IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 13 March 2025, accepted 4 May 2025, date of publication 13 May 2025, date of current version 21 May 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3569565

RESEARCH ARTICLE

X-SPIDE: An eXplainable Machine Learning Pipeline for Detecting Smart Ponzi Contracts in Ethereum

LUCA PENNELLA[1], FABIO PINELLI[2], AND LETTERIO GALLETTA[2]
[1]Department of Economics, Business, Mathematics and Statistics, University of Trieste, 34127 Trieste, Italy
[2]IMT School for Advanced Studies Lucca, 55100 Lucca, Italy

Corresponding author: Letterio Galletta (letterio.galletta@imtlucca.it)

Luca Pennella's PhD scholarship has been funded by the European Union-Next Generation EU. Fabio Pinelli and Letterio Galletta have been supported by Project SEcurity and RIghts in the CyberSpace (SERICS) PE0000014, funded by the European Union-NextGenerationEU under the National Recovery and Resilience Plan M4C2 I1.3., CUP: D67G22000340001.

ABSTRACT Blockchain technology is revolutionizing digital asset exchange by eliminating the need for central authority control. However, the decentralized nature of blockchain attracts malicious actors, leading to the proliferation of financial scams, with Ponzi schemes being particularly prevalent. Consequently, there is a growing need to develop automatic detection mechanisms for such scams. So far, the problem has been tackled by considering only classifier performances and with limited focus on the explanation and interpretation of the results. However, interpretability and explainability are crucial when classifier decisions may have economic consequences. This paper introduces X-SPIDE (*XAI Smart Ponzi Identification and Detection*), an explainable machine learning pipeline for Ponzi scheme detection within the Ethereum blockchain that aims to find the trade-off between performance and explainability. X-SPIDE allows for comparing the results of different classifiers, identifying a small set of features that offer strong performance, and understanding how these features contribute to classification — highlighting specific characteristics of malicious contracts. Moreover, we introduce and make publicly available a new comprehensive dataset comprising 7446 smart contracts, incorporating features derived from transaction history, creation, and deployment bytecodes to train and test our pipeline.

INDEX TERMS Anomaly detection, blockchain, code features, eXplainable AI, fraud detection.

I. INTRODUCTION

Blockchain is revolutionizing how individuals and companies exchange digital assets without the control of a central authority. This technology has been successfully exploited for deploying new economic applications, e.g., cryptocurrencies [1] and Decentralized Finance [2]. However, shortly after this technology gained widespread adoption and its economic significance grew, it began to attract the interest of malicious users seeking to exploit the pseudonymity of these platforms and the absence of regulation [3]: on the one hand, they utilize cryptocurrencies to facilitate untraceable currency transfers, evading scrutiny by authorities; on the other hand, they deliver scams to deceive honest users willing to make revenue through cryptocurrencies. Nowadays, many types of scams can be found on blockchain platforms, such as exploits, hacks, and phishing [4]. It is estimated that fraudulent activities in the Bitcoin space amassed a minimum of 11 million USD between 2011 and 2014 [5]. Among the various scams, Ponzi schemes have approached the blockchain world, first on Bitcoin [5] and more recently on Ethereum [6]. *Ponzi schemes* are fraudulent investment operations where older investors obtain returns from new investors' money rather than legitimate business activities. Although the actual conditions to gain money depend on the specific rules of the scheme, a common feature is that participants who want to redeem their investments have to make new participants join the scheme. Participants who

The associate editor coordinating the review of this manuscript and approving it for publication was Thanh Ngoc Dinh.

join later are the most likely to lose their money. Thus, the development of automatic techniques to counter these scams is required to protect average users and to allow them to participate safely in the blockchain economy. Our focus here is on Ethereum, one of the most famous blockchain systems, and smart contracts to deliver Ponzi schemes called *smart Ponzi contracts*. Although several papers have addressed Ponzi scheme detection on Ethereum [7], [8], [9], [10], [11], [12], there is still no conclusive solution. On the one hand, there is a lack of publicly available datasets that can be used to train and test effective classifiers. On the other hand, the problem has been tackled so far by considering only the optimization of the classifier's performance as a target, paying no attention to explaining and interpreting classification results. Interpretability and explainability are crucial when classifier decisions have economic consequences. An automatic and explainable technique for classifying smart contracts is needed to fill such gaps, which can be safely used as the backbone for developing new fraud detection tools.

A. CONTRIBUTIONS

In this paper, we contribute towards filling this gap by introducing X-SPIDE (*XAI Smart Ponzi Identification and Detection*), an explainable machine-learning pipeline to detect smart Ponzi contracts. Besides the classification of contracts and Ponzi scheme detection, X-SPIDE allows for comparing the results of different classifiers trained on different datasets. Moreover, it provides a procedure to refine the set of features while maintaining comparable classification performance to the original feature set. Finally, it relies on *eXplainable Artificial Intelligence* (XAI) techniques and tools to explain the model and carefully inspect misclassified contracts. We train and test our pipeline on a new and reusable dataset featuring 7446 unique real-world smart contracts. Within this dataset, 6566 (88.18%) are identified as non-Ponzi contracts, and 880 (11.82%) as Ponzi contracts. The dataset contains information about the transaction history of the contracts as well as their bytecode. We apply a thorough cleaning process on this dataset to detect and remove duplicates and non-compliant contracts (some bytecodes were not retrievable as the contracts appeared destroyed or were not recoverable on the blockchain). Then, we split the clean dataset into eight distinct datasets characterized by different bytecode features but the same transaction features. More precisely, four datasets incorporate features extracted from the *creation bytecode*, i.e., the code used to create and deploy the contract on the blockchain (creation code) plus the code of the contract itself (runtime code) and transaction features. The other four datasets instead include solely features obtained from the *deployed bytecode* (runtime code only) and the transaction features. We conduct several experiments on these datasets to train and test our machine learning pipeline. As a side effect, our experiments also allow us to answer the following six research questions: (1) Which is the pair (model, dataset) that provides the highest Recall?; (2) How does the usage of creation code and deployed code impact classification performance?; (3) What is the impact of account features on the classification performance?; (4) Does a subset of features exist that ensures performance comparable to the entire dataset? (5) What are the most important features?; (6) What are the characteristics of the misclassified contracts?

In conclusion, the main contributions of our paper can be summarized in three main aspects: first, we define a new machine learning pipeline that incorporates eXplainable Artificial Intelligence (XAI) outputs to elucidate model behavior. Second, we introduce a new dataset that is publicly available for use. Last, through experiments, we analyze the different performances between created and deployed bytecode, investigate the significance of features, and analyze the key characteristics of misclassified contracts.

B. STRUCTURE OF THE PAPER

In Section II, we provide the reader with the relevant background to understand the technical development of the paper. In Section III, we compare our work with the relevant literature. Section IV formally defines our pipeline. In Section V, we introduce our datasets, our features, and how we collect them. Section VI reports on our experimental evaluation and the results we achieved. Finally, in Section VII, we draw some conclusions and discuss possible future work.

This is an extended version of the conference paper [13]. We extend the previous work in several directions: (1) we precisely define our pipeline X-SPIDE, characterizing what the input and the output of each stage are; (2) we introduce here a large dataset comprising more smart contracts and updated values of the features; we also devise a systematic process that cleans the dataset by addressing duplicates and non-compliant contracts; (3) our machine learning pipeline work considering both transaction and bytecode features; (4) we study how different types of bytecode impact classification performance; (5) we experiment with a different set of feature combinations (see Table 2), selecting the best one; (6) we study the interpretability of our model using different XAI techniques such as Shapley values and partial dependency plots, and we provide an interpretation of the results considering the smart contract's bytecode.

II. BACKGROUND

Here, we provide the required information about blockchain systems, the Ethereum platform, and Ponzi schemes to clarify the context and the terminology for readers unfamiliar with this application domain.

A. A GLIMPSE OF ETHEREUM

A blockchain platform is a dynamic network of peer-to-peer nodes that maintain a replicated data structure, called the *blockchain*, that globally records the occurrence of certain events. The network nodes own a local copy of the blockchain and update it upon reception of special messages, called

transactions. To ensure the blockchain's consistency, these systems rely on a consensus protocol that imposes a total order on the updates performed by the nodes.

Ethereum is one of the most famous blockchain platforms. In the recent version, it uses a Proof-of-Stake consensus algorithm where special nodes, called validators, propose blocks containing transaction bundles. The consensus is designed to be cryptographically and economically secure, requiring potential attackers to hold a significant amount of ether (ETH in short), the Ethereum cryptocurrency. A reward system incentivizes honest participation, while penalties deter malicious behavior among stakers.

A distinguishing feature of Ethereum is that it provides a decentralized *virtual machine* that can execute programs, called *contracts* and written in the EVM bytecode language. The EVM bytecode is a low-level language where a program/-contract is a sequence of instructions, each characterized by its own opcode and its operands on the stack. From a technical point of view, Ethereum implements a state transition system. The current state of this transition system is stored on the blockchain and includes all the accounts and their balances. There are two types of accounts: the *external accounts* that users control and *contract accounts* that are controlled by a contract's bytecode. Intuitively, contracts can be seen as objects in object-oriented languages: they have fields representing the state of the contract and a set of functions that users or other contracts can invoke. We discuss an example of a smart contract in the next sub-section. An account is uniquely identified by its *address*. Every account is equipped with a particular field, called the *balance*, which represents the amount of ETH owned by the account. Users can send transactions, called *external transactions*, to the Ethereum network. Through transactions, a user can *(i)* create new contracts; *(ii)* invoke a function of a contract; *(iii)* transfer Ether to contracts or other users. All external transactions are recorded on the blockchain. When a contract receives an external transaction, it reacts by executing one of its functions that, in turn, fires other transactions. These transactions are called *internal transactions* and are not recorded on the blockchain but still impact the balance of users and the state of other contracts.

B. SMART PONZI CONTRACTS

Ponzi schemes are classic frauds concealed as "high-yield" investment programs. The initiator of the scheme generates returns for existing investors through revenue paid by new investors rather than from legitimate business activities or profits of financial trading. More in general, the U.S. Securities and Exchange Commission (SEC)[1] defines Ponzi schemes as:

> A Ponzi scheme is an investment fraud that involves the payment of purported returns to existing investors from funds contributed by new investors. Ponzi scheme organizers often solicit new investors by promising to invest funds in opportunities that generate high returns with little or no risk. With little or no legitimate earnings, Ponzi schemes require a constant flow of money from new investors to continue. Ponzi schemes inevitably collapse, most often when it becomes difficult to recruit new investors or when a large number of investors ask for their funds to be returned.

[1] See https://www.sec.gov/spotlight/enf-actions-ponzi.shtml

Although the actual conditions to gain money depend on the specific rules of the scheme, a common feature is that a user who wants to redeem her investment has to make new users join the scheme. In this way, the schemes create a pyramid of investors, where the initiator is at the top, and the investors at level $l + 1$ compensate for the investment of those at level l. Once a scheme collapses because no more investors join it, those at the top levels of the pyramid gain money, while those at the bottom lose it.

