IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 30 December 2022, accepted 20 January 2023, date of publication 23 January 2023, date of current version 30 January 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3239403

RESEARCH ARTICLE

NFTs for Open-Source and Commercial Software Licensing and Royalties

MOHAMMAD MADINE[1], KHALED SALAH[1], (Senior Member, IEEE), RAJA JAYARAMAN[2], AND JAMAL ZEMERLY[1], (Senior Member, IEEE)

[1]Department of Electrical Engineering and Computer Science, Khalifa University of Science and Technology, Abu Dhabi, United Arab Emirates
[2]Department of Industrial and Systems Engineering, Khalifa University of Science and Technology, Abu Dhabi, United Arab Emirates

Corresponding author: Raja Jayaraman (raja.jayaraman@ku.ac.ae)

This work was supported by the Khalifa University of Science and Technology under Award CIRA-2019-001.

ABSTRACT Software licenses are legal agreements of sale and usage among software developers and clients. Such legal agreements are crucial to effectively manage ownership and protect the rights of involved parties. Today's software licensing mechanisms are mostly centralized and do not address the ever-increasing issues and complexities of modern software that may include multiple licenses, open-source distribution, rewarding other contributors of external software libraries, and utilizing royalty payments for monetization. As a result, developers have lost confidence in the existing software licensing models, and many software projects are failing due to lack of funding and royalty payments. This paper addresses such issues and complexities by proposing a novel decentralized software licensing system based on Non-Fungible Tokens (NFTs) and blockchain. The proposed licensing system is applicable to both commercial and open-source software. We use NFTs as digital tokens that encapsulate software code and their artifacts by minting them as unique valuable assets that allow developers to store and manage them on a blockchain ledger. With NFTs, developers can register and license their code, monetize it on NFT marketplaces, and earn royalties from other software projects that use their code. We present system architecture, relevant sequence diagrams, and develop aggregation algorithms for Ethereum smart contracts with ERC-1155 NFTs. Furthermore, we perform functional validation of our system and analyze the cost of its adoption. We also analyze the security of the solution and show how its applicability can be generalized and extended. We have made our smart contract code and related testing scripts publicly available on GitHub.

INDEX TERMS Software licensing, software royalties, open-source software, NFTs, blockchain, Ethereum, smart contracts.

I. INTRODUCTION

Computer software of all types, licensing models, and price ranges, is ubiquitous across all countries, industries, and devices. By default, copyright protection laws restrict access to using any software [1]. Consequently, developers distribute their software under a licensing model that defines the permissible and restricted uses based on factors such as the territory and time, as well as the type of use, such as commercial use, modification, and redistribution [2]. Licenses fall into two categories, proprietary and open-source. Proprietary licensing allows developers to distribute paid versions of closed-source software commercially, making revenue generation straightforward [3], [4]. Open-source licensing preserves what is known as the four essential freedoms that aim to make the software more accessible [2], [5]. Developers choose the licensing strategy for their software based on many factors, including the extent of contribution, competition in the market, cost of distribution, and ability to leverage revenue from integration and support services [6], [7], [8].

Even though developers are increasingly more aware of the importance of effectively licensing their software, substantial limitations thwart the adoption and confidence in licensing [1]. First, software licensing can become overcomplicated when it comes to multiple licenses. Multi-licensing is beneficial to appease conflicting dependencies and satisfy

The associate editor coordinating the review of this manuscript and approving it for publication was Pedro R. M. Inácio.

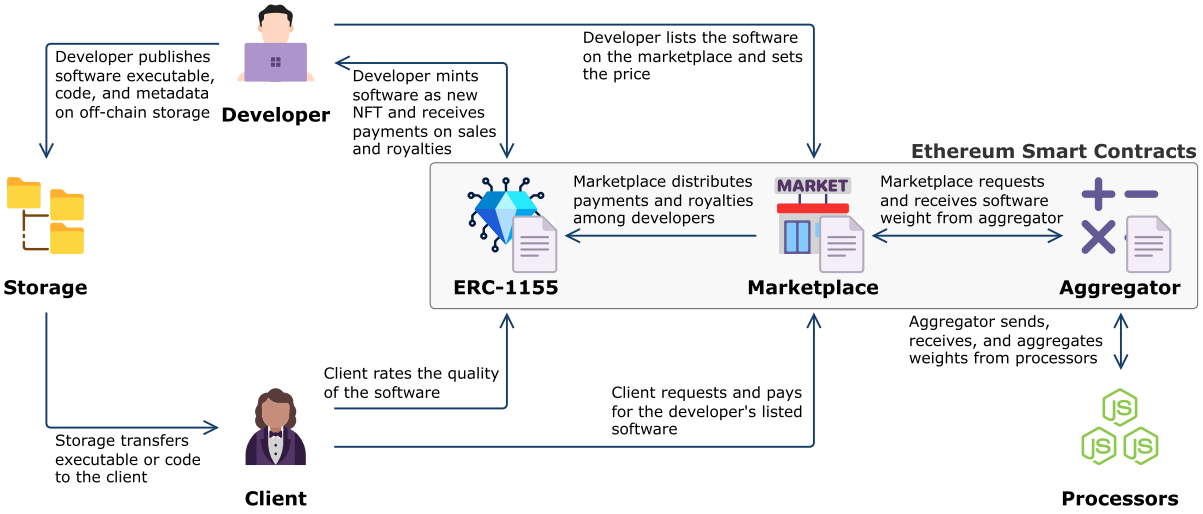

FIGURE 1. System overview of the proposed blockchain-based solution for software licensing and royalties.

a wide range of users in a way that balances profitability and accessible software. Therefore, many developers, especially in dependency-reliant languages such as JavaScript, employ a multi-license model for their software and yet face license incompatibility issues [9], [10]. Second, most open-source projects are certain to fail, wasting contributions that otherwise would flourish the software development. In the last decade, open-source software has gotten more popular than proprietary software, pushing community-driven contributions for high-quality monopoly-free software more than ever before [2], [11]. Despite that, loss of interest and lack of funding resources are the main reasons developers pull the plug on more than 50 percent of open-source projects [12].

Blockchain is a technology that enables decentralized ledgers that record transactions on peer-to-peer nodes, all governed by a consensus mechanism. Cryptographic and staking mechanisms ensure that no single entity can feasibly alter any transaction without altering all subsequent records. These features allow for building secure, transparent, and resilient systems without the need for a central authority [13]. This technology can establish the groundwork for solutions that address licensing issues. Such solutions also take advantage of the more recent advancements in blockchain, namely smart contracts and Non-Fungible Tokens (NFTs). Smart contracts are externally-triggered methods that execute on decentralized validator nodes. Ethereum formally introduced the smart contracts concept into blockchain, along with Solidity, which became the most popular language for writing smart contracts [14], [15]. NFTs are unique assets managed by special standardized smart contracts called ERC-721 and ERC-1155. NFTs in an ERC-721 smart contract accommodate one type of asset per deployment, whereas those in an ERC-1155 smart contract can have multiple types of assets within the same deployment. In addition, ERC-1155 enables batch transfer of NFTs and offers a more flexible alternative to ERC-721 [16].

Blockchain is only suitable for a narrow set of applications in its isolated form. Hence, it is common to integrate it with supplementary technologies such as decentralized storage, incentivized external processing nodes, and reputation systems. Decentralized storage alleviates the limited space for blockchain transactions. InterPlanetary File System (IPFS) is one of the most popular decentralized storage solutions, providing means to upload and publish content-addressable assets on a peer-to-peer network [17], [18]. External processing nodes extend the functionality of smart contracts with options to perform computationally-expensive tasks, retrieve information from external sources and APIs, and commit to timer actions. Since the external processes do not run on Ethereum Virtual Machines (EVMs), they must receive additional incentives from external parties such as the end-user [19], [20]. A reputation system is a technique to keep track of the behavior of entities in the blockchain network. For example, the most common reputation system performance measurement lets users rate each other after finalizing a transaction [21].

Blockchain, along with other decentralization technologies, has an outstanding potential to address software licensing concerns. In this paper, we propose an NFT-based software licensing solution with support for multi-licenses to open a streamlined and profitable path for the developers of open-source and proprietary software. Figure 1 highlights a high-level overview of our proposed NFT-based solution. Our contributions in this paper are summarized as follows:

- We investigate and propose a fully decentralized solution for managing software licenses and royalties using blockchain and ERC-1155 NFTs, allowing developers to list their software under multiple licensing models.

