UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO

FELIPE ZIMMERLE DA NÓBREGA COSTA

DISTRIBUTED REPOSITORY FOR SOFTWARE PACKAGES USING BLOCKCHAIN

Recife

2022

FELIPE ZIMMERLE DA NÓBREGA COSTA

DISTRIBUTED REPOSITORY FOR SOFTWARE PACKAGES USING BLOCKCHAIN

A Ph.D. Thesis presented to the Center for Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Philosophy Doctor in Computer Science. Area: Computing Theory

Adviser: Ruy José Guerra Barretto de Queiroz

Co-Adviser: Leopoldo Motta Teixeira

Recife

2022

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

C837d Costa, Felipe Zimmerle da Nóbrega
Distributed repository for software packages using blockchain / Felipe Zimmerle da Nóbrega Costa. – 2022.
107 f.: il., fig., tab.

Orientador: Ruy José Guerra Barretto de Queiroz.
Tese (Doutorado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2022.
Inclui referências e apêndices.

1. Teoria da computação. 2. Blockchain. I. Queiroz, Ruy José Guerra Barretto de (orientador). II. Título.

004 CDD (23. ed.) UFPE - CCEN 2023-27

Felipe Zimmerle da Nobrega Costa

“Distributed Repository for Software Packages Using Blockchain”

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação. Área de Concentração: Teoria da Computação

Aprovado em: 09/12/2022.

_____

Orientador: Prof. Dr. Ruy José Guerra Barretto de Queiroz

BANCA EXAMINADORA

_____

Prof. Dr. Carlos André Guimarães Ferraz
Centro de Informática / UFPE

_____

Prof. Dr. Kiev Santos da Gama
Centro de Informática / UFPE

_____

Prof. Dr. João José Costa Gondim
Departamento de Ciência da Computação / UnB

_____

Prof. Dr. Carlos Alberto Kamienski
Centro de Matemática, Computação e Cognição / UFABC

_____

Prof. Dr. Rodrigo Elia Assad
Departamento de Estatística e Informática / UFRPE

I dedicate this thesis to all my family, friends, and professors who support me in getting here.

Acknowledgements

[felipe]$ ipfs cat QmZaiYiinMcs5daMKD4BAETszZ6zzBiYDR5V9TKsFYH4cW

Thanks to all my friends and family who supported me during this work. Especially my beloved wife Clara and my daughter Mariana for supporting me in many different ways, as so my parents Flavia and Domingos.

Thanks to Eusebio who took care of my lovely dogs Mog and Bisteka during all classes and events.

An special thanks to Ruy, that has been advising me for more than ten years, first in my master's and now in my Ph.D.; it goes without saying that you also inspire me in sports ;)

Years back, in my graduation days, my advisor (Carlos, cabm) gave me a friend's dissertation as an example to follow. Guess what? This very same work was based on the structure of Leopoldo's thesis. Thank you, Leopoldo!

Alex, you have been a good friend helping me in different manners during all this work. I miss the Nokia days.

Countless lunches at Brazzettus to discuss how we could hack Blockchain and meet consensus without the financial incentive. Thank you, Gustavo, for brainstorming and discussing the craziest ideas.

Lastly, thank you, Victor. You truly believed in the potential of this work and helped in different manners. This work is where it is because of your support. I wish we could have one more beer.

Abstract

A package repository is an essential piece of a software ecosystem where packages and interdependencies are published together with security updates. In free and open-source software, the software repositories are frequently hosted and maintained using donations or contributions in the form of computational power or financial aid. The technical solution adopted to absorb the computational power donation limits on its design, prohibiting small donors from participating with their contributions. The lack of contributions directly implies limiting repository functionalities. This work proposes a package repository using Blockchain evaluated through real-world simulations and statistics. The Blockchain described has its consensus algorithm crafted to befit the purpose of a package repository without financial appeal. The consensus algorithm relies on a forger party where peers are semi-randomly selected using a protocol to agree on the forger node. Also, the proposed Blockchain keeps a compatible layer with the traditional repositories, easing its adoption. With the adoption of the proposed Blockchain, the repositories could benefit from the computational power of small contributors, thus enabling more features for their end-users. Furthermore, this work presents a package search over peer-to-peer, computed on untrusted nodes, yet guaranteeing that the results are trusted. In this work, we present tests with a Blockchain holding more than 250 thousand packages, published over more than ten years of the ArchLinux distribution. Finally, we present a functional Blockchain that cohesively exposes more than four million package releases published over more than seventeen years of the PyPi catalog.

Keywords: blockchain; distributed computing; software packages.

Resumo

Um repositório de pacotes é uma parte essencial de um ecossistema de software em que pacotes e interdependências são publicados juntos com atualizações de segurança. No software livre e de código aberto, os repositórios de software são frequentemente hospedados e mantidos por meio de doações na forma de poder computacional ou de ajuda financeira. A solução técnica adotada para absorver as doações de poder computacional são tecnicamente limitadas, proibindo a contribuição de pequenos doadores. A falta de contribuições implica diretamente em limitações das funcionalidades do repositório. Este trabalho propõe um repositório de pacotes usando Blockchain avaliado por meio de simulações e estatísticas do mundo real. O Blockchain descrito tem seu algoritmo de consenso elaborado para atender ao propósito de um repositório de pacotes sem apelo financeiro. O algoritmo de consenso depende de uma festa de forjamento em que os pares são selecionados aleatoriamente usando um protocolo para concordar com o nó forjador. Além disso, o Blockchain proposto mantém uma camada compatível com os repositórios tradicionais, facilitando sua adoção. Com a adoção do Blockchain proposto, os repositórios poderiam se beneficiar do poder computacional de pequenos contribuidores, permitindo assim mais recursos para seus usuários finais. Além disso, este trabalho apresenta uma busca de pacotes peer-to-peer, computada em nós não confiáveis, mas garantindo que os resultados sejam confiáveis. Neste trabalho, apresentamos testes com um Blockchain contendo mais de 250 mil pacotes, publicados ao longo de mais de dez anos da distribuição ArchLinux. Por fim, apresentamos um Blockchain funcional que expõe de forma coesa mais de quatro milhões de lançamentos de pacotes publicados ao longo de mais de dezessete anos do catálogo PyPi.

Palavras-chave: blockchain; computação distribuída; repositório de pacotes.

List of Figures

List of Tables

Contents

1 Introduction

Modern software development is commonly developed on top of different reusable components, which are constantly evolving (2). Those components are often distributed in the format of packages. Stanzas of meta-information describe packages (3). A Package Manager often interprets meta-information by creating a simple interface for the user to manage the *download*, *installation*, *update*, and *removal* of packages. Frequently, package managers also provide functionality for *searching* available packages and *dependency solving* (4).

In Free and Open Source Software (FOSS), a catalog of packages is also known as a repository. Repositories and the infrastructure to host it are often maintained by a community. Hence, it is essential to count on community members to contribute in the form of donations or by establishing partnerships to guarantee the healthy functioning of the repository. The lack of computing power may threaten the independent functionality of those repositories, or even the repository itself. For example, on Python's most extensive software catalog PyPI/PIP, search functionality has been disabled since 2020 (5).

Distribution architecture for repositories "in the wild" (6; 7; 8; 9; 10) is either limited or nonexistent. Using mirrors
(11) does not encourage minor or occasional contributors, thus limiting contributions to a few community members willing to share larger amounts of computing power. Facilitating contribution in computing power has numerous advantages, including having peers close to each other, reducing network latency, and allowing the creation of a cache via a Peer-to-Peer network (12; 13; 14). The combination of both kinds of contributors, those with great power and minor or occasional contributors, can considerably increase the amount of computing power available for the repositories. As a consequence, enabling off-load processing on the network.

Considering the nature of open source projects, where data belongs to the community (15), ideally, no central authority should concentrate the power to control the repository. Each contributor should hold some responsibility, just as in a Blockchain. Decentralized, distributed, and often public, a Blockchain consists of a sequence of blocks where a given block contains the cryptographic hash of the previous block (16). The calculation of the block hash considers

the previous block hash. Therefore, validating one block implies validating its predecessor until the first block. In a Blockchain, the decision on what is published on each block are made upon Distributed Consensus. In a Peer-to-Peer network, the unknown and unauthenticated peers must agree on the next block based on simple rules.

The Blockchain proposed in this work guarantees the authenticity of the package vouched by the distribution that the user chooses to trust. An adversary will not be able to present a package that is not on the Blockchain or change a package to trick the user into installing malicious software. To ensure that, the Blockchain has the following features:

(a) All published packages are digitally signed.

(b) All the vouch actions are digitally signed, ensuring: Authentication, Integrity, and Non-repudiation.

(c) The addition or removal of users as members of the trails is also based on digital signatures.

(d) The block is also signed with the forgery digital signature.

(e) The forgery is chosen upon popularity (as demonstrated in the tests). Therefore, keeping the most interested in the correctness of the Blockchain is responsible for generating the next block.

(f) Popularity is granted to verified downloads only.

In this work, our **objective** is to assist anyone on the Internet to share a small portion of computing power for a package repository. Ultimately, we assist in removing the figure of a central node by using a Blockchain. This work presents the following contributions:

i **Blockchain**. Unlike financial-based Blockchains, where the incentive is merely financial, the Blockchain hereby presented considers the consistency of package publication based on the popularity of the stakeholders. In our Blockchain, every block holds a Merkle root; this ensures that a given package version is held on a given block. Nevertheless, we also introduce the native support to exchanging packages over the p2p network (12).

ii **Distributed Consensus**. Unlike other Blockchains, we present a method for an agreement based on Stake that is ultimately connected to distribution popularity. Considering the FOSS model of *Web-Of-Trust* (17), we propose a thrust worth of the packages based on reputation.

iii **Forger Selection**. The forger vocalizes the mutual agreement on the package publication. The forger is semi-randomly selected, given a group of possible forgers. We propose an algorithm for the group of forgers to randomly choose one of them for block publication.

The Blockchain proposed in this work will also enable understanding of supply chain security issues (18; 19), as the history of every package publication and download statics will be available. The history of the Blockchain is public, immutable, and downloadable by anyone interested, allowing the history to be retrieved and processed as security researchers may want. We organize the remainder of this work as follows:

- **Chapter 2** reviews theoretical concepts adopted and used within this work;
- **Chapter 3** presents a new method for distributed consensus: Proof-of-Download (PoDl). The PoDl securely counts the number of successful downloads in a given file. This chapter presents a manner to provide a recipe to the server, confirming that a download was finished and verified;
- **Chapter 4** introduces a Blockchain developed to host software packages with autonomous decisions on which package to publish. The Blockchain is meant to work over a Peer-to-Peer (p2p) network without privileged nodes. This chapter also introduces a test case used as an test comparing the designed Blockchain in an actual use case scenario.
- **Chapter 5** discusses our evaluation of the proposed Blockchain. This chapter presents a detailed study comparing the PyPi repository versus the ones published in our test, understanding the differences in publication times, and presenting the attacker figure. This chapter has also discuss the evaluation of the PoDls.
- **Chapter 6** states the contributions of this work, including the list and descriptions of the published tools.
- **Chapter 7** presents the final considerations, including related and future works.

1.1 Related Work

The PPIO Edge Cloud Group shows a technique called *Proof-of-Download*. The *Proof-of-Download* led by the Edge Cloud Group is meant to verify the integrity of the download as opposed to inform the server that a download has safely occurred (20). PPIO's *Proof-of-Download* has no immediate relation with the PoDl, except for the fact that, as collateral, PoDl also validates the integrity of the download on the client side.

The combination of Blockchain and IPFS is the target of different publications. Dale and J. Liu suggested using a file-sharing over p2p (12); The authors highlight the p2p feasibility and some advantages of the traditional server-client approach (12). Although they have used the IPFS, their principal repository was still a central piece pointing the clients towards the suitable IPFS Content Identifier (CID). On the other hand, in this work, we also propose the file exchange and package meta-data using the IPFS as the backbone, but using distributed consensus (**Section 4.7.1**) to avoid the figure of a central piece.

Still in the context of IPFS, Shivansh Kumar et al. suggest using IPFS and Blockchain to store medical records (21). Their work emphasizes the resilience of a Blockchain to Denial-of-service (DoS) attacks by not having a centerpiece. Thus, turning it more resistant to possible attackers and service outages. The author also covers secure patient data storage in compiling the existing solutions and detailing their architecture. In the same way, proposed in their work, our work does not rely on the central piece, making it resilient to DoS attacks. Their work also considers distributing the patient's records using off-chain storage via IPFS. However, unlike our proposal, no Blockchain is built for their purpose; instead, they rely on Ethereum and smart contracts; therefore, their functioning mainly depends on financial incentives.

The combination of file storage and Blockchain is also the subject of a study by Nishara Nizamuddin, presenting a decentralized document version control using Ethereum and IPFS (22). The work also states how multiple versions of documents can be saved collaboratively in a Blockchain. There is no Blockchain proposed to befit this subject, but the utilization of Ethereum with Solidity smart contracts. Different from our work, where the Blockchain block holds the IPFS information, in their work, they have a resolution of smart contracts to point towards the IPFS CIDs. Unlike the solution detailed in our work, Nizamuddin's somewhat

depends on financial incentives relying on the Ethereum Blockchain.

Also using Ethereum with Smart contracts, Gavin D'mello and Horacio González–Vélez presented a Distributed Software dependency management using Blockchain, and IPFS (23). In their work, they have used the NPM package repository for an evalutation (**Section 2**). Their work is focused on availability, where the package dependencies are stated in a pre-defined format. There is no consensus achieved based on the context of the package publications. Also, there is no concept of distributions vouching for packages. Unlike their work, in our work, we built a Blockchain to suit the purpose of hosting package repositories. In our work, there is no external dependency on a third-party Blockchain. So, the contribution of computing power is not split into the network but is strictly given to support the package repository. The package dependency resolution in our work follows the model used by PyPi that is known to be working and well-tested in production. Another important factor to consider is that our work does not depend on financial incentives of any kind. As mentioned in their work, the scalability of the solution is somewhat dependent on Ethereum gas.[1]

Reproducible Builds are a new method for Proof-of-Work (PoW), presented by M. Numan Ince et al. (25). The idea is that the approver will compile all the published packages sources to obtain a binary that shall be equal the the original published binary. The approver then publishes the package meta-data and binaries on the IPFS. In the work there are no details on block publication interval or the effort to conduct the relase of the packages. Also, there is no mentions to Blockchain characteristics such as time in between package publication or blocks size. The incentive to the validators is framed in a gamification format, where scores are added to the validators with more assertive contribution. Although the idea of package building seems to be functional, it is rather ineffective, inheriting one of the biggest criticism of the Bitcoin, which is efficiency (26). Depending on the package, the compilation could be computationaly intensive, since it might also depend on other packages, and this may take even more time and effort to bulid. Perhaps those with more computer power will always be the first to publish the package. Differently from the Reproducible Builds method, the PoDl is meant to work effortless as it

[1] Gas is the fee required to successfully conduct a transaction or execute a contract on the Ethereum Blockchain platform. Fees are priced `gwei`, a fraction of the cryptocurrency ether (10^{-9} ETH) Gas is used to pay validators for the resources needed to conduct transactions (24)

tampers the file with negligent amount of bytes. The dirty removal (**Section 3.1**) also validates the download as collateral, thus, leading to a more efficient manner. Moreover, the figure of the impostor is never cited on their work (25).

2 Background

In this chapter, we introduce the essential concepts used in this work. Packages repositories are explained in **Section 2.1**, while **Section 2.2** details concepts on p2p communication. One-way hash functions are exposed in **Section 2.3**, and finally, the InterPlanetary File System is discussed in **Section 2.4**, followed by a detailed description of Blockchain in **Section 2.5**.

2.1 Package Repositories

Publishers or organizations usually maintain repositories to distribute their own software, or software compatible with their platform. Some repositories may befit the particular developer language community, others the entire Linux distribution. **Table 1** lists examples of popular FOSS repositories. Most repositories also provide tools intended to search for, install and otherwise manipulate software packages.

Table 1 – Example of Package Repositories provided or used by Linux distributions and Development communities.

	Repo	Site
Linux Distro	Ubuntu	`https://help.ubuntu.com/community/Repositories`
	Gentoo	`https://packages.gentoo.org/`
	ArchLinux	`https://archlinux.org/packages/`
	Debian	`https://www.debian.org/distrib/packages`
	Fedora	`https://src.fedoraproject.org/`
Dev. Communities	Python	`https://pypi.org/`
	Ruby	`https://rubygems.org/`
	JavaStript	`https://www.npmjs.com/`
	Go	`https://go.dev/doc/modules`
	Rust	`https://crates.io/`
	Perl	`https://www.cpan.org/`

Source: Elaborated by the author (2022)

Since the repositories vouch for the packages they provide, some repositories are considered malware-free (27). Oftentimes, the package permission is coupled with the authorization in the system, which significantly reduces the threat of malware. Digital signatures together with Public Key Infrastructure (PKI) (28) or Web Of Trust (WoT) (29) integrate threat models to protect the integrity and authenticity of the files provided in package repositories, and so

allowing the existence of untrusted mirrors. Mirrors are copies of the central repository in geographically distributed servers. Companies often host their internal mirror for public repositories, significantly reducing the time to consume new software and general Internet bandwidth usage.

Linux distributions depend heavily on package repositories to provide their users with a straightforward manner for installing and updating new software. The repositories are often configurable regarding package stability and allow the selection of an end-point server (mirror). The mirror choice is usually automatically set to the geographically closest server in an attempt to reduce latency. Thus, the mirror schema naturally imposes a hierarchy on the distribution. Distributed architecture in the repositories "*in the wild*" (6; 7; 8; 9; 10) is either limited or nonexistent.

A slightly different kind of software repositories are those behind **application stores** (30; 31; 32). In essence, all repositories are meant to provide software for their users. Different from the FOSS repositories, the **application stores** provide the functionality for users to buy or subscribe to applications (or services) on their Mobile Phones, Tablets, or Computers. Thus, **application stores** are not in the scope of this work.

