Proceedings of the 5th International Conference on Signal Processing and Machine Learning
DOI: 10.54254/2755-2721/108/2025.20839

Research on Security Vulnerability Detection of Blockchain Smart Contracts Based on Machine Learning

Yinuo Wu[1,a,*]

[1]*Nanjing University of Aeronautics and Astronautics, Liyang City, Jiangsu Province, 213300, China*
a. 162320304@nuaa.edu.cn
**corresponding author*

Abstract: Blockchain technology, characterized by its immutability, decentralization, transparency, security, and traceability, has shown vast potential for applications in secure IoT communications and data protection through its deployed smart contracts. While machine learning-based code generation systems aim to automate high-quality programming solutions, they face significant challenges when addressing blockchain-related issues. This paper analyzes the limitations of machine learning in identifying vulnerabilities within blockchain smart contracts and proposes robust solutions. To achieve this, this paper suggests organizing multiple security experts for labeling, developing efficient labeling tools, employing semi-supervised learning to reduce dependency on labeled data, and establishing a continuous update mechanism for labeled datasets to adapt to evolving threat landscapes. To address the scarcity of training samples in the blockchain domain, this paper introduces a method for generating additional Solidity smart contract training samples using data augmentation techniques. Given that traditional data augmentation methods are not suitable for Solidity, the approach involves converting Solidity contracts into Python code for processing, then reverting them back to Solidity post-augmentation, with provided code examples. Furthermore, leveraging established non-blockchain code to train blockchain-related models enhances model performance and generalization capabilities. These strategies effectively tackle the issue of insufficient training samples in the blockchain field and offer new perspectives on the conversion between Solidity and Python.

Keywords: Blockchain, machine learning, semi-supervised learning, data augmentation techniques

1. Introduction

Blockchain is a database technology whose immutability is maintained by cryptographic techniques and consensus mechanisms [1]. As a decentralized, open, secure, and traceable distributed ledger, blockchain lacks a central control authority, ensuring fairness and transparency in transactions. Each block contains a copy of the data, which can be verified and updated by any participant. Blockchain has evolved through three distinct stages of development. Blockchain 1.0: This phase began with the emergence and development of Bitcoin, the first cryptocurrency. It primarily focused on recording transactions and ensuring data security, uniqueness, and immutability [2]. Blockchain 2.0: Represented by Ethereum, this stage integrated and deployed smart contracts, marking its application

DOI: 10.54254/2755-2721/108/2025.20839

in enterprise-level use cases [3][4]. Blockchain 3.0: This phase extended blockchain applications beyond cryptocurrencies, addressing societal governance issues across various industries.

As the core component of Blockchain 2.0, smart contracts have vast potential in fields such as the Internet of Things (IoT). Once deployed, smart contracts are immutable and execute predefined conditions autonomously. The code conditions embedded in the contract are executed precisely. Because Ethereum's smart contracts operate on a public blockchain, transactions can be audited and assets tracked in real time. Moreover, personal privacy is well-protected through techniques like hash algorithms, fostering widespread trust in smart contracts and promoting their adoption across diverse domains.

In recent years, blockchain technology has achieved remarkable progress in IoT applications, covering areas such as secure communication, anonymous protocols, firmware upgrades, and smart home protection. For instance, early studies implemented secure communication and anonymous protocols among IoT devices using the Bitcoin blockchain, ensuring data privacy and integrity. Similarly, blockchain technology has been leveraged to optimize IoT firmware upgrades, enhancing efficiency and reducing risks associated with centralization. Additionally, scholars have proposed blockchain-based solutions to ensure Docker image authenticity [5], protect smart homes [6], and manage security in multi-tier network structures [7], effectively improving system security and reliability. Notably, researchers have explored methods enabling resource-constrained, low-power devices to communicate with Ethereum blockchain nodes through gateways, ensuring seamless integration into the blockchain network [8]. In scenarios such as data transactions, privacy protection, supply chain security, and applications in homes, battlefields, and healthcare, blockchain technology demonstrates immense potential. By providing transparent and highly traceable data management, it significantly enhances the overall security of IoT systems.

In summary, blockchain technology not only offers novel solutions for addressing core issues in IoT—such as secure communication, firmware upgrades, and data protection—but also fosters the healthy development of the IoT ecosystem through mechanisms like smart contracts.