• We design, illustrate and explain our proposal using system architecture and sequence diagrams that cover the key scenarios for how developers and clients may interact with each other, including open-source and proprietary software use cases.

• We present a solution that incorporates reputation, payments, and royalties into NFTs to increase confidence in software projects, provide means to fund them with perpetual and subscription licenses, and pay developers of low-level dependencies automatically and transparently.

• We develop the key functionalities of our proposal using Solidity smart contracts and deploy them on Ethereum testing networks. We verify the correctness of the code and use it for cost analysis. Our code is publicly available on GitHub .[1]

The remainder of this paper consists of six sections. Section II investigates the state-of-the-art research concerned with software licensing. In Section III we propose our NFT-based approach, whereas in Section IV we demonstrate the implementation details of our solution. Later in Section V, we deploy our smart contracts for evaluation and analysis, followed by discussing our findings in Section VI. Finally, Section VII wraps up our proposed novel contributions and results.

II. RELATED WORK

In this section, we delve into the previous literature that discusses and proposes solutions for the software license concerns, namely the multi-license management, open-source funding, and distribution of royalties. To our knowledge, no prior research has targeted software licensing using blockchain and NFTs. Therefore, we explore the literature pertaining to each concern independently.

A. MULTI-LICENSE MANAGEMENT

Moraes et al. [9] studied common incompatibilities among open-source licenses, compared file-level and project-level license options, and recommended that developers use a file-level multi-licensing model for their code to overcome compatibility issues and offer less restrictive access options. As part of the license management and enforcement, other research proposed using the Markov chain to flag abnormal user behavior, binary analysis tools to detect code cloning, and a Trusted Execution Environment (TEE) to ensure compliance with license agreements [22], [23], [24]. Regarding leveraging blockchain, Stepanova and Erinš [3] suggested using the technology to maintain the history of the software licenses and those who hold the legitimate license, whereas Chiu et al. [25] proposed using Ethereum and IPFS for software validation and integrity.

B. OPEN-SOURCE FUNDING

Gaetano [26] studied the potential profitability of open-source software developers and concluded that contributions to open-source are profitable, especially for those who provide additional hardware products and support.

Multiple platforms for funding open-source projects have been established over the past decade. The most notable and prevalent among these is Liberapay [27], which relies on indirect and recurring donation-based transactions among the contributors and developers. The contributors donate to the platform, and the platform periodically uses its open-source algorithm to distribute the funds among its registered accounts of developers and open-source projects. GitHub Sponsors is a more recent commercial approach to the problem, allowing sponsors to make recurring or single-time contributions to open-source projects hosted on GitHub [28]. Other funding platforms include Open Collective and Tidelift. One major drawback of the existing centralized solutions is their reliance on third-party payment processors such as Stripe and PayPal, undermining the confidence in delivering the contributions in part or whole [27], [28].

Gitcoin is a decentralized and blockchain-based funding platform for software projects and bug bounties. Gitcoin's Grants feature focused on crowdfunding open-source software, providing means to support developers with direct one-time or quarterly payments. The payments are collected from a matching pool distributed among developers using the quadratic funding algorithm [29], [30].

C. DISTRIBUTION OF ROYALTIES

Software distribution often takes the form of a sale or a licensing agreement; however, considering that software is also an intellectual property, it is valid to think about how the developers can collect royalties for providing their original binaries and source code. Furthermore, because of the nature of software development, it is common for developers to incorporate others' software as libraries and dependencies. Therefore, it is crucial to bear in mind to fairly distribute the royalties among all the contributing pieces of codes [31].

The rise of NFTs has provided an attractive solution for protecting intellectual property rights [16], [32]. Researchers have also suggested leveraging NFTs to let creators earn royalties on each successful sale, or resale of their intellectual property [33], [34], [35].

III. NFT-BASED SOLUTION

Herein, we delineate the elements and architecture our proposed solution comprises. In addition, we describe the mechanism under which the developers license their software, clients obtain the software, and smart contracts distribute payments among contributors.

A. COMPONENTS AND ENTITIES

The elements in the proposed system are of two types: 1) Decentralized network components encompassing the smart contracts and storage, and 2) Entities that interact with the components. The foundational element in our solution is

[1] https://github.com/AnonGitter20221117/nft-for-software-licensing

blockchain, as it is the primary destination all other elements directly or indirectly integrate with.

1) BLOCKCHAIN AND SMART CONTRACTS

Blockchain is the decentralization station in our solution. We leverage this technology to store data and transactions related to the software distribution immutably and execute lightweight processes in a trusted manner. We cluster the data and processes into three Ethereum-based smart contracts: ERC-1155, marketplace, and aggregator.

1) **ERC-1155**: This smart contract implements the ERC-1155 multi-token standard and gets deployed by the developer once per software. The smart contract maintains all the software distribution options as different types of tokens and keeps track of the clients who obtained each distribution. From the clients' perspectives, they can rate the software distributions they obtain.

2) **Marketplace**: This smart contract manages software listings and tracks their hierarchy of dependencies. In the marketplace, developers publish their software and specify properties such as the price, clients purchase a listed software, and contributors withdraw their share of profits from the client's payment.

3) **Aggregator**: This smart contract controls the requests to and responses from off-chain processing nodes. The smart contract announces requests to its registered nodes and efficiently aggregates the nodes' responses into a single numerical result. The requests are initiated first by other smart contracts that wish to execute infeasible processes on-chain.

2) DECENTRALIZED AND CENTRALIZED STORAGE

The employment of storage solutions in our design is twofold: 1) Store the ERC-1155's metadata files, and 2) Store the assets referenced in the metadata, such as the distribution terms and conditions, the license text, the binaries, and the open-source code.

The solution requires developers to maintain the metadata files on a public content-addressable decentralized storage network, such as IPFS, and to reference the files using a name-addressable service, such as IPNS or DNSLink. As for the assets, the developers may upload them to any storage solution they consider the most suitable, such as a private cloud solution or a public repository. However, it is in the interest of the developers to upload the assets on a solution of high quality-of-service. Otherwise, a negative experience by the clients will reflect on the software rating.

3) ENTITIES AND DECENTRALIZED APPLICATIONS

The active entities in our solution are the developer, the client, and the processors. All the entities eventually interact with the smart contracts and therefore require a Decentralized Application (DApp) to execute such interactions.

1) **Developer**: Developers are the authors of software codes. They upload and maintain their software on storage solutions, deploy ERC-1155 smart contracts, mint various software distributions, publish their software on the marketplace, and optionally provide off-chain activation services to the clients.

2) **Client**: Clients are users and buyers of software. They request and pay for the software listed on the marketplace, own the software distribution NFT on the ERC-1155 smart contract, and interact with the storage solution to retrieve the software.

3) **Processor**: Processors are general-purpose computation nodes. They register on the aggregator smart contract, listen to and participate in requests, execute the operation, and provide a response back to the contract.

Paid entities, i.e., the developer and the processor, maintain a reputation score to reflect their performance. Clients can rate each developer's software distribution once per update. On the other hand, clients and developers can rate each processor once per interaction. The processor's score also behaves as a requirement for the requests since they may impose a minimum score for any processor willing to participate. For this solution, we opt to use a lightweight reputation system, diverging from the literature in which there exists more sophisticated general-purpose and NFTs-specific solutions [36], [37].

B. SOFTWARE PUBLISHING AND PURCHASING MECHANISMS

We divide the typical sequence of interactions into two sets: 1) Software publishing and 2) Software purchasing. For each of the two sets, we visualize the interactions in a sequence diagram and describe the steps in detail.

1) SOFTWARE PUBLISHING INTERACTIONS

For this scenario of publishing software, we showcase two developers: The first of which publishes an open-source code (software A) on the IPFS network, and the second publishes a commercial executable (software B) on private cloud storage. The executable has a dependency on the open-source code. We assume both developers have Ethereum addresses and have established connections with the system's networks. Figure 2 displays the sequence of interactions among the entities and the components.