2.1.1 Package manager

A package manager is a set of software tools that support and automates the process of installing, configuring, and removing computer programs for a computer in a consistent manner (33). The first package managers, originated around 1994, had no dependency checks, yet drastically simplified the process of adding and removing software from a running system (34; 35). In 1995, Perl's CPAN provided dependency resolution, making it even easier to download and install software (36).

Given the importance of the package repositories to foment the collaboration in FOSS, it is not uncommon to find package manager releases coupled with the language itself (37; 38). Sometimes, the package manager can also be part of the language (embedded) (39). **Table 2** details popular package managers.

Table 2 – Example of Package managers on (a) Linux distributions and (b) Development communities.

(a)

	Repo	Package Manager
Linux Distro	Ubuntu	apt
	Gentoo	emerge
	ArchLinux	pacman
	Debian	dpkg/apt
	Fedora	yum
	Manjaro	pacman

(b)

	Repo	Package Manager
Dev. Commun.	Python	PyPi
	Ruby	gems
	JavaStript	npm
	Go	(embedded)
	Rust	crates
	Perl	cpan

Source: Elaborated by the author (2022)

2.2 Peer-to-Peer Communication

A Peer-to-Peer (p2p) network is one in which the participants (referred to as peers or nodes) communicate directly, on more or less "equal footing". This does not necessarily mean that all peers are identical; some may have different roles in the overall network. However, one of the defining characteristics of a p2p network is that they do not require a privileged set of *servers* which behave entirely differently from their *clients*, as is the case in the predominant client/server model (40).

Because the definition of p2p networking is broad, many different kinds of systems have been built under the umbrella of p2p. The most culturally prominent examples are likely the file-sharing networks like BitTorrent (41), and, more recently, the proliferation of Blockchain networks that communicate in a p2p fashion (40).

While p2p networks have many advantages over the client/server model, some challenges are unique to the p2p model. Among those:

- **Transport:** The most basic layer in a p2p network. The transport layer is responsible for the transmission/reception of data from peers. At this layer, the transported data is unknown.
- **Identity:** Cryptography is used to uniquely identify every node in a network and also guarantee non-repudiation of the messages. As the keys can be easily generated, nodes' identities are not necessarily revealed to other peers. A concerned node can rejoin the network with a new pair of keys whenever it is convenient.
- **Security:** Guarantee message integrity in a public environment is the challenge addressed in a

p2p network with different protocols, as TLS 1.3, and Noise (42).

- **Peer Routing:** In a peer routing system, a peer can search for the address he is looking for or send an inquiry to another peer who is more likely to have the answer. Contacting more and more peers increases the chances of finding the target peer. Usually, nodes view the network in their own routing tables, enabling them to answer routing queries from others. A popular routing algorithm is Kademlia (43).
- **Content Discovery:** The peers publish in the network resources they are offering. Those resources are later routed via the network when requested by other peers.
- **Messaging / PubSub:** Broadcasting to the network is an indispensable resource for many p2p implementations. In a multi-purpose network, the broadcast happens over Publish-Subscribe (pubsub) system where interested nodes subscribe to specific subjects, and others post on that subject—limiting the target nodes to the subscribed ones. One of the most popular protocols is the Gossip protocol (44).

2.3 Hashing

The hash is a function that maps data of arbitrary size to fixed-sized values. Hashes, also called one-way functions, have different applications for computer security, including digital signature (45) and password storage and verification (46). Hashes are also used in Bitcoin as the main component on the computation of the distributed consensus (**Section 2.5.2.5**) and also to store transactions on Blocks using Merkle trees (**Section 2.3.1**).

A special type of hash, called Cryptography hash, has some distinct properties that a regular hash function may not have, including:

- **Pre-image resistance:** Given a hash value h, it should be computationally unfeaseable to find any message m such that $h = hash(m)$. This concept is related to a one-way function.
- **Second Pre-image resistance:** Given an input m_1, it should be computationally unfeaseable to find a different input m_2 such that $hash(m_1) = hash(m_2)$. This property is sometimes referred to as weak collision resistance.

- **Collision resistance:** It should be computationally unfeaseable to find two different messages m_1 and m_2 such that $hash(m_1) = hash(m_2)$. Such a pair is called a cryptographic hash collision. This property is sometimes referred to as strong collision resistance. It requires a hash value at least twice as long as that required for pre-image resistance; otherwise, collisions may be found by a birthday attack (47; 48).

An example of hash algorithm is *sha256*. For any input given, the output is a string with 256 bits. It is virtually impossible to find two inputs that lead to the same hash output. **Fig. 1 (a)** exemplifies the output of a *sha256* hash. The algorithm will provide a different output by changing the phrase slightly. As demonstrated in **Fig. 1 (b)**.

Figure 1 – Example of the *sha256* output for the inputs: (a) “Rocket Man", (b) “Rocket Man 1" and, (c) “Rocket Man 10".

(a) sha256(“Rocket Man”) =
“57f6e03588382e8d28f1c8f555ec1d677f7edac11afc29ce1228d3d715ead1cf”

(b) sha256(“Rocket Man 1”) =
“3a39d36a80f495064bedcae3d464bb2d2a99d5e195ea019a67797f3b4bc9dbf5”

(c) sha256(“Rocket Man 10”) =
“05c54d35056b0dbf6a5fb77c7157aefdf729f2dd500393e473b25deaeda12b92”

Source: Elaborated by the author (2022)

In **Fig. 1 (b)** it was added a *nonce* to the string illustrated at **Fig. 1 (a)**. The *nonce* is used in cryptography to avoid common hash output due to very common inputs. With different *nonce*, the hash will be different. The *nonce* with *sha256* is used in the Blockchain’s Proof-Of-Work, explained in detail on **Section 2.5.2.5**.

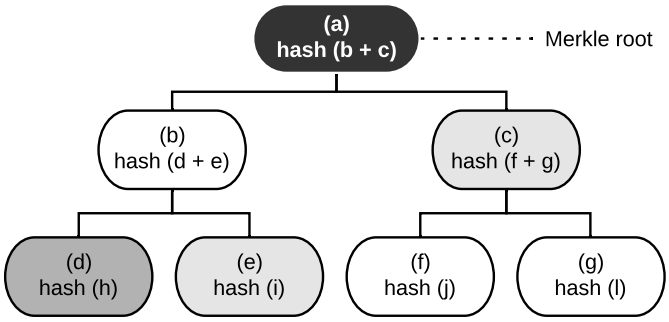
2.3.1 Merkle tree

Merkle tree is an example of a data structure that uses hashes as its kernel. A Merkle tree or Hash tree verifies the contents of large data structures efficiently and safely (49). Merkle tree reduces the size needed to store the verification that a given data belongs to a given structure.

As illustrated on **Fig. 2** verifying whether data belongs to a tree depends on proof; that proof is given by the hashes of some leaves on the tree and other essential data, listed: (50)

i The root hash of the Merkle tree;

Figure 2 – States a Merkle tree whereas the proof of information (d) is highlighted. One that presents: d + hash (i) + hash (c) and the Merkle root (a) can confirm that (d) is presented on the such tree.



Source: Elaborated by the author (2022)

ii The hash values to be verified;

iii The paths from the root to the nodes containing the values under consideration;

The proof size increases as the amount of information grow, making proof size $O(log2n)$ where n is the number of elements stored on the tree.

The utilization of Hash trees is quite broad, as an example of the file-system Btrfs (51) and ZFS (52). The Bitcoin transaction storage uses a variation of the Merkle tree, name Fast Merkle tree. Fast Merkle tree suggests skipping some intermediates hashes verification while considering a prove (50). Another notable utilization of a Hash Tree variation is in the InterPlanetary File System (53).

2.4 InterPlanetary File System

The InterPlanetary File System (IPFS) is a p2p hypermedia protocol designed to make the Internet faster, safer, and more open (54). In a P2P network such as IPFS, if one node is down, other nodes in the network can serve needed files (53).

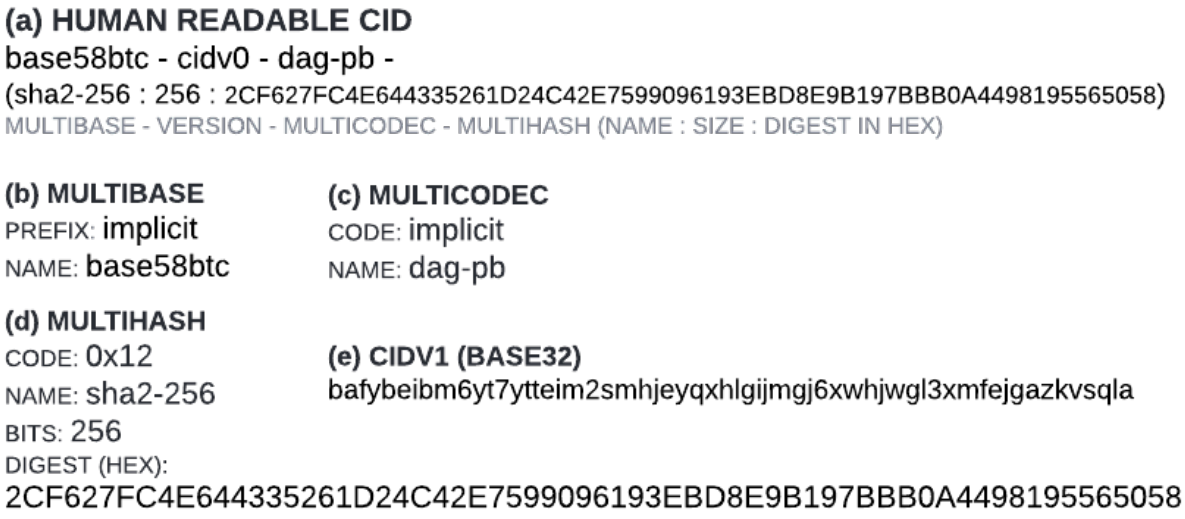
IPFS is a distributed system for storing and accessing files, websites, applications, and data (55). IPFS synthesizes successful ideas from previous p2p systems, including DHTs (56), BitTorrent (57), Git (58). The contribution of IPFS is simplifying, evolving, and connecting proven techniques into a single cohesive system, greater than the sum of its parts (59).

IPFS uses content addressing to identify content, as opposed to where it is located.

Every piece of content included in a IPFS has a Content Identifier. The CID is merely the file representation in the format of a multi-base string as presented in the **Fig. 3**; The CID includes:

- **Multibase:** The encoding that was used on the CID string-encode representation of the original byte representation (60).
- **Multicodec:** Identifier for the codec used on the ID generation.(61)
- **Multihash:** Identifies the hash type and size used in the CID computation.(62)

Figure 3 – Example of all the details on the CID QmRNAMFjhAfmq8NT2zLbSeGh3cMoQHg5ZwStbqf74he9aP. (a) Description of every bit of the CID. (b) Multibase explained. (c) Multi codec details. (d) Multhash information. (e) CID v1 on Base32 format. (1)



Source: Elaborated by the author (2022)

By default, the IPFS uses the hashing algorithm *sha256*. However, the CID is responsible for stating the algorithm. Hence, virtually any hashing algorithm is supported. Multiple hashing functions can co-exist on a single IPFS network. One important characteristic of content addressing is that:

i Any difference in the content will produce a new CID;

ii The same content will produce the same CID, regardless of the computing node.

When a file is uploaded to IPFS, other peers can access the file using the file CID. Once uploaded, the file owner will no longer have control over the file and to whom else the hash will be shared. The user who uploads a file cannot control access to the uploaded file once the hash address has been shared (53).

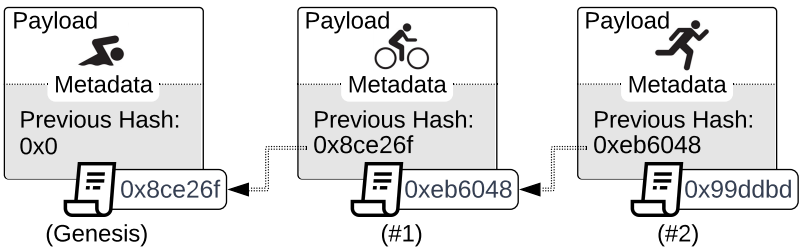
In order to download a file from an IPFS network, a user can be either part of the network as a node or use a gateway. Any node can present itself as a gateway (63). The gateways are implemented to create a compatible layer between IPFS and popular protocols such as HTTP, allowing anyone to fetch a file from an IPFS network using the HTTP protocol.

2.5 Blockchain

Blockchain is a decentralized, distributed and often public digital ledger (16). It consists of a sequence of blocks, where a given block contains the cryptographic hash of the preview block. The calculation of the block hash takes into consideration the previous block hash. Therefore, validating one block is also validating its predecessor until the first block. The first block is also known as *Genesis* Block.

In addition to the predecessor's block hash, the block also contains a payload. The **payload** is the data that is meaningful to the final application. The predecessor's hash is part of the block structure, also known as **metadata**. **Payload** and **metadata** are detailed on the **Fig. 4**. In **Fig. 4**, it is also possible to notice the block interconnection.

Figure 4 – Blockchain blocks structure. Every block contains the hash of its predecessor.



Source: Elaborated by the author (2022)

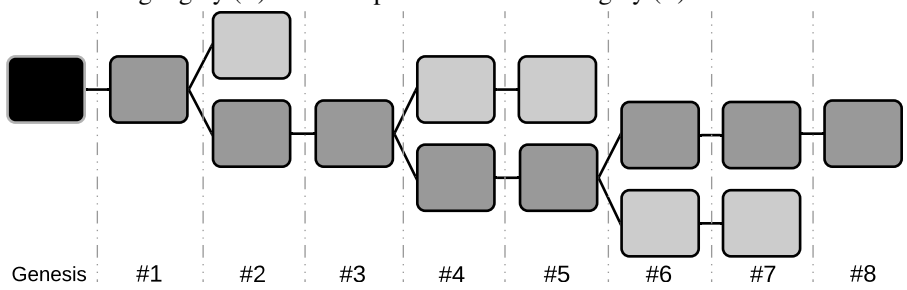
In cryptocurrencies such as Bitcoin (64), the block usually holds a Merkle tree (64) that contains the transactions. In Bitcoin, the block also includes the following metadata: time stamping, the previous block hash, version of software/protocol, nonce, and difficulty target, and

finally, the Merkle root (64).

The generation of a new block relies on the network consensus. Therefore, the time in between the block generation is not precise. Network consensus may depend on the resolution of a cryptographic puzzle, explained on **Section 2.5.2**.

As blocks contain the hash of their previous block, the data in any given block cannot be changed unless the subsequent blocks are also changed. Although the blocks have a single parent, a block may have multiple children. Each of those child blocks refers to and contains the hash of the same preview block. This scenario happens on a **Blockchain fork** or **Soft fork**. The fork can be a consequence of two (or more) peers proposing a block at (almost) the same time. That is a temporary situation, as new blocks will eventually be generated atop those, and the longest chain is considered the best history. A diverging chain is illustrated in **Fig. 5**.

Figure 5 – Example of a Blockchain diverging at heights 2, 4, and 6. Black (■) is the Genesis block. Light grey (□) are the orphan blocks. In dark grey (▩) is the main chain.



Source: Elaborated by the author (2022)

Orphan is the block not selected for inclusion in the main chain. There is no guarantee that any particular entry will remain in the best version of history forever. As the incentives are made to extend new blocks, or opposite to overwrite, it is unlikely that old blocks become orphans.

Another type of fork is a **hard fork**. It arises out of changing the way to generate new blocks. Therefore, it is considered a **hard fork** when the old method of validation for blocks can no longer validate new ones. **Hard fork** is an essential mechanism to implement new features and fixes in a Blockchain. If the nodes do not agree with the **hard fork**, there will be a split where the Blockchain will assume two different consensuses, one for each version. Therefore two different Blockchains.

In the case of a fork, the block height could temporarily point towards two or more blocks

while the block hash identifies the block uniquely and unambiguously.

2.5.1 Network Structure

A Blockchain is decentralized using a p2p network, where the participants are authenticated by their collaboration power, mitigating the possibility of unlimited reproducibility intrinsic to a digital asset (65). On a Bitcoin p2p network, every node is treated equally. There is no hierarchy, and there are no centralized or master nodes. Every node on Bitcoin is an equal peer. The network has a random topology running over TCP, where nodes are randomly peered with other random nodes (64).

Every participant in the p2p is capable of validating any given block. The validation of the Blockchain is relatively inexpensive (65). The hash of the block in question must be computed to be validated. If the calculated hash matches the hash registered in the parent block, it is valid, and every subsequent block is consequently valid.

2.5.2 Distributed Consensus

The consensus mechanism needs to ensure that the decision upon the newest block is acceptable and fair to all legit nodes, as it will define the truth of the Blockchain. This process of generating a new block is referred to as **mining**.

A valuable property in general likely to have in a Blockchain is the capability to not concentrate the power on a selected peer or peers but make the network autonomous, without a central authority. Generally speaking, the consensus mechanism should be secure enough that it is more profitable to cooperate than to subvert.

The mining process in Bitcoin uses **PoW**, which consists in guaranteeing that the odds to be selected to craft the next block is given to the node that puts more "work" on it. The new block has to follow a set of basic rules; otherwise, it will be treated as invalid by the network. Together with **PoW**, there are other methods of Proof, such as: **AlgoRand**, **Ripple**, **Proof-of-Authority (PoA)**, and **Proof-of-Stake (PoS)**.

2.5.2.1 Proof-of-Stake

In Blockchain terms, Stake is what the user has and pledges to participate in the decision on the next block. Unlike the name suggests, the consensus is not arbitrated exclusively by the one who holds more resources but by a set of arbitration "voters" that decide. Thus, making sure that it is decentralized in multiple peers.

In *PoS*, the miner of a new block is known as the forger. Before the election, to participate in the selection party, the forger has to deposit some tokens into the network, using it as collateral to vouch for the block.