The spread of cryptocurrency and smart contracts has created new opportunities to deploy this kind of fraud. Indeed, it is possible to find samples of smart contracts implementing Ponzi schemes, called *smart Ponzi contracts*, deployed on the main blockchain platforms like Ethereum and Bitcoin. This paper focuses on the Ethereum platform. According to the literature [7] smart Ponzi contracts have several attractive features that make them useful for delivering scams:

1) The initiator of a smart Ponzi could stay anonymous, since deploying the contract on the blockchain and withdrawing money from it only requires an Ethereum account that does not reveal her real identity.
2) Once deployed on the blockchain, smart contracts are "unmodifiable" and "unstoppable". Thus, no central authority could terminate the execution of the scheme, seize the money, and refund the victims.
3) Since the code of smart contracts is public and immutable, and its execution is automatically enforced by the blockchain platform, investors may believe that no one can take advantage of their money and that they could eventually gain the declared interests.

The most significant feature of a smart Ponzi is the policy used to redistribute new investments among participants, i.e., how the money flows. This requires a smart Ponzi to maintain a data structure storing participants' information and to implement a strategy for redistributing dividends.

Identifying redistribution behavior is crucial to classify a contract as a smart Ponzi. Also, it is challenging because many other kinds of contracts, e.g., gambling games, may have similar behavior, which may induce many false positives. Bartoletti et al. [6] proposed the following four requirements to classify a smart contract as a Ponzi scheme:

R1 the contract redistributes money to the investors according to a given logic;
R2 the contract receives money only from the investors;

R3 each investor makes a profit if a certain number of investors subsequently join the contract, investing a certain amount of money;
R4 the later an investor joins the contract, the higher the risk for the investor to incur a loss.

A smart contract is classified as a smart Ponzi when it satisfies *all* four requirements. Note that requirement R1 rules out contracts that provide users with some assets but do not implement a logic to distribute them to participants, e.g., tokens; requirement R2 ensures that a participant invests a certain amount in joining the contract; requirement R3 demands a constant flow of new investments for investors to make a profit; requirement R4 characterizes the fraudulent nature of smart Ponzi contracts because it reflects the fact that making a profit for investors is likely impossible after a certain point in time: too many victims must join the scheme for the contract to have enough money to reward all the participants. Thus, the scheme collapses when this happens. Note that the requirements above impose no condition on whether the money was received or not by the initiator of the scheme. We will study the need for such a condition in our experimental evaluation of Section VI.

Typically, smart Ponzi contracts are categorized into four types according to their redistribution strategies: *Tree-scheme*, *Chain-scheme*, *Waterfall-scheme*, and *Handover-scheme*. Since we do not consider these categories here, we refer the interested reader to the relevant literature [6].

We now clarify a smart Ponzi contract through a small yet real example. Consider the code snippet in Figure 1. The code is written in Solidity, a high-level, object-oriented programming language primarily used for creating smart contracts. It features static typing and supports inheritance, reusable libraries, and complex user-defined types, ultimately compiling into EVM bytecode. The code is extracted from a contract named *Multiplier* that is deployed on Ethereum with the address

`0x30D1B797365F936300055A704A902124467B8b14`

and implements a chain scheme. The code snippet resembles a class declaration and consists of two parts: the declaration of fields and data, and the declaration of two functions. Lines 2 to 18 are constant and variable definitions used to record the state of the contract: `PROMO` is the address of the initiator of the contract; `PROMO_PERCENT` is the percentage of the investment that each participant pays to the initiator as a fee; `MULTIPLIER` is a constant denoting how much participant's investment must be multiplied; the structure `Deposit` records the address, the amount of deposit, and the expected reward of each investor. The array `queue` stores all the investors in order of arrival, and the integer `currentHead` is the index of the head of the queue. The contract receives Ether from investors through a transaction, and when this happens, it executes the special function with no name, called *fallback function* at line 20.

The received Ether is automatically transferred to the contract balance that is accessible to the programmer via the read-only variable **`balance`**. Information about the received transaction is accessible via the special object **`msg.`** In this case, the fallback function, if the received amount is positive (i.e., **`msg.value`** > 0), records the address (**`msg.sender`**), the investment and the expected reward of the investor (computed taking into account the factor `MULTIPLIER`) in the queue (line 23). Then, it calculates and pays the fee to the initiator (lines 28-27). Finally, it pays previous investors through the private function `pay`. This function (lines 35-61) scans the queue and pays the expected revenue to the investors as long as the balance of the contract is positive. Once an investor has received her reward, she is removed from the queue. From this example, it is easy to see that the scheme's initiator receives some money for each new investor and that investor *A* will be repaid only when she is at the front of the queue. This event occurs only when enough new investors join the scheme to repay the old investors preceding *A* in the queue.

III. RELATED WORK

Since the inception of Bitcoin in 2009, cryptocurrencies and blockchain systems have attracted the attention of cybercriminals, who exploit them to carry out potentially untraceable scams. Since the full transaction history is publicly available and provides accurate records of user behavior, several papers [4], [8], [9], [10], [11], [12], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26] have proposed machine learning techniques to detect possible frauds and scams. Below, we describe the contribution of these papers, and then we highlight the distinctive features of our work compared to them.

Bartoletti et al. [7] study the problem of identifying Ponzi schemes in Bitcoin using data mining techniques. In particular, they released a public dataset of Bitcoin addresses and an open-source tool to build such a dataset; then, they apply different classification algorithms (Random Forest, Bayes Network, and Ripper) and systematically evaluate them to identify the best discriminating features for detecting Ponzi schemes in Bitcoin. In a subsequent paper, Bartoletti et al. [6] consider smart Ponzi contracts in Ethereum. First, they define four behavioral criteria that characterize a contract as a smart Ponzi and produce a dataset with several samples satisfying such criteria. Then, they perform several analyses by hand on a subset of their dataset, e.g., about the security of their code and the fairness of the distribution policy. Chen et al. [8] provide a reusable dataset with real-world samples and evaluate different classifiers to determine which presents the best classification performance. They use two different classes of features: the account features taken from the transaction history and code features extracted from the contract's bytecode. Chen et al. [10] propose SADPonzi, a detection approach based on symbolic execution. This approach analyzes the bytecode of contracts to extract semantic information about the execution and

```
contract Multiplier {
  //Address of the "promoter" of the contract: she receives fee for each transaction
  address constant private PROMO = 0x5D5fe29339592eEb51c43E54F0a81cA7642B6d2b;
  //Percent received by the "promoter"
  uint constant public PROMO_PERCENT = 7;
  //How many percent for your deposit to be multiplied
  uint constant public MULTIPLIER = 121;

  //The deposit structure holds all the info about the made deposits
  struct Deposit {
      address depositor; //The depositor address
      uint128 deposit;   //The deposit amount
      uint128 expect;    //How much we should pay out (initially it is 121% of deposit)
  }

  Deposit[] private queue;  //The queue of investors
  uint public currentHead = 0; //The index of the first depositor in the queue.

  //This function receives all the deposits stores them  in the queue and make immediate payouts
  function () public payable {
      if(msg.value > 0){
          //Add the investor into the queue. Mark that he expects to receive 121% of deposit back
          queue.push(Deposit(msg.sender, uint128(msg.value), uint128(msg.value*MULTIPLIER/100)));

          //Send some promo to enable this contract to leave long-long time
          uint promo = msg.value*PROMO_PERCENT/100;
          PROMO.send(promo);

          //Pay to first investors in line
          pay();
      }
  }

  //Used to distribute all the money on contract to the first investors starting from the head of queue
  function pay() private {
      // The current balance of the contract
      uint128 money = uint128(address(this).balance);

      //Cycle on the queue
      for(uint i=0; i<queue.length; i++){
          uint idx = currentHead + i;  //the index of the currently first investor

          Deposit storage dep = queue[idx]; //the info of the first investor

          if(money >= dep.expect){  //If we have enough money on the contract to fully pay to investor
              dep.depositor.send(dep.expect); //Send money to him
              money -= dep.expect;            //update money left

              //the investor is fully paid and she is removed from the queue
              delete queue[idx];
          }else{
              //No enough money, so partially pay to investor
              dep.depositor.send(money); //Send to her the money left
              dep.expect -= money;       //Update her expected amount
              break;
          }
      }

      currentHead += i; //Update the index of the current first investor
  }
}
```

FIGURE 1. Example of smart ponzi contract: the code of the *Multiplier* contract.

identifies investor-related transfer behavior and the distribution strategies adopted by the scheme. SADPonzi performs the classification only by looking at the code, not the transaction history. Wang et al. [9] propose a classifier based on Long-short Term Memory Network for detecting smart Ponzi. Fan et al. [16] propose PonziTect, a smart Ponzi contract detection method based on ordered boosting that classifies contracts considering only their bytecode. They adopt data augmentation to solve the problem of imbalanced samples among the two classes by increasing the proportion of smart Ponzi contracts at the boundary. Lou et al. [11] use convolutional neural networks to build a smart Ponzi detector. They focus mainly on bytecode features, and their pipeline is quite standard: First, they transform smart contracts into single-channel images and then adopt the spatial pyramid pooling method to ensure that the generated images have

the same size. Ibba et al. [17] builds a machine learning model that uses account features and the bytecode of the contracts. They also consider Solidity source code and apply text classification techniques to extract further features to use in the classification. They tested their approach with decision trees, support vector machines, and naive Bayes. Jin et al. [18] propose HFAug, a generic Heterogeneous Feature Augmentation module that can be adapted to various existing Ponzi detection methods. The module captures heterogeneous information associated with account behavior. Zheng et al. [12] propose MulCas, a method for detecting smart Ponzi schemes that works solely on the bytecode. Peng et al. [19] use eight classification algorithms such as Logistic Regression, Decision Trees, Support Vector Machine, Random Forests, Extremely Randomized Trees, Gradient Boosting Machines, XGBoost, and LightGBM to build a model that detects Smart Ponzi schemes. They focus solely on the functionality based on the opcodes of smart contracts on Ethereum. Zhang et al. [23] introduce a method for identifying smart Ponzi using an enhanced version of the LightGBM algorithm. Their approach combines bytecode features with user transaction data and opcode frequencies. Chen et al. [20] focused on extracting features from user accounts and operation codes from smart contracts to build a classification model for detecting latent Ponzi schemes. Jin et al. [22] propose a dual-channel early warning framework dubbed Ponzi-Warning. Their proposal performs feature extraction and fusion on code and transaction features. They present a temporal evolution augmentation strategy for generating transaction graph sequences, increasing the data scale, and introducing temporal information. Yu et al. [21] model Ponzi scheme identification and detection as a node classification task and propose a detection model based on graph convolutional networks. Jacinta et al. [24] propose an approach that detects Ponzi schemes on Ethereum using random forest, neural network, and K-nearest neighbor. Cai et al. [25] devised a methodology that employs a graph to represent transaction-related semantics within a smart contract and uses a graph convolutional network to identify potential Ponzi-like transaction patterns. Wang et al. [26] present a detection approach that integrates opcode context analysis with the Adaptive Boosting (AdaBoost) algorithm. The methodology employs the n-gram algorithm to capture extensive opcode features related to contracts and integrates them with features derived from contract accounts. Below, we compare our work with the above-cited papers. This paper's main novelties and distinctive features are the introduction of an explainable machine learning pipeline and the study of the impact of different types of bytecodes on classification performances. Moreover, while other papers in the literature focus solely on optimizing classification performance, we prioritize finding the right balance between performance and interoperability. We believe that this focus becomes crucial when decisions can impact economic systems like Ethereum. More precisely, we consider both account and code features to detect smart Ponzi schemes. Additionally, we enhance the existing dataset by introducing new transaction features to capture better requirements R1-R4 of Section II and incorporating additional contracts. Furthermore, we integrate various Explainable Machine Learning techniques to foster a transparent understanding of the decision-making of our model. Going beyond traditional analysis, we scrutinize the classifier's response to misclassified contracts. Through these techniques, we aim to pinpoint a subset of features demonstrating comparable performance and contribute to an enhanced understanding and interpretation of the classification model. Additionally, we encourage further investigation and exploration within this domain by publicly making our dataset available for further research. Finally, Feng et al. [27] propose IDPonzi, an interpretable model for detecting Ponzi schemes on the blockchain. Compared to our approach, their dataset is smaller than ours, containing 200 Ponzi schemes, whereas our dataset includes 880. Moreover, their model utilizes only bytecode features and achieves lower performance in terms of Recall. From an explainability perspective, in contrast to theirs, our pipeline includes not only SHAP values but also partial dependence plots. Table 1 provides an overview of the characteristics of various related works and highlights the key differences, both among themselves and in comparison to this paper. A checkmark (✓) means the presence of a specific feature, while a crossmark (✗) indicates its absence.