Phase 1 (Upload Software A (Open-Source)):

1.1 - 1.2) *Developer A* adds the open-source code to the *IPFS storage*, which returns the Content Identifier (CID) of the asset. The developer then prepares the metadata file and publishes it to the *IPNS network*, which returns the path of the file.

1.3 - 1.4) *Developer A* deploys a new *ERC-1155 smart contract A* on the public Ethereum network. After the EVMs validate the contract, they send its address to the developer.

1.5) *Developer A* adds a distribution model to *ERC-1155 smart contract A*, declaring that software A is open-source, and payments on using it are royalty-based.

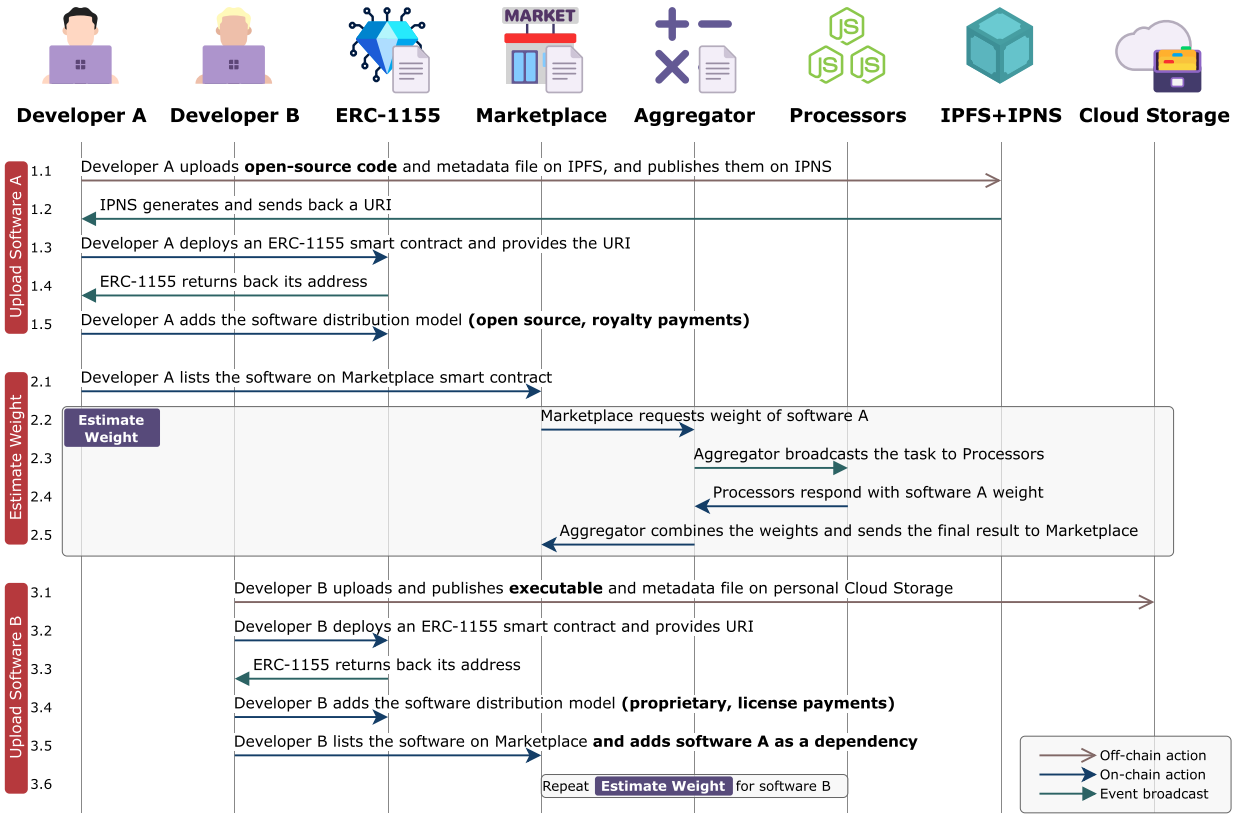

FIGURE 2. Sequence diagram showing interactions for uploading and minting software.

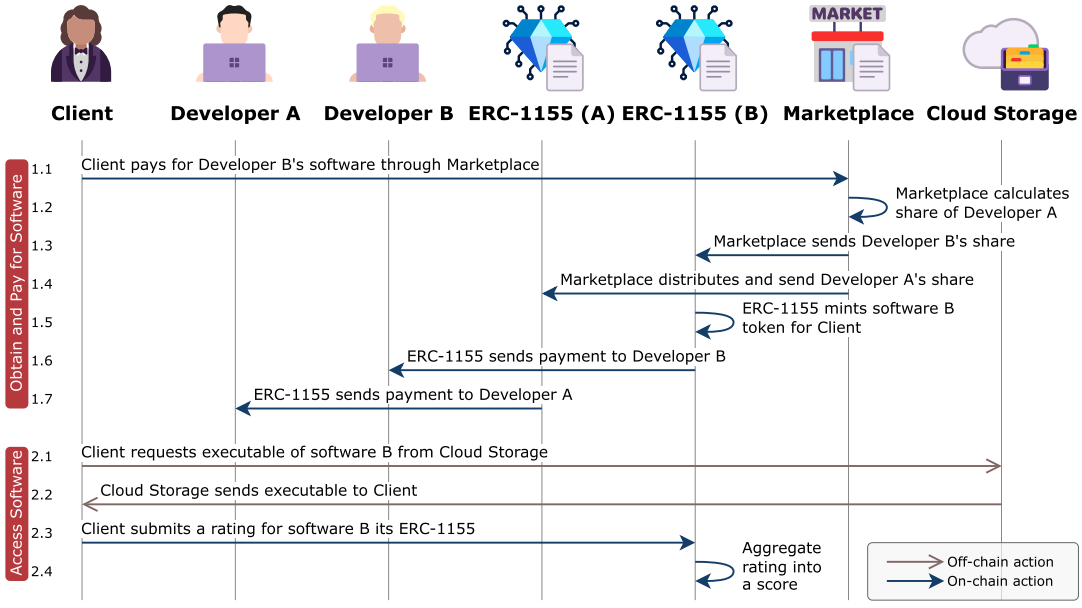

FIGURE 3. Sequence diagram showing interactions for obtaining software and distributing royalties.

Phase 2 (Estimate Weight of Software A):

2.1) *Developer A* lists the distribution model of software A on *Marketplace smart contract*.

2.2) *Marketplace smart contract* requests estimating the weight of software A from *Aggregator smart contract*.

2.3 - 2.4) *Aggregator smart contract* broadcasts the weight request to all *Processors*. The *Processors* fetch the software from the *IPFS storage* and estimate the weight based on multiple factors, such as project size (number of lines of code), and GitHub stars and forks.

Then, they return their estimates to *Aggregator smart contract*.

2.5) *Aggregator smart contract* filters out *Processors* of low reputation score, updates the reputation scores of accepted *Processors*, computes the weight, and returns the weight to *Marketplace smart contract*.

Phase 3 (Upload Software B (Commercial)):

3.1) *Developer B* adds the executable and metadata file to the private *Cloud storage*.

3.2 - 3.3) *Developer B* deploys a new *ERC-1155 smart contract B* on the public Ethereum network, which returns the contract address.

3.4) *Developer B* adds a distribution model to *ERC-1155 smart contract B*, declaring that software B is proprietary, and payments on using it are license-based.

3.5) *Developer B* lists the distribution model of software B on *Marketplace smart contract*, and adds software A as a direct dependency. Steps 2.2 to 2.5 repeat to estimate the weight of software B.

2) SOFTWARE PURCHASING INTERACTIONS

This scenario builds up on the previous one, as it illustrates a client purchasing software B, which results in both developers splitting the payment, considering that software B is dependent on software A. Figure 3 depicts the interactions in the scenario.

Phase 1 (Obtain and Pay for Software B):

1.1) *Client* purchases software B through *Marketplace smart contract*.

1.2) *Marketplace smart contract* calculates the payment share of software A as a dependency for software B.

1.3 - 1.4) *Marketplace smart contract* distributes and sends the payments to *ERC-1155 smart contract A* and *ERC-1155 smart contract B*.

1.5) *ERC-1155 smart contract B* mints a new NFT of the purchased distribution for *Client*.