The more a user stakes, the better the chances of being selected. Possibly malicious users will not act against the network, as it compromises the value of their tokens. Therefore, losing more assets than perhaps winning.

This process is key to selecting the right user to forge the block. This semi-random selection could be based on different factors, including **Randomized block selection**, **Coin age-based selection**, and **Delegated Proof-of-Stake**.

Randomized block selection: The forger is chosen based on a formula that combines the lowest hash value, and the size of the stake (66).

Coin age-based selection: A combination of randomization with a "coin age" factor. The "coin age" is determined by the age of the coin times the number of days that a coin has been held. Originally, coins older than 30 days were eligible to compete in the next block generation. Older and larger sets of coins have a greater probability of signing the next block. Once the block is signed, the age of the winning coin is set to zero. Old coins (older than 90 days) are not eligible to sign a block (67).

Delegated Proof-of-Stake (DPoS): In *delegated Proof-of Stake* the nodes have a reputation based on Stake. Only the 21 most reputed nodes can participate in the transaction validations and block generation. The reputation may vary; nodes can lose or conquer reputation.(67).

2.5.2.2 Proof-of-Authority

In *PoA*, transactions and blocks are validated and approved by validators. The validators are capable of adding data to blocks using special software. Individuals earn the right to become a validator based on reputation. To keep being a validator, the user must cooperate, as they can lose the status out of a lousy reputation (68).

2.5.2.3 Ripple

Every few seconds, all nodes run the Ripple consensus algorithm, maintaining correctness and agreement on the network. Once the agreement is reached, the ledger is considered "closed". If there is no fork on the network, the last-closed ledger maintained by all nodes in the network will be identical (69).

2.5.2.4 AlgoRand

For AlgoRand, decentralization means not having to trust a centralized entity as a single source of truth in the network. The responsibility to run and maintain the network falls to ordinary users. For that, AlgoRand utilizes the Byzantine Agreement: a communication protocol that allows the users of a distributed system to reach consensus in the presence of malicious actors (70).

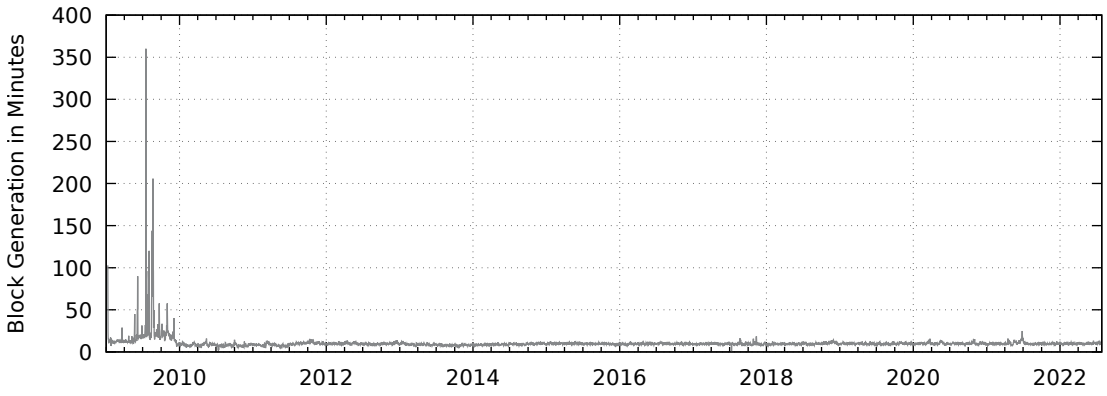
2.5.2.5 Proof-of-Work

In Bitcoin, any member of the network can generate a new block. There is no election or fixed moment where the consensus occurs. Instead, the consensus is achieved from the asynchronous interaction of thousands of independent nodes, all following simple rules. Once the cryptographic puzzle is solved, the new block can be announced by whoever solves it.

The puzzle relies on the computation of a hash with particular characteristics; those characteristics directly impact the amount of computation necessary to find the hash. Since the network power is known (In terms of hash per second), the parts are adjusted to have a strong probability of having a solution in a given time frame. In Bitcoin, it is expected to have a block

on average every 10 minutes (64). On **Fig. 6** it is possible to see the historical values of the amount of time that was necessary to generate Bitcoin blocks (71).

Figure 6 – Amount of time in between Bitcoin blocks generation.



Source: Cieśla, Kacper (2019) (71)

The first transaction in the block is a payment to the user who has produced the block. There is an incentive for those who generate the new blocks to encourage peers to participate in the mining business. This incentive is paid out in tokens, which later can be exchanged for money — the number of tokens halves every 210,000 blocks. Eventually, the value will be zero, and this incentive will no longer exist. There is also an incentive in the form of fees left by the transaction's sender.

To illustrate a puzzle, imagine if there is a need to find a hash where the first character is zero. Given the data from the **Fig. 1 (a)** and **(b)**, it is possible to change the *nonce*, until finding a hash that starts with zero. That is a relatively easy task. It is possible to notice that the *nonce* 10 will generate the hash output of **Fig. 1 (c)**.

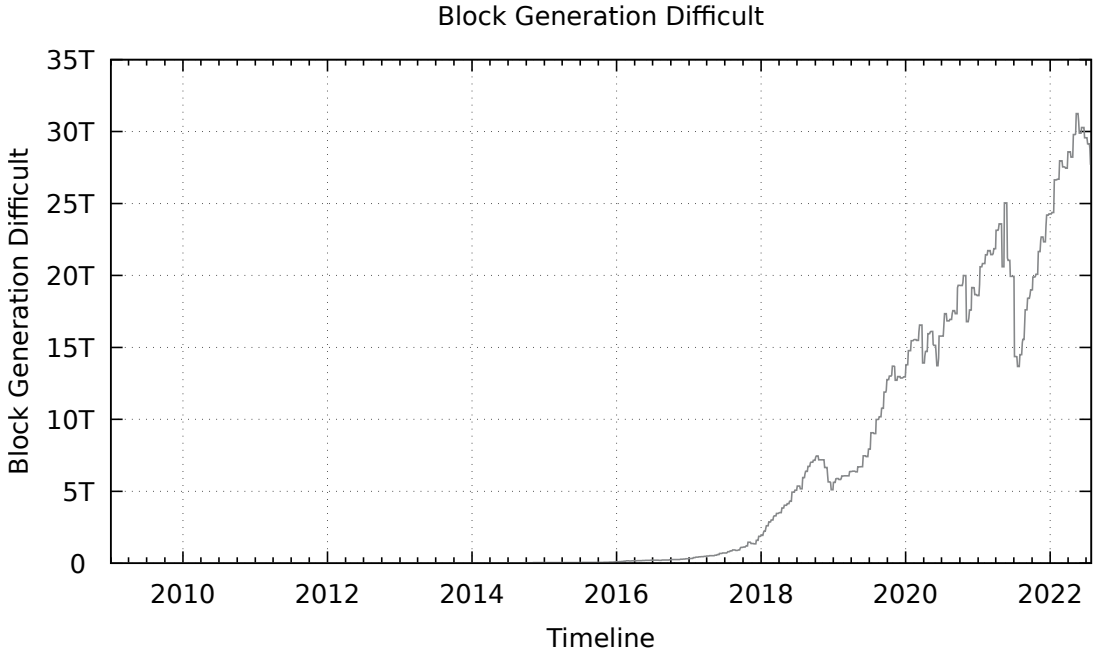
Considering the hash is evenly distributed, it is expected to have a hash starting with 0 once every 16 hashes. In numerical terms, that is the same as finding a hash that has a value less than 0x10000000000000000000000000. That is an example of a Bitcoin challenge target.

Notice that the smaller the target is, the more difficult it is to find a hash that has a value lower than it. Once the *nonce* is found, anyone can rapidly and inexpensively validate that the *nonce* meets the target.

Bitcoin's *Proof-of-Work* is very similar to the problem above, where the input of the hash is given by the block herders components, which include, among other things, the timestamp,

nonce and Merkle tree. The network sets the target/challenge, considering the hash/power amount, to probabilistic generate blocks in a near-fixed time. **Fig. 7** illustrates the difficulty of generating blocks in the history of Bitcoin's Blockchain.

Figure 7 – Difficulty to generate Bitcoin blocks.



Source: Cieśla, Kacper (2019) (71)

3 PoDl: A new consensus method

In this chapter, we present *Proof-of-Download (PoDl)*, a secure manner to count the number of successful downloads of a given file. Later, the amount of downloads is used to securely compute the popularity of file downloads, ultimately associated with their publishers. Finally, popularity is used to establish the eligibility of peers (distribution/trails members) to issue new blocks.

3.1 A Successful Completed Download

As discussed in **Section 2.5.2**, there are several known methods of obtaining distributed consensus in a Blockchain network. The PoDl is one of those methods. It demands a secure way to establish the number of downloads of a given file. The download counter is just incremented when the user effectively downloaded and authenticates the target file, giving no chance for the user to fake a download. In this subsection, we present a mechanism to ensure that a download is only counted when it is completed successfully.

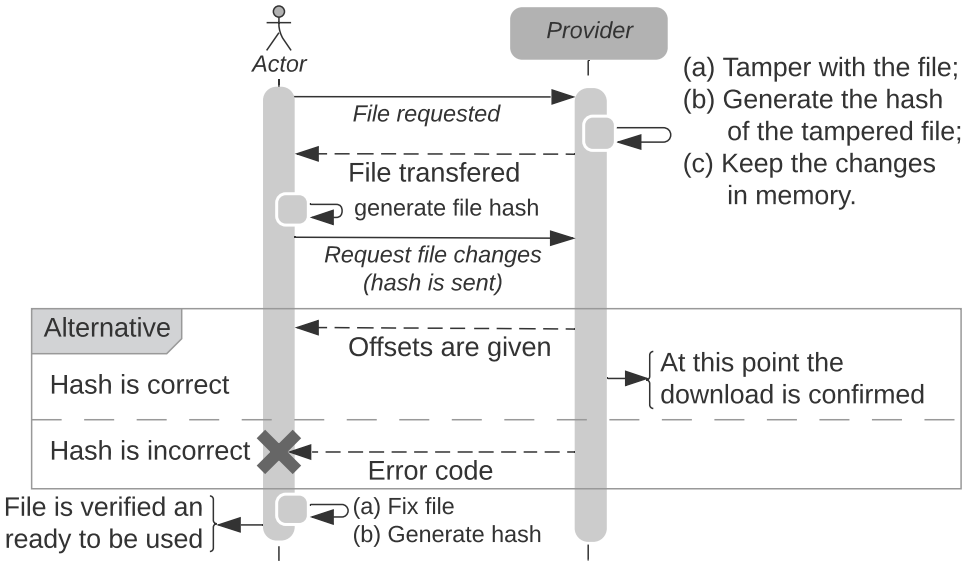
To understand how *PoDl* securely counts the number of downloads, assume that there are two peers and one artifact: a *user*, a *provider*, and a **file**. The *user* is the one who downloads the file. As the name suggests, the *provider* is the one who provides the file, while the **file** is the subject to be downloaded. **Fig. 8** illustrates the interaction of the peers on a sequence diagram.

It is notable on the diagram at **Fig. 8** that there are a few steps that make this process differ from a regular download. The first step is **file tamper**. This method aims to add random blocks of bytes (dirty) in random places of the target file. Not too many to avoid increasing the overall file size, but enough to make the file corrupted. The **Fig. 9** details the corrupted file in contrast with the original file.

Once tampered with, the file acquires some important characteristics that further support the process of verifying that the download successfully happened, as follows.

- There will be a hash for the tampered file that differs from the original file.
- The tampered file is corrupted, therefore, useless to the final application. The dirt

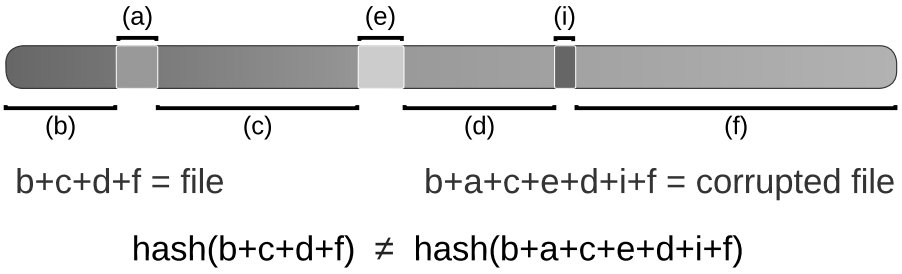
Figure 8 – Sequence diagram demonstrating a file download with *Proof-of-Download*.



Source: Elaborated by the author (2022)

needs to be removed for the file to be usable.

Figure 9 – Example of tampered file. The junction of the blobs (b), (c), (d), and (f) composes the original file, while the blobs (b), (a), (c), (e), (d), (i), and (f) makes the tampered file.



Source: Elaborated by the author (2022)

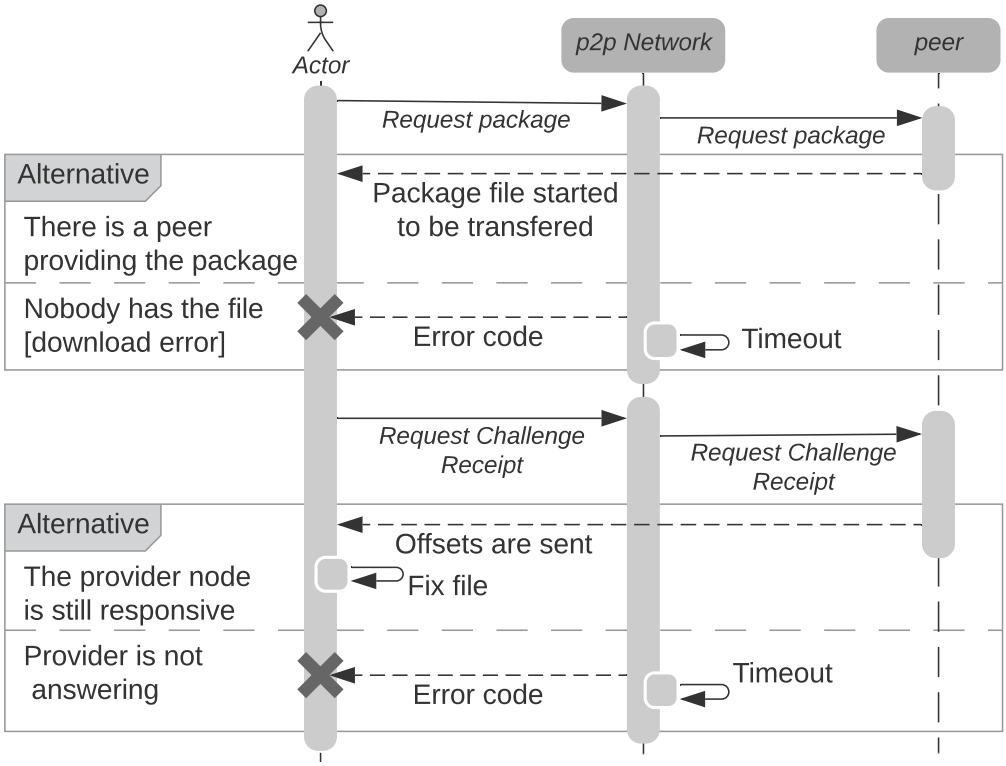
That premise about the tampered file forces the *user* to ask the *provider* the offsets for the dirty content. Otherwise, the data will be useless. The *provider* only reveals those when secure hash of the tampered file is presented. To have the correct hash, the *user* has no choice but to download the tampered file.

Using the suggested method, we guarantee that the download happened with minimal effort. The overhead in terms of size tends to be negligible. The *user* has to generate a secure hash of the file, regardless of our check, as it is necessary to validate that the data was not

corrupted. The *provider* could tamper with the file on stream while providing the content to the *user*, and the hash computation can take place on the same stream.

To have a package downloaded, the *user* must ask for a package on the network. Suppose there is a *provider* with such a package. The download will take place with the first *provider* who announces the existence of the package, as illustrated on the sequence diagram on **Fig. 10**. Once the download is finished, the package can be fixed and finally be made available to the final *user*.

Figure 10 – Simplification of the Proof-of-Download sequence diagram over p2p.



Source: Elaborated by the author (2022)

It is also essential to note that this process could happen in the clear, with no need for security or trusted channels. Anyone who may witness the process will also be able to validate it. Alternatively, this process could be replaced with Public Key Encryption (72); however, the hash approach is the most straightforward and, yet, with excellent efficiency. Moreover, the hash process is easy for multiple validations and better fits while receiving the file from various peers as in a p2p network.

Considering the files to be compressed files (73), we suggest a dirty of 16k for every 500kb of file. The dirty shall be generated on randomly generated numbers (74). During this work, the dirty size, distribution, and randomness values were not exhausted. Understandably,

one may use file correction techniques such as CRC correction (75) to reconstruct the original file. However, such a procedure uses the same, if not more, computation than engaging the challenge proposed. Yet, there is no potential attack considering such an overthrow.

3.1.1 Proof-of-Download as Consensus Method

As the packages belong to distribution, the popularity of the distributions is given due to their package's popularity. Like in PoS, where the stake is a financial asset, in PoDl, the assets are the popularity of the download as it pairs with the package popularity. Ultimately, the members of the most popular distributions or trails are eligible to participate in the semi-random selection to forger a new block. The semi-random selection is explained in-depth on **Section 4.7.1**.

Not necessarily all download records shall be kept, but those within the context of a block forgery. As the popularity could be computed considering pre-processed results from the predecessor block. Details on how the popularity ratio is calculated are explained at **Section 4.7.2**.

3.1.2 Proof-of-Download on a p2p Network

As all the steps necessary to guarantee a secure download can be validated by whoever may be observing the transactions, there does not constitute an impediment to having the same suggested protocol adopted in a p2p manner. Notice, however, that the ones who increment the download counter will be the network itself (represented by the forger), in contrast to the *provider*. For that, the *provider* needs to broadcast a receipt message confirming the download and giving a chance to any other peer on the network to verify it, including the peer that will eventually generate the block in which the popularity numbers will be present, as detailed on **Fig. 10**. The signatures of provider is used to garantee the authenticity of the download receipt, the request of the download is also observed by the forger.