IV. METHODOLOGY AND PIPELINE DESIGN

This section introduces X-SPIDE, an interpretable machine-learning pipeline for identifying smart Ponzi contracts. Our pipeline goes beyond just classifying contracts and detecting Ponzi schemes, it also allows for *(i)* an easy comparison of the results produced by various classifiers trained on different datasets; *(ii)* feature refinement to reduce the feature set used by the model while ensuring that the classification performance remains comparable to the original set; *(iii)* explainability through XAI techniques and tools to clarify the model's decisions and thoroughly examine any misclassified contracts. More precisely, our pipeline consists of three stages: S_1 model and dataset selection; S_2 feature selection; S_3 model explanation. The first stage S_1 receives as input a set of classifiers C, a set of datasets D, and an evaluation metric m (e.g., Recall, Precision, AUC, etc.). It trains and tests all combinations of classifiers and datasets using a grid-search and cross-validation procedure and returns in output the classifier $c_i \in C$ (with its optimized hyperparameters) and the dataset $d_j \in D$ that outperforms the other combinations according to the metric m.

Formally, let CL be the set of classifiers, DS the set of datasets, M the set of evaluation metrics, we define the function $S_1 : \wp(CL) \times \wp(DS) \times M \mapsto CL \times DS$ as follows:

$$
\begin{aligned}
&S_1(C, D, m) = (c_i, d_j) \\
&\text{where} \quad c_i \in C \text{ and } d_j \in D \\
&\text{and} \quad \forall c_k \in C \setminus \{c_i\}. \forall d_k \in D \setminus \{d_j\}.\ m(c_i, d_j) \geq m(c_k, d_k)
\end{aligned}
$$

TABLE 1. A summary of the comparison between related work and this paper.

Authors	Used Methods	Year	Code Features	Account Features	Creation vs. Deployed Bytecode	Data availability	XAI
Chen et al. [20]	Boosting-based algorithm	2018	✓	✓	✗	✓	✗
Chen et al. [8]	Boosting-based algorithm, random forest	2019	✓	✓	✗	✓	✗
Peng et al. [19]	Boosting-based algorithm, Random Forest	2020	✓	✗	✗	✗	✗
Fan et al. [16]	Boosting-based algorithm	2020	✓	✗	✗	✗	✗
Lou et al. [11]	Convolutional neural networks	2020	✓	✗	✗	✗	✗
Chen et al. [10]	Symbolic execution	2021	✓	✗	✗	✓	✗
Wang et al. [9]	Long-short Term Memory Network	2021	✓	✓	✗	✗	✗
Ibba et al. [17]	Decision trees, SVM, and naive Bayes	2021	✓	✓	✗	✗	✗
Zhang et al. [23]	Enhanced version of LGBM	2021	✓	✓	✗	✗	✗
Yu et al. [21]	Graph convolutional networks	2021	✗	✓	✗	✗	✗
Jin et al. [22]	Dual-channel Early Warning Framework	2022	✓	✓	✗	✗	✗
Zheng et al. [12]	Multi-view Cascade Ensemble algorithm	2022	✓	✓	✗	✓	✗
Jacinta et al. [24]	Random forest, neural network, and K-nearest neighbor	2023	✓	✗	✗	✗	✗
Cai et al. [25]	Graph convolutional networks	2023	✓	✓	✗	✗	✗
Wang et al. [26]	Boosting-based algorithm	2023	✓	✓	✗	✗	✗
Feng et al. [27]	Boosting-based algorithm	2024	✓	✗	✗	✗	✓
This paper	Decision Tree, random forest, and LGBMC	2024	✓	✓	✓	✓	✓

This stage produces the best pair of classifiers and a dataset, and allows for other analyses: we can investigate how different classifiers perform on various datasets, gaining insights from the features within each dataset. Moreover, considering different metrics, the stage helps understand which dataset is more suitable for optimizing a specific metric. For instance, a certain dataset with specific features might yield a higher recall than others. This dataset could be a better choice if the analyst aims to minimize false negatives by ensuring that as few threats as possible go undetected.

The second stage of the pipeline S_2 takes in input the classifier c_i and the dataset d_j produced by S_1, and it finds a subset of the features of d_j that increases or, at least, preserves the performance achieved by the classifier c_i when trained on d_j. Intuitively, S_2 adopts the Recursive Feature Elimination (RFE) algorithm that trains and tests c_i in several runs to detect the less contributing features to be removed in the next run. This process continues until the performances remain within a given threshold ϵ or the number of features to be considered reaches 10. Formally, let F_d denote the set of

features of a dataset d, we define the function $S_2 : CL \times DS \times M \mapsto DS$ as follows:

$$S_2(c, d, m) = d'$$
$$\text{where} \quad |m(c, d') - m(c, d)| \leq \epsilon \text{ and } F_{d'} \subseteq F_d \text{ and } \epsilon \geq 0$$

The stage S_2 of the pipeline allows us to identify a smaller set of features that ensures a good classification performance with respect to the optimized metric, to work with a more manageable dataset, and to improve the time for training and testing. Moreover, this stage also suggests which features mainly contribute to an accurate classification.

The stage S_3 provides results that can be used to describe the model behavior. In particular, S_3 computes the Shapley values Sh, partial dependencies PD for each smart contract for all considered features, and the misclassified contracts MC. More formally, we can define S_3 as the following:

$$S_3(c, d) = (Sh, PD, MC)$$

where c is a classifier and d is a dataset. We describe below how S_3 computes the elements of the resulting triple. The computation of Shapley values Sh assesses the contribution of each feature to the model's predictions, evaluating the model against all possible feature combinations. In practice, the Shapley value of a feature i for a specific instance x (a smart contract in our case), in symbols $Sh(x, i)$, is calculated as follows:

$$Sh(x, i) = \sum_{S \subseteq \{1,\ldots,p\} \setminus \{i\}} \frac{|S|!(p - |S| - 1)!}{p!} [c(x_S \cup \{i\}) - c(x_S)]$$

where

- S represents a feature subset excluding feature i;
- $|S|$ denotes the cardinality of the set S;
- x_S is the sample x restricted to features in subset S;
- $c(x_S \cup \{i\})$ is the classifier prediction when feature i is included;
- $c(x_S)$ is the classifier prediction without feature i;
- p is the total number of features.

Intuitively, the formula above computes the contribution of feature i by comparing predictions with and without i across all possible feature subsets S, weighed by the subset size and total feature count. Finally, these contributions are summed to determine the Shapley value for feature i. Therefore, the set Sh includes all the $Sh(x, i)$ for all features i and all the instances x in the dataset d. Shapley values are considered as XAI global and local models.

The partial dependencies PD determine the relationship between a feature i and the predicted class, i.e., Ponzi or non-Ponzi. The partial dependence of a feature i is computed as follows:

$$PD(i) = \frac{1}{|d|} \sum_{m=1}^{|d|} \left(\frac{1}{|S|} \sum_{w \in S} c(x[i \mapsto w]) \right)$$

where

- $|d|$ is the total number of samples in the dataset;
- S is the set of values taken by the feature i in the dataset d;
- $|S|$ is the size of S;
- $x[i \mapsto w]$ is the sample x where the value of the feature i is replaced with w (the values of the other features are unchanged).

Intuitively, the formula above is computed by varying the value of the feature of interest and calculating the average classifier prediction while keeping the value of other features constant. These values are useful for interpreting classifier responses to feature changes and can reveal non-linear or complex relationships between the features and the model output. The partial dependencies are XAI global models since they consider all instances and give a statement about the global relationship of a feature with the predicted outcome.

The set of Sh allows us to plot the model's global behavior as beeswarm plots of the feature importance. The set of DP helps investigate the relationship between the target response and a selected set of input features through Partial dependence plots (PDPs). Finally, stage S_3 helps study misclassified contracts MC and gain further insights regarding the behavior of the model and the specificity of these contracts. In particular, it applies SHAP on the subsets False Positive and False Negative independently, then it retrieves their code, enabling manual inspection by analysts, and finally, it supports the study of the feature importance for specific contracts.

Figure 2 summarizes pipeline and describes what each stage takes as input and returns as output. In Section VI, we implement our pipeline using Python and the Scikit-learn library, and we adopt XAI library Shap[2] and partial dependence plots (PDPs) [28].

V. DATASET AND FEATURE DESCRIPTIONS

The contracts used to train and test our pipeline for the classification of Ponzi and non-Ponzi contracts are obtained by merging the lists of addresses and the corresponding labels of [6], [8], [10], [12], and [18]. We obtained a new list with 7962 unique contracts. However, some bytecodes were not retrievable as the contracts appeared destroyed. After compiling a comprehensive list of accessible smart contracts, we download the values of the relevant features. Upon completing this initial analysis and cleanup, we obtain a dataset with 7446 contracts, comprising 6566 non-Ponzi and 880 Ponzi contracts.

Each contract is enriched with 5 types of features: (1) creation bytecode with absolute frequency; (2) creation bytecode with weighted frequency; (3) deployed bytecode with absolute frequency; (4) deployed bytecode with weighted frequency; (5) account and transaction history features. The first four types are obtained by analyzing the bytecode associated with each smart contract. The last

[2]https://shap.readthedocs.io/en/latest/index.html

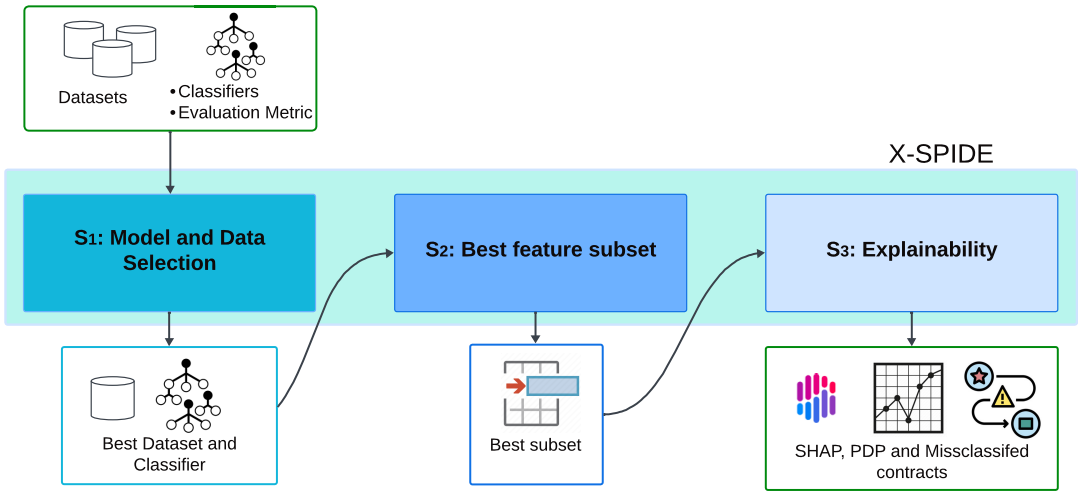

FIGURE 2. Structure and dataflow of the X-SPIDE pipeline.

one takes into account the information associated with each smart contract about its activities on the blockchain. We used eight possible combinations of these families of features, and each combination led to a dataset. Applying X-SPIDE to different combinations of features allows us to understand the contribution of each feature's family to the classification process and compare the relative performance. In the following, we describe the process and the type of features we extracted from the bytecode and transaction activities.