1.6) *ERC-1155 smart contract B* transfers the received payment share to *Developer B*.

1.7) *ERC-1155 smart contract A* transfers the received payment share to *Developer A*.

Phase 2 (Access Software):

2.1 - 2.2) *Client* communicates with the *Cloud storage* to request software B executable and other necessary activation files. In return, the *Cloud storage* validates the request and supplies the files to *Client*.

2.3) *Client* submits a rating of software B distribution to *ERC-1155 smart contract B*.

2.4) *ERC-1155 smart contract B* updates the score of the distribution by aggregating the previous ratings and the newly received ones.

IV. IMPLEMENTATION

Herein, we discuss the critical algorithms we use to implement the smart contract methods. In Figure 4 we break down the algorithms into three contract classes, each comprising state variables and methods. The relations between state variables determine the overall organization of the class, and the relations between classes determine the overall system architecture.

Algorithm 1 `rate`: Update the Rating Score of a Ratable Object

Input: `ratableId` ← Identifier of a Ratable object
Input: `rating` ← Signed integer rating value
Require `ratableId` exists
Require caller is allowed to rate `ratableId`
Require caller did not already rate `ratableId`
Require `rating` is within range
$\texttt{score} \leftarrow \frac{\texttt{score} \times \texttt{ratings} + \texttt{rating}}{\texttt{ratings} + 1}$, where:
    `score` ← The Ratable variable's score
    `ratings` ← The number of previous ratings
$\texttt{ratings} \overset{+}{\leftarrow} 1$
Update `ratableId`'s properties to `score` and `ratings`
Mark that the caller has rated `ratableId`

The Software contract class implements the ERC-1155 standard, which includes methods to manage the NFT Uniform Resource Identifier (URI), NFT balances, and transfer of NFT ownership. Besides, Software contains state variables to keep track of the developer, distributions, and clients. The contract houses three methods to handle its variables:

- `createDist`: Generates a new distribution identifier (`dist`). Calling this method is exclusive to the developer.
- `updateDist`: Updates the modifiable properties of the distributions, such as the distribution hash. Exclusive to the developer.
- `grantDist`: Calls the ERC-1155 `mint` method to grant a distribution NFT to a client. Exclusive to the Marketplace contract.

Each distribution in the Software contract complies with a library called Ratable, allowing the clients who obtained the software to submit a rating via a `rate` method. Algorithm 1 details how ratings are added, where `rating`, `score` $\in [\![-10\,000, 10\,000]\!]$ and `ratings` $\in \mathbb{Z}_{\geq 0}$. From the perspective of the client, the rating ranges from -100.00 (dissatisfied) to 100.00 (satisfied). However, since Solidity does not support float point operations, the 2 decimal places become part of the integer value by multiplying it by a factor of 100.

The Marketplace contract class contains state variables to keep track of its owner, an Aggregator smart contract, and the listings of software distributions. There are five methods in the contract to manage operations between the clients and developers:

- `configure`: Specifies which Aggregator smart contract this Marketplace integrates with. Exclusive to the Marketplace owner.

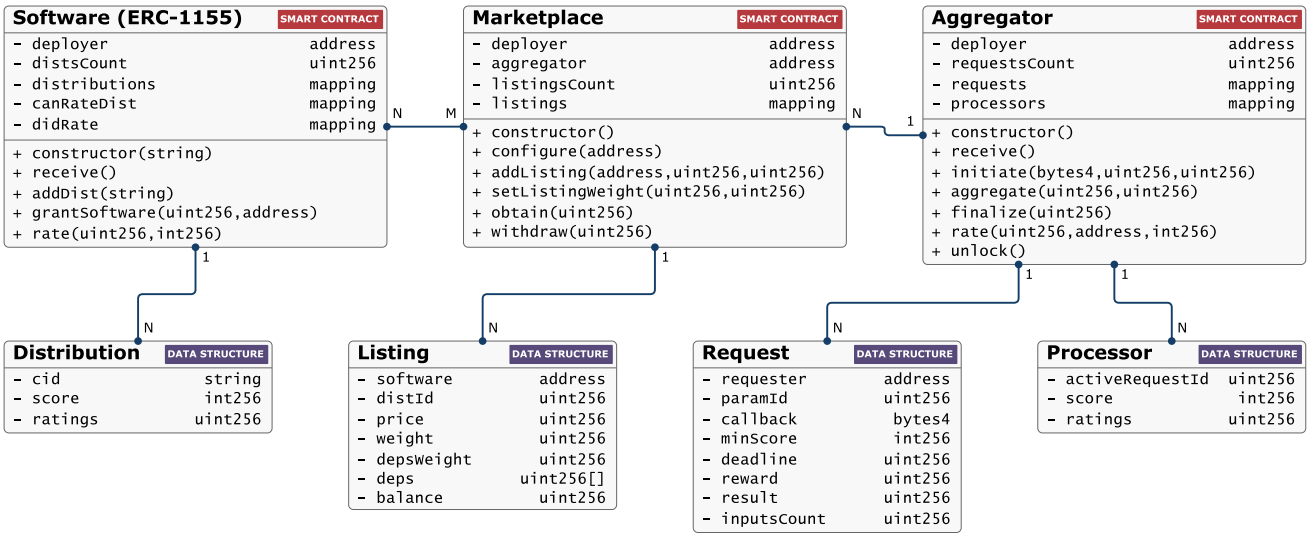

FIGURE 4. Class diagram showing the Software, Marketplace, and Aggregator contract classes and their corresponding data structures.

Algorithm 2 `addListing`: Create a New Software Distribution Listing in the Marketplace

Input: `software` ← Address of a Software contract
Input: `dist` ← Identifier of a specific distribution
Input: `deps` ← Array of other listings
Require `software` is a proper Software contract
Require caller is the developer of `software`
Require `dist` exists in `software`
Require listing does not already exist
Require all listings in `deps` exist and have weight
`/* Define a listing identifier from the keccak hash of software and dist */`
`listingId` ← `keccak256(software, dist)`
Create new `listing` object, where:
    `software` ← The input `software`
    `dist` ← The input `dist`
    `deps` ← The dependencies `deps`
    `depsWeight` $\leftarrow \sum_{d \in \texttt{deps}} (\texttt{weight} + \texttt{depsWeight})$
Call `initiate` method of Aggregator, where:
    `paramId` ← The identifier `listingId`
    `callback` ← The Marketplace method `configure`
    `minscore` ← The minimum processor score of 0
    `within` ← The request duration 1 hour
Event: Broadcast `listingId` to listening nodes

Algorithm 3 `obtain`: Obtain a Software Distribution

Input: `listingId` ← Identifier to a listing
Require `listingId` exists
Require caller does not obtain `listingId`
Require payment matches the `listingId` price
`listing` ← Listing at `listingId`
`listing.balance` $\xleftarrow{+}$ Transaction payment
Call `grant` method of `listingId.software`, where:
    Distribution is `listing.dist`
    Owner is transaction caller

- `addListing`: Generates a new distribution listing and returns a unique listing identifier (`listingId`). Handles setting the distribution weight, in alliance with the Aggregator `initiate` and `finalize` methods. Finally, this method aggregates the total weights of the entire dependencies tree into `depsWeight` variable. Algorithm 2 shows the detailed procedures of this method, where `software` and `dist` are the developer's Software smart contract address and distribution identifier, and `deps` is an array of listings representing the dependencies.
- `setProperties`: Allows the developer to set additional and optional properties to the listing, such as the price.
- `obtain`: Allows the client to acquire a software distribution `listingId`. Algorithm 3 explains the method operations in detail.
- `withdraw`: Distributes the collected revenue of a listing among the developer and dependencies based on the weights of each. Algorithm 4 clarifies the withdrawal operations.