This paper aims to analyze the limitations of machine learning in detecting vulnerabilities in blockchain smart contracts and propose practical solutions to address these challenges.

2. Principles

A code generation system based on machine learning aims to produce high-quality programming solutions automatically. The entire process is divided into five main steps: pretraining, fine-tuning, understanding the problem, generating solutions, and filtering and submission.

(1) Pretraining: First, the system performs pretraining using large-scale datasets, typically sourced from open-source code repositories like GitHub or specialized datasets. For code generation tasks, the training data consists of diverse code snippets spanning multiple programming languages and application scenarios. By training on a vast amount of textual data, the model learns fundamental syntax structures and advanced semantic information, laying a solid foundation for subsequent tasks.

(2) Fine-tuning: To better adapt the model to specific tasks, the system employs a fine-tuning strategy. Specifically, by training on sample datasets that include problem descriptions and corresponding solutions, the model parameters are further optimized, enhancing its ability to address specific types of problems. To improve the model's generalization and robustness, these sample datasets are widely collected from various sources, including but not limited to online programming communities, forums, and tutorials. This ensures coverage of diverse features, categories, and noise levels.

(3) Understanding the Problem: When a user submits a programming query, the system first utilizes natural language processing (NLP) techniques to analyze it, accurately capturing the core

DOI: 10.54254/2755-2721/108/2025.20839

requirements and boundary conditions of the problem. The objective of this stage is to ensure that the subsequently generated solutions precisely meet the user's requirements.

(4) Generating Solutions: Based on the problem's understanding, the system automatically generates multiple potential solution codes. These codes are produced through the model's complex internal algorithmic logic, exploring various approaches to solving the problem. The number and diversity of generated solutions depend on the model's design and configuration.

(5) Filtering and Submission: Finally, the system conducts a series of filtering operations on all generated solutions, including but not limited to removing evidently erroneous code and grouping similar solutions together. On this basis, a voting mechanism (such as human review) may be applied to further determine the optimal solution. Before submission to the user, the system conducts multiple rounds of testing on the selected solution to ensure its correctness and stability. Only those codes that pass all test cases are ultimately adopted and provided to the user. This process not only guarantees the quality of the output but also enhances user satisfaction.

3. Issues

However, the aforementioned system encounters several challenges when addressing blockchain-related problems, as outlined below:

(1) Complexity of Fault-Tolerance Mechanisms: Blockchain systems require high fault-tolerance to address malicious nodes and network failures. This increases the complexity of error detection, as the system must accurately identify and handle errors in uncertain environments. AI models must maintain robustness in highly dynamic and uncertain conditions, posing higher demands on their design and training.

(2) Multidisciplinary Knowledge and Training Requirements: Blockchain technology spans multiple fields, including cryptography, distributed systems, economics, and law. This necessitates AI models to possess extensive knowledge and training to understand and address complex problems across these domains. While current AI models perform well in specific fields, they face limitations in interdisciplinary applications.

(3) Diversity of Blockchain Systems: Blockchain systems are highly diverse, encompassing public chains, consortium chains, and private chains, each with distinct architectures and security mechanisms. This diversity increases the difficulty of model generalization, requiring tailored training and adjustments for different blockchain types. Existing AI models often struggle to operate universally across multiple blockchain systems and typically require custom development for each system.

(4) Scarcity of Smart Contract Datasets:

a. Due to blockchain's open-source and immutable nature, smart contracts typically undergo rigorous review before deployment, resulting in a relative scarcity of vulnerable contracts. This leads to a lack of datasets containing vulnerabilities for AI model training. The insufficient volume of training data hinders AI models from learning effective methods for identifying and repairing smart contract vulnerabilities, thereby impacting their performance in this domain.

b. For several reasons, recent smart contract code is often not open-sourced. In the early development stages, smart contract code may be immature, containing uncertainties and risks that could lead to user misunderstandings or misuse if prematurely open-sourced. Additionally, smart contracts may depend on proprietary libraries or services that prohibit open-sourcing. Some smart contracts may also be bound by confidentiality agreements, further limiting the availability of training datasets.

c. The annotation process for smart contract code affects both the quantity and quality of datasets. Vulnerabilities in open-source code are often not explicitly labeled and require post-analysis by professionals to identify and annotate them. During annotation, contextual information such as the

DOI: 10.54254/2755-2721/108/2025.20839

contract's business logic and invocation relationships must also be considered. A lack of contextual information in annotations can prevent models from accurately understanding the nature of vulnerabilities, thereby reducing the accuracy and effectiveness of vulnerability detection.