A. CODE FEATURES

Bytecode is an abstract instruction set designed for efficient execution by a software interpreter or a virtual machine. Unlike human-readable source code, bytecode is expressed in a numerical format: the bytecode comprises a series of bytes, each of which refers to a specific operation whose semantics is specified by the Ethereum yellow paper [29]. For example, the operation with opcode 0x02 represents the multiplication at the EVM level (denoted with the name MUL).

The reason for adopting bytecode is its widespread use in various aspects of smart contracts analysis, such as contract vulnerability and category classification [6], [29], [30]. In particular, we first define 4 types of features, taking into account the bytecode associated with each smart contract.

The first two feature sets consider the creation bytecode: the first set is obtained by counting the number of occurrences of each specific opcode in the code (call it absolute frequency); whereas the second considers the percentage of the occurrences (call it weighted frequency). The other two sets of features are similar, but consider the deployed bytecode.

We compute the values of these features as follows. We gather the creation bytecode associated with each smart contract from the online service etherscan.io,[3] and the deployed bytecode from the online service infura.io.[4] Additionally, we disassemble the bytecode into an equivalent series of opcodes with *evmdis*,[5] a tool to disassemble all instructions in bytecode. Furthermore, we unify opcodes with the same logical function into a single one. For example, the various variants of the PUSH instruction (from PUSH1 to PUSH32) are considered a single generic instruction PUSH. Thus, we associated each contract with two lists of opcodes, one for the creation bytecode and one for the deployed bytecode. For both lists, we compute the absolute and weighted frequency of each opcode, obtaining 4 lists to be associated with the corresponding smart contract.

A summary of the steps we implemented for extracting code features is in Figure 3. At the end of the extraction, we obtained 76 different features corresponding to the frequency of different opcodes within the various contracts analyzed.

B. ACCOUNT FEATURES

The intrinsic characteristics of a Ponzi scheme determine its behavior. There are at least three characteristics of Ponzi contracts: (1) these contracts usually send Ether to accounts once investing to the contract; (2) some accounts receive more payments than investments. For example, the creator who charges fees frequently from the contract; and (3) the contract balance may be low, as a Ponzi scheme always tries to maintain an image of fast and high returns.

Below, we report the list of features for each contract:

1) *Balance*: the difference between the amount of ETH in input and the ETH in output;
2) *Lifetime*: the difference between the time of the first and the last transaction made or received;
3) *Tx_in*: the number of input transactions;

[3]https://etherscan.io/

[4]https://www.infura.io/

[5]https://pyevmasm.readthedocs.io/en/latest/index.html

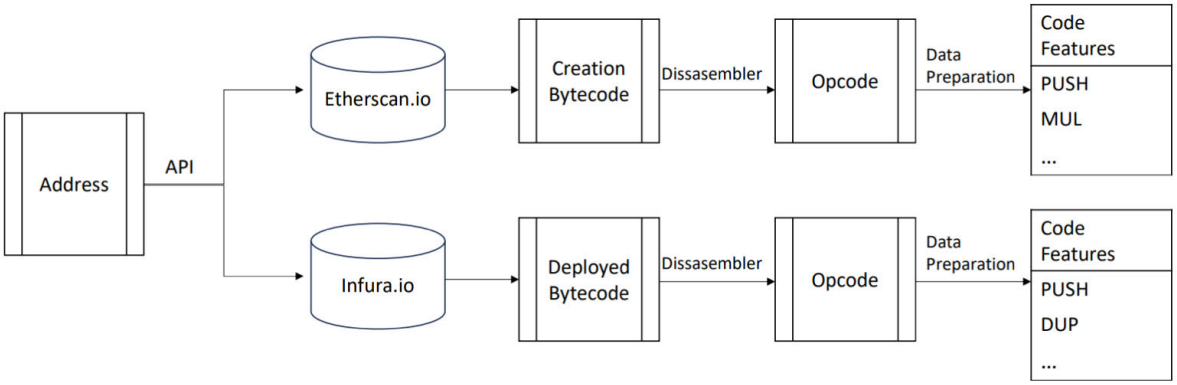

FIGURE 3. Scheme of the framework implemented for opcode extraction.

4) *Tx_out*: the number of output transactions;
5) *Investment_in*: the number of input transactions that deposit an amount of ETH in the contract;
6) *Payment_out*: the number of output transactions paying an amount of ETH;
7) *#addresses_paying_contract*: the number of distinct addresses that paid the contract;
8) *#addresses_paid_by_contract*: the number of distinct addresses paid by the contract;
9) *Mean_v1*: the average of the differences of the number of input/output transactions from/to the same address;
10) *Mean_v2*: the average of the differences of the amount of ETH received and paid by the contract involving the same address;
11) *Sdev_v1*: the standard deviation of the differences in the number of input and output transactions involving the same address;
12) *Sdev_v2*: the standard deviation of the differences between the amount of ETH in and out involving the same address;
13) *Paid_rate*: the ratio between Tx_in and Tx_out;
14) *Paid_one*: the ratio between the number of investors paid many times and the number of total investors;
15) *Investment_in/Tx_in*: the ratio between Investment_in and Tx_in;
16) *Payment_out/Tx_out*: the ratio between Payment_out and Tx_out;
17) *Percentage_some_tx_in*: the percentage of active days with at least one input transaction during the contract lifetime;
18) *Sdev_tx_in*: the standard deviation of the number of transactions per day;
19) **Percentage_some_tx_out**: the percentage of active days with at least one output transaction during the contract lifetime;
20) **Sdev_tx_out**: the standard deviation of the number of transactions in output per day;
21) **Initiator_get_eth_wo_investing**: this feature is 1 if the contract initiator has earned ETH without any investment, 0 otherwise;
22) **Initiator_get_eth_investing**: this feature is 1 if the initiator has earned ETH investing in the contract, 0 otherwise;
23) **Initiator_no_eth**: this feature is 1 if the Initiator has obtained no ETH investing in the contract, 0 otherwise.

The first 18 features are inherited from the literature, while the last 5 in **bold** are the ones we introduce. Below, we briefly comment on some of the features, explaining the rationale behind them. For example, Feature 14 estimates how often a contract interacts with accounts it already knows. A high value of this feature means more interactions (we expect it to happen for a smart Ponzi). Features 15 and 16 aim to capture the requirement R3: the contract redistributes money to the investors according to a given logic by measuring the percentage of transactions distributing ether among Ethereum addresses. Since we expect that non-Ponzi contracts present a lower percentage than Ponzi ones, these features are uniquely based on Ether exchanges. Features 17 and 19 monitor the number of input and output transactions per day over time. A small value of Feature 21 (respectively 19) indicates that the contract was active for a few days, considering input transactions (output, respectively, for Feature 19). On the contrary, a high value means the contract presented a more regular activity over its lifetime. We also consider the standard deviation of the number of daily transactions to capture the variability during the contract life. Feature 15 considers input and output transactions, respectively. This feature measures whether a contract sends or receives transactions only in a few days or regularly. In particular, we expect that Ponzi contracts will have a short lifetime in which they will receive several user investments. Indeed, since making a profit for investors is likely impossible after a certain time, the scheme collapses, and the contract will no longer receive new investments. The last three features, 21,

22, and 23, verify whether the initiator received money from the contract. Indeed, in smart Ponzi contracts, the initiator usually receives a certain amount of money, even without an initial investment. We add these features to study whether receiving a certain amount of money identifies this fraud. Therefore, we expect most contracts labeled as Ponzi to have Feature 21 and Feature 22 equal to 1. On the contrary, Feature 23 equals 1 for non-Ponzi contracts since it is quite unusual to find the initiator of a smart Ponzi that receives no Ether. Our experimental evaluation shows that the new features improve the classification performance. These experiments and a thorough description of the features are reported in the conference version of this paper [13].

C. DATASETS

As previously stated in stage S_1 of the pipeline, we process 8 datasets, selecting the best one, and we assess the impact of various features on classification quality. Subsequently, a ninth dataset was derived to identify a concise and efficient set of features during the stage S_2.

In particular, we aim to evaluate how the creation and deployed bytecode influence the classification task. Therefore, we consider two groups of datasets, one containing features extracted from the creation bytecode and another containing features related to the deployed bytecode.

The datasets utilizing creation bytecode are as follows:

- *Transaction plus frequency opcode* dataset, which includes the absolute frequency of opcodes and account features (7446 smart contracts x 99 features).
- *Transaction plus weighted opcode* dataset, which includes the weighted frequency of opcodes and account features (7446 smart contracts x 99 features).
- *Only frequency opcode* dataset, containing solely the absolute frequency of creation opcodes (7021 smart contracts x 76 features).
- *Only weighted opcode* dataset, incorporating the weighted frequency of creation opcodes (7021 smart contracts x 76 features) exclusively.

During the cleaning process of the creation bytecode, we identified 415 duplicate samples with the associated labels. Additionally, we encountered 10 instances where distinct labels were linked to the same bytecode at varying addresses. Since we want to devise a fully automated process, and since there is no best policy to solve these conflicts, all these samples were discarded.

Whereas the following datasets use the deployed bytecode:

- *Transaction plus frequency opcode* dataset, which includes the absolute frequency of opcodes and account features (7446 smart contracts x 99 features).
- *Transaction plus weighted opcode* dataset, which includes the weighted frequency of opcodes and account features (7446 smart contracts x 99 features).
- *Only frequency opcode* dataset, comprising uniquely the absolute frequency of deployed opcodes (6621 smart contracts x 76 features).
- *Only weighted opcode* dataset, presenting only the weighted frequency of deployed opcodes (6621 smart contracts x 76 features).

During the cleaning process of the deployed bytecode, we discovered 819 duplicate bytecodes, including their associated labels. Moreover, there were 6 occurrences where distinct labels were mapped to the same bytecode at different addresses. As with the creation bytecode, we discarded all such samples to maintain a fully automated process.

The rationale behind these decisions is to account for and manage the presence of duplicated bytecodes and their relationship with different addresses and their respective transaction histories.

Table 2 summarizes the presence or absence of the different types of features across the datasets. A checkmark (✓) denotes the inclusion of a type of feature, while a crossmark (✗) indicates its absence. Additionally, a ninth dataset is generated during stage S_2 by minimizing the number of features through the application of RFE. This dataset comprises a subset of account and code features, specifically 47 variables based on the deployed bytecode. Further details about this procedure are in Section VI.