The Aggregator contract class manages a set of processor nodes and requests using five methods. The processor nodes comply with the Ratable library, allowing requesters to rate its

Algorithm 4 `withdraw`: Receive and Distribute Listing Revenue

Input: `listingId` ← Identifier to a listing
Require `listingId` exists
Require `listingId` has accumulated balance
Require caller is the developer of `listingId`
`listing` ← Listing at `listingId`
`/* Define compact local variables for readability */`
`lB` ← `listing.balance`
`lW` ← `listing.weight`
`lTW` ← `listing.weight` + `listing.depsWeight`
foreach `dep` in `listing.deps` **do**
  `dTW` ← `dep.weight` + `dep.depsWeight`
  `/* Distribute the dependencies' share of accumulated balance among themselves */`
  $\texttt{dep.balance} \xleftarrow{+} \frac{\texttt{lB} \times \texttt{dTW}}{\texttt{lTW}}$
end
`listing.balance` ← 0
`/* Send the listing's share of accumulated balance to its developer */`
Transfer $\frac{\texttt{lB} \times \texttt{lW}}{\texttt{lTW}}$ to the `listing` owner

Algorithm 5 `aggregate`: Receive Processor Input and Perform Cumulative Averaging

Input: `requestId` ← Identifier to a request
Input: `input` ← Unsigned integer value
Require `requestId` exists
Require processor is not locked
Require processor rating score is acceptable
Require request time limit has not passed
$\texttt{result} \leftarrow \frac{\texttt{result} \times \texttt{inputs} + \texttt{input}}{\texttt{inputs} + 1}$, where:
  `result` is the request result
  `inputs` is the number of participating processors
$\texttt{inputs} \xleftarrow{+} 1$
Update `requestId`'s properties to `result` and `inputs`
Update processor active request to `requestId` and lock state to `true`

performance according to Algorithm 1. The remaining class methods are:

- `initiate`: Receives aggregation requests from external callers and creates a proper `request` with a unique identifier `requestId`, which returns to the caller.
- `aggregate`: Accepts numeric response from a processor node and aggregates it dynamically into the request, as Algorithm 5 details. Exclusive to processors.

Algorithm 6 `unlock`: Reset Processor Lock State and Transfer Incentives

Require processor is locked
Require request is finalized
`/* A request moves to finalized state after the deadline and rating periods pass */`
`requestId` ← Processor's active request
`reward` ← Payment to processors of `requestId`
`inputs` ← Number of participations in `requestId`
Update processor active request to `0` and lock state to `false`, indicating it is unlocked
Transfer $\frac{\texttt{reward}}{\texttt{inputs} + 1}$ to the processor

- `finalize`: Calls back the requester to send the final result. The first processor to trigger the method after the request time limit receives an incentive identical to other processors that submit inputs. Exclusive to processors.
- `unlock`: Allows the processor to participate in other requests in addition to transferring incentives. Algorithm 6 delineates the method. Exclusive to processors.

V. TESTING AND EVALUATION

After building and optimizing the smart contracts based on the algorithms, we evaluate the code to verify that it can provide the proper functionality outcomes, has resiliency against erroneous inputs, and lacks major programmatic security flaws.

Our evaluation phase uses Hardhat version 2.10.1, a development environment that enables running scripts to compile, deploy, and test Solidity code. The code relies on two dependencies: `ERC1155.sol` and `Address.sol`, both of which we import from OpenZeppelin contracts version 4.7.2. The smart contract compiler is Solidity Compiler (Solc) version 0.8.9, with 1 000 optimization runs. As for the deployment environment, we use a local Ethereum testing network housing 11 Ethereum accounts to deploy the smart contracts and make transaction requests. Table 1 unfolds the roles and addresses of the Ethereum accounts.

A. UNIT TESTING

The unit testing is the first set of functional validation step that we execute to ensure the smart contract operate as per the requirements of our design. We use Mocha framework to test each method of the three Solidity smart contracts, by enumerating over all the scenarios in which the methods may succeed or revert. A smart contract method reverts as a result of an input error, so that the transaction does not register in the ledger, saving the caller from losing their Ether payments. By the end of the unit testing we go through 80 different set of inputs for the methods, of which 55 are revert cases.

The Software contract method `addDist` reverts when the caller is not the developer, or when the hash input already

TABLE 1. Entities we use in testing and their truncated Ethereum addresses.

Entity	Address	Entity	Address
Aggregator deployer	0xf39F	Processor 1	0x9965
Marketplace deployer	0x7099	Processor 2	0x976E
Developer A	0x3C44	Processor 3	0x14dC
Developer B	0x90F7	Processor 4	0x2361
Client	0x15d3	Processor 5	0xa0Ee
		Processor 6	0xBcd4

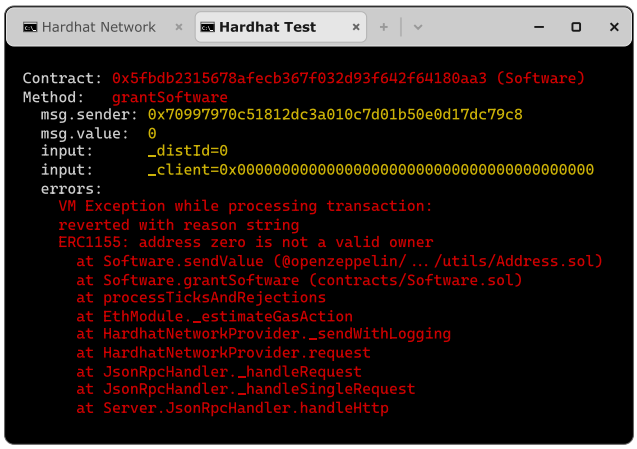

FIGURE 5. Snapshot of a failed transaction logs, showing the Software smart contract throwing an error due to invalid inputs.

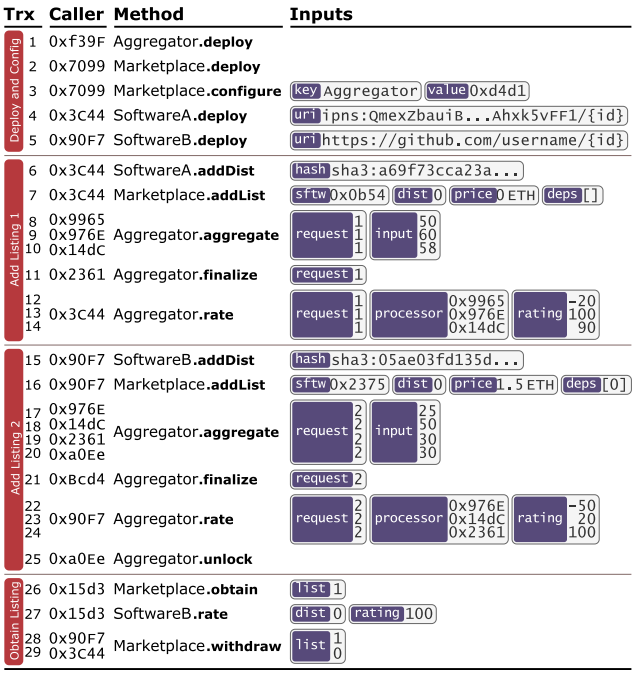

FIGURE 6. On-chain transactions and parameters we execute during testing.

exists. The `grantSoftware` internal method reverts if the client already owns the software token, if the caller is not a trusted Marketplace, or if the Marketplace grants a non-existing distribution, to a non-ERC1155Receiver contract or the zero address. The `rate` method reverts if the caller already submitted a rating, if the rating is for an unobtained or nonexistent distribution, or if the rating is out of the $[\![-10\,000, 10\,000]\!]$ range. Figure 5 depicts one of the revert cases, in which the Marketplace calls `grantSoftware` with the client being the zero address. In such scenario, the ERC-1155 contract catches the error and logs the reason.

The Marketplace contract method `configure` reverts if the caller is not the Marketplace deployer, if there is a non-finalized request, or if the Aggregator address matches the existing one, belongs to an Externally-Owned Account (EOA), or matches the zero address. The `addListing` method reverts if the caller is not the software developer, if the software distribution already exists, or if any of the dependencies do not exist or without a weight. The method `setProperties` reverts in the condition of an invalid or already weighted listing, and if the caller is not the trusted Aggregator contract. The method `obtain` reverts if the listing does not exist, if the caller already obtains the listing, or if the payment does not match the listing price. Finally, the `withdraw` method reverts if the listing does not exist, or has no balance, or if the caller is not the developer of the software.