4. Solutions

To overcome the challenges in smart contract code annotation and vulnerability detection, the following solutions are proposed:

4.1. Organizing Multiple Security Experts for Annotation

By involving multiple security experts in the annotation process, consistency and accuracy can be ensured. Experts can collaboratively review and discuss potential vulnerabilities, thereby improving the quality and reliability of the annotations.

4.2. Developing Efficient Annotation Tools

The workflow of smart contract annotation tools involves several steps, from source code to the final analysis report. First, a lexer breaks the source code into a series of basic units, or tokens, which can include keywords, identifiers, and operators. Next, a parser uses these tokens to construct a tree-like data structure called an Abstract Syntax Tree (AST) that reflects the syntactic structure of the source code. Once the AST is constructed, the tool performs semantic analysis, such as type checking and scope validation, ensuring the code is not only syntactically correct but also logically valid.

During this process, tools may provide features like syntax highlighting, applying different colors or styles to code elements (e.g., keywords, strings, comments) to aid readability. Additionally, error detection mechanisms report compile-time errors during syntax and semantic analysis, while runtime errors may require predictive methods like simulated execution or static analysis.

In the current development of blockchain smart contracts, commonly used annotation tools include:

Remix IDE, a web-based Solidity integrated development environment (IDE), offers features such as syntax highlighting, auto-completion, real-time error alerts, and static analysis. It also supports various plugins, such as debuggers and formal verification tools.

Truffle Suite, not only a development framework but also an IDE based on Ganache, supports the development, compilation, deployment, and testing of smart contracts. It integrates additional code annotation features when used with editors like VS Code.

Solgraph, a visualization tool, generates function call graphs of Solidity smart contracts, aiding developers in understanding interactions between contracts.

Additionally, some non-blockchain annotation tools can be adapted for use with blockchain code after appropriate modifications. For instance, VS Code can support smart contract development through specific plugins. The "Solidity" plugin provides features such as syntax highlighting, code completion, and error detection. The "Hardhat for VS Code" plugin works with the Hardhat framework, offering additional development conveniences. Products from JetBrains (e.g., IntelliJ IDEA, WebStorm, PyCharm) can also support smart contract development when plugins are installed, such as the Sublime Text can install Solidity syntax highlighting and other supporting plugins through package managers like Package Control. Moreover, integrating static analysis tools like Slither, MythX, and Oyente can automatically detect security vulnerabilities and code quality issues during development. For example, using Hardhat to integrate Slither and MythX, or Truffle to integrate Oyente. Enhanced smart contract awareness, such as parameter information, function signature hints, go-to-definition, and find-all-references, can also be implemented to allow developers to browse and understand the code more quickly. The design and development of efficient annotation tools to assist annotators can significantly enhance annotation efficiency and quality. These tools should incorporate

DOI: 10.54254/2755-2721/108/2025.20839

intelligent features such as automated prompts and context awareness to reduce the workload for annotators.

4.3. Employing Semi-Supervised Learning

Semi-supervised learning is a machine learning approach that utilizes a small amount of labeled data (annotated data) and a large amount of unlabeled data (unannotated data) for training.

Common semi-supervised learning methods include:

Self-Training: The model is initially trained using a limited amount of labeled data. It then predicts the labels of unlabeled data and selects the predictions with the highest confidence as "pseudo-labels." These pseudo-labeled data are incorporated into the labeled dataset, and the model is retrained. This process can be iterated until the model converges or a predefined stopping criterion is met.

Co-Training: Often applied to scenarios with multi-view or multimodal data, this method involves training two or more models, each focusing on different views of the data. Each model is trained on labeled data from its respective view and then predicts labels for unlabeled data. Subsequently, each model selects high-confidence predictions from the other model to augment its labeled dataset.

Generative Adversarial Networks (GANs): GANs can be applied to semi-supervised learning tasks. The generator creates data samples, and the discriminator not only distinguishes real data from generated data but also classifies the real data. In this way, the discriminator learns useful information from unlabeled data while performing classification tasks.