VI. EXPERIMENTAL EVALUATION

In this section, we explore the results obtained by applying X-SPIDE to the 8 datasets that were introduced earlier. These results, together with the inspection of the intermediate stages, answer the following research questions:

RQ1 What configuration (comprising model and dataset) maximizes Recall performance?

RQ2 How do the different types of code features of Section V, derived from creation and deployed bytecode, impact the classification of smart contracts?

RQ3 Does the inclusion of account features improve classification performance compared to using code features alone?

RQ4 Does there exist a subset of sub-features that demonstrates comparable performance to the entire dataset?

RQ5 What are the most important features in detecting Smart Ponzi contracts, and how do they influence the classification outcome?

RQ6 What common peculiarities are found in misclassified contracts, i.e., false positives and false negatives, and how do they influence the incorrect classification?

A. STAGE 1: BEST MODEL AND DATASET SELECTION

Here, we describe the stage S_1 of X-SPIDE and answer **RQ1**, **RQ2** and **RQ3**. As said, S_1 takes as input a set of classifiers C, a set of datasets D, a metric m, and assesses the performance of the classifiers in C trained on the datasets in D to identify the pair (classifier, dataset) that outperforms the other combinations according to m. In our experiments, the set of classifiers C includes Decision Tree [30], Random Forest [31], and the Light Gradient Boosting Machine Classifier (LGBMC) [32]. We selected these algorithms because

TABLE 2. A summary of the features included in the various datasets.

Dataset	Account Features	Frequency Opcode	Weighted Opcode	non-Ponzi Sample	Ponzi Sample	Total
Creation Bytecode						
Transaction plus frequency opcode	✓	✓	✗	6566	880	7446
Transaction plus weighted opcode	✓	✗	✓	6566	880	7446
Only frequency opcode	✗	✓	✗	6258	763	7021
Only weighted opcode	✗	✗	✓	6258	763	7021
Deployed Bytecode						
Transaction plus frequency opcode	✓	✓	✗	6566	880	7446
Transaction plus weighted opcode	✓	✗	✓	6566	880	7446
Only frequency opcode	✗	✓	✗	5952	669	6621
Only weighted opcode	✗	✗	✓	5952	669	6621
Best features set	✓	✗	✓	6566	880	7446

they offer strong classification performance, especially with tabular and unbalanced datasets [33]. Additionally, their behavior is easier to explain compared to more complex methods like neural networks. In summary, they strike a good balance between classification performance and explainability. Whereas for the datasets D, we use the ones introduced in Section V.

To perform the model selection, stage S_1 of X-SPIDE uses a grid search procedure with cross-validation to fine-tune the hyperparameters and thus optimize each classifier's performance. The grid search splits each dataset into an 85% training set and a 15% test set, with stratification based on the target variable. In our experiments, we employ two optimization metrics (i.e., the parameter m of S_1): Area Under the Curve (AUC) and Recall. The cross-validation process of the pipeline splits the training data into ten folds. The model (i.e., classifier and corresponding hyperparameters) undergoes training and testing ten times, using each fold as a validation set. We compute the average score for the metric of interest over these ten tests. Finally, stage S_1 outputs the best pair (classifier, dataset), such that the classifier $c_i \in C$ (with its optimized hyperparameters) and the dataset $d_j \in D$ outperform the other combinations according to the metric m. Specifically, the chosen classifier presents the highest average AUC when optimizing for AUC or the highest average Recall when optimizing for it. For the sake of generality, the pipeline, to perform its optimality selection, also computes the standard metrics *Accuracy*, *AUC*, *F1*, *Precision*, and *Recall* on the test set, considering the optimal hyperparameter values for each dataset and classifier. The results of this step are reported in Table 3. The table shows the previously mentioned metrics for all possible combinations of datasets and classifiers where the results optimize AUC and Recall. Typically, when dealing with fraud detection, security issues, and similar cases, Recall is the metric one wants to maximize since capturing as many fraudulent instances as possible is important. Maximizing Recall ensures we effectively identify all fraudulent cases or security breaches within our system. Table 3 shows that the LGBMC classifier, optimized for Recall on *Transaction plus weighted opcode* from deployed bytecode, performs the best. The optimal pair is highlighted in **purple**. This pair represents the output of the stage S_1. In particular, the hyperparameters obtained with the grid search for LGBMC are the following: 140 estimators, maximum depth of 15, learning rate of 0.1, subsampling of 0.5 for columns, regularization alpha of 0.2 (L1 regularization term), and regularization lambda of 1 (L2 regularization term).

Answers to RQ1:

- When considering the metrics as a whole, the LGBMC classifier achieves the best performance on the *Transaction plus weighted opcode dataset* from deployed bytecode with recall optimization (Table 3).

We further investigate the results of Table 3 to extract useful insights on the classification performance, related to the characteristics of the datasets, and to answer questions **RQ2** and **RQ3**. First, the results of the classification task show generally very good performance in all the considered metrics. In more detail, we observe that results obtained using the datasets generated by the deployed bytecode exhibit generally higher Recall than the ones obtained by the creation bytecode. There are two exceptions (2 cases over 12 experiments) when we apply Decision Tree and LGBMC on *Only frequency opcode* from creation bytecode. This consideration leads us to conclude that datasets derived from deployed bytecodes tend to provide higher Recall performance regardless of which classifier is adopted. This is also confirmed by Table 4, which summarises the top classifiers' performance across the four datasets. The table is computed by averaging the results obtained across the deployed or creation bytecode datasets for each metric. From the results, it emerges that *deployed bytecode* datasets exhibit better average performance in Recall and Accuracy. This is reasonable, considering the deployed bytecode contains only the code executed at runtime when some contract methods are invoked. Instead, the creation code also includes the code necessary for the creation and deployment of the contract

TABLE 3. Results from the grid search comparing three classifiers across datasets, including *Transaction plus frequency opcode*, *Transaction plus weighted opcode*, and *only frequency opcode*, considering both creation and deployed bytecode.

Creation Bytecode	Metric Classifier	AUC	Accuracy	F1	Precision	Recall
Recall Optimization						
Transaction plus frequency opcode	Decision Tree	0.832	0.930	0.705	0.705	0.705
	LGBMC	0.978	0.962	0.841	0.841	0.841
	Random Forest	0.970	0.961	0.809	0.949	0.705
Transaction plus weighted opcode	Decision Tree	0.849	0.942	0.747	0.768	0.727
	LGBM	0.975	0.961	0.835	0.828	0.841
	Random Forest	0.980	0.962	0.812	0.959	0.705
Only frequency opcode	Decision Tree	0.859	0.933	0.713	0.667	0.765
	LGBMC	0.973	0.957	0.812	0.782	0.843
	Random Forest	0.971	0.953	0.747	0.892	0.643
Only weighted opcode	Decision Tree	0.818	0.929	0.657	0.636	0.680
	LGBMC	0.950	0.958	0.767	0.863	0.690
	Random Forest	0.955	0.964	0.804	0.881	0.740
Auc Optimization						
Transaction plus frequency opcode	Decision Tree	0.832	0.930	0.705	0.705	0.705
	LGBMC	0.973	0.962	0.824	0.925	0.742
	Random Forest	0.974	0.962	0.817	0.959	0.712
Transaction plus weighted opcode	Decision Tree	0.849	0.942	0.747	0.768	0.727
	LGBMC	0.976	0.965	0.840	0.919	0.773
	Random Forest	0.979	0.965	0.837	0.935	0.758
Only frequency opcode	Decision Tree	0.859	0.933	0.713	0.667	0.765
	LGBMC	0.975	0.955	0.805	0.770	0.843
	Random Forest	0.971	0.953	0.747	0.892	0.643
Only weighted opcode	Decision Tree	0.827	0.923	0.667	0.633	0.704
	LGBMC	0.969	0.954	0.776	0.817	0.739
	Random Forest	0.976	0.959	0.792	0.891	0.713
Deployed Bytecode	Metric Classifier	AUC	Accuracy	F1	Precision	Recall
Recall Optimization						
Transaction plus frequency opcode	Decision Tree	0.841	0.928	0.706	0.686	0.727
	LGBMC	0.973	0.961	0.835	0.828	0.841
	Random Forest	0.964	0.962	0.812	0.959	0.705
Transaction plus weighted opcode	Decision Tree	0.873	0.944	0.769	0.757	0.780
	LGBMC	**0.973**	**0.962**	**0.840**	**0.825**	**0.856**
	Random Forest	0.974	0.964	0.829	0.951	0.735
Only frequency opcode	Decision Tree	0.825	0.933	0.673	0.657	0.690
	LGBMC	0.954	0.953	0.771	0.752	0.790
	Random Forest	0.950	0.959	0.757	0.928	0.640
Only weighted opcode	Decision Tree	0.818	0.929	0.657	0.636	0.680
	LGBMC	0.950	0.958	0.767	0.863	0.695
	Random Forest	0.955	0.964	0.804	0.881	0.740
Auc Optimization						
Transaction plus frequency opcode	Decision Tree	0.841	0.928	0.706	0.686	0.727
	LGBMC	0.975	0.970	0.866	0.902	0.833
	Random Forest	0.964	0.962	0.812	0.959	0.705
Transaction plus weighted opcode	Decision Tree	0.873	0.944	0.769	0.757	0.780
	LGBMC	0.974	0.968	0.859	0.887	0.833
	Random Forest	0.974	0.964	0.829	0.951	0.735
Only frequency opcode	Decision Tree	0.825	0.933	0.673	0.657	0.690
	LGBMC	0.961	0.951	0.782	0.886	0.700
	Random Forest	0.954	0.967	0.811	0.947	0.710
Only weighted opcode	Decision Tree	0.818	0.929	0.657	0.636	0.680
	LGBMC	0.950	0.958	0.767	0.863	0.690
	Random Forest	0.955	0.964	0.804	0.881	0.740

on the blockchain, thus, the malicious behavior may be less evident.

Answers to RQ2:

- Generally, the deployed bytecode provides better classification performances than the creation bytecode alone (Table 3) when we consider the Recall as a metric to be optimized.
- All the datasets provide very good classification performances for the various classifiers according to the considered metrics (Table 3).

TABLE 4. Comparison of performance metrics between created bytecode and deployed bytecode.

Pipeline	Mean AUC	SD AUC	Mean Accuracy	SD Accuracy	Mean F1	SD F1	Mean Precision	SD Precision	Mean Recall	SD Recall
Deployed Bytecode	0.966	0.013	0.958	0.004	0.807	0.033	0.807	0.035	0.810	0.048
Creation Bytecode	0.971	0.007	0.957	0.005	0.816	0.022	0.852	0.051	0.806	0.036

We consider the different performance achieved by the dataset containing or not transaction features to address **RQ3**. We observe two different behaviors depending on whether the dataset includes the creation or deployed bytecode. For the first case, the performance of the classifiers seems not to be impacted by transaction features, regardless of the optimized metric. For instance, considering the recall optimization results (top rows of Table 3), the recall achieved using the dataset *Transaction plus frequency opcode* is lower than the one obtained with *Only frequency opcode* for two cases out of three. An opposite behavior emerges considering the other group of datasets that contains code features from deployed bytecode, namely *Transactions plus weighted opcode* and *Only weighted opcode*. In these cases, the former shows higher performance than the latter. The datasets with transaction features always show better results in the case of deployed bytecode, regardless of the type of classifier and optimized metric.