3.1.3 The Challenger and the Challenged as One

The threat model for PoDl does not consider the case where the challenge is the same computer as the challenger; in this case, the challenge could be resolved without a download, breaking the premise that a download has to be continuously performed. The puzzle validator shall verify that challenger and challenged are two different peers. Alternatively, challenges shall be issued by peers considered to be trusted in the given time frame, thus mitigating the possibility of fake downloads.

As per its digital nature, the challenger and challenged can be virtual computers in the same host or the same network. Even so, a download still needs to be performed. Therefore, as computational efforts are spent, this download must be valid. This problem is also discussed in **Section 5.1.3.3** where the threat model for evaluating this work is detailed.

4 A Distributed Repository for Software Packages

This chapter presents a feature-rich Blockchain that includes file distribution in the InterPlanetary File System (IPFS) and consensus achieved by PoDls. The **Section 4.1** introduces what information we store in the Blockchain, while the **Section 4.2** further detail how the Blockchain operations supports to the package publication. The **Section 4.3** details how blocks use IPFS CIDs to publish package information. The roles on the different types of p2p nodes are presented on **Section 4.4**. The section **Section 4.5** details the characteristics of how data is published on the Blocks. **Section 4.6** introduces the package search over untrusted peers on the p2p network. Finally, the consensus agreement achieved over the forger party is detailed in **Section 4.7**.

4.1 Introduction

This section introduces a Blockchain that is based on stake. However, the stake in this Blockchain is not tokens or money, but popularity. The popularity is held by groups or organizations that vouch for software packages (distributions). The peers are any user or computer engaged on the Blockchain's p2p network, either by consuming or providing resources.

None of the peers in our Blockchain have financial incentives. The benefit is in the packages publication, in the distributions' best interest. The distribution users later consume the packages, and a reward is granted to the associated distribution for every confirmed download. The interest in having a trustworthy source for packages is mutual and interdependent.

The popularity factor measures how significant a distribution user base is. The popularity is computed based on the reward given for the distribution at every confirmed package download. Ultimately, the distributions with the most extensive user base are most interested in keeping the quality of the Blockchain, therefore, trusted to forge new blocks.

The proposed Blockchain is described as holding package information from different distributions (trails), so package managers can later consume this information to install, upgrade, and downgrade computer programs. The Blockchain is also meant to provide the search functionality being processed on the network.

Several Blockchain implementations already use IPFS as a method for distributed content (76; 77; 78; 79; 80). The block's payload is somewhat limited in size, infeasible to hold larger information such as package meta-information (described at **Section 2.1**) or the package sources. The payloads are usually limited to a hash that is later used to compose the block Merkle tree (**Section 2.3.1**). Therefore, it makes necessary to have an out-of-the-tree content provider.

The utilization of IPFS is natural, as its p2p nature allows easy setup and contributions in the bandwidth and storage space format. As the network expands, it increases the chances of having geographically close nodes, possibly reducing latency and increasing transmission speed. The setup-less network cache is easy to deploy, serving an entire local network without the need for setup on the clients. The files are served/reachable after their content (in the format of CIDs), therefore auto-verifiable and easy to store on a Blockchain.

The file can be fetched from untrusted peers on the p2p network, as the verification can be performed using the file CIDs, stored on the Blockchain. Every block also holds CID pointers to digital documents, including the package summary, in total it is expected to have blocks of 225 bytes, containing: *Block Version*, *Block Timestamp*, *Merkle root*, *Previous block hash*, *popularity list*, *Package summary*, and the *Block signature*, as detailed in **Table 3**.

Table 3 – Block structure on the proposed Blockchain. 225 Bytes wide, the blocks are bigger than a normal Blockchain block due to support for the popularity list and Packages Summary.

Size	Field Name	Description
1	Version	Version number
4	Timestamp	Creation timestamp
32	Merkle tree	Merkle tree root
32	Previous block	Hash for the previous block
46	Popularity list	CID for latest popularity list
46	Package summary	CID for the latest package list
64	Block signature	Ed25519 Forger's signature

Source: Elaborated by the author (2022)

The package summary and popularity list are published together within every block at the cost of fewer bytes. As the transaction information on package releases is published on the Merkle tree of every block, both popularity and package release files could be computed, yielding the Blockchain history. The computation comes at a cost, which may pose a difficulty for more minor or ad-hoc contributors—providing such an already computed list speeds up the

contributions, removing the barrier of downloading the block's information and indexing the summarization.

For the sake of simplicity, the actors and artifacts of the proposed Blockchain are named as follows.

- **User:** A pair of public/private keys that a human or machine can represent. The public key, at times, can be associated with a handle.
- **Trail:** A label given to an identity vouching for a package (distribution). The trails are under the control of a user.
- **Package:** Every piece of software that holds a name and a version. Every package has a publisher and a user. Eventually, the package could be part of the trail.
- **Peer:** Anyone or anything using the network to either search, download, or provide a package.
- **Provider:** A peer that is on the network to host the packages.

4.1.1 The Package summary

The package summary is a Comma-Separated Values (CSV) (81) list, whereas all of the already published packages on the Blockchain are described in its latest version. In the package summary document, it is expected to be encountered the package version, package description, and block proof for every package. Such information is later used for peers that want to search packages or provide the search functionality. The package summary file also makes it possible to look up the latest versions. **Appendix B** shows an example of a summary file.

As the list is available in a p2p manner via IPFS, the clients may opt to provide the list partially to whatever size fits the amount of computing power they are willing to share. Therefore, limiting the number of resources used to process a file whose size is arbitrary. Likewise, the local search can be done in a stream fashion as the download goes.

4.1.2 The Popularity list

The validation of stakeholders (possible block forgers) can be made more accessible by using the summarization file. The file contains information on every trail eligible for publication. The clients only need to compute part of the Blockchain to identify possible publishers with such a list. They allow easy look-up by any peer concerned about a block's authenticity.

The popularity list is a CSV file containing the trail and popularity rate it may have at the given height. Like the package summary, this file has an arbitrary size. However, the list is ordered by popularity, making the most consulted distributions on top of the file. Thus, allowing the partial download and lookup to the CSV list.

4.2 Blockchain Mechanics

Adding new packages, new trails, and vouching for packages are part of the daily actions of the proposed Blockchain. That information is also carried as part of the Blockchain data and will be checked by the forger and the users within the block generation and validation.

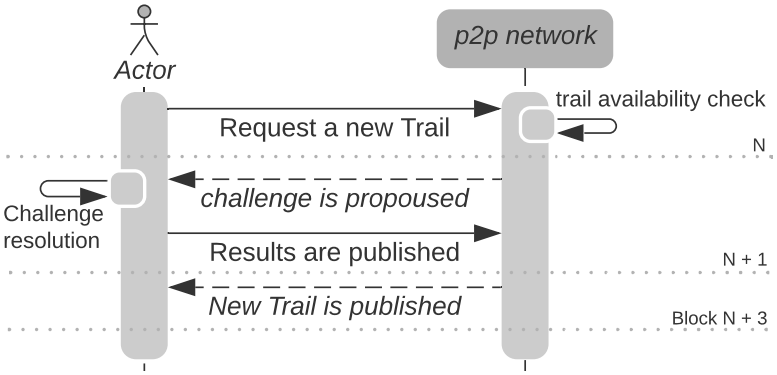
4.2.1 Creating a new Trail

Any Blockchain user can announce a new trail. No authority is needed to validate it or impose restrictions. However, requesting a name that another user already takes is not possible.

We suggest the limited number of ten new trails por block to avoid the massive generation of new trails. The preference is given to users who are already part of a popular trail. The generation is also taken in two blocks: **request** and **response**. In the request phase, the user asks the network for the trail, presenting the public key. If eligible to be in the next block, the next block will contain an encrypted challenge and can only be read by whoever has the corresponding private key. The second step is to provide this result in the p2p network. If the challenge is resolved, the trail will be published in the second block.

If everything proceeds with the regular flow (no exceptions), the trail is formalized, and at a third block, it will be possible to add more users to the trail. **Fig. 11** contains a sequence diagram that describes the creation of a new trail.

Figure 11 – Sequence diagram that illustrates the creation of a new trail.



Source: Elaborated by the author (2022)

4.2.2 Deleting a Trail

The trail is deleted whenever no more users are holding the ownership. If that happens, the trail name will be vacant for the following user who claims it. Notice that the packages vouched by a given trail are still valid, regardless of whether valid users hold the trail authority. After deletion, a new trail with the same name will be considered a second one.

4.2.3 Adding a User to a Trail

A challenge-response protocol is associated with adding every user, to guarantee that a fake user (invalid) will not hold a trail. Furthermore, access to a giving trail will not be taken by someone not interested in having such access. The user challenge is conditioned to:

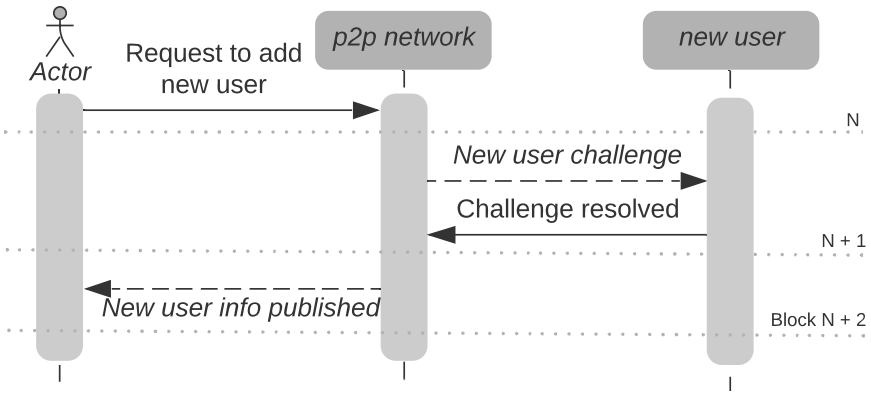
- Some user who already has privileges on the given trail grant access to a second user.
- The second user in question accepts the invitation by solving a cryptographic puzzle.

The challenge-response problem demands the user to use its private key. When added to a trail, the user is known as trail member. **Fig. 12** illustrates the sequence diagram that describes adding a user to a trail.

4.2.4 Removing a user from a Trail

Differently from adding a user to a trail, there is no puzzle for removing the user. Instead, a member of the trail, including the member himself, can publish the removal information on the

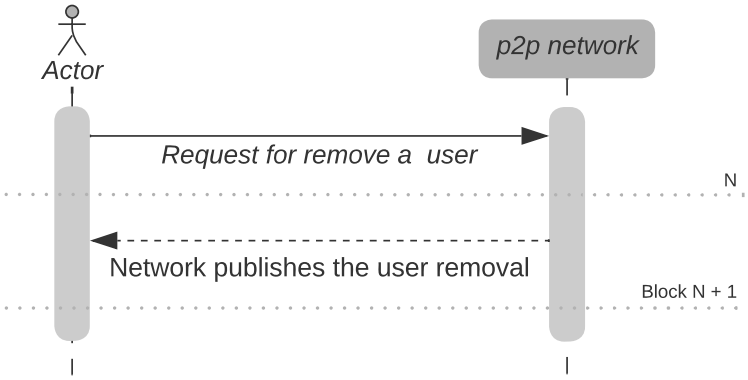
Figure 12 – Sequence diagram that illustrates the addition of a user to a trail.



Source: Elaborated by the author (2022)

Blockchain. Notice that it is technically possible to have a nested trail. That may happen when the trail is held only by users without access to their privative keys. If that happens, the user will not be able to remove himself from the trail. **Fig. 13** illustrates the sequence flow that describes the removal of a user from a given trail.

Figure 13 – Sequence diagram that illustrates the removal of a user from a trail.

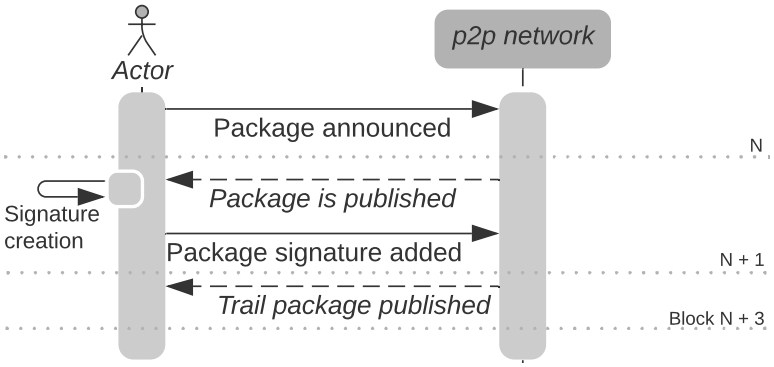


Source: Elaborated by the author (2022)

4.2.5 Adding packages to a Trail

After a package is added to the Blockchain, a trail member can vouch for it. The vouch is a simple signature provided by any trail member. The signature is made for the composition of the package checksum and the trail name. **Fig. 14** has a sequence diagram that illustrates the addition of a package to a trail. Once the package is vouched for, it will be part of a trail repository.

Figure 14 – Diagram illustrates the addition of a package to the Blockchain after the package is added to a trail.



Source: Elaborated by the author (2022)

Blocks with too many packages may increase the proof size, leading the forgers to a possible DoS attack. An arbitrary limit of one hundred packages per block was established to avoid such scenarios, especially by unpopular trails (recently created, for instance). This number was proven to be efficient in our experiments (**Section 5.1.2**), yet, not big enough to cause a gag situation. The preference to be published is given considering the popularity of each trail, such that popular trails have precedence.

4.3 File and Meta-data storage

All package meta-data and source files are meant to be distributed over IPFS. As the CID represents the data, it automatically is suitable as self-validation of the file integrity. The self-validation property makes it possible to retrieve files from public sources yet guarantees their integrity. The CIDs are published on the blocks, later fetched by the users by different means together with the proof that the data belongs to a given block.

Critical infrastructure in server rooms or public Internet access, where restrictions are imposed on Internet access, may rely on IPFS web gateways (**Section 2.4**) for downloading packages and meta-information. Ultimately, the gateways are nodes on the p2p network, accounting for the download popularity. The utilization of web gateways is discouraged, as it implies not taking full advantage of p2p benefits.

Guaranteeing the availability of the files on the IPFS network is critical to avoid dangling packages. Multiple IPFS servers are expected to host the same file, either because a node used

the file and kept a copy or because the node has the file downloaded to serve others. Mirrors can easily be deployed as a new IPFS node. Different mirror strategies can be adopted, for instance, only mirroring the most recent packages instead of mirroring the entire Blockchain data.

Atop the considerations on the expected mirrors, the publication depends on simple rules to guarantee the file's existence while the block is forged. In the event of a package publication, the publisher is expected to have the package available on an IPFS node. If available, the package is also provided by the block forger, making sure to have at least two nodes on the IPFS providing the package. Without having the package files (source or meta-information), the forger discards the package. If any block got to be published without the associated files, it is automatically considered invalid.

Considering the software life-cycle, it is expected that packages eventually become deprecated. The files for deprecated packages may not be distributed or available on the IPFS. Deprecated packages may imply security risks, so not serving those could benefit the users. (82).

The deprecated package: Different metrics could be used to understand package deprecation. Deprecation comes at discretion of the distribution that vouches for the package. Here we guide on how to layout package depreciation metrics on a Blockchain. Having the blocks being tentatively published on an equal spaced time allows us to understand the publication time based on the tip of the Blockchain, therefore having the block publish age. One option for distributions is the re-publication of the package (with the same version) at certain fixed age (or block interval). If the distribution does not re-publish old packages at a given time, automatically consider those as deprecated. Likewise, packages that depend on deprecated ones shall be treated as deprecated.

The missing package: Network failure and server outage are two of the many reasons why a package may be unavailable. The availability is somewhat proportional to the number of mirrors. Thus, the easy way to avoid missing packages is to provide at least one copy of the file on a reliable server. One interested in providing such a mirror server may have to walk over the digital ledger, downloading every single file to its instance of the IPFS node. While constructing the mirror, it is also possible to identify the already missing files. The distribution may want to ponder walking the ledger, having as a metric the availability of its files considering external

factors such as the geographical position of the potential users.

IPFS and Proof-of-Download: Both regular Hypertext Transfer Protocol (HTTP) download and the retrieval of a file via IPFS are feasible to adhere to a PoDl model, the implementation has no technical limitation or impediment. However, the implementation demands an effort outside the scope of this work. Nevertheless, considering it got implemented, the timing for new version adoption may impact the well functioning of PoDl; therefore, the Blockchain deployment shall consider milestones where the necessity of full PoDl adoption is irrelevant. This is further discussed on **Chapter 5**.

4.4 Peer-to-Peer Network

Considering the goal of having an autonomous Blockchain and avoiding any centerpiece on the communication, p2p was a natural choice. Notice that the node's communication happens in a different network from the IPFS. The IPFS is only used for the file transfers as having the files distributed over a standard IPFS comes with the benefit of counting with the already deployed infrastructure, holding about 2 (two) million new users per week (83).

One of the challenges in a p2p network is ensuring that all nodes are part of the same network. Nodes can find each other on a local area network, benefiting from low latency and fast local area network transfer without overusing the Internet. Nevertheless, the Internet node look-up is vital to ensure network integration globally. The node finder method may vary depending on the characteristics of the node's network.

For the local area network node finding, mDNS (84) was chosen, as it allows nodes to find each other with zero configuration by using a multi-cast system of the DNS records. For the Internet node finder, the bootstrap allows nodes to find each other by querying pre-fixed nodes; once connected to these hard-coded nodes, the local nodes tables are filled with other connected nodes (85). Once the node is part of the network, all of the resources (and nodes) look-up are made through a protocol based on Kademlia DHT (43). The nonexistence of nodes may imply the deprecation of the network as a whole. A fallback on the WEB API is expected by clients facing problems in reaching the p2p network.