The Aggregator smart contract method `initiate` reverts if the caller is an EOA, if the request already exists and is active, or if the specific minimum score is outside the range $[\![-10\,000, 10\,000]\!]$. The `aggregate` method reverts if the transaction proceeds the request deadline, if the request does not exist, or if the caller is locked or has a low score. Revert cases of the `finalize` method include a nonexistent request, locked or low score caller, or calling on a finalized request or calling before its deadline. Similarly, `rate` reverts if the request is nonexistent, non-finalized, or overdue requests, if the processor did not participate or already received a rating, if the rating is out of range, or if the caller is not the one who made the request. As for the `unlock` method, it reverts if the caller is already unlocked, or if the active request is yet not finalized.

B. INTEGRATION TESTING

The smart contract functional validation mimics the scenario we presented in Section III. The scenario consists of four phases, the first of which is contract deployment and configuration, followed by two phases of adding and publishing the software in the marketplace, and the last is obtaining the software. We report all the method calls that take part in the testing in Figure 6.

1) DEPLOY AND CONFIGURE

The deployment begins with the Aggregator and then the Marketplace smart contracts, returning `0xd4d1` and `0xf6c3` as contract addresses, respectively. Neither of those transactions (trx) supplies additional inputs to the constructor methods (trx 1-2). However, the Marketplace deployer establishes a connection to the Aggregator so that the Marketplace

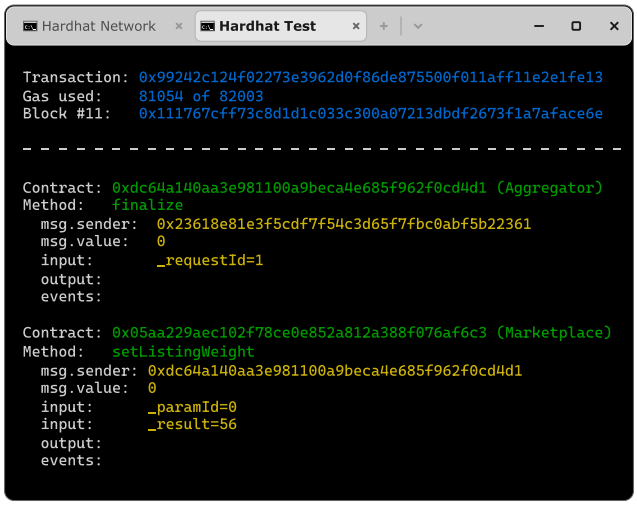

FIGURE 7. Snapshot of a transaction logs, showing the weight of a software.

can trigger it on new software listing requests (trx 3). In anticipation of the software deployments, the software developers deploy the smart contracts SoftwareA and SoftwareB, providing the IPNS and HTTPS links as URIs (trx 4-5).

2) ADD LISTING 1

The software developer adds the distribution to SoftwareA, then publishes it on the Marketplace as a new listing, providing a 0.5 ETH incentive for processors to estimate the weight (trx 6-7). Three processors submit weight estimates, which the Aggregator averages to 56 and returns to the Marketplace (trx 8-11). Figure 7 shows the weight the Marketplace receives. Finally, the software developer rates the processors individually (trx 12-14).

3) ADD LISTING 2

Phase 3 is similar to phase 2; however, the software developer chooses to list the distribution for 1.5 ETH, sets listing 1 as a dependency, and pays 0.3 ETH to the processors (trx 15-16). For the weight estimation, four processors respond with an average of 33 (trx 17-21), and the developer rates three of them back (trx 22-24). After the rating period passes, the fourth processors manually requests to unlock the account (trx 25).

4) OBTAIN LISTING 2

A client pays 1.5 ETH for listing 2 through the Marketplace, granting a license to use the software (trx 26). The client submits a rating of 100 for SoftwareB distribution 0 (trx 27). The developer of SoftwareB withdraws 0.556 ETH as revenue from the Marketplace, and similarly for SoftwareA with revenue of 0.944 ETH (trx 28-29).

C. STATIC SECURITY ANALYSIS

In conjunction with writing the Solidity smart contracts, we quantitatively analyze the security of the implementation using Slither, an open-source static analyzer [38]. The tool automatically looks for vulnerabilities using 80 detectors of common attacks. Slither categorizes the vulnerabilities into five levels of impact: high, medium, low, informational, and optimization, and two levels of confidence: high and medium. By the end of the development and analysis cycles, only the false positive optimization detectors remain in our code, in addition to six medium-impact and four low-impact vulnerabilities in the OpenZeppelin libraries. Table 2 clarifies the output of Slither detections, along with their impact, confidence, which smart contract is vulnerable, and how many times the detector triggered.

TABLE 2. Output of Slither security analysis on the solution's smart contracts.

Detector	Impact	Conf.	Smart Contract	Freq.
reentrancy-events	Low	Medium	`Marketplace.sol`	1
uninitialized-local	Medium	Medium	`ERC1155.sol`	4
unused-return	Medium	Medium	`ERC1155.sol`	2
variable-scope	Low	High	`ERC1155.sol`	4

VI. DISCUSSIONS

In this section, we discuss the cost and security of the proposed architecture and explore additional domains that can take advantage of this solution.

A. COST ANALYSIS

To get an insight into the feasibility of the system, we record and report the transaction costs while issuing repeated Ethereum requests to the network, covering all methods of the three smart contracts. Table 3 summarizes the methods and their costs to call. The gas cost reflects the time and space complexity according to the EVMs, whereas the USD cost is the fiat representation of the gas cost, which differs depending on the market's gas and Ether prices. The prices as of October 27th, 2022, are 10 Gwei per gas unit and 1 550 USD per Ether. Although the gas costs are mostly steady over time, the USD costs fluctuate more and in a less predictable manner.

The highest costs trace back to the contract deployments, which is reasonable considering they are only deployed once per system (Aggregator and Marketplace) or software (Software). The methods with the highest costs are `addDist`, `addListing`, and `obtain`, due to making cross-contract calls and heavily modifying state variables. The remaining methods cost much lower, with an average of 0.86 USD per invocation.

$$c_{\min} = nc_{\texttt{aggregate}} + c_{\texttt{finalize}} \tag{1}$$

An additional cost that we can calculate is the incentive users must pay to publish their software. Equation 1 shows the minimum payment to processors $c_{\min}$, where $c_{\texttt{m}}$ is the cost of method `m`, and n is the expected number of processors to participate in the distribution weight estimation task.

B. LATENCY AND THROUGHPUT ANALYSIS

The main factor in determining the latency and throughput of our system is the blockchain network we use. In Ethereum,

TABLE 3. Transaction cost and throughput of smart contract deployment and user-invokable methods.

Contract	Method	Cost [Gas]	Cost [USD]	Throughput [TPS]
Aggregator	deploy	559 961	8.68	2.23
	`aggregate`	41 986	0.65	29.77
	`finalize`	67 053	1.04	18.64
	`rate`	73 530	1.14	17.00
	`unlock`	28 164	0.44	44.38
Marketplace	deploy	629 672	9.76	1.99
	`configure`	43 970	0.68	28.43
	`addListing`	227 404	3.52	5.50
	`obtain`	106 599	1.65	11.73
	`withdraw`	62 700	0.97	19.94
Software	deploy	1 724 481	26.73	0.72
	`addDist`	182 193	2.82	18.12
	`rate`	68 973	1.07	59.52

the expected latency of executing the methods, which is the average time between reaching the network's pool of pending transactions and the first confirmation, is 6 seconds. This latency value is a direct result of the network's characteristics of adding a new block of transactions every 12 seconds (a slot). Additionally, this value assumes the caller of the method pays a competitive gas price.

The throughput of the methods also depends on the characteristics of the blockchain network, such as the average block capacity (in gas) and slot time. Table 3 shows the throughput estimations for each method, given the current 15 000 000 gas units block capacity and 12 seconds slot time.

C. SECURITY

We study the security of our proposed solution based on the formal threat-risk assessment model laid out by Homoliak et al. [39]. The model analyzes each component of the blockchain architecture stack, consisting of the network, consensus, replicated state machine, and application layers. We choose this model over the domain-risk alternative proposed by Lee et al. [40], as it reflects our architecture more directly.

1) NETWORK LAYER

Our solution adopts the Ethereum public blockchain network. As a result, it entertains high availability, decentralization, and openness. On the other hand, the nodes are prone to Domain Name Service (DNS) and traffic routing manipulations which can escalate to preventing them from connecting to the network, causing an **eclipse attack** [41]. Although improbable, concentrated cyberattacks on the network can theoretically cause a **Denial of Service (DoS)** on the consensus nodes and resources.