Answers to RQ3:

- Although there are subtle variations in performance across datasets, the dataset incorporating account features exhibits slightly superior performance compared to those utilizing only code features (Table 3).
- When optimizing Recall, LGBMC consistently demonstrates superior performance across all metrics (Table 3).
- When optimizing AUC, LGBMC outperforms other classifiers, maintaining high values across all metrics (Table 3).

B. STAGE 2: FEATURE SELECTION AND MOST RELEVANT FEATURES

The stage S_2 of X-SPIDE receives as input the best pair (model, dataset) (i.e., LGBMC, *Transaction plus weighted opcode dataset* from deployed bytecode with Recall optimization) and determines if there exists a subset of features that improves the quality of the model. This stage returns a reduced subset of features and allows us to answer **RQ4**. To address such a problem, stage S_2 considers the number of features as a further hyperparameter to be optimized. In practice, it performs a grid search procedure with cross-validation to optimize the Recall and the number of features.

To tune this last hyperparameter, we start taking all the features of the considered dataset, perform a grid search and a 5-fold cross-validation with the other hyperparameters of the LGBMC classifier, and optimize for Recall, obtaining the best-performing hyperparameter configuration.

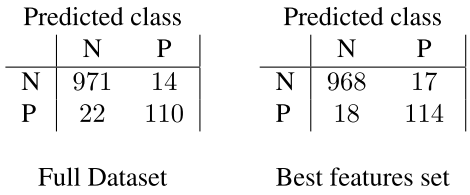

FIGURE 4. Confusion matrices: confusion matrices of the best classifier on *Transaction plus weighted opcode dataset* (Full Dataset) and *Best features set*, where we indicate with N and P the non-Ponzi and Ponzi classes, respectively.

As for the previous experiments, the dataset is divided into an 85% training set (6566 samples) and a 15% test set (880 samples). Then, we adopt the *Recursive Feature Elimination* (RFE) algorithm to remove, at each run, the less important feature. This process continues until we have only 10 features left to evaluate. As a result, we obtain the highest mean Recall with a subset of *Transaction plus weighted opcode* from the deployed bytecode dataset, which consists of 47 features, including eight related to transactions and the remainder to bytecode. The LGBMC classifier trained with the RFE algorithm outperforms previous solutions with only 47 features. The best-performing feature set with LGBMC achieved the following metrics: AUC: 0.977, Accuracy: 0.969, F1: 0.867, Precision: 0.870, Recall: 0.864. This is significant considering the smaller size of the dataset compared to other datasets that included both transactions and bytecode (99 features) or only frequency opcode (76 features), as shown in Table 3. Stage S_2 outputs the resulting dataset, which we call *Best feature set*. The hyperparameters for the selected classifier are the following: 140 estimators, a maximum depth of 15, a learning rate of 0.1, a subsample of columns set to 0.8, alpha (L1 regularization term) of 0.2, and lambda (L2 regularization term) set to 1.

Additionally, we examine the confusion matrices of the model using all the features against *Best features set* (see Figure 4). The confusion matrix summarizes the number of correctly classified samples, namely true positives (TP); the number of false positives (FP); the number of false negatives (FN); and the number of true negatives (TN). From the matrices in the figure, we observe that the classifier trained with *Best features set* presents a higher number of TP and a lower number of FN. More precisely, the figure reveals that 4 false negatives are now correctly classified.

Answers to RQ4:

- The classification outcome seems to depend more on information from the contract code than the transaction

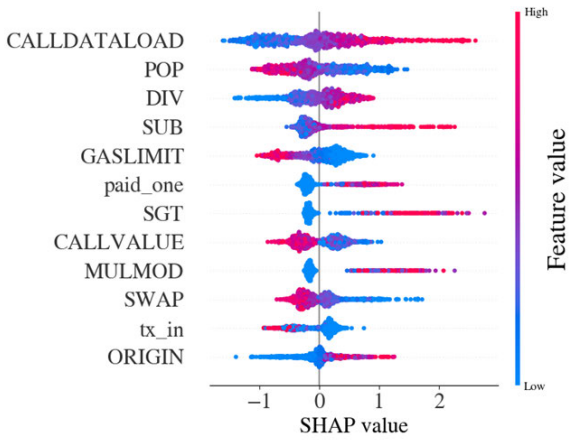

FIGURE 5. Shapley Values of Best feature set: the feature importance of the top 12 features in terms of Shapley values.

behavior, even if this last one is relevant for the classification (Table 3).

- The best set of features consists of 47 features, including account and code features. The top twelve include ten code features and two transaction features (Figure 5).
- The LGBMC classifier trained with the RFE algorithm outperforms previous solutions while utilizing only 47 features.
- The classifier trained with the best features increases the number of correctly classified contracts (Figure 4).

C. STAGE 3: MODEL EXPLANATION WITH XAI

The stage S_3 of X-SPIDE takes in input the best classifier from S_1 and the dataset obtained with S_2, then it computes the Shapley values *Sh*, the partial dependences *PD*, and the misclassified contracts *MC*. The first two results of S_3 help understand the importance of the features for the classification model. The last result, namely, the misclassified contracts *MC*, undergoes further investigation to identify the features that misled the classifier.

The Shapley values can be used to visually investigate the impact of the most important features on classification. Partial dependences can be represented in plots called Partial Dependence Plots (PDPs) to analyze the relationship between the target response and specific features of interest while marginalizing the values of all other input features. Below, we describe the analyses on the results of S_3 to explain our model results and their classification performance.

1) FEATURE IMPORTANCE

The sets *Sh* and *DP* allow us to measure how individual features influence the behavior of Ponzi and non-Ponzi contracts and also help answer **RQ5**.

Figure 5 shows the beeswarm plot of the feature importance based on Shapley values for the test set. The plot provides a concise summary of how the top features in the dataset influence the model output. Each instance is represented by a single dot, positioned on the x-axis based on its corresponding Shapley value. The dots 'pile up' along each feature row, indicating the density. The color of the dots represents the original value of the feature, with blue indicating lower values and red showing higher values. We can determine each feature's positive or negative effect on the prediction outcome by analyzing the plot.

The figure shows that a high value of *CALLDATALOAD*,[6] *SUB*,[7] *MULMOD*,[8] *SGT*,[9] *paid_one* positively impacts the classification, indicating a higher likelihood of being labeled as a smart Ponzi contract. Conversely, a low value of *GASLIMIT*,[10] *POP*,[11] *CALLVALUE*,[12] and *tx_in* assists the classifier in identifying the instance as a smart Ponzi contract. Furthermore, it is interesting that two of the top twelve features concern the transaction history.

Partial dependence plots (PDPs) provide a useful visual and analytical tool to explore the relationship between the target response and a selected set of input features. In our case, we focus on the top twelve features identified through the Shapley values. In Figure 6, we present the PDP graphs for each feature within the top 12 features related to the target variable, whether it is classified as Ponzi or non-Ponzi. For instance, the PDP plot for the *CALLDATALOAD* feature in Figure 6 illustrates the impact of the proportion of *CALLDATALOAD* opcodes on the likelihood of a smart contract being classified as Ponzi. A higher proportion of *CALLDATALOAD* opcodes is associated with more positive cases in the target variable. This could be explained by considering that a conventional smart Ponzi contract retains a data structure in its storage, which stores users' addresses and investments. These data are included in the transaction input and must be stored in the contract's maintained data structure. They must initially be pushed onto the stack and subsequently loaded. Similar patterns can be observed for *paid_one*. This feature may positively affect classifying an instance as a smart Ponzi because, typically, such contracts do not provide users with many services besides investing some money and, in case, repaying them. Also, the features *DIV*, *MULMOD*, and *SGT* positively impact the classifier, but giving an intuitive explanation of the reason is tricky.

Conversely, features like *GASLIMIT* negatively correlate with the target response. This could be explained by the fact that this instruction appears the same in the code of all smart contracts, both Ponzi and non-Ponzi.

It is important to note that these observations focus on individual features, considering one feature at a time.

[6]The instruction loads 32 bytes of the transaction data on the virtual machine stack.

[7]The instruction subtracts the top two elements of the stack and pushes the result back on the stack.

[8]The instruction performs a modulo multiplication of the top 2 elements with the 3rd element and then pushes the result back on the stack.

[9]The instruction signed greater-than comparison.

[10]The instruction gets the block gas limit.

[11]The instruction removes and returns a word from the stack.

[12]The instruction gets deposited value by the instruction/transaction responsible for this execution.

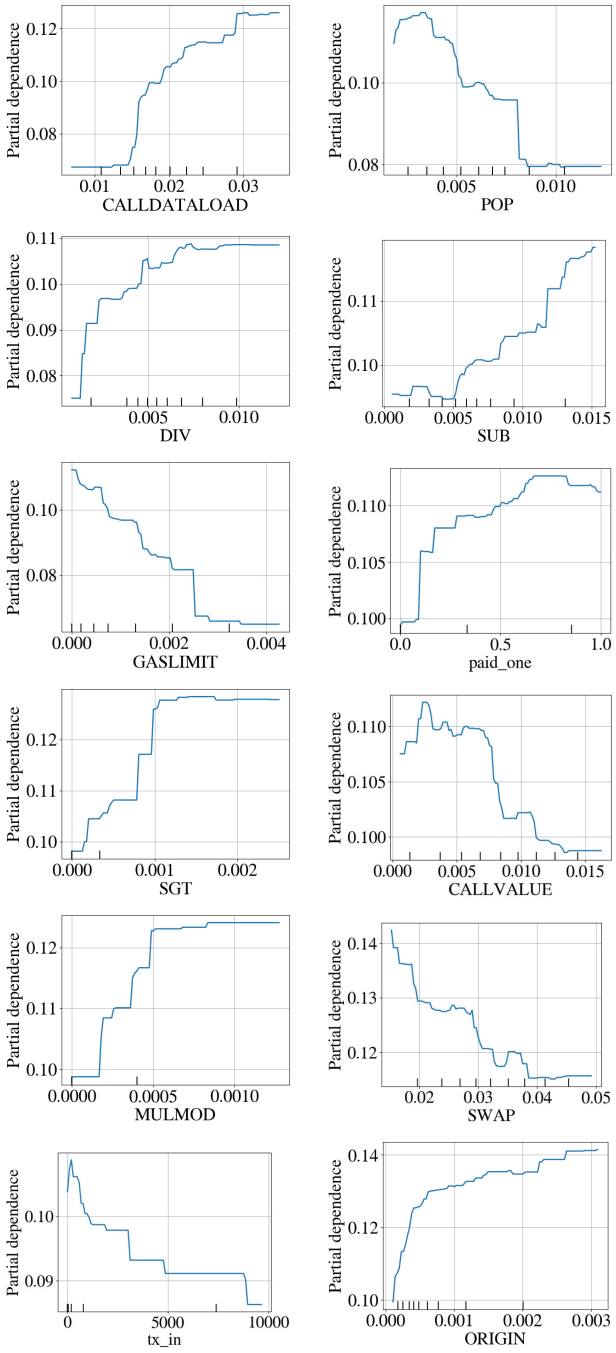

FIGURE 6. Partial Dependence plot of the top 12 features.

