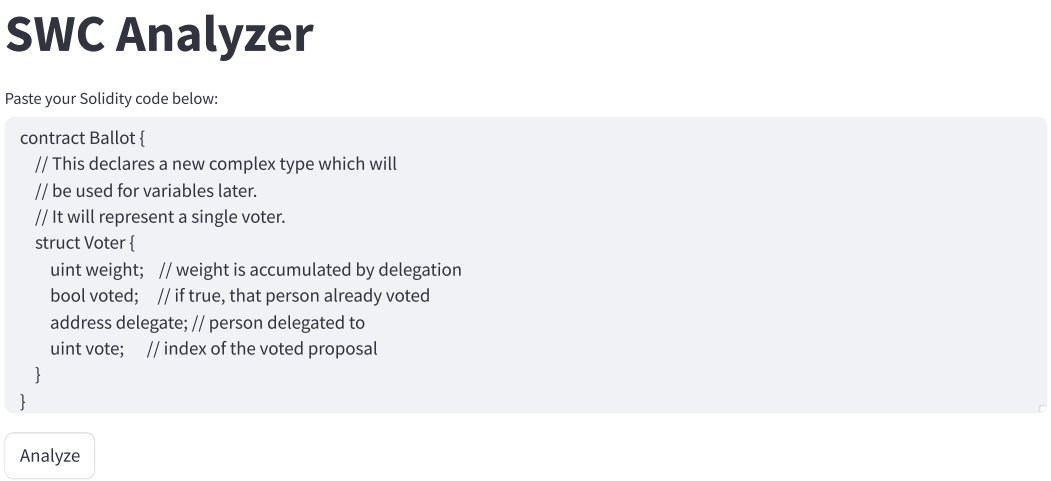

Figure 4: Smart contract as the input in the software tool.

For the evaluation we selected a set of curated smart contracts—either from public vulnerability collections or manually collected—to ensure coverage of all six STRIDE categories. Each contract was designed or annotated to exhibit one or more semantically recognizable patterns that correspond to our implemented heuristics, as presented in Figure 3.

For example, contracts that use `tx.origin` in their authorization logic were included for spoofing detection; contracts vulnerable to unverified `delegatecall` were selected for spoofing; contracts lacking verifiable logging mechanisms for sensitive actions were used for denial of service assessment; and so on. Each contract was submitted to the system via the interface, where the input source code was parsed, analysed, and processed through the engine's rules.

The system successfully identified threat indicators based on heuristic matches and generated structured output data describing the corresponding STRIDE category, location in the code, and rationale associated with the matching pattern. The classification output was manually reviewed against known vulnerabilities in each contract to ensure consistency and correctness of logic. This process confirmed that the heuristics behaved as expected and correctly associated semantic patterns with the appropriate threat types.

Although this evaluation confirms that the system operates as designed on targeted contracts, it does not yet provide quantitative performance metrics such as precision, recall, false-positive rate (FPR), or false-negative rate (FNR). These metrics are essential for assessing robustness at scale and will be incorporated in future work through systematic benchmarking on larger

Table 2
Example of a list of identified threats in the tool, classified according to the STRIDE methodology.

Stride	Title	Description	Mitigation
Information Disclosure, Tampering	Function Default Visibility	Functions that do not have a function visibility type specified are `public` by default. This can lead to a vulnerability if a developer forgot to set the visibility and a malicious user is able to make unauthorized or unintended state changes.	Functions can be speci-fied as `external`, `public`, `internal`, or `private`. Choose the appropriate visibil-ity to reduce the attack surface.
Denial of Service, Repudiation	Unchecked Call Return Value	The return value of a message call is not checked. Even if the called contract fails, execution resumes, possibly leading to unexpected logic.	Always check the return value of low-level calls to ensure correct behaviour.
Denial of Service	Assert Violation	The `assert()` function should only assert invariants. Reaching a failing `assert()` may indicate a bug or misuse for input validation.	Use `require()` for input validation. Fix bugs that allow invalid states.
Tampering	Arbitrary Jump with Function Type Variable	If assembly is used, an attacker might reassign a function type variable to malicious instructions.	Minimize assembly usage. Do not allow arbitrary assignment of function pointers.
Uncategorized	Unexpected Ether Balance	Contracts may misbehave if they assume a specific Ether balance. Ether can be sent forcibly via `selfdestruct` or mining.	Avoid strict equality checks for contract balance.
Information Disclosure	Unencrypted Private Data On-Chain	`private` variables can still be read from the blockchain. Attackers can extract sensitive state.	Store private data off-chain or encrypt it before storing.

datasets (e.g., from Etherscan, SmartBugs, or curated audit reports). In addition, while this evaluation used a threat-focused dataset, future testing will include general-purpose and benign contracts to estimate baseline false-positive rates. Finally, although STRIDE provides a clear conceptual framework, it was originally designed for traditional IT systems. Certain blockchain-native threats—such as front-running, gas-griefing, miner extractable value (MEV), or timestamp dependence—are not fully captured by STRIDE. As such, we plan to extend our methodology by integrating domain-specific threat taxonomies, such as those from DASP or newer blockchain security ontologies.