Graph-Based Methods: In this approach, data points are treated as nodes in a graph, with edges connecting similar data points. Graph regularization assumes that connected nodes should share similar labels. Algorithms such as random walks on the graph can propagate label information, enabling unlabeled data points to obtain labels.

Consistency Regularization: This requires the model to make consistent predictions for small perturbations of the input data. For example, the model should produce similar predictions for both the original and augmented versions of the same data point. This enables the model to learn the underlying structure of the data, even without explicit labels.

These methods illustrate how semi-supervised learning leverages unlabeled data to reduce dependency on large labeled datasets. By integrating reward mechanisms, semi-supervised learning can enhance a model's generalization capabilities and improve training efficiency with limited annotated data.

4.4. Establishing a Continuous Update Mechanism for Annotated Data

To address the evolving threat landscape, it is essential to establish a mechanism for continuously updating annotated data to reflect new vulnerabilities and attack techniques. This can be achieved by regularly collecting and annotating the latest vulnerability data, ensuring that models remain adaptable to emerging threats.

4.5. Generating More Training Samples Through Data Augmentation Techniques

Using data augmentation techniques to generate additional training samples can improve the generalization and robustness of a model. The core idea of data augmentation is to create new, plausible training samples from existing data through transformations or other methods, rather than simply duplicating the original data. This approach enhances the diversity and size of the training dataset, thereby improving model performance.

The application of data augmentation techniques to Solidity code differs from that for image or text data. Solidity is a programming language with a strict syntax and structure that must be adhered to for successful compilation and execution as intended. Nonetheless, it is possible to generate more

DOI: 10.54254/2755-2721/108/2025.20839

training samples through certain modifications that do not alter the functionality of the contract, such as changing parameters, adding comments, or renaming variables. These operations can enhance the diversity of training data while maintaining the contract's functionality.

Below are some examples illustrating such data augmentation techniques: (The following examples are written in Solidity.)

- Modifying Numeric Constants in Contracts

Numeric constants, such as initial balances or transaction limits, can be modified randomly or within a specified range. Care must be taken to ensure the modified values are reasonable and do not introduce logical errors into the contract.

```
contract MyContract {                  //Original Contract
uint public myBalance = 100 ether;
function add(uint amount) public returns (uint) {
        myBalance += amount;
        return myBalance;}
}
contract MyContractAugmented {     //Data-Augmented Contract
uint public myBalance = 250 ether; // Modified initial balance

    function add(uint amount) public returns (uint) {
        myBalance += amount;
        return myBalance;}
}
```

- Renaming Variables

By renaming variables (including state variables and function parameters), new training samples can be generated while maintaining the same functionality.

```
contract MyContract {                //Original Contract
      uint public balance;
      function updateBalance(uint newBalance) public {
        balance = newBalance; }
}
contract MyContractAugmented {    //Data-Augmented Contract
      uint public accountBalance;       // Variable name changed
      function setAccountBalance(uint newAccountBalance) public
{ // Function name and parameter name changed
        accountBalance = newAccountBalance; }
}
```

- Adding/Removing Comments

Adding or removing comments in a contract is a simple yet effective method of data augmentation. While this does not impact the program's behavior, it alters the code's presentation.

```
contract MyContract {         //Original Contract
    uint public value = 42;}
contract MyContractAugmented {   //Data-Augmented Contract
```

DOI: 10.54254/2755-2721/108/2025.20839

```
    uint public value = 42; // Removed comment}
```

However, the Solidity language is not suitable for operations like flipping or any type of image processing. Solidity is primarily used for writing smart contracts on the Ethereum blockchain, excelling in handling logic, state management, and transaction verification. Due to performance constraints (limited computational power of the Ethereum Virtual Machine (EVM), which is not suitable for executing complex mathematical operations and intensive data processing), high storage costs, and insufficient privacy guarantees, Solidity is not ideal for computationally intensive tasks.