However, they provide valuable insights into the marginal effects of each feature on the target response.

Answers to RQ5:

- By Figures 5 and 6 it is clear that opcode features such as *CALLDATALOAD*, *SUB*, and *MULMOD* with high Shapley values may identify some low-level behavior of smart Ponzi contract, e.g., maintaining a data structure to store users' investments or redistributing the reward to investors.
- By Figures 5 and 6 we see that opcode features such as *GASLIMIT*, *POP* and *CALLVALUE* with low Shapley values may indicate that those instructions are not sufficient to discriminate between the two classes because they appear in the same way in the code of all smart contracts.
- Figures 5 and 6 show that transaction features such as *paid_one* have high Shapley values, whereas *tx_in* with low Shapley Values may identify some behavioral aspects of smart Ponzi contracts, e.g., these contracts may receive few input transactions with a certain amount of money and tend to pay just one investor.

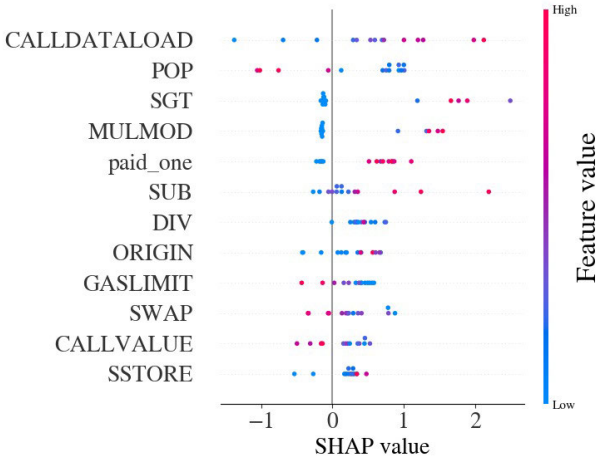

FIGURE 7. Shapley values of false positives: The feature importance of the top 12 features in terms of Shapley values.

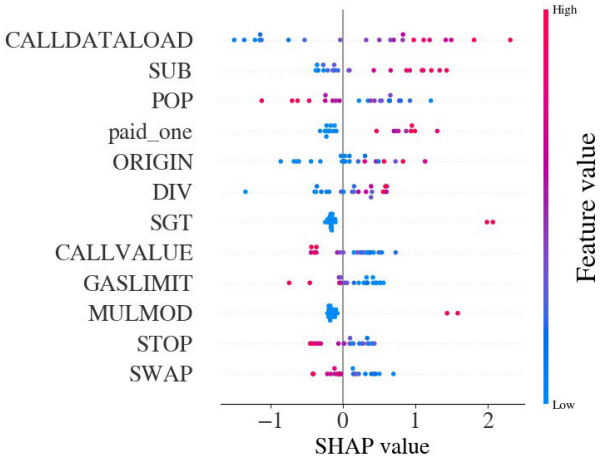

FIGURE 8. Shapley values of false negatives: The feature importance of the top 12 features in terms of Shapley values.

2) ANALYSIS OF MISCLASSIFIED CONTRACTS

The stage S_3 of X-SPIDE, also leverages XAI techniques to automatically analyze misclassified contracts and to identify which contract characteristics contribute to misclassification. Initially, S_3 applies SHAP to the contracts in the subsets *False Positive* and *False Negative*. Subsequently, given the small number of misclassified contracts, S_3 selects them and returns their code, enabling their manual inspection. Finally, S_3 also supports the analysis of specific contracts. Later in this section, we specifically apply it to two pairs of contracts, one

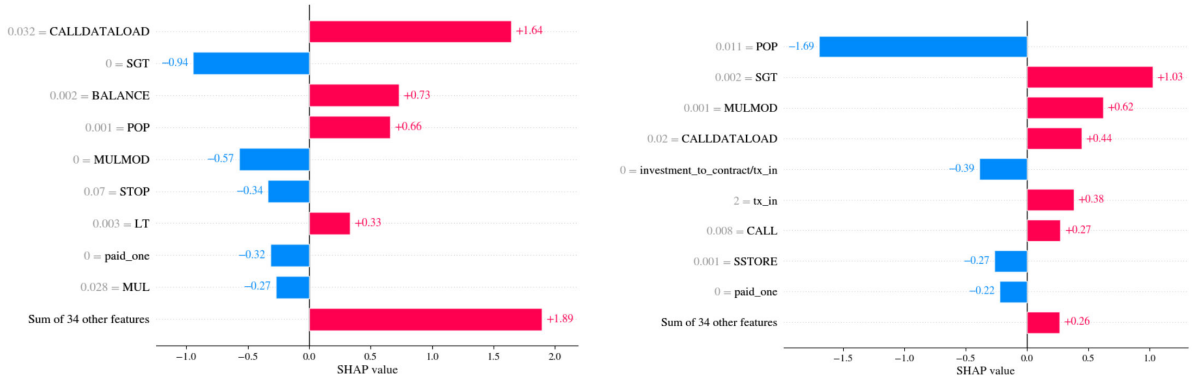

FIGURE 9. Bar plot of false positive contracts: Plots showing the impact of features in contract classification.

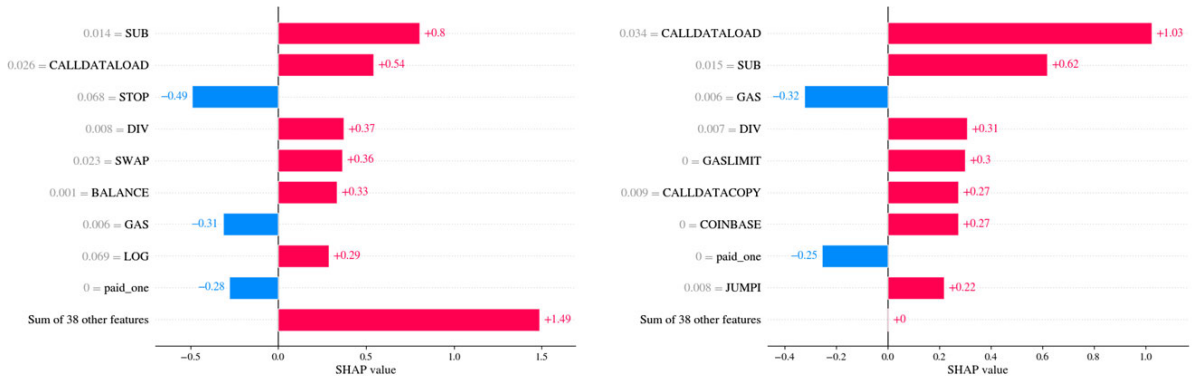

FIGURE 10. Bar plot of false negative contracts: Plots showing the impact of features in contract classification.

for False Positives and one for False Negatives, highlighting their peculiar characteristics through SHAP investigation. Finally, we answer to research question **RQ6**.

As said, we apply SHAP to the subset of False Positive contracts. The results are in Figure 7, where we report the most important features. Comparing this plot with the one in Figure 5, we note that the feature *tx_in* is replaced by the *SSTORE* one, while the other most important features remain the same. Considering this small set of False Positive contracts, we observe that the *MULMOD* and *SGT* features become more important for Ponzi classification. However, the values of the features push the classifier to label these contracts as Ponzi, making it difficult to distinguish them from the real Ponzi.

To better understand the nature of these contracts, S_3 supports manual inspection. More precisely, it retrieves the addresses of the misclassified contracts[13] and permits an analyst to inspect their verified source code, published on etherscan.io, when available.[14] From this manual inspection, we notice that most False Positive contracts are related to gambling games, lotteries, and tokens. The classifier may have misclassified them because they share characteristics with Ponzi schemes, such as receiving payments from users, storing user addresses in internal data structures, paying fees to the contract creator, and implementing a mechanism to pay out a winner or redeem tokens. However, other similar contracts (tokens, games, and lotteries) are correctly classified in our dataset.

Moreover, stage S_3 allows an analyst to select specific contracts and generates SHAP's bar plots. As an example, we analyze here two false positive contracts.[15] In this scenario, we expect to observe features commonly associated with Ponzi contracts. In the top plot of Figure 9, the features *CALLDATALOAD* and *POP* exhibit values similar to those found in Ponzi contracts, and both contribute heavily to the classifier's decision to assign a target value of 1. Conversely, the features *paid_one* and *SGT* have low values, typically associated with non-Ponzi contracts. A similar pattern is observed in the second false positive contract in Figure 9. Notably, features such as *SGT*, *MULMOD*, and *CALLDATALOAD* (highlighted in red) strongly influence the classifier to result in a Ponzi classification. On the other hand, the features *POP*, *Investment_in/tx_in*, and *paid_one* push the classifier towards a non-Ponzi classification.

For the False Positives, our analysis indicates that these contracts exhibit several features typically associated with

[13] Available in our online repository.

[14] We cannot provide a conclusive analysis for contracts whose source code is not public.

[15] Addresses 0×927a4e90c3728f04cc373cd4c445daafa9e54d, f70xd51a87caaa567677abac451b00e0a0a18a992b49.

smart Ponzi contracts, making it difficult for the classifier to correctly label them.

The same process is applied to False Negative contracts. The results of the SHAP analysis are in Figure 8: also, for this set, the most important features match those of Figure 5 except for *tx_in*, which is replaced by the feature associated with the instruction *STOP*. For the set of False Negative contracts, we notice that the feature ORIGIN is the most relevant, while the feature *DIV* lost its importance. Regarding the overall characteristics, these contracts differ from the typical Ponzi by a low value of *CALLDATALOAD*, *SGT*, and *MULMOD*, pushing the classification towards Not-Ponzi.

Then, stage S_3 retrieves the code of the False Negative to enable the manual inspection of the contract code. From a manual inspection, we found that their code is complex with intricate program logic. However, for some of the contracts where we are able to understand their behavior: reading the code makes clear that they either hide a pyramid scheme within a seemingly legitimate contract type (such as a gambling game or token) or use a complex or overly simplistic logic for distributing rewards. In some cases, the author deliberately conceals the contract's true nature to avoid detection by users.

Finally, we select two false negative contracts misclassified by the model.[16] As shown in Figure 10, the SHAP values for these contracts are relatively low, indicating that the classifier tends to label them as non-Ponzi contracts due to the absence or limited presence of significant Ponzi-related features. For instance, features like *SUB* and *CALLDATALOAD* have values similar to non-Ponzi contracts, which might explain why the classifier failed to detect their Ponzi-like characteristics.

Answers to RQ6:

- The analysis reveals that false positives have some behavioral aspects similar to Ponzi schemes. The analysis of the features with XAI techniques (Figure 7 and 9) confirms that feature values resemble typical characteristics found in Ponzi contracts, so this explains why the classifier labeled them as Ponzi.
- The manual inspection reveals that false negatives are contracts that hide a pyramid scheme inside a non-fraudulent contract, or implement a singular distribution logic. The analysis of the features with XAI techniques (Figure 10) confirms that such contracts often lack features typically associated with Ponzi schemes, so this explains why the classifier labeled them as non-Ponzi.