To strengthen the empirical foundation of our methodology, we conducted a quantitative validation experiment focused on measuring classification accuracy at scale. This experiment assesses the system's performance using standard metrics from information retrieval and

software analysis, including:

- **Precision (P):** the proportion of identified threats that are true positives.
- **Recall (R):** the proportion of all actual threats that were correctly identified.
- **False Positive Rate (FPR):** the proportion of benign patterns incorrectly flagged as threats.
- **False Negative Rate (FNR):** the proportion of true threats missed by the system.

The evaluation is based on two benchmark datasets:

1. **Ground-truth vulnerability dataset:** contracts from known vulnerability collections such as SmartBugs, Trail of Bits audits, and the SWC Registry. Each vulnerability is manually labelled with its STRIDE category and source location.
2. **General-purpose contract corpus:** a randomly sampled set of verified smart contracts from Etherscan, assumed to be mostly benign, to estimate baseline false-positive rates.

Each contract was automatically analysed by our system, and the output was compared to the ground-truth annotations. This evaluation provides objective insight into the practical reliability and coverage of our methodology, and will help guide future refinement of heuristics, reduction of overwriting, and integration of additional detection logic for blockchain-native threats.

6. Discussion

The proposed methodology introduces a transparent, heuristics-based framework for classifying STRIDE threats in Solidity smart contracts. Unlike traditional static analysers or machine learning-based systems, our approach explicitly bridges the gap between low-level vulnerability detection and high-level threat modelling. By combining semantically relevant code patterns with hand-picked heuristics, the system enables developers, reviewers, and security experts to interpret and follow threat classifications in a structured and conceptually sound manner.

A key advantage of this methodology is the use of the STRIDE model as a basis for thinking about smart contract threats. Models such as SWC Registry [6] and SCSVS [26] are oriented toward technical control enumeration rather than conceptual threat abstraction. Our system improves semantic richness and security context by mapping Solidity code artifacts to STRIDE categories, which are essential for architectural risk analysis and regulatory reporting. We acknowledge that heuristic mapping between SWC threats and STRIDE categories is still an area that requires further formalization and objective assessment. Establishing a consistent and validated link between low-level vulnerability patterns and high-level threat categories is essential to support tool standardization, interoperability, and benchmark comparability. However, this work goes beyond the scope of the current study, which focused primarily on the development and presentation of a functional proof-of-concept tool. We believe that refining this mapping is a promising direction for future work.

In addition, the use of hand -defined heuristics to cover different benefits in terms of interpretation, audit and modularity. Unlike neural systems, where decisions about the classification require post-hoc interpretative techniques, our approach provides a direct overview of each rule

and its semantic basis. This supports that it uses cases where human confirmation, appearances and formal reasoning, such as code reviews, compliance audit and security certificates are required.

Our proposed tool complements already exiting tools by offering a higher level of abstraction focused on semantics of threats. When normal tools report findings in terms of implementing of the code, our system explains these patterns and the conditions of the opposite capacity, offensive surfaces and safety goals framed by style.

Nevertheless, more conceptual and technical restrictions remain. Static analysis cannot capture all context behaviours and the heuristic is manually maintained. In addition, while a stepping frame preview, structured good, may not fully cover the blockchain risks -specific concerns, such as a value that can be drawn by a miner (MEV). Future work should take into account the hybrid threats of threats and the inclusion of information on the implementation or formal specifications.

This methodology proves that the heuristic analysis, in line with the step, can be as a bridge between signals at the level of code and semantics at the level of threat, which offers a structured, interpretative and widespread approach to a smart contract. It lays the foundations for more integrated tools that are aware of a model that supports secure software for the blockchain program that goes beyond isolated defect detection.

7. Conclusion

This article represents a new approach for the classification of threats and smart contracts, with the use of manually defined heuristic in line with the STRIDE threat modelling model. With semantic patterns specific to STRIDE, with structured threats, the proposed system offers transparent, extensive and conceptual justified alternatives to existing vulnerability tools.

Unlike black-box machine learning approaches or low-level static analysers, our methodology provides high interpretation and safety findings in the trace. Each classification is anchored in a clearly defined heuristic, which facilitates the certification of experts, compliance with regulations and integration into development flows. The use of STRIDE helps concluding on offensive surfaces that bridge the gap during the analysis of the source code and the design of architectural security.

While the current approach focuses on reviewing the static code, future work may include dynamic analysis, integration with formal techniques for verifying and enriching a heuristic base with automated mining sources of threats such as SmartBugs.

Ultimately, this methodology contributes a reproducible and practical framework for elevating smart contract security analysis from pattern detection to structured threat reasoning. It opens the way to blockchain engineering, where threat models and domain knowledge are included in the automated safety tools.

Acknowledgments

The authors acknowledge financial support from the Slovenian Research and Innovation Agency (Research Core Funding No. P2-0057).