Every node on the p2p network has its ID; that ID is a correspondent to its public key.

So, if one node wants to communicate with the other node, they can find each other via their ID. Likewise, their respective IDs expose the resources (files or capability). The capabilities of each node are represented depending on the node type. The private node key is also used to sign the packages and blocks.

The network contains different types of nodes. All are meant for different functionality. The different nodes may share specific resources, depending on the commitment/available resources: *CPU*, *Disk Space*, or *Bandwidth*. **Table 4** presents the different types of nodes and the expected resource to be shared.

The nodes that demand more *Disk Space* are meant to run on storage servers where the *CPU* and memory may be limited, yet, enough to provide the resources on the network. The *CPU* intensive nodes targets personal computers or virtual servers where the disk space is minimal, but not *CPU*. This scenario avoids the necessity of two machine instances for storage and *CPU* to deploy a contribution. The *Bandwidth* is frequently limited in upload, which can be a concern, especially for personal computers in home usage, hence the possibility to contribute with it or not.

Table 4 – Different types of nodes on the Blockchain

Node Type			resources to share		
			CPU	Disk Space	Bandwidth
(i)	Supply	Search	Heavily	None	Minimal
(ii)	Supply	Storage	Minimal	Heavily	Heavily
(iii)	Supply	Blockchain	Moderate	Moderate	Moderate
(iv)	Client		-	-	-
(v)	Forger		-	-	-

Source: Elaborated by the author (2022)

The forger node (**Table 4-v**) is a particular node meant to forge new blocks. The block forger needs to perform a block signature using a key in which the privilege of being part of a popular trail was granted. The client node (**Table 4-iv**) does not cooperate with the network as it may be running in a device with limited resources (such as an IoT device); it is meant to consume the packages and search functionality.

Implementation-wise, the contributors or supply nodes can support the three types of contribution (or any combination of **Table 4-i to iii**) in the same node. Some limited resource

contributors, such as the **search** node (**Table 4-i**), are minimalist enough to run in a browser, as shown in **Section 6.1.2.5**.

The p2p network provides via the nodes two different resources, namely **A. Package search** and **B. Blockchain update**. **Table 5** shows an example.

Table 5 – CID used by the nodes for flagging resource availability: (a) Package search. (b) Blockchain update.

-	CID
(a)	QmTp9VkYvnHyrqKQuFPiuZkiX9gPcqj6x5LJ1rmWuSySfU
(b)	QmWoWUjG717ARJDqyxZttVnnh3nmWcFJgXtwHLZifkZi87

Source: Elaborated by the author (2022)

4.5 Blockchain Characteristics

Releasing a new block every two hours, we suggest having a blockchain with blocks of 225 bytes, containing *Block Version*, *Block Timestamp*, *Merkle root*,*CID for latest popularity list*, *CID for the latest package list*, *Ed25519 Forger's signature*, and the *Previous block hash*, as detailed in **Table 3**.

Although the download size is 225 bytes, the information is contained in 32 bytes after verification. Only the block hash needs to be saved. By keeping all block hashes in a file, the offset denotes the block number (**Equation** 4.1). Therefore, the local data for block validation is proportional to the size of the chain in blocks. The blocks can always be downloaded and later compared to the saved hash or downloaded by its hash. Keeping the hash for elderly blocks may not be necessary as those may hold outdated or discontinued software. The amount of block hashes to keep is at the user's discretion, as the blockchain can always be downloaded as necessary.

$$f(x) = x \times 32 \tag{4.1}$$

where:

$f(x)$ = Offset for the Trusted Block Hash

x = Block height

4.5.1 Blockchain update

All nodes that depend on an up-to-date Blockchain version shall update as frequently as two hours (block publication time). The updates could either be via the p2p network or via WEB. The updates are expected to happen by downloading the entire block Merkle tree. The nodes entitled to mirror the Blockchain (**Table 4-iii**) would also provide the details on every information published on the Merkle tree. Over and above, peers can listen to the block publication broadcast, keeping a live updated version of the ledger.

The p2p mirror nodes are meant to provide the updates per block; The block height can be used to identify blocks. It is possible to have more than one valid block at a given height. Alternatively, blocks can be retrieved by their unique identifier, the block hash. That block hash can be later validated at the ledger's most significant chain. Additionally, mirrors can validate the integrity of each block and all the provided information.

Updates on the package source files (or meta-data) are only necessary when packages need to be installed or updated. Mirrors that may want to host packages source files and meta-data (**Table 4-ii**) can keep an IPFS copy of every file on the chain.

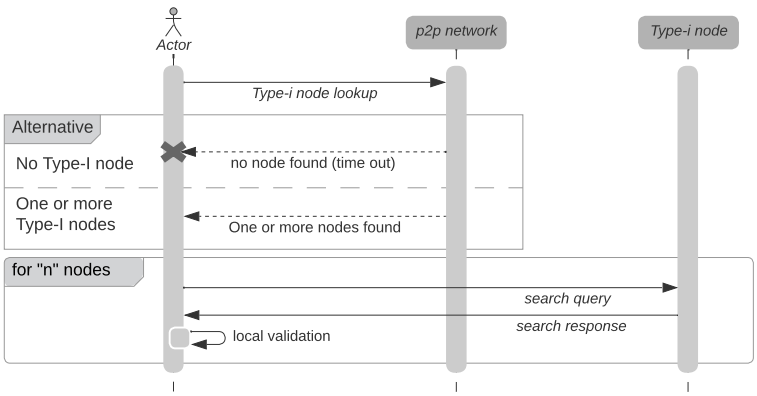
In practice, the package manager will update the tree on every operation, letting the user know if packages are available for update. The chain update should be faster if frequently updated. Otherwise, cumulative updates may delay the update process.

4.6 Package Search

As the name suggests, the main goal of the functionality is to provide a search under every available package on the Blockchain. To perform a search, the client first has to identify the nodes that provide the search functionality: over the p2p network, the client has to look up the "search resource" (**Table 5**) using the Kademlia DHT. Once identified, the client will place a search request for the selected search node (**Table 4-i**). The look-up is meant to happen over the **package name** and **package description**. The search supports limited regular expression patterns. Limitation aimed at avoiding ReDoS attacks (86). The sequence diagram for the search operation is illustrated in **Fig. 15**.

As a return of every search result, the search node provides a name, version, description,

Figure 15 – Diagram that illustrates the search operation using the Blockchain p2p network.



Source: Elaborated by the author (2022)

and block proof for every match. The proof establishes that the given information belongs to a given block. The proof, as illustrated in **List. 1**, contains all the hashes required to establish whether a given data belongs to a Merkle root. By downloading the block, the user can confirm that it is trustful by comparing its hash with the local register; furthermore, the Merkle root is used to validate that the information obtained is part of the block.

Listing 1 – Proof giving for every search result. Used by the client to establish the authenticity of the data.

```
{
 "block":15804,
 "info":"AeJniV9kxOd/vOEz2BW9JRp42gkuczLd6v9u31bb3UgGcGlw/zAuMg==",
 "hashes":[
  {
   "type":1,
   "hash":"31aef495829f334f76dcb9dd02794d539de350eb50551803c3d2d37e0851137e"
  },
  {
   "type":-1,
   "hash":"13189bb8eb34bdc30d5746460c507e20421865ddf0675891250b21b87fc66c49"
  }
 ]
}
```

Source: Elaborated by the author (2022)

Search as WEB API: Via p2p gateway or local database, websites can provide the search functionality. Regardless of the method used, the proof for every record must be presented, providing the user who requested the search a receipt for later validation. The web interface

is essential for infrastructures where Internet access is limited. The p2p gateway can present itself for the network as a regular node, performing the search and giving back the search results to the end user. The local database can be downloaded, given the last block's summary list, or computed by the Blockchain history.

Local search: In cases where storage size and processing power are not limited, nodes can keep the last summary list saved, updating as new packages are released. Within the saved list, the users could search in their local database. The CPU power availability is expected to have faster results within that method.

Building the search list: Any peer that is willing to cooperate with the network by providing the search functionality (**Table 4-i**) has to either build or download its search table. Computing the search table means indexing the Blockchain by package, keeping the latest versions of all packages and their respective names, descriptions, and versions.

A summary list is also published on every block via IPFS. When downloaded from a doubtful peer, the list hash can be validated against the CID posted on the last block—allowing a node to retrieve the latest package information without needing to compute the packages on the entire Blockchain.

4.6.1 Search Result Threat Model

The threat model considers the search to prevent bogus results from being presented. For that, user participation is needed, as discussed in what follows.

- The user shall validate that the search keyword is part of the package description or name.
- The user must validate that the package information was published on the given block.

The user shall disregard the search result if such criteria are not met. Furthermore, the user shall consider the latest version in the package summary file on the tip block. It is up to the user to install a package different from the most recent version of a given package. The concealment of package information may be to the user's perception via the different responses on the network. Ultimately, the client may consider the results for multiple searches.

The negative response while searching for network nodes that provide the search functionality could be due to the nonexistence of search nodes (**Table 4-i**). The former may indicate the deprecation of the network.

4.7 Distributed Consensus

There are a few nodes that could issue a block, and those are the ones most interested in having a consistent network: the block forger (**Table 4-v**). Since the network is made of nodes that do not necessarily trust each other, it must agree on the block publications and their contents. The consensus is obtained via pre-established rules over the p2p network.

Forger nodes possess a private key in which the correspondent public key is part of a trail in which the popularity grants permission to possible issue blocks. The permission is temporary within a block forgery time frame, given the trail popularity ultimately associated with the user. A summary of the popular numbers of each trail is published on every block (popularity list). Popularity changes on every block.

Blocks published on the network without a valid signature from a valid forger shall be disregarded. Atop of the forger signature, for a block to be considered valid, it has to follow a minimal set of rules, listed below.

- The forger signature on the block must be valid.
- All package publication and trail operations signatures must be valid.
- The block respects the limits on the number of new trails and packages.
- Trail members have correctly solved the trail puzzle.
    - All publishers have to be validated.
- Files need to be reached over the IPFS network. The forger must provide a copy.
- Summary for packages and tokens must be valid accordingly to the Blockchain history.

Blocks that do not meet one of the above mentioned items are automatically regarded as invalid and should not be trusted by the users. Since all validation data are public, block validation can be held by any peer on the network, including other forgers.

Since it is expected to have more than one forger interested in publishing a block, there is a criterion in which a forger is randomly elected to forge a block. This computation follows an agreement labeled, the forger party.

4.7.1 The Forger Party

Every 118 minutes, one of the forgers initiates the forger party. The forger party is the process where a forger gets selected to forge a block. The party is divided into phases, namely **Participation announcement**, **Ticket revealing**, **Block publication**. All those phases are subject to time-outs. The party takes no longer than two and a half minutes.

Participation announcement: Participation in the forger party is not mandatory. If the forger happens to be interested in forging a block, it has to announce such interest. The announcement of participation is combined with a payload later used to select the forger node. This payload is a *sha256* hash representing the concatenation of the message timestamp and two 32-bit numbers picked at random **Equation** 4.2, so-called Ticket.

$$\text{Ticket} = \text{sha256}(\text{timestamp} + \text{rand}_{\text{I}} + \text{rand}_{\text{II}}) \tag{4.2}$$

The computed hash is later signed (using the forger key) and published on the p2p network for all listeners. Meanwhile, the forger also listens for the other nodes' publications and saves each publication in a table. If the signature does not match or is not given by a valid forger (member of a popular trail), the publication will be automatically discarded.

The participation announcement takes no more than 60 seconds. After that, any new message will be discarded. Likewise, messages published before 118 minutes, after the publication of the latest block will also be discarded.

Ticket revealing: Each forger reveals the Ticket over the p2p network one minute after the hash

publication. Every other saves this Ticket information on the table. The Ticket information is validated; if there is a mismatch between the Ticket and the published hash, the data is discarded. In this step, the timestamp will also be validated; it must be befitting to the message publication.

Once the tables are filled, every forger has a similar table. The first random number must sort the table; lower numbers first. By so, every forger (with access to the same announcements) has a common position on the table.

After sorting the table (**Table 6**), an XOR operation takes place, considering all of the randomly generated numbers, the result of this operation leads to a seed number, the remainder of the division of the seed number by the number of forgers in the party is the elected node, considering its position on the table.

Table 6 – Example of party-table information, sorted by rand-I. Notice that this table is held by each participant in the party while electing the forger for a block.

#	name	ts	rand-I	rand-II	hash
1	Cubert	1608216371	10	19	0ebe27ebb...d6690bad7
2	Nibbler	1608216334	15	1	f89dc13a2...b0e94020b
3	Heber	1608216340	40	100	80cbbf7b6...5db692084

Source: Elaborated by the author (2022)

When two (or more nodes) publish the same random number, the second random number can be used to untie the selection. Suppose the second number also happens to be equal. In that case, the untie is given to the node where the name is lowest, considering its name's ASCII characters.

The users of the four most popular distributions are allowed to participate in the forger party (Considering the popularity ratio posted on the previews block). The remaining users shall be disregarded. Since this forger party happens in plain sight, every other node can validate. A diverging tree is expected if a node considers a different list of forgers. One tree will be chosen for the next block forger party. The tip common to more forgers will prevail.

Block publication: Once the node becomes aware of its selection, the announcement of the new block is expected on the p2p network. If an impostor forger becomes selected, it will not publish invalid data, as the block will be invalid. On the occasion of never publishing the block, the chain will be fixed on the next forger party, whereas a valid forger will likely be picked within 2 hours.

4.7.2 Popularity Rate

The popularity rate is directly proportional to the number of downloads. Every block computes the popularity considering the current download rate and previous popularity. As expressed in the **Equation** 4.3. P stands for popularity while p_p is previous popularity, c_p is currently popularity, and t is total amount of downloads of a given trail.

$$\mathrm{P} = ((\frac{\mathrm{p_p}}{100} \times \mathrm{t}) \times 0.3) + (\mathrm{c_p} \times 0.7) \tag{4.3}$$

The popularity of each trail is published on every block. Therefore any user will be made aware of it. As each download gets confirmed, the number of downloads on a given trail increases to be used on the following block publication. Every forger keeps its count of the number of downloads. The values on the amount of download in between the forgers may vary, but, presumably, the elected forger will give a good approximation of the download numbers, as it is in the forger's best interest to keep the consistency of the Blockchain.

In this scenario, fake clients need to download the package making it more expensive than the challenge generation, therefore mitigating the possibility of having counterfeit downloads. The cost of the challenge is inexpensive, as the package needs integrity validation nevertheless.

5 Evaluation

This chapter describes the experiments conducted for the validation of the Proof-of-Download and also the Blockchain details described on **Chapter 4**. We hereby describe two different simulations crafted to validate the specificities of the proposed Blockchain. The first test (see **Section 5.1**) is meant to test the Blockchain mechanics (**Section 4.2**) and the Proof-of-Download (**Chapter 3**). The second test targets the forger party (**Section 4.7.1**) considering aspects while deployed on a geographical disposed p2p network in agreement to **Chapter 4**. The first simulation uses the package base from ArchLinux, whereas the second uses Python's PyPi software base.

5.1 First Test: Evaluating PoDl and Blockchain Mechanics

The main objective of the test hereby described is to guarantee the healthy functioning of the Blockchain Mechanics and verify that the Proof-Of-Download is correctly used to calculate the distribution popularity. Thus, the blocks got forged by the correct peers. Nevertheless, the test was also meant to understand package publication delay and Blockchain size.

In this test a evaluation Blockchain, where all packages from the ArchLinux (87) distribution were hosted. Hence, it was possible to understand the feasibility of having a distribution in the proposed format. With the testal Blockchain, it is also possible to analyze factors such as block size, perfect timing for block generation, and format and disposition of the data for the different trails. Most importantly, the evaluation validates the rules proposed on **Chapter 3** and **Section 4.2**.

ArchLinux was chosen because it has a good balance of stability and package update frequency. It also has a third-party repository Arch User Repository (AUR) (88), which will be handled in future work. The simulation deals with almost 5000 users publishing packages over ten years of ArchLinux package history.

Some characteristics were laid aside while generating the Blockchain, such as package signature generation. The signatures are already proven to work, and are the subject of other works (89). Hence, signatures were not considered in the simulation or threat model. As

collateral putting the signature aside increase the speed of the simulation.

The simulation software does not test the p2p network; instead, it tries to achieve consensus. The package creation time is irrelevant within the block interval as the acknowledgment of the package's existence to every peer is given on the block publication. Therefore, eventual delays in a package publication may put the package to the next block but irrelevant within the block window.

The presence of a bad actor trying to subvert the consensus does not need to be expressed, as every peer advocates on its behalf. Nevertheless, a joint agreement is meant to be met by the proposed rules.

During the tests it was not collected numbers on the downloads' performance nor the files' availability among the contributors. Dale and J. Liu (12) published relevant work on package repositories using p2p where the statics on the download performance is shown, considering the file availability.

5.1.1 The Test Blockchain

In ArchLinux, the package creation recipes are placed in a Git (90) repository (91). The package receipt contains all information necessary to validate and compile the package sources. An example of a recipe file or PKGBUILD is illustrated at **List. 2**. The metadata expressed in the PKGBUILD file is attributed to a given set of known variables that are further used to compile the package.