2) CONSENSUS LAYER

Under Ethereum's new proof of stake consensus mechanism, the blockchain network announces the validator node addresses ahead of publishing a new block, making them susceptible to **targeted DoS** attacks. Besides this risk, the new consensus mechanism makes previous attacks significantly more expensive and less rewarding, including the **51% attack**.

3) REPLICATED STATE MACHINE LAYER

Ethereum does not confiscate the user identity and parameters of the transaction logs. Therefore, in our solution, we abstain from dealing with private or personally identifying data. However, our implementation uses Solidity, which suffers from **language vulnerabilities** on its own. In addition, the code can contain unchangeable **implementation vulnerabilities**, necessitating thorough technical investigation using static and dynamic code analysis tools, such as Slither and Echidna (see Section V) [42], [43].

4) APPLICATION LAYER

This layer houses five components that we analyze separately.

- Non-custodial *wallets*, which are the most secure among wallet options, are susceptible to **private key theft** through malware, keyloggers, and social engineering [41].
- Fiat-cryptocurrency *exchanges* are inherently centralized and pose a **Single-Point-of-Failure (SPoF)**.
- Compromised *processor nodes* can cause **data tampering**, and they rely on an imperfect aggregation model that does not counteract **freeloading attacks** [39], [44].
- IPFS-based Filecoin [45] *storage* uses proof-of-spacetime and proof-of-replication to prevent **availability attacks**, **Sybil attacks**, **de-duplication attacks**, **outsourcing attacks**, and **generation attacks**.
- The proposed *reputation system* prevents **bad-mouthing**, and **ballot-stuffing** attacks with compulsory purchase of the software and non-reimbursable transaction fees prior to submitting a rating. However, processor nodes and software developers can perform **whitewashing** by discarding their identity of poor rating for a new one with a neutral state and a rating of 0. Additionally, a **misbehaving developer** of cloud-based software may unjustly refuse or revoke the client's license.

D. GENERALIZATION

Our design of the architecture and algorithms is aligned with a software distribution setting, enabling effective management of the licenses and fair sharing of the royalties. Nevertheless, we can transfer and expand this solution to other industries and domains with few changes to the stakeholders and minimal tweaks to the system design. One area with great potential is license agreements within governmental or business-to-business operations, such as property lease contracts, trademark licensing, and art royalties. Instead of the developer and client stakeholders as in our proposal, the new stakeholders can be government institutions, or business entities with multiple underlying owners using a multi-signature wallet [46]. Furthermore, the distribution of royalties can be based on a settled ratio, or the decision of a committee that operates as a decentralized autonomous organization [47]. These solutions typically require a high level of confidentiality, encouraging the use of private and permissioned blockchain networks,

such as Hyperledger Fabric, alongside local storage that encrypts data by default.

Another aspect in which other systems can take advantage of our solution is using the individual modular components we are proposing, including the Ratable library, Aggregator smart contract, and Marketplace's efficient hierarchical distribution of assets. In our solution we design these components and integrate them for the goal of decentralized software distribution and monetization. However, blockchain-based systems in the supply chain, healthcare, insurance, and other fields can utilize these features, as they overcome common limitations in blockchain systems, especially Ethereum.

VII. CONCLUSION

To address the issues existing in today's software licensing, we proposed a decentralized solution based on NFTs and Ethereum blockchain smart contracts to enable verifiable software ownership, direct purchase payments, and royalty distribution, all in a trusted, secure, and immutable manner. Our approach employed the ERC-1155 standard to tokenize software distributions as unique NFTs, which developers can list on decentralized NFT marketplaces for clients to purchase. In addition, developers can earn from an additional monetization stream, by receiving royalty payments when other developers use the software as a dependency. Our solution brings trusted and enforceable license agreements to protect the rights of developers and clients, and revives the open-source software model with new revenue streams. We demonstrated the effectiveness of our system as we implemented, deployed, and evaluated three Solidity smart contracts, which incorporated all the key functionalities of our solution. The smart contracts mint software distributions as NFTs, list them on a marketplace, and estimate the weight of the software to build a dependency tree, allowing fair distribution of royalties. The results of our testing and evaluation showed that all code components functioned as expected in default scenarios and in cases where entities behaved maliciously or erroneously. The results of our cost and security analyses showed that our system is feasible, cost-efficient, and resilient against known cyberattacks. As a future work, we plan to develop an end-to-end system for transparent and monetizable software distribution, and integrate the solution with well-known open-source software repositories such as GitHub and BitBucket.

REFERENCES

[1] Y. G. Grange, T. Jurges, J. Schnabel, N. P. F. Lorente, and M. Fußling, "Best licensing practices," 2020, *arXiv:2012.12994*.

[2] M. Ballhausen, "Free and open source software licenses explained," *Computer*, vol. 52, no. 6, pp. 82–86, Jun. 2019, doi: 10.1109/MC.2019.2907766.

[3] V. Stepanova and I. Erins, "Blockchain-based model for software licensing," in *Proc. 4th Int. Conf. Syst. Rel. Saf. (ICSRS)*, Nov. 2019, pp. 30–34, doi: 10.1109/ICSRS48664.2019.8987715.

[4] H. Kaminski and M. Perry. (2007). *Open Source Software Licensing Patterns*. [Online]. Available: https://ir.lib.uwo.ca/csdpub/10

[5] E. DeBrie and D. Goeschel, "Open source software licenses: Legal implications and practical guidance," The Nebraska Lawyer, Mar./Apr. 7–13, 2016.

[6] S. Li, H. K. Cheng, Y. Duan, and Y.-C. Yang, "A study of enterprise software licensing models," *J. Manag. Inf. Syst.*, vol. 34, no. 1, pp. 177–205, Jan. 2017, doi: 10.1080/07421222.2017.1297636.

[7] T. August, H. Shin, and T. I. Tunca, "Generating value through open source: Software service market regulation and licensing policy," *Inf. Syst. Res.*, vol. 29, no. 1, pp. 186–205, Mar. 2018, doi: 10.1287/isre.2017.0726.

[8] T. August, W. Chen, and K. Zhu, "Competition among proprietary and open-source software firms: The role of licensing in strategic contribution," *Manag. Sci.*, vol. 67, no. 5, pp. 3041–3066, May 2021, doi: 10.1287/mnsc.2020.3674.

[9] J. P. Moraes, I. Polato, I. Wiese, F. Saraiva, and G. Pinto, "From one to hundreds: Multi-licensing in the JavaScript ecosystem," *Empirical Softw. Eng.*, vol. 26, no. 3, p. 39, Mar. 2021, doi: 10.1007/s10664-020-09936-2.

[10] D. A. Almeida, G. C. Murphy, G. Wilson, and M. Hoye, "Investigating whether and how software developers understand open source software licensing," *Empirical Softw. Eng.*, vol. 24, no. 1, pp. 211–239, Feb. 2019, doi: 10.1007/s10664-018-9614-9.

[11] L. A. Barba, "Defining the role of open source software in research reproducibility," 2022, *arXiv:2204.12564*.

[12] S.-Y. T. Lee, H.-W. Kim, and S. Gupta, "Measuring open source software success," *Omega*, vol. 37, no. 2, pp. 426–438, 2009, doi: 10.1016/j.omega.2007.05.005.

[13] M. Krichen, M. Ammi, A. Mihoub, and M. Almutiq, "Blockchain for modern applications: A survey," *Sensors*, vol. 22, no. 14, p. 5274, Jul. 2022, doi: 10.3390/s22145274.

[14] W. Zou, D. Lo, P. S. Kochhar, X. B. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu, "Smart contract development: Challenges and opportunities," *IEEE Trans. Softw. Eng.*, vol. 47, no. 10, pp. 2084–2106, Oct. 2021, doi: 10.1109/TSE.2019.2942301.

[15] S. N. Khan, F. Loukil, C. Ghedira-Guegan, E. Benkhelifa, and A. Bani-Hani, "Blockchain smart contracts: Applications, challenges, and future trends," *Peer-Peer Netw. Appl.*, vol. 14, no. 5, pp. 2901–2925, Sep. 2021, doi: 10.1007/s12083-021-01127-0.