VII. CONCLUSION AND FUTURE WORK

This paper presented X-SPIDE, an automatic explainable machine learning pipeline for detecting smart Ponzi contracts on Ethereum. X-SPIDE consists of three stages. The first stage S_1 assesses the performance of multiple classifiers trained using different datasets to identify the best pair (model, dataset), and allowed us to study how the different bytecode types impact the classification of Ponzi and non-Ponzi contracts. The results of our experiments show that deployed bytecode achieves better performance. Given the best pair (model, dataset), the second stage S_2 identifies a small yet effective set of features that ensures a good classification quality and enables us to explain the classifier's decisions. The last stage S_3 performs two different analyses relying on XAI techniques. The first investigates how the identified features impact classification and how they are related to possible fraudulent behavior. The second consists of an inspection of misclassified contracts. These analyses not only enhance the interpretability of our findings but also reinforce the robustness of our pipeline. Moreover, to train and test our pipeline, we released new reusable datasets with 7446 unique real-world smart contracts that can be used for future research. The dataset contains transaction history and bytecode information and captures several behavioural aspects of contracts during their lifetime.

Future work aims to extend our pipeline in different directions. First, we intend to optimize the best feature extraction procedure. Then, we plan to improve our approach to extract opcode features by the bytecode. In particular, we aim to identify high-level behavioural patterns from sequences of instructions and possibly apply deep learning techniques to minimize the feature engineering effort. As a further direction, we will study how to make our pipeline robust against smart Ponzi contracts that use adversarial evasion techniques to conceal their fraudulent nature, e.g., hiding inside apparently harmless contracts. We plan to apply techniques similar to malware evasion countermeasures [34]. Moreover, we will study how the performance of our pipeline degrades over time as the Ponzi scheme evolves, and we will devise a retraining strategy that allows us to update our model, taking into account new training samples. Additionally, we plan to refine the requirements of being a smart Ponzi proposed by Bartoletti et al. [6]. Another line of research will focus on identifying active Smart Ponzi contracts nearing collapse and refining our features to capture the financial posture of contracts. We will explore the generalizability of this approach to other contract types and other forms of scams on Ethereum, such as phishing, as one of the most promising. Finally, we plan to integrate our model inside an existing Ethereum Wallet to warn the user when she is about to invest in a suspicious contract.

DATA AVAILABILITY

The datasets and the notebooks used for the experiments presented in this article are available online at https://github.com/LucaPennella/x-spide-smart-ponzi-detection

REFERENCES

[1] Satoshi Nakamoto. (2008). *Bitcoin: A Peer-to-Peer Electronic Cash System*. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[16] Addresses `0xbdb8b73aea0c43118ce8834c91d50ae8bbd5ed`, `32 0 × 1e997b4a256e11071167a1d08e98a9f5dde0bf72`.

[2] E. Napoletano. (2021). *Decentralized Finance is Building a New Financial System*. Accessed: 2022. [Online]. Available: https://www.nasdaq.com/articles/decentralized-finance-is-building-a-new-financial-system-2021-04-02
[3] T. Moore, "The promise and perils of digital currencies," *Int. J. Crit. Infrastruct. Protection*, vol. 6, nos. 3–4, pp. 147–149, Dec. 2013.
[4] M. Bartoletti, S. Lande, A. Loddo, L. Pompianu, and S. Serusi, "Cryptocurrency scams: Analysis and perspectives," *IEEE Access*, vol. 9, pp. 148353–148373, 2021.
[5] M. Vasek and T. Moore, "There's no free lunch, even using Bitcoin: Tracking the popularity and profits of virtual currency scams," in *Financial Cryptography and Data Security*. Cham, Switzerland: Springer, 2015, pp. 44–61.
[6] M. Bartoletti, S. Carta, T. Cimoli, and R. Saia, "Dissecting Ponzi schemes on Ethereum: Identification, analysis, and impact," *Future Gener. Comput. Syst.*, vol. 102, pp. 259–277, Jan. 2020.
[7] M. Bartoletti, B. Pes, and S. Serusi, "Data mining for detecting Bitcoin Ponzi schemes," in *Proc. Crypto Valley Conf. Blockchain Technol. (CVCBT)*, Jun. 2018, pp. 75–84.
[8] W. Chen, Z. Zheng, E. C.-H. Ngai, P. Zheng, and Y. Zhou, "Exploiting blockchain data to detect smart Ponzi schemes on Ethereum," *IEEE Access*, vol. 7, pp. 37575–37586, 2019.
[9] L. Wang, H. Cheng, Z. Zheng, A. Yang, and X. Zhu, "Ponzi scheme detection via oversampling-based long short-term memory for smart contracts," *Knowl.-Based Syst.*, vol. 228, Sep. 2021, Art. no. 107312.
[10] W. Chen, X. Li, Y. Sui, N. He, H. Wang, L. Wu, and X. Luo, "SADPonzi: Detecting and characterizing Ponzi schemes in Ethereum smart contracts," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 49, no. 1, pp. 35–36, Jun. 2021.
[11] Y. Lou, Y. Zhang, and S. Chen, "Ponzi contracts detection based on improved convolutional neural network," in *Proc. IEEE Int. Conf. Services Comput. (SCC)*, Nov. 2020, pp. 353–360.
[12] Z. Zheng, W. Chen, Z. Zhong, Z. Chen, and Y. Lu, "Securing the Ethereum from smart Ponzi schemes: Identification using static features," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 5, pp. 1–28, Sep. 2023.
[13] L. Galletta and F. Pinelli, "Explainable Ponzi schemes detection on Ethereum," in *Proc. 39th ACM/SIGAPP Symp. Appl. Comput.*, Apr. 2024, pp. 1014–1023.
[14] X. F. Liu, X.-J. Jiang, S.-H. Liu, and C. K. Tse, "Knowledge discovery in cryptocurrency transactions: A survey," *IEEE Access*, vol. 9, pp. 37229–37254, 2021.
[15] A. Trozze, J. Kamps, E. A. Akartuna, F. J. Hetzel, B. Kleinberg, T. Davies, and S. D. Johnson, "Cryptocurrencies and future financial crime," *Crime Sci.*, vol. 11, no. 1, pp. 1–35, Dec. 2022.
[16] S. Fan, S. Fu, H. Xu, and C. Zhu, "Expose your mask: Smart Ponzi schemes detection on blockchain," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2020, pp. 1–7.
[17] G. Ibba, G. A. Pierro, and M. Di Francesco, "Evaluating machine-learning techniques for detecting smart Ponzi schemes," in *Proc. IEEE/ACM 4th Int. Workshop Emerg. Trends Softw. Eng. Blockchain (WETSEB)*, May 2021, pp. 34–40.
[18] C. Jin, J. Jin, J. Zhou, J. Wu, and Q. Xuan, "Heterogeneous feature augmentation for Ponzi detection in Ethereum," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 69, no. 9, pp. 3919–3923, Sep. 2022.
[19] J. Peng and G. Xiao, "Detection of smart Ponzi schemes using opcode," in *Proc. Blockchain Trustworthy Syst., 2nd Int. Conf. (BlockSys)*, Jan. 2020, pp. 192–204.
[20] W. Chen, Z. Zheng, J. Cui, E. Ngai, P. Zheng, and Y. Zhou, "Detecting Ponzi schemes on Ethereum: Towards healthier blockchain technology," in *Proc. World Wide Web Conf. World Wide Web (WWW)*, 2018, pp. 1409–1418.
[21] S. Yu, J. Jin, Y. Xie, J. Shen, and Q. Xuan, "Ponzi scheme detection in Ethereum transaction network," in *Proc. Blockchain Trustworthy Syst., 3rd Int. Conf. (BlockSys)*, Jan. 2021, pp. 175–186.
[22] J. Jin, J. Zhou, C. Jin, S. Yu, Z. Zheng, and Q. Xuan, "Dual-channel early warning framework for Ethereum Ponzi schemes," in *Proc. Big Data Social Comput., 7th China Nat. Conf. (BDSC)*, Jan. 2022, pp. 260–274.
[23] Y. Zhang, W. Yu, Z. Li, S. Raza, and H. Cao, "Detecting Ethereum Ponzi schemes based on improved LightGBM algorithm," *IEEE Trans. Computat. Social Syst.*, vol. 9, no. 2, pp. 624–637, Apr. 2022.
[24] I. J. Onu, A. E. Omolara, M. Alawida, O. I. Abiodun, and A. Alabdultif, "Detection of Ponzi scheme on Ethereum using machine learning algorithms," *Sci. Rep.*, vol. 13, no. 1, p. 18403, Oct. 2023.
[25] J. Cai, B. Li, J. Zhang, and X. Sun, "Ponzi scheme detection in smart contract via transaction semantic representation learning," *IEEE Trans. Rel.*, vol. 73, no. 2, pp. 1117–1131, Jun. 2024.
[26] M. Wang and J. Huang, "Detecting Ethereum Ponzi schemes through opcode context analysis and oversampling-based AdaBoost algorithm," *Comput. Syst. Sci. Eng.*, vol. 47, no. 1, pp. 1023–1042, 2023.
[27] X. Feng, Q. Shi, X. Li, H. Liu, and L. Wang, "IDPonzi: An interpretable detection model for identifying smart Ponzi schemes," *Eng. Appl. Artif. Intell.*, vol. 136, Oct. 2024, Art. no. 108868.
[28] J. H. Friedman, "Greedy function approximation: A gradient boosting machine," *Ann. Statist.*, vol. 29, no. 5, pp. 1189–1232, Oct. 2001.
[29] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, pp. 1–32, Jan. 2013.
[30] J. R. Quinlan, "Induction of decision trees," *Mach. Learn.*, vol. 1, no. 1, pp. 81–106, Mar. 1986.
[31] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.
[32] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T. Liu, "LightGBM: A highly efficient gradient boosting decision tree," in *Proc. Adv. Neural Inf. Process. Syst.*, Dec. 2017, pp. 3149–3157.
[33] I. D. Mienye and Y. Sun, "A survey of ensemble learning: Concepts, algorithms, applications, and prospects," *IEEE Access*, vol. 10, pp. 99129–99149, 2022.
[34] A. Afianian, S. Niksefat, B. Sadeghiyan, and D. Baptiste, "Malware dynamic analysis evasion techniques: A survey," *ACM Comput. Surv.*, vol. 52, no. 6, pp. 1–28, Nov. 2019.

LUCA PENNELLA is currently pursuing the Ph.D. degree in applied data science and artificial intelligence with the University of Trieste. He is a Teaching Assistant with the University of Trieste, he contributes to courses in statistics and machine learning. He has a background in statistical sciences, economics, and data science, with experience in data analysis across consulting and academic settings. His research interests include the application of machine learning techniques and quantitative methods in blockchain, survey data, decentralized finance, and web 3.0.

FABIO PINELLI received the M.Sc. degree in computer science and the Ph.D. degree in information engineering from the University of Pisa, in 2005 and 2010, respectively. He was a Research Scientist with the IBM-Research Ireland and a Senior Data Scientist with Vodafone Italia. He is currently an Associate Professor with the SySMA Group, IMT School for Advanced Studies Lucca. His main research interests include data mining and machine learning methods applied to different domains, and economics in urban environments.

LETTERIO GALLETTA is currently an Assistant Professor with the IMT School for Advanced Studies Lucca. Previously, he was a Postdoctoral Researcher with the Department of Computer Science, University of Pisa. His research activity mainly focuses on language-based security, i.e., using techniques from programming languages, compilers, and formal verification to address security problems. He applied these techniques to different fields, such as adaptive software, the Internet of Things, and more recently, secure compilation, firewalls, and smart contracts.