Listing 2 – Relevant parts of the PKGBUILD file for the OpenSSH package.

```
pkgname=openssh
pkgver=7.9p1
pkgrel=1
pkgdesc='Premier connectivity tool for remote login with the SSH protocol.'
url='https://www.openssh.com/portable.html'
license=('custom:BSD')
arch=('x86_64')
makedepends=('Linux-headers')
depends=('krb5' 'OpenSSL' 'libedit' 'ldns')
optdepends=('Xorg-xauth: X11 forwarding'
            'x11-ssh-askpass: input passphrase in X')
validpgpkeys=('59C2118ED206D927E667EBE3D3E5F56B6D920D30')
source=("https://ftp.openbsd.org/pub/OpenBSD/OpenSSH/portable/
            ${pkgname}-${pkgver}.tar.gz"{,.asc}
        'sshdgenkeys.service'
        'sshd@.service'
        'sshd.service'
```

```
18          'sshd.socket'
19          'sshd.conf'
20          'sshd.pam')
21  sha256sums=('6b4b3ba2253d84ed3771c8050728d597c91cfce898713beb7b64a305b6f11aad'
22              'SKIP'
23              '4031577db6416fcbaacf8a26a024ecd3939e5c10fe6a86ee3f0eea5093d533b7'
24              '3a0845737207f4eda221c9c9fb64e766ade9684562d8ba4f705f7ae6826886e5'
25              'c5ed9fa629f8f8dbf3bae4edbad4441c36df535088553fe82695c52d7bde30aa'
26              'de14363e9d4ed92848e524036d9e6b57b2d35cc77d377b7247c38111d2a3defd'
27              '4effac1186cc62617f443854151030 21f72f674f8b8e26447fc1139c670090f6'
28              '64576021515c0a98b0aaf0a0ae02e0f5ebe8ee525b1e647ab68f369f81ecd846')
```

Source: Vinet, Judd and Griffin, Aaron (2019) (91)

Along with the ten years of package publication, the PKGBULID files suffered changes in how the file was read, especially with the support for having variables set with other variables' values. Considering the evolution of the PKGBUILD file, little changes were applied to certain PKGBUILDs, allowing the files to be parsed regardless of their version. This normalization does not negatively impact our analysis, as the analysis is not considering how the packages are built; consequently, most of the information on the PKGBUILD is ignored, except for package names and versions. Thus, the work of **Downloading and processing the packages information** takes place before **Constructing the Blockchain**.

5.1.1.1 Downloading and processing the packages information

ArchLinux packages Git repository contains all the changes in every distribution package. For every change, there is a new commit. A single commit may also hold modifications for different packages. Within the package recipe, there are, among other things, the package name and version (as demonstrated on **List. 2 line 1-2**). There is no package index; instead, there is the repository history.

By enumerating all the commits from the package repository, it was possible to parse the changes and identify the history of each package for later inclusion in a Blockchain. A Python script was created to iterate over all repository commits to understand the differences in each package's PKGBUILD. The package recipe was parsed for every new package change, and the 'version' was saved and indexed by the change date.

Initially, the package parser used Bash (92), recognizing the values for the variables

"`pkgver`" and "`pkgname`". However, as part of the evolution of the PKGBUILD file, the support for considering variables external to the PKDBUILD file and compositions of variables made the Bash parser inconsistent. This led to the creation of a dedicated parser to normalize the data, targeting the package's name and data version. When applicable, the distribution's name was set to "archlinux". Once all the package changes were identified and normalized, the information was indexed in files given the package release date.

5.1.1.2 Constructing the Blockchain

The simulation considers the package publication date to reconstruct the history as if the package were being published on the Blockchain given its original publication time. Fast forwarding the publication of all packages considering the package index built on the first step. The consensus, the target of the simulation, is achieved by the end of each block, considering the rules stated in **Section 4.2** based on an educated guess on the number of downloads.

A total of 76 trails were added to the simulation. Four of those are more relevant for our tests: **archlinux**, **pypy**, **perl**, **ruby**. The complete list of trails used on the simulation is available at **Table 7**. To decide which package goes to which trail, there is a regular expression (also listed on **Table 7**) for each trail. If the regular expression matches the package name, the package is considered part of the trail. Notice that a package can be vouched by more than one trail.

The vouch action is, naturally, placed after the addition of each package. During the simulation, the vouch action was programmed to happen from the following block to four blocks ahead; The decision of when to vouch for respects the probability stated on **Table 8**. That is meant to represent a real-world scenario where vouching for packages necessarily occurs after the package publication, but not necessarily immediately after the publication. This is due to different reasons, including possible network failures. The values on **Table 8** were arbitrarily chosen based on simplifying a Fibonacci sequence (93). The vouch timing will implicate when the package is available to the user, a critical metric, mainly for security updates. Therefore, this makes it extremely important to be represented.

The genesis block for the Blockchain was created using a timestamp 40 minutes before the first package publication, giving the simulation the necessary time to create the ArchLinux

Table 7 – Regular expression is used to check whether a package belongs to a given trail. (a) Trails with custom regular expressions. (b) Trails where the regular expression consists in the first four digits of the trail name.

(a)

Trail with specific regex - Trail name (Regex)			
archlinux (.*)	perl (perl)	pypy (py)	ruby (rb)

(b)

Other Trails - Regex are the first four digits			
ALTLinux	Ark Linux	BasicLinux	CentOS
Conectiva	Debian	Devil-Linux	Dyne:Bolic
Feather	Freesco	Frugalware	Gentoo
IPCop	Kanotix	Knoppix	Linux Scratch
Lycoris	Manjaro	Pardus	PHLAK
Puppy Linux	SLAX	Source Mage	SuSE
Turkix	Univention Corp	Whitebox Linux	Amigo Linux
BackTrack	BeatrIX	ClusterKnoppix	CRUX
DamnSmallLinux	DragOnLinux	Elive	Fedora
Freespire	G2Linx	Goodgoat	IpodLinux
Kate OS	Kubuntu	Lunar Linux	Mandriva
MEPIS	PCLinuxOS	PocketLinux	Red Hat
SmoothWall	Sun JDS	SystemRescue	Ubuntu Linux
VectorLinux	Yellow Dog	BioKnoppix	Zenwalk Linux
Floppix	Gnoppix	Morphix	Kurumin
Red Hat Ent	TopologiLinux	Yoper	BLAG
DeLi Linux	Foresight	GoboLinux	Linspire
muLinux	Slackware	TurboLinux	Xandros

Source: Elaborated by the author (2022)

trail by following the trail creation steps (**Section 4.2.1**). The interval between the block generation was chosen to be 20 minutes. Reducing the block publication value can cause the Blockchain to produce too many blocks, increasing the disk space necessary for validation and general overhead data. Increasing the publication period affects the timing of when the user will have the package availability and the amount of data held on each block, making the Merkle tree proof bigger. This tradeoff can be observed on **Fig. 16 and 19**.

The educated values assumed for confirmed download numbers were independent for the four most relevant trails, illustrated on **Table 9**. Those numbers mimic what was observed in real-world scenarios.

Atop the expected package data added to each block, the simulation also adds meta-data information to understand how the consensus was achieved, allowing the debugging and understanding of each block consensus. An illustration of a block is available on **List. 3**.

Table 8 – Probability used in the simulation to perform the vouching for a given package.

Block	Prob.
N + 1	60%
N + 2	20%
N + 3	10%
N + 4	10%

Source: Elaborated by the author (2022)

Table 9 – Range utilized to simulate the number of confirmed downloads.

Trail	archlinux	pypy	perl	ruby	others
Minimum	200000	10000	100000	100000	0
Maximum	300000	400000	100500	100900	100000

Source: Elaborated by the author (2022)

Listing 3 – Example of a Blockchain block inside the simulator.

```
{
    "forger": "Poppy",
    "metadata": {
        "amount_of_packages": 1,
        "amount_of_valid_trails": 4,
        "everybody_that_can_forge_this_block": [
            { "popularity": 38.603180292110558, "trails": ["pypy"]     , "user": "Ava"    },
            { "popularity": 36.285204258730523, "trails": ["archlinux"], "user": "Poppy"  },
            { "popularity": 13.269007771346679, "trails": ["Perl"]     , "user": "Leo"    },
            { "popularity": 11.842607677812248, "trails": ["fal"]      , "user": "Amelia" }
        ],
        "popularity_at_generation": [
            { "name": "fal"      , "pop":  5.2051448090322765 },
            { "name": "archlinux", "pop": 39.2003394897262822 },
            { "name": "Perl"     , "pop": 15.8103337175138812 },
            { "name": "pypy"     , "pop": 39.7841819837275565 }
        ]
    },
    "number": 150,
    "packages": [
        {
            "package": {
                "arch": ["i686", "x86_64"],
                "depends": "python",
                "license": ["GPL"],
                "md5sums": "1af233c6fa0a68851bc6155b2f563c30",
                "name": "bzr",
                "parser": "regexp v1.0",
                "pkgrel": 1,
                "pkgver": "1.3",
                "pkgdesc": ["A decentralized revision control system"],
                "source": "http://bazaar-vcs.org/releases/src/bzr-$pkgver.tar.gz",
                "url": "http://www.bazaar-vcs.org"
```

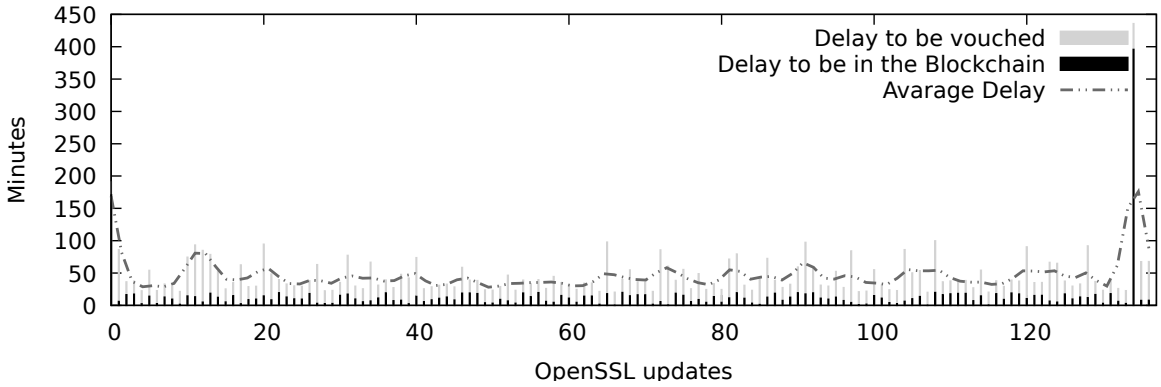
```
            },
            "publisher": {"name": "Pammi", "signature": "signature goes here"}
        }
    ],
    "popularity": [
        { "name": "fal"      , "pop": 11.842607677812248 },
        { "name": "archlinux", "pop": 36.285204258730523 },
        { "name": "Perl"     , "pop": 13.269007771346679 },
        { "name": "pypy"     , "pop": 38.603180292110558 }
    ],
    "trails": []
}
```

Source: Elaborated by the author (2022)

5.1.2 Results

The simulation of the Blockchain was fundamental to validate the proposal of the Blockchain mechanics (**Section 4.2**) and Proof-of-Download (**Chapter 3**). Within the simulator, it is possible to validate the score on the distribution popularity at any time. It is also possible to verify that all packages for ArchLinux were correctly added to its trail. As expected, some packages have a significant delay in publishing due to limitations on the number of packages per block. **Fig. 16** shows the delay on the OpenSSL publication.

Figure 16 – Delay in having the OpenSSL package available to trail users. In the chart, it is possible to observe the delay for the OpenSSL package to be available in the Blockchain and the delay for the package to be vouched and finally available to the user. The average delay is also demonstrated.

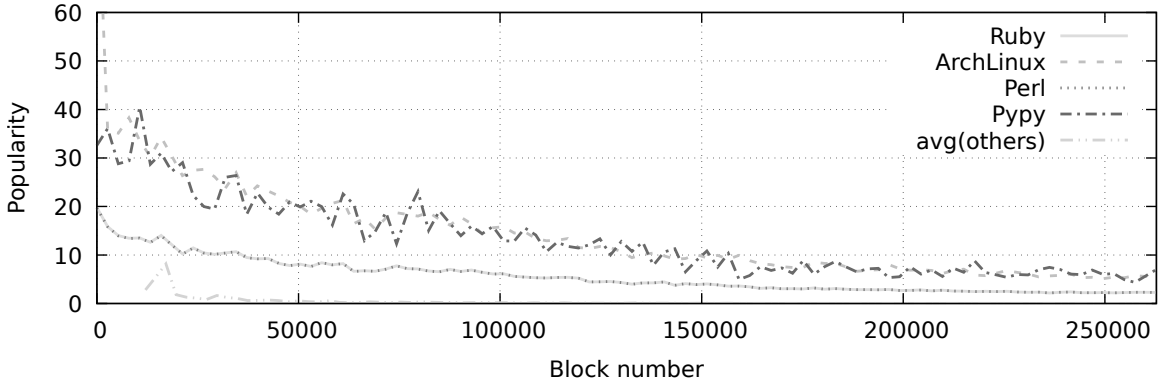


Source: Elaborated by the author (2022)

On **Fig. 17.**, it is possible to find different trails' reputations during the block generation. The reputation is consistent, confirming no chance of unpopular trails generating the blocks.

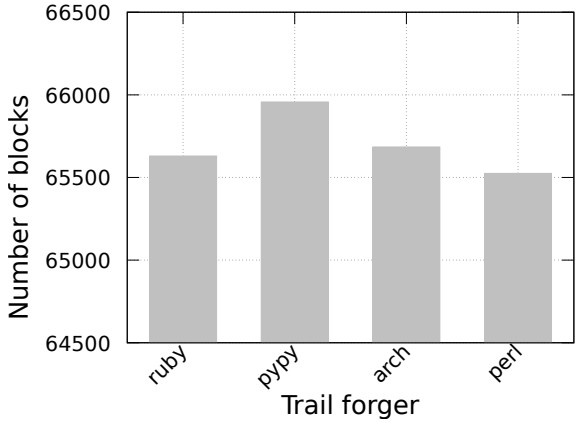
This mitigates the possibility of a malicious user creating a new trail to somehow tamper with block forgery.

Figure 17 – The popularity of the trails along the block generation. The chart highlighted the popularity of the trails: Ruby, ArchLinux, Perl, and PyPi. The average for all other trails is also shown.



Source: Elaborated by the author (2022)

Figure 18 – The number of Blocks forged by trail. In the chart, it is possible to notice that only the four most popular trails forged blocks.

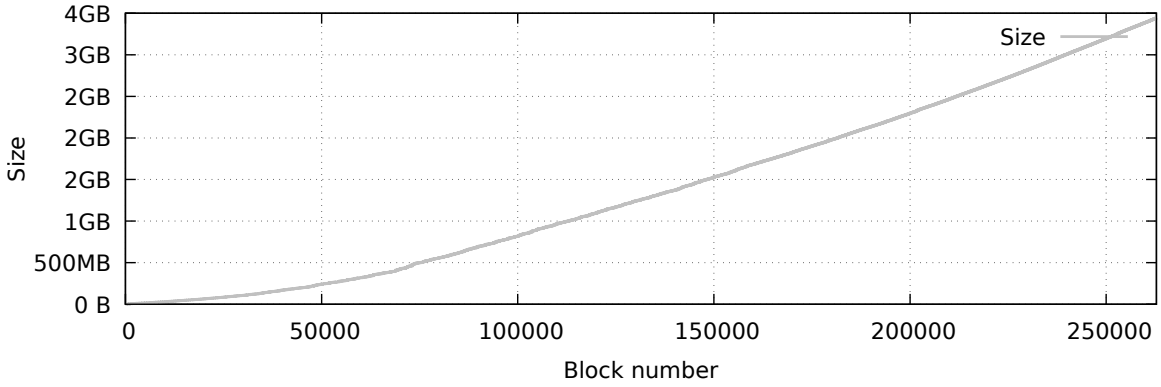


Source: Elaborated by the author (2022)

It is also possible to notice that distributions with less popularity did not manage to forge any package, showing the infeasibility to create new trails to tamper with the Blockchain. **Fig. 18** summarizes all forge blocks.

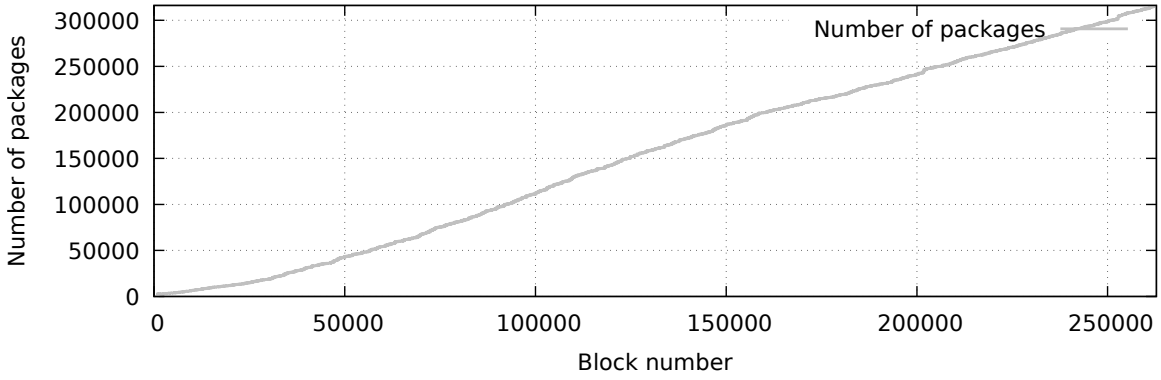
The size of the Blockchain structure was not expressive, given the amount of data stored. In a production environment, the JSON format could be replaced with more optimal data. JSON was utilized in the simulation for easy readability. **Fig. 19** illustrates the size of the Blockchain increasing with each block.

Figure 19 – Blockchain size by block. In the chart, it is possible to observe the storage size of the Blockchain versus the block height.