[16] G. Wang and M. Nixon, "SoK: Tokenization on blockchain," in *Proc. 14th IEEE/ACM Int. Conf. Utility Cloud Comput. Companion*. New York, NY, USA: Association for Computing Machinery, Dec. 2021, pp. 1–9, doi: 10.1145/3492323.3495577.

[17] M. Legault, "A practitioner's view on distributed storage systems: Overview, challenges and potential solutions," *Technol. Innov. Manag. Rev.*, vol. 11, pp. 32–41, Jul. 2021, doi: 10.22215/timreview/1448.

[18] E. Daniel and F. Tschorsch, "IPFS and friends: A qualitative comparison of next generation peer-to-peer data networks," *IEEE Commun. Surveys Tuts.*, vol. 24, no. 1, pp. 31–52, 1st Quart., 2022, doi: 10.1109/COMST.2022.3143147.

[19] H. Al-Breiki, M. H. U. Rehman, K. Salah, and D. Svetinovic, "Trustworthy blockchain oracles: Review, comparison, and open research challenges," *IEEE Access*, vol. 8, pp. 85675–85685, 2020, doi: 10.1109/ACCESS.2020.2992698.

[20] L. Breidenbach, C. Cachin, A. Coventry, A. Juels, and A. Miller. (Feb. 2021). *Chainlink Off-Chain Reporting Protocol*. [Online]. Available: https://research.chain.link/ocr.pdf

[21] M. Madine, K. Salah, R. Jayaraman, Y. Al-Hammadi, J. Arshad, and I. Yaqoob, "AppXchain: Application-level interoperability for blockchain networks," *IEEE Access*, vol. 9, pp. 87777–87791, 2021, doi: 10.1109/ACCESS.2021.3089603.

[22] P. Venalainen, "Detecting software license violations," M.S. thesis, Metropolia Univ. Appl. Sci., Finland, May 2021.

[23] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra, "Finding software license violations through binary code clone detection," in *Proc. 8th Work. Conf. Mining Softw. Repositories*. New York, NY, USA: Association for Computing Machinery, May 2011, pp. 63–72, doi: 10.1145/1985441.1985453.

[24] S. Costea and B. Warinschi, "Secure software licensing: Models, constructions, and proofs," in *Proc. IEEE 29th Comput. Secur. Found. Symp. (CSF)*, Jul. 2016, pp. 31–44, doi: 10.1109/CSF.2016.10.

[25] W.-Y. Chiu, L. Zhou, W. Meng, Z. Liu, and C. Ge, "ActAnyware-blockchain-based software licensing scheme," in *Blockchain and Trustworthy Systems*, H.-N. Dai, X. Liu, D. X. Luo, J. Xiao, and X. Chen, Eds. Singapore: Springer, 2021, pp. 559–573.

[26] L. Di Gaetano, "A model of corporate donations to open source under hardware-software complementarity," *Ind. Corporate Change*, vol. 24, no. 1, pp. 163–190, Feb. 2015.

[27] Liberapay. (Oct. 2019). *Liberapay—Payment Processors*. [Online]. Available: https://en.liberapay.com/about/payment-processors
[28] GitHub. (Mar. 2019). *GitHub Sponsors Additional Terms*. [Online]. Available: https://docs.github.com/en/site-policy/github-terms/github-sponsors-additional-terms
[29] GitCoin. (Sep. 2017). *GitCoin—Discover and Fund Public Goods*. [Online]. Available: https://gitcoin.co/grants
[30] V. Buterin, Z. Hitzig, and E. G. Weyl, "A flexible design for funding public goods," *Manag. Sci.*, vol. 65, no. 11, pp. 5171–5187, Nov. 2019, doi: 10.1287/mnsc.2019.3337.
[31] (Jan. 2016). *Outsourced Guru—Royalty-Source Code*. [Online]. Available: https://outsourcedguru.wordpress.com/2016/01/10/royalty-source-code/
[32] P. Ç. Aksoy and Z. Ö. Üner, "NFTs and copyright: Challenges and opportunities," *J. Intellectual Property Law Pract.*, vol. 16, no. 10, pp. 1115–1126, Dec. 2021, doi: 10.1093/jiplp/jpab104.
[33] Q. Wang, R. Li, Q. Wang, and S. Chen, "Non-fungible token (NFT): Overview, evaluation, opportunities and challenges," 2021, *arXiv:2105.07447*.
[34] B. Bodo, A. Giannopoulou, P. Mezei, and J. Quintais, "The rise of NFTs: These aren't the droids you're looking for," *Eur. Intellectual Property Rev.*, vol. 44, no. 5, pp. 267–282, 2022.
[35] S. M. H. Bamakan, N. Nezhadsistani, O. Bodaghi, and Q. Qu, "Patents and intellectual property assets as non-fungible tokens; Key technologies and challenges," *Sci. Rep.*, vol. 12, no. 1, p. 2178, Feb. 2022, doi: 10.1038/s41598-022-05920-6.
[36] H. R. Hasan, K. Salah, A. Battah, M. Madine, I. Yaqoob, R. Jayaraman, and M. Omar, "Incorporating registration, reputation, and incentivization into the NFT ecosystem," *IEEE Access*, vol. 10, pp. 76416–76433, 2022, doi: 10.1109/ACCESS.2022.3192388.
[37] M. Madine, K. Salah, R. Jayaraman, A. Battah, H. Hasan, and I. Yaqoob, "Blockchain and NFTs for time-bound access and monetization of private data," *IEEE Access*, vol. 10, pp. 94186–94202, 2022, doi: 10.1109/ACCESS.2022.3204274.
[38] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," 2019, *arXiv:1908.09878*.
[39] I. Homoliak, S. Venugopalan, D. Reijsbergen, Q. Hum, R. Schumi, and P. Szalachowski, "The security reference architecture for blockchains: Toward a standardized model for studying vulnerabilities, threats, and defenses," *IEEE Commun. Surveys Tuts.*, vol. 23, no. 1, pp. 341–390, 1st Quart., 2021, doi: 10.1109/comst.2020.3033665.
[40] J. H. Lee, "Systematic approach to analyzing security and vulnerabilities of blockchain systems," Ph.D. dissertation, Massachusetts Inst. Technol., Cambridge, MA, USA, 2019. [Online]. Available: https://dspace.mit.edu/handle/1721.1/121793
[41] (Jul. 2022). *A Security Framework for Blockchain Applications*. [Online]. Available: https://halborn.com/a-security-framework-for-blockchain-applications
[42] A. Groce and G. Grieco, "Echidna-parade: A tool for diverse multicore smart contract fuzzing," in *Proc. 30th ACM SIGSOFT Int. Symp. Softw. Test. Anal.* New York, NY, USA: Association for Computing Machinery, Jul. 2021, pp. 658–661, doi: 10.1145/3460319.3469076.
[43] (Mar. 2019). *Blockchain Security A Framework for Trust and Adoption*. [Online]. Available: https://www.kennisdclogistiek.nl/publicaties/blockchain-security-a-framework-for-trust-and-adoption
[44] N. Chondamrongku, J. Sun, and I. Warren, "Formal security analysis for blockchain-based software architecture," in *Proc. Int. Conf. Softw. Eng. Knowl. Eng. (SEKE)*, Pittsburgh, PA, USA, Jul. 2020.
[45] J. Benet and N. Greco. (2018). *Filecoin: A Decentralized Storage Network*. [Online]. Available: https://filecoin.io/filecoin.pdf
[46] W. Powell, S. Cao, T. Miller, M. Foth, X. Boyen, B. Earsman, S. Del Valle, and C. Turner-Morris, "From premise to practice of social consensus: How to agree on common knowledge in blockchain-enabled supply chains," *Comput. Netw.*, vol. 200, Dec. 2021, Art. no. 108536, doi: 10.1016/j.comnet.2021.108536.
[47] A. I. Sanka, M. Irfan, I. Huang, and R. C. C. Cheung, "A survey of breakthrough in blockchain technology: Adoptions, applications, challenges and future research," *Comput. Commun.*, vol. 169, pp. 179–201, Mar. 2021, doi: 10.1016/j.comcom.2020.12.028.