Declaration on Generative AI

During the preparation of this work, the authors used ChatGPT in order to identify and correct grammatical errors, typos, and other writing mistakes, and DeepL to translate text from another language into English or vice versa. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] M. R. Amin, M. F. Zuhairi, M. N. Saadat, A survey of smart contracts: security and challenges, Int. J. Netw. Secur 29 (2020) 3.

[2] V. Y. Kemmoe, W. Stone, J. Kim, D. Kim, J. Son, Recent advances in smart contracts: A technical overview and state of the art, IEEE Access 8 (2020) 117782–117801.

[3] W. Zou, D. Lo, P. S. Kochhar, X.-B. D. Le, X. Xia, Y. Feng, Z. Chen, B. Xu, Smart contract development: Challenges and opportunities, IEEE transactions on software engineering 47 (2019) 2084–2106.

[4] N. Ivanov, C. Li, Q. Yan, Z. Sun, Z. Cao, X. Luo, Security threat mitigation for smart contracts: A comprehensive survey, ACM Computing Surveys 55 (2023) 1–37.

[5] S. Katsikeas, E. R. Ling, P. Johnsson, M. Ekstedt, Empirical evaluation of a threat modeling language as a cybersecurity assessment tool, Computers & Security 140 (2024) 103743. doi:10.1016/J.COSE.2024.103743.

[6] Smart contract weakness classification (swc), 2020. URL: https://swcregistry.io/, date accessed: 5/29/2025.

[7] F. Mi, C. Zhao, Z. Wang, S. M. Halim, X. Li, Z. Wu, L. Khan, B. Thuraisingham, An automated vulnerability detection framework for smart contracts, Distributed Ledger Technologies: Research and Practice (2023).

[8] J. Ottati, G. Ibba, H. Rocha, Comparing smart contract vulnerability detection tools., in: BENEVOL, 2023, pp. 9–16.

[9] G. Iuliano, D. Di Nucci, Smart contract vulnerabilities, tools, and benchmarks: An updated systematic literature review, arXiv preprint arXiv:2412.01719 (2024).

[10] S. W. Bahar, Advanced security threat modelling for blockchain-based fintech applications, arXiv preprint arXiv:2304.06725 (2023).

[11] M. A. Rouzbahani, H. G. Garakani, Blockchain security threats: Survey, in: 2024 11th International Symposium on Telecommunications (IST), IEEE, 2024, pp. 171–175.

[12] M. Wohrer, U. Zdun, Smart contracts: security patterns in the ethereum ecosystem and solidity, in: 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE), IEEE, 2018, pp. 2–8.

[13] VulnDetector, Vulndetector: Your ultimate vulnerability scanner, 2025. URL: https://vulndetector.io/, date accessed: 6/5/2025.

[14] Slither, Slither, the smart contract static analyzer, 2025. URL: https://github.com/crytic/slither, date accessed: 6/5/2025.

[15] SmartCheck, Smartcheck – a static analysis tool that detects vulnerabilities and bugs

in solidity programs (ethereum-based smart contracts, 2025. URL: https://github.com/smartdec/smartcheck, date accessed: 6/5/2025.
[16] Oyente, Oyente: An analysis tool for smart contracts, 2025. URL: https://github.com/enzymefinance/oyente, date accessed: 6/5/2025.
[17] Mythril, Mythril is a symbolic-execution-based securty analysis tool for evm bytecode., 2025. URL: https://github.com/ConsenSysDiligence/mythril, date accessed: 6/5/2025.
[18] ContractFuzzer, The ethereum smart contract fuzzer for security vulnerability detection (ase 2018), 2025. URL: https://github.com/gongbell/ContractFuzzer, date accessed: 6/5/2025.
[19] Remix, Remix ide is a no-setup tool with a gui for developing smart contracts., 2025. URL: https://remix-project.org/?lang=en, date accessed: 6/5/2025.
[20] Manticore, Manticore is a symbolic execution tool for the analysis of smart contracts and binaries., 2025. URL: https://github.com/trailofbits/manticore, date accessed: 6/5/2025.
[21] sFuzz, sfuzz, 2025. URL: https://github.com/duytai/sFuzz, date accessed: 6/5/2025.
[22] MadMax, Madmax, 2025. URL: https://github.com/nevillegrech/MadMax, date accessed: 6/5/2025.
[23] J. Fernández, J. A. Macías, Heuristic-based usability evaluation support: A systematic literature review and comparative study, in: Proceedings of the XXI International Conference on Human Computer Interaction, Interacción '21, Association for Computing Machinery, New York, NY, USA, 2021. URL: https://doi.org/10.1145/3471391.3471395. doi:10.1145/3471391.3471395.
[24] OWASP, 2025. URL: https://owasp.org/www-community/Threat_Modeling_Process#stride, date accessed: 6/5/2025.
[25] T. of Bits, crytic/not-so-smart-contracts: Examples of solidity security issues, 2023. URL: https://github.com/crytic/not-so-smart-contracts.
[26] C. Diligence, Smart contract security verification standard (scsvs), https://github.com/consensys/SCSVS, 2025. Date accessed: 22/6/2025.