To address this issue, this paper proposes a method of converting Solidity smart contracts into Python for processing and then converting the augmented samples back into Solidity. The process is divided into the following three steps:

- Convert Solidity Contract to Python

Original Solidity Contract:

```
pragma solidity ^0.8.0;
contract VulnerableContract {
      uint256 public balance;
      function deposit() public payable {
        require(msg.value > 0, "Deposit value must be greater than
0");
        balance += msg.value;}
      function withdraw(uint256 amount) public {
        require(amount <= balance, "Insufficient funds");
        (bool sent, ) = msg.sender.call{value: amount}("");
        require(sent, "Failed to send Ether");
        balance -= amount;}
}
```

Corresponding Python Code:

```
class VulnerableContract:
    def __init__(self):
        self.balance = 0
    def deposit(self, amount):
        if amount <= 0:
            raise ValueError("Deposit value must be greater than
0")
        self.balance += amount
    def withdraw(self, sender, amount):
        if amount > self.balance:
            raise ValueError("Insufficient funds")
        print(f"Sending {amount} to {sender}") # Simulate sending
Ether
        self.balance -= amount
```

- Use Python to Augment Samples and Generate More New Samples

Next, we will create new test cases by modifying the contract's logic or parameters. For instance, we can randomly change deposit amounts or try various withdrawal operations.

DOI: 10.54254/2755-2721/108/2025.20839

Data Augmentation Example:

```
import random
def augment_contract(contract, num_samples=10):
    samples = []
for _ in range(num_samples):# Randomly select operation
        operation = random.choice(['deposit', 'withdraw'])
        if operation == 'deposit':
            amount = random.randint(1, 100)
            contract.deposit(amount)
            sample = f"deposit({amount});"
        else:
            amount = random.randint(1, contract.balance)
            contract.withdraw('0xUser', amount)
            sample = f"withdraw(0xUser, {amount});"
        samples.append(sample)
    return samples
vulnerable_contract = VulnerableContract()# Create contract
instance
samples = augment_contract(vulnerable_contract)for s in samples:
    print(s)
```

- Importing Processed Samples Back into Solidity Contracts or Saving in Formats Usable by Solidity

The final step is to convert these augmented sequences back into Solidity format and integrate them into a new test contract.

```
Integrating Augmented Samples into Solidity:
pragma solidity ^0.8.0;
contract TestVulnerableContract {
      uint256 public balance;
      function deposit() public payable {
        require(msg.value > 0, "Deposit value must be greater than
0");
        balance += msg.value;}
      function withdraw(uint256 amount) public {
        require(amount <= balance, "Insufficient funds");
        (bool sent, ) = msg.sender.call{value: amount}("");
        require(sent, "Failed to send Ether");
        balance -= amount;}
      function test() public { // Test function
        // Insert augmented sample calls here}
}
```

Additionally, the issue of high storage costs can be addressed through layer-2 solutions. Layer-2 solutions refer to additional frameworks or protocols built on top of the base blockchain (Layer-1). These solutions aim to address performance limitations such as slow transaction speeds, high fees, and poor scalability by moving some processing tasks from the main chain to Layer-2. This significantly improves the efficiency and throughput of the system while maintaining decentralization.

DOI: 10.54254/2755-2721/108/2025.20839

4.6. Non-Blockchain Code Training Blockchain Models

Utilizing mature non-blockchain code to train models related to blockchain. In the blockchain domain, various machine learning and deep learning models can be effectively used to enhance data analysis, security monitoring, and decision optimization. First, natural language processing (NLP) models, such as pre-trained BERT and RoBERTa, can be utilized to classify blockchain-related textual data, including comments in smart contracts, whitepapers, and forum posts. Named entity recognition (NER) models can extract key entities from blockchain transaction records, such as addresses and transaction IDs. Image recognition models, particularly convolutional neural networks (CNNs), although seemingly unrelated to blockchain, are very useful for analyzing visual representations of blockchain data, such as transaction graphs and wallet activity heatmaps.

For time series data processing, recurrent neural networks (RNNs) and long short-term memory networks (LSTM) can predict transaction patterns and price changes on the blockchain. Transformer models are also highly suitable due to their ability to handle long sequential data, making them ideal for analyzing long-term transaction histories or blockchain datasets. Anomaly detection models, such as autoencoders and isolation forests, can identify abnormal behaviors in blockchain networks, including fraudulent transactions and malicious activities. Recommendation systems, whether collaborative filtering-based or content-based, can recommend relevant blockchain projects, tokens, or smart contracts to users.

Reinforcement learning (RL) models can optimize decision-making processes in blockchain networks, for example, helping miners choose optimal transaction packaging strategies. Financial models, such as time series prediction models (ARIMA, Prophet) and risk assessment models, can forecast cryptocurrency price trends and assess investment risks. Additionally, general machine learning models, like random forests and support vector machines, can be applied to various classification and regression tasks, such as predicting transaction success rates and risk levels of smart contracts.