Source: Elaborated by the author (2022)

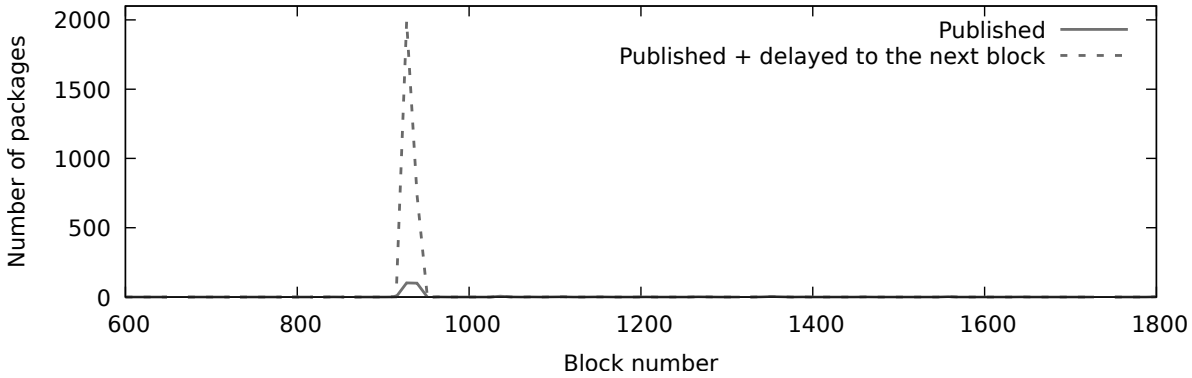
Figure 20 – Number of packages in the Blockchain. The chart represents the increase in the number of packages per block.



Source: Elaborated by the author (2022)

At block 923 (**Fig. 21**), one can observe the first time that the number of published packages hits the block limit. It is important to notice that the most popular users (given that they are members of popular trails) got their packages published first. The others got the packages published on best effort on the upcoming blocks. **Fig. 20** presents the number of packages on the Blockchain.

Figure 21 – Number of packages published per block (From 600 to 1800). This chart illustrates the number of packages published between blocks 600 and 1800, where a peak could be observed. This peak represents a considerable number of packages that got published simultaneously by the ArchLinux distro.



Source: Elaborated by the author (2022)

5.1.3 Threat model

The Blockchain proposed in this work guarantees that the user will download the package vouched for the distribution that he/she chose to trust. An adversary will not be able to poison the Blockchain with “trusted" packages that could contain malicious software. To ensure that, the Blockchain has the following features:

(a) All published `packages` are digitally signed.

(b) All the vouch actions are digitally signed, ensuring: Authentication, Integrity, and Non-repudiation.

(c) The addition or removal of users as members of the trails is also based on digital signatures.

(d) The block is also signed with the forgery digital signature.

(e) The forgery is chosen upon popularity (as demonstrated in the tests). Therefore, keeping the most interested in the correctness of the Blockchain is responsible for generating the next block.

(f) Popularity is granted to verified downloads only.

5.1.3.1 Creating the most popular trail

It is feasible to create a new trail that will be more popular than the authentic ones. However, to do that, it will be necessary to have a more significant amount of downloads than the authentic ones for quite some time. Thus, the effort to tamper with the block generation increases proportional to the importance of the authentic trails, given their popularity.

5.1.3.2 Certificate hijacking

A revoke mechanism still needs to be proposed. In the case of a digital certificate hijack, the attacker can impersonate a trail member.

5.1.3.3 Provider and the user on the same computer

If the provider and user are on the same computer, the download could be placed effortlessly or it may not be necessary to download at all. This use case leads to a scenario in which a malicious increase in the number of downloads is possible. This edge case was not considered during the simulation. In production environments, it is regarded the possibility of having the challenges placed by a trusted peer (member of a popular trail). Nevertheless, this user shall not deliver more challenges than expected, otherwise abused. The parameters on the number of challenges per peer and other calibration that may seem necessary were not the target of this simulation nor work, thus shall be treated in future work.

5.2 Second Test: Evaluating Blockchain and the Forger Party

This test aims to understand the healthy and secure function of the forger party on how resilient it is considering elements such as faulty networks and the presence of bad actors.

Nevertheless, it is also expected to understand the capability of the Blockchain to hold the number of packages proposed in the test.

The test now presented consists of adding a set of nodes acting on the construction (bootstrap) and maintenance/update of the Blockchain, mimicking the packages distributed on the PyPi repository. The test also counts w845 BRADLEY AVE UNIT 1845 BRADLEY AVE UNIT 1ith the figure of the impostor nodes. The impostors have no goal other than to play as adversaries, attempting to subvert the Blockchain.

Unlike the simulation shown on **Section 5.1**, the simulation here disregards the popularity ratio in favor of a fixed stake, where the forger party is placed under constrained timing. Here, the aspects of the Blockchain mechanics (**Section 4.2**) are also disregarded in favor of a monolithic Blockchain, serving only the purpose of PyPi data. With that, we expect to prove the consistency of the forger party being tested on a production-like p2p environment.

At the very least, it is expected that the Experimental Blockchain will have all the packages published on the PyPi repository, in addition to any other package added by the publisher, straight on the Blockchain.

The Blockchain simulation was planned to count on having twelve different nodes. Those nodes fell under categories depending on their responsibility on the Blockchain. The categories are described below.

- **Whistleblower**: The main responsibility of the whistleblower is to announce on the Blockchain new packages publications. This is done either by going over the publication history of the PyPi repository or by tracking new publications. Node type is defined in **Table 4-iv**.
- **Forger**: The forger nodes are the ones with stakes to publish blocks. Those will compete on the forger party (**Section 4.7.1**) to release new blocks. Node type is defined in **Table 4-v**.
- **Impostor**: Adversary nodes trying to subvert the Blockchain. Detailed in **Table 11**. Node type is defined in **Table 4-iv**.
- **Listen**: The listen nodes are listening to the p2p network, observing packages/block publication to feed their database. The Listen database will later be used to verify the consistency of the package's availability. Node type is defined in **Table 4-i,ii,iii and iv**.

Each category had a different number of living nodes: four nodes for whistleblowers, three nodes for forgers, three for impostors, and finally, two listing nodes. The nodes were distributed in different geographical locations to simulate a real-world scenario where network latency pays a difference. The nodes compete with each other on the package and block publication. The nodes have received nicknames, intending to generate easy-to-read statistics, as illustrated on **Table 10**. Also, on **Table 10** the geographical position of the nodes is revealed.[1]

Table 10 – Details on each of the nodes used on the Blockchain evaluation.

#	Name	Region	Description
Whistleb.	Amy	UK-I	Publish new packages
	Hermes	US	
	Kif	Canada	
	Leela	UK	
Forgers	Cubert	UK	Blocks Forgers
	Nibbler	US	
	Heber	Canada	
Impostor	Zapp	UK	Submit packages with different authorship
	Zoidberg	US	Submit packages with wrong signature
	Mon	Canada	Delay secure updates
List.	Scruffy	UK	Watch and keep package publication
	Morbo	Canada	

Source: Elaborated by the author (2022)

The whistleblower nodes trust each other. They also named the other three whistleblowers as trusted publishers for a given package on every initial publication. That way, they will also compete to publish further package updates.

Under the simulation, the "fixed" stakes are divided equally between the tree forgers: 333 for each forger. Nibbler is pre-selected to be the forger for the genesis block. The subsequent blocks were the product of the forger party exactly how it is described on **Section 4.7.1**. The source for the construction and test of the Blockchain came from two distinct places: the package history and the continuous updates.

5.2.1 Feeding the Blockchain

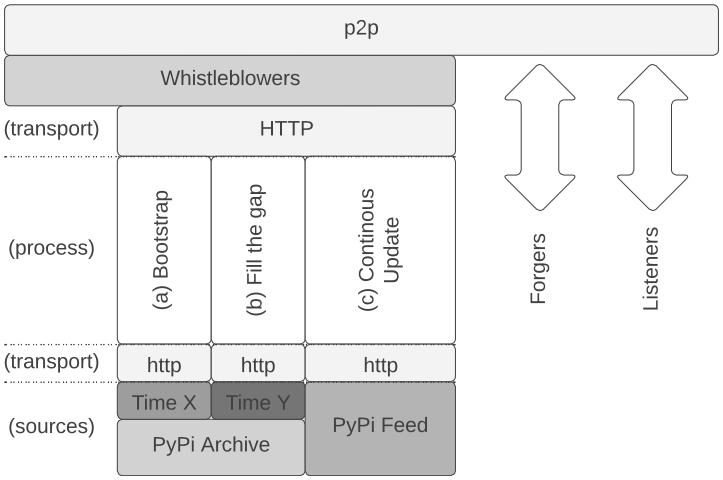
The mechanism to feed the Blockchain was based on a producer-consumer pipe, whereas items got to be removed from the pipe whenever confirmed to be processed by a node, allowing

[1] Any resemblance of any of the names of the nodes to any TV series is mere coincidence.

the node competition. The consumers are the network nodes (whistleblowers), listening to broadcast messages on the p2p network and responding accordingly. The forger considers the package information that came with the first whistleblower. Naturally, packages are validated before getting accepted.

The **Fig. 22** demonstrate the dispositions of the producers and consumers in this Blockchain feeding process. The producers are divided into three groups based on the source of information. The first group is the one that enumerates the data from the PyPi history (bootstrap). The second was considering the continuous update of the packages. Finally, a third process fills the gap for the interval between the PyPi snap shoot to the moment the Blockchain started to get the continuous update.

Figure 22 – Pipeline for feeding the Blockchain with the different processes: (a) bootstrap, (b) sync process, and (c) continuous update.



Source: Elaborated by the author (2022)

5.2.1.1 Bootstrap: Feeding the Blockchain with PyPi data

All the data from the PyPi repository are available on their WEB API. However, to avoid abuse on servers (that are supported by donations), there is some restriction on the number of parallel downloads, such as download frequency. Considering the limitation and volume of data, scrapping all data was challenging.

The PyPi API counts with a package index to indicate every package published on PyPI, presented in a JSON file format, as illustrated on **List. 4**. The scrapping process downloads all

package information, saving a JSON file for every package. We applied normalization to save packages with names that could fit in the used file system (ext4) (94). The process to download the package information was gentle (respecting the API limits) and counted with four different crawlers, having a total processing time of 25 minutes and 12 seconds to download over 6.2G of data finishing the snapshot on May 06, 2022, 14:10:19 GMT-4.

Listing 4 – PyPi JSON that describes the package modsecurity. The API endpoint was:

`https://pypi.org/pypi/package_name/json`


```
{    "info": {
        "author": "Cyril Jouve",
        "author_email": "jv.cyril@gmail.com",
        "bugtrack_url": null,
        "classifiers": [
            "License :: OSI Approved :: GNU General Public License v3 or later (GPLv3+)",
            "Programming Language :: Python :: 3", "Programming Language :: Python :: 3.10"        ],
        "description": "",
        "description_content_type": "",
        "docs_url": null,
        "download_url": "",
        "downloads": {"last_day": -1, "last_month": -1, "last_week": -1},
        "home_page": "",
        "keywords": "",
        "license": "GPL-3.0-or-later",
        "maintainer": "",
        "maintainer_email": "",
        "name": "ModSecurity",
        "package_url": "https://pypi.org/project/modsecurity/",
        "platform": "",
        "project_url": "https://pypi.org/project/modsecurity/",
        "project_urls": null,
        "release_url": "https://pypi.org/project/modsecurity/0.1.1/",
        "requires_dist": null,
        "requires_python": ">=3.9.7,<4.0",
        "summary": "",
        "version": "0.1.1",
        "yanked": false,
        "yanked_reason": null
    },    "last_serial": 12316778,
    "releases": {
        "0.1.0": [ ... ],
        "0.1.1": [
            {
                "comment_text": "",
                "digests": {
                    "md5": "9bc5e9f5050cb6e10a246d3810336bb8",
                    "sha256": "c6fe382081f19fb304edd54dd3408c0d1077e7b1c6c93dc9354213a5425be252"
                },
                "downloads": -1,
                "filename": "ModSecurity-0.1.1-cp310-cp310-musllinux_1_1_x86_64.whl",
                "has_sig": false,
                "md5_digest": "9bc5e9f5050cb6e10a246d3810336bb8",
                "package type": "bdist_wheel",
                "python_version": "cp310",
                "requires_python": ">=3.9.7,<4.0",
                "size": 4398269,
```

```
                "upload_time": "2021-12-15T21:43:26",
                "upload_time_iso_8601": "2021-12-15T21:43:26.223626Z",
                "URL": "https://files.pythonhosted.org/packages/9e/29/71
    ↪ fa9785783cf663d109706ea3b694ecdc395ed1838135d170bc161436f0/modsecurity-0.1.1-cp310-cp310-musllinux_1_1_x86_64.whl"
    ↪ ,
                "yanked": false,
                "yanked_reason": null            } ] },
    "URLs": [        {
            "comment_text": "",
            "digests": {
                "md5": "9bc5e9f5050cb6e10a246d3810336bb8",
                "sha256": "c6fe382081f19fb304edd54dd3408c0d1077e7b1c6c93dc9354213a5425be252"
            },
            "downloads": -1,
            "filename": "ModSecurity-0.1.1-cp310-cp310-musllinux_1_1_x86_64.whl",
            "has_sig": false,
            "md5_digest": "9bc5e9f5050cb6e10a246d3810336bb8",
            "package type": "bdist_wheel",
            "python_version": "cp310",
            "requires_python": ">=3.9.7,<4.0",
            "size": 4398269,
            "upload_time": "2021-12-15T21:43:26",
            "upload_time_iso_8601": "2021-12-15T21:43:26.223626Z",
            "URL": "https://files.pythonhosted.org/packages/9e/29/71
    ↪ fa9785783cf663d109706ea3b694ecdc395ed1838135d170bc161436f0/modsecurity-0.1.1-cp310-cp310-musllinux_1_1_x86_64.whl"
    ↪ ,
            "yanked": false,
            "yanked_reason": null        } ],
    "vulnerabilities": []
}
```


Source: Elaborated by the author (2022)

As the packages were already hosted in JSON format, no normalization was needed on the package data. However, the release date for each package was used to organize the data into a Blockchain structure. At first, the script identified the oldest package/release among the downloaded packages. The oldest package timestamp (`config` / Mar 21, 2005, 15:59:25) was used to create the genesis block. For every package release, the timestamp was computed to a block, respecting two hours between each block creation and the genesis block publication time. The script appended the package release information on a JSON file named after the block number.

Concerning that all package data was aligned with their respective blocks, a different process was used to broadcast the information to the whistleblowers. The whistleblowers collected the received package information and publishing as new packages for the attention of the forgers. Consequently, creating a snapshot of the PyPi database at a given time. Nevertheless,

it was still necessary to continue updating the Blockchain with newer package publications outside the package history. The block publishing interval of two hours was simulated during the simulation in about four seconds per block. On average, it took about four seconds per block, fast-forwarding the block publication to 3 days and 12 hours of execution. A total of 75062 blocks were published/used in the evaluation.

5.2.1.2 Continuing Update of the Packages Releases

Although bootstrapping the Blockchain was shown to be effective, it was somewhat abusive. As the resources for PyPi are limited, the continuous download of the entire PyPi database would be unfair. More elegantly, a continuous process spotting only the new updates would be a good fit; that is precisely what is provided by PyPi updates RSS feed (95). Available at `https://pypi.org/rss/updates.xml`, the PyPi update feed contains the most recently released packages.

Every two minutes, a script downloads PyPi's update feed and sent to the different whistleblowers in the same way that was done while the Blockchain was being bootstrapped. Those produce nodes are known by the whistleblower that, once authenticated, the data will proceed with the package announcement/broadcast to the forgers.

If a package is claimed - published by the author straight on the Blockchain - the whistleblower will not be able to update it unless the whistleblower is also marked as publisher by the original publisher. Packages already claimed can only be published by whoever claims them or is authorized by the original claimer. The network may count on various whistleblowers; it is all right for them to compete with each other on package publishing, but they must be trusted as package publishers. The block forger should avoid any conflict by picking the first package announced.

There is no guarantee on the update frequency of the PyPi feed. Therefore, a package may be published on PyPi but not immediately on the Blockchain. Furthermore, there is a time gap between when the repository snapshot was taken and when the Blockchain was ready to receive updates. A sync process fills that gap.

5.2.1.3 Sync: missing package verification

The verification to check if a package is missing on the Blockchain consists of downloading the packages from the PyPi database and comparing whether they are present or not on the Blockchain. If not present, the packages are added via whistleblower altogether with the packages for the next block. In this case, it is expected a delay. The package will only appear in a later block, therefore being late to be available to the user. The best effort is used to add the packages to the Blockchain. It is guaranteed that every package will be published in the long run. Eventually, packages may not be published on the immediate next block.

This third process runs with an extended interval to avoid draining the resources from PyPi servers. Hence the need for three different processes: bootstrap, continuing updates, and verification.

5.2.2 The Whistleblower

Apart from listening to the network for updates, the whistleblower is also responsible for parsing the package JSON, downloading every package file, and providing a copy on an IPFS node. Once the CID of the files are generated, the whistleblower adds an extra entry to JSON, mapping the URI with the CID content. In a way, the user was provided with both options: regular download or download via IPFS whenever available. This flexibility was necessary to reduce the amount of space necessary for storing all the package files during the simulation. An example of files to be inserted on the IPFS can be found in **List. 4, lines 53 and 76**.

The whistleblower must validate that package information is coming from a trusted source. If information is not authenticated or corrupted, it must be discarded. Likewise, every package publication sent by the whistleblower shall be digitally signed, otherwise discarded by forgers.

5.2.3 The Listening Nodes

The listening nodes are responsible for creating a parallel database containing the information on all the packages by listing to block publications. This information is later used to compare the data published on the PyPi network to that we published on the Blockchain.

Additionally, the listening nodes will also be providing search functionality and Blockchain mirror on the p2p network **Table 4-i,ii,iii and iv**.

Important to note that these nodes are unaware of the Blockchain construction stage. They are meant to be executed at the beginning of the Blockchain creation process and kept alive until the end of the simulation. Those nodes are creating their local database to facilitate the statistics generation. That will later be used to understand the package publication timing.

5.2.4 Bad actors: The adversary nodes

The threat model is hereby defined by three actors: Zapp, Zoidberg, and Mon. Those three nodes are responsible for behaving lousily in different manners, as described in **Table 11**.

Table 11 – The tree adversary nodes used in the Blockchain simulation followed by each attack description.

Node	Description
Zapp	Re-assemble package publication having its name as authorship. The signature matches, but the whistleblower must prevail.
Zoidberg	Publishes valid packages but posing as a whistleblower. The signature will not match.
Mon	Flood publication of old packages, distracting the forger.

Source: Elaborated by the author (2022)

Every 10 seconds, the Zapp node retrieved the first package published on the p2p network, re-branding the signature as he had published the package (regardless of the original publisher). Zapp is not trusted as the publisher of any package, nor is it recognized as a whistleblower. In the same 10 seconds, Zoidberg published a package with an invalid signature. In the Zoidberg attack, the original author's signature was changed in the first byte, leading to a corrupt signature. Mon's attack was the most relevant, as the node was publishing old packages (at most 100 randomly selected packages from previous blocks) with valid signatures, leading the forger to look up if the packages had already been published.

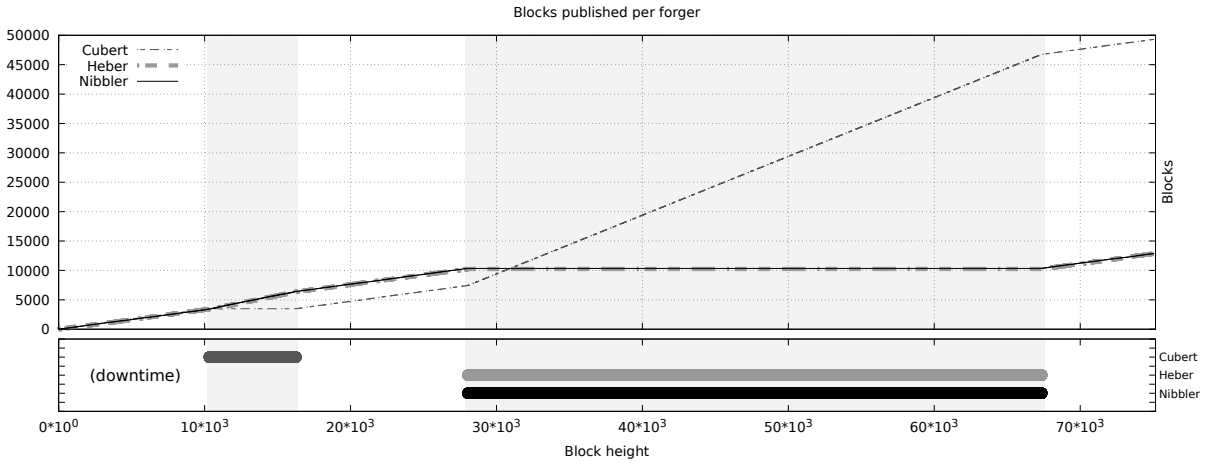
5.2.5 Results

The presence of the impostor nodes was recognized as a great value in the development of the test, as it surfaced implementation errors otherwise not noticed. The impostors helped

to understand that the signature validation made at the package announcement arrival is less memory intensive. Thus, making the forger nodes less prone to DoS attacks for invalid signatures or “fake whistleblowers”.

The forger implementation had to be changed several times for package lookup optimization. In the end, a parallel GDBM.[2] Before the GDBM adoption, Mon’s attacks led to CPU spikes due to intensive data lookups.

Figure 23 – Number of times a forger got selected in the forger party versus the block height. The chart represents the number of blocks forged by each forger in the test: Cubert, Heber, Nibbler. The node’s downtime (not participating in the forger party) is highlighted.



Source: Elaborated by the author (2022)

The problems that became evident in the simulation are unlikely to happen in a real-world scenario. The four seconds per block on simulation leads to nodes’ CPU utilization of 100% per almost all the simulation, something that would not happen in sparse intervals such as two hours per block. The cited problems would not represent a threat. Nevertheless, apart from the possible collateral damage, the attacks proved to fail in their primary objective. As shown on **Table 12**, the Zapp node did not manage to publish any package. Also, no package was observed to be published with an invalid signature. Ultimately, Mon’s attack did not manage to fake any package version.

The semi-random selection for block forgers proved to be even, as the distribution seems not to favor any particular node, as demonstrated on **Fig. 23**. Network issues and power outages

[2] GNU dbm (or GDBM, for short) is a library of database functions that use extensible hashing and work similarly to the standard UNIX dbm.

Table 12 – Number of packages published by each whistleblower (or bad actor).

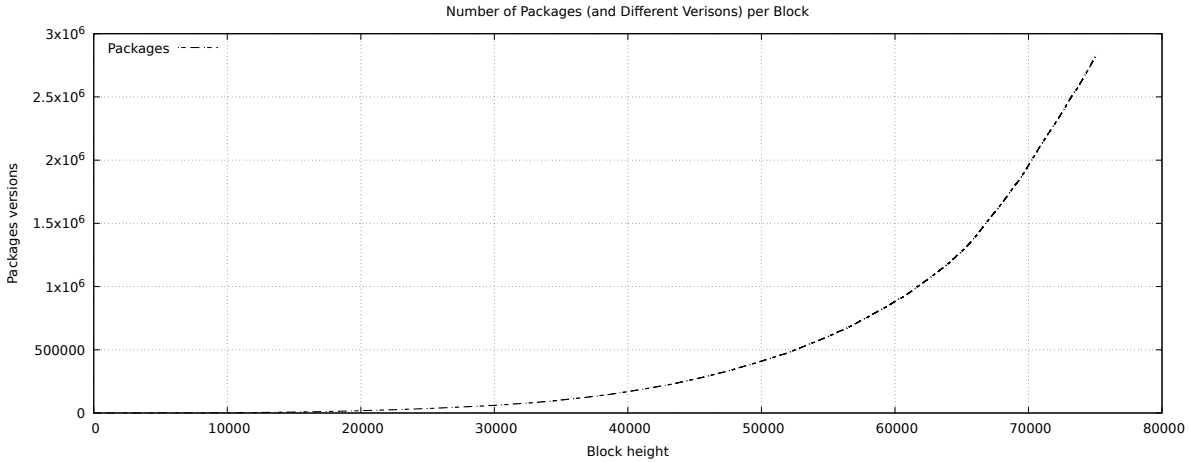
Amy	Hermes	Kif	Leela	Zapp	Total
701026	688197	689855	748870	0	2827948
24.79%	24.34%	24.39%	26.48%	0%	

Source: Elaborated by the author (2022)

led to downtime for the nodes, Heber and Nibbler. As expected, Heber and Nibbler's downtime did not affect the well functioning of the Blockchain; Cubert was publishing blocks without damage.

Finally, the number of packages encountered in the PyPI repository matches the number of packages published in the Blockchain. There are no divergences in the package publications. We observe that 2,827,948 packages (or new package versions) were published on the snapshot taken on May 24th, 2022. **Fig. 24** illustrates the growth of the package publication on PyPi, consequently, the growth of packages published on the Blockchain.

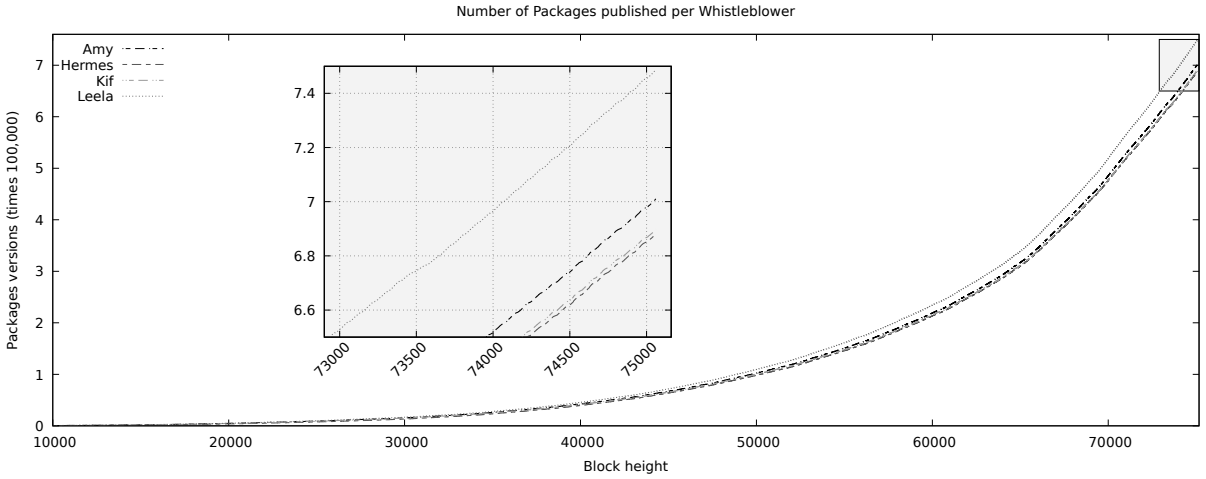
Figure 24 – Number of packages and new versions published along the block height. In this chart, it is possible to observe the number of new packages held by the Blockchain at every height.



Source: Elaborated by the author (2022)

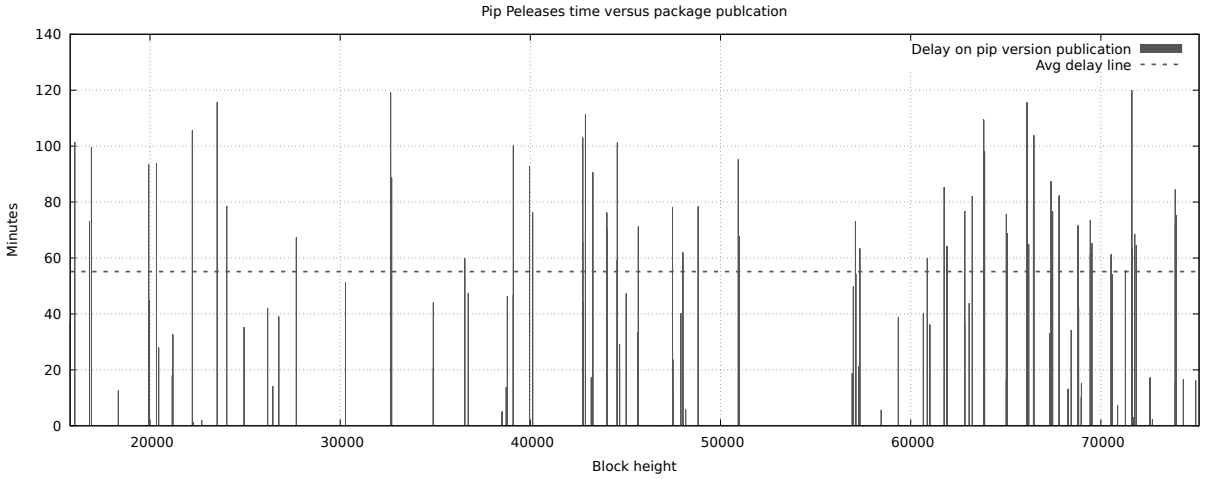
The competition among the whistleblowers did not seem to favor any particular node, as all nodes seem to have a very similar amount of packages published. In the **Fig. 25** it is possible to understand the amount of package publication among the four different whistleblowers. Towards the end of the Blockchain, Leela has a slight discrepancy in packages published. The difference seems to increase after block 70,000, where Leela had less latency to reach forgers Herbe and Nibbler. However, the difference is negligible, given the number of published packages.

Figure 25 – Amount of new packages published per whistleblower. In the chart, it is possible to observe the number of packages published by each of the whistleblowers used in the evaluation: Amy, Hermes, Kif, Leela. Highlighted the interval between blocks 73000 and 75000



Source: Elaborated by the author (2022)

Figure 26 – Delay on Pip releases concerning the correspondent block publishing. The average time is highlighted in the dashed line. The delay illustrated in the chart represents the time difference between the package submission on the Blockchain and the package publication conditioned to the Block publication.

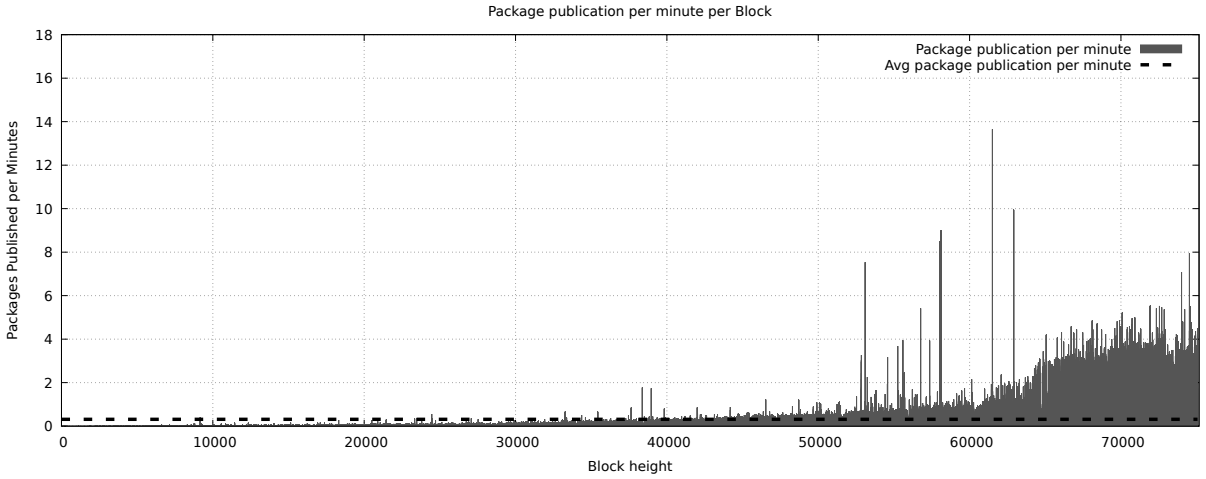


Source: Elaborated by the author (2022)

The `pip` package was chosen as an example to understand the amount of time between the package publication (by the whistleblower) and the package availability to the final user. The Blockchain associates the package available to the block publication (approximately every two hours). **Fig. 26** illustrates all `pip` releases together with the delay between the package and the block publication. Notice the average delay of 55 minutes. An excellent time compared with

the mirrors practice, where, in practice, servers are “synced” during the night for easy network traffic.

Figure 27 – Average on package publication per minute on every block. The chart represents the capacity of package publication on the Blockchain using Transactions Per Second (TPS) metric.



Source: Elaborated by the author (2022)

The TPS usually seems to be a critical factor on every Blockchain. In this test, the TPS was considered low. **Fig. 27** illustrates the average package publication per minute on every block. The highest amount of publication was given at block 61507, where on average, 13.65 packages were published per minute.

6 Software Contributions

This chapter introduces the Capivara package manager and the Web API and command line utilities that make part of the ideas fomented during the execution of this work available to the PyPi community. Capivara is a utility initially used as an experiment that later became a feature-rich package repository over Blockchain. In this chapter, we also describe the PyPi Blocks website.

6.1 Capivara: The Package Manager

The Capivara implementation was used to validate the theory described in this document. Initially, it was meant to be a simulation, but it was later made into a production tool serving packages to the community. The Capivara package manager aims to provide the same functionality as Python Pip, but over a Blockchain. Currently, it only supports Python/PyPi. However, extending it to support other script languages or any package distribution is feasible.

The first step towards mimicking Pip's behavior inside the Blockchain was to nourish the Blockchain with all the information already available on the PyPi databases. All the details on the process to feed the Blockchain are described in **Section 5.2.1**.

6.1.1 The PyPi Blocks API

To facilitate the understanding of the Blockchain, allowing debugging the p2p interactions, an HTTP RESTful API (96) was created. This API is meant to export all the functionalities of the p2p network. Later, this API was also used by nodes with network restrictions (**Section 2.4**). The API implementation as its specifications is publicly available on the Capivara website (97). Any node may as well deploy and provide its API version as the code is available on GitHub. Furthermore, this API is used to feed information on the block explorer on the Capivara website.

Among other elements, the API provides: block data retrieve, block header retrieve, block package summary, block stake summary, package data retrieve, and storage retrieve.

6.1.1.1 Block Data retrieve

Delivery of all data related to a given block. Including individual information on every release and meta-data. The format and details are illustrated on **List. 7**. Notice the particular endpoint called *latest*, which presents the latest published block.

API endpoint: GET /API/v1/block/[block number]
GET /API/v1/block/latest

6.1.1.2 Block header retrieve

It is similar to block data retrieval, except that it only returns the block header. The example response is listed on **List. 9**.

API endpoint: GET /API/v1/block/[block number]/header
GET /API/v1/block/latest/header

6.1.1.3 Block package summary

Provides the CID for the summary of every latest version of the published packages by block number. That is the list of packages meant to be used on the package search (**Section 4.1.1**). The example response is listed on **List. 5**, the example of a summary file is illustrated on **List. B**.

API endpoint: GET /API/v1/block/[block number]/summary
GET /API/v1/block/latest/summary

6.1.1.4 Block forger summary

Provides the CID for the summary of the possible forgers list by block number. That is the list of the forgers by block. The example response is listed on **List. 6**. Notice that this version is not yet using the PoDl. Here the stacks are used to select the eligible forgers. As described is **Section 5.2**. An exemple of a list of forgers can be found on **List. 12**.

API endpoint: GET /API/v1/block/[block number]/forgers
GET /API/v1/block/latest/forgers

6.1.1.5 Package data retrieve

Provides all information related to a given package. It includes all released versions, meta-data information, and proof for each piece of information. The example response is listed on **List. 8**.

API endpoint: GET GET/API/v1/package/[package name]

6.1.2 Capi website - PyPi Blocks site

Capi's website has multiple objectives, including informing the user of the technical details behind Capivara Blockchain and providing a block explorer where users can navigate the blocks and published packages. The website also serves as a gateway for storage retrieval (**Section 2.4**). Finally, the site has a statistics section where the users can find data on the Blockchain and package publishing.

6.1.2.1 Technical Background

One of the aspects of the project is to incite discussion on possible features and general improvements. The website provides a basic understanding of Blockchain and the data structures used by Capivara Blockchain, enlightening new users to learn the basics and jump straight ahead on the discussion on GitHub.