The steps to implement these models typically involve selecting a pre-trained model that performs well in the relevant domain and then fine-tuning it using labeled data from the blockchain field to meet specific task requirements. Continuous evaluation and iteration are necessary to continuously optimize the model's performance, ensuring it remains efficient and accurate in the ever-changing blockchain environment. This mechanism for ongoing updates is crucial for maintaining model relevance and effectiveness.

Through transfer learning techniques, models trained in non-blockchain domains can be migrated to blockchain domains, accelerating the model training process and enhancing its performance on specific tasks.

By employing the above methods, critical issues in smart contract annotation and vulnerability detection can be effectively addressed, improving the accuracy and robustness of models and providing stronger security for smart contracts.

5. Conclusion

This paper analyzed the limitations of machine learning in smart contract vulnerability mining and provided feasible solutions. However, some issues still require further research:

(1) Development of Annotation Tools: Design and develop efficient annotation tools and auxiliary annotation tools to enhance the efficiency and reduce the cost of annotation by professionals. These tools should be context-aware and provide highlighting suggestions.

(2) Establishing a Continuous Update Annotation Data Mechanism: This involves real-time updates of vulnerability codes, linking with a database storing vulnerability codes. The updated vulnerability codes, annotated in real-time, are then stored in datasets for subsequent AI training.

DOI: 10.54254/2755-2721/108/2025.20839

(3) Solidity and Python Mutual Conversion: Converting smart contracts from Solidity to Python for processing and then converting the augmented samples back to Solidity. This process is theoretically feasible but presents challenges in practice. Once the logic of smart contracts becomes complex, several issues arise:

a. Low Effectiveness in Translating Solidity to Python: Solidity is a specialized language for writing smart contracts, with many data types specific to blockchain environments for which there are no direct equivalents in Python. Solidity is a statically-typed language requiring all variable types to be determined at compile time, while Python is dynamically typed, allowing changes in variable types during runtime. These differences may lead to inconsistencies in the translated code logic, resulting in low effectiveness in Solidity-to-Python translation.

b. Complexity in Converting Python Back to Solidity: Python's dynamic typing system allows for variable type changes during runtime, which Solidity does not support. This necessitates the addition of numerous type annotations and checks when converting from Python to Solidity, making the process cumbersome and error-prone. Due to the immutability of smart contracts, even if the Solidity code appears correct after conversion from Python, it requires thorough security reviews and testing.

References

[1] *YIN, M., MALKHI, D., REITER, M. K., & et al. (2019). HotStuff: BFT Consensus in the Lens of Blockchain. arXiv:1803.05069.*

[2] *Chen, W., & Zheng, Z. (2018). Blockchain data analysis: status, trend and challenges. Journal of Computer Research and Development, 55(9), 29-46.*

[3] *Bartoletti, M., & Pompianu, L. (2017). An empirical analysis of smart contracts: Platforms, applications and design patterns. In Financial Cryptography and Data Security (pp. 494-509). Cham, Switzerland: Springer, vol. 10323.*

[4] *FU, M. L., WU, L. F., HONG, Z. L., & et al. (2019). Research on mining technology of smart contract security vulnerabilities. Journal of Computer Applications, 39(7), 1959-1966.*

[5] *Xu, Q., Jin, C., Rasid, M.F.B.M., Veeravalli, B., & Aung, K.M.M. (2017). Blockchain-based decentralized content trust for docker images. Multimedia Tools and Applications, 126.*

[6] *Dorri, A., Kanhere, S.S., Jurdak, R., & Gauravaram, P. (2017). Blockchain for IoT security and privacy: The case study of a smart home. In Proceedings - 2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops), 618-623.*

[7] *Li, C., & Zhang, L.J. (2017). A blockchain based new secure multi-layer network model for the Internet of Things. In Proceedings - 2017 IEEE 2nd International Congress on Internet of Things (ICIOT 2017), 3341.*

[8] *Zylmaz, K.R., & Yurdakul, A. (2017). Integrating low-power IoT devices to a blockchain-based infrastructure. In Proceedings of the Thirteenth ACM International Conference on Embedded Software (EMSOFT 17) Companion, 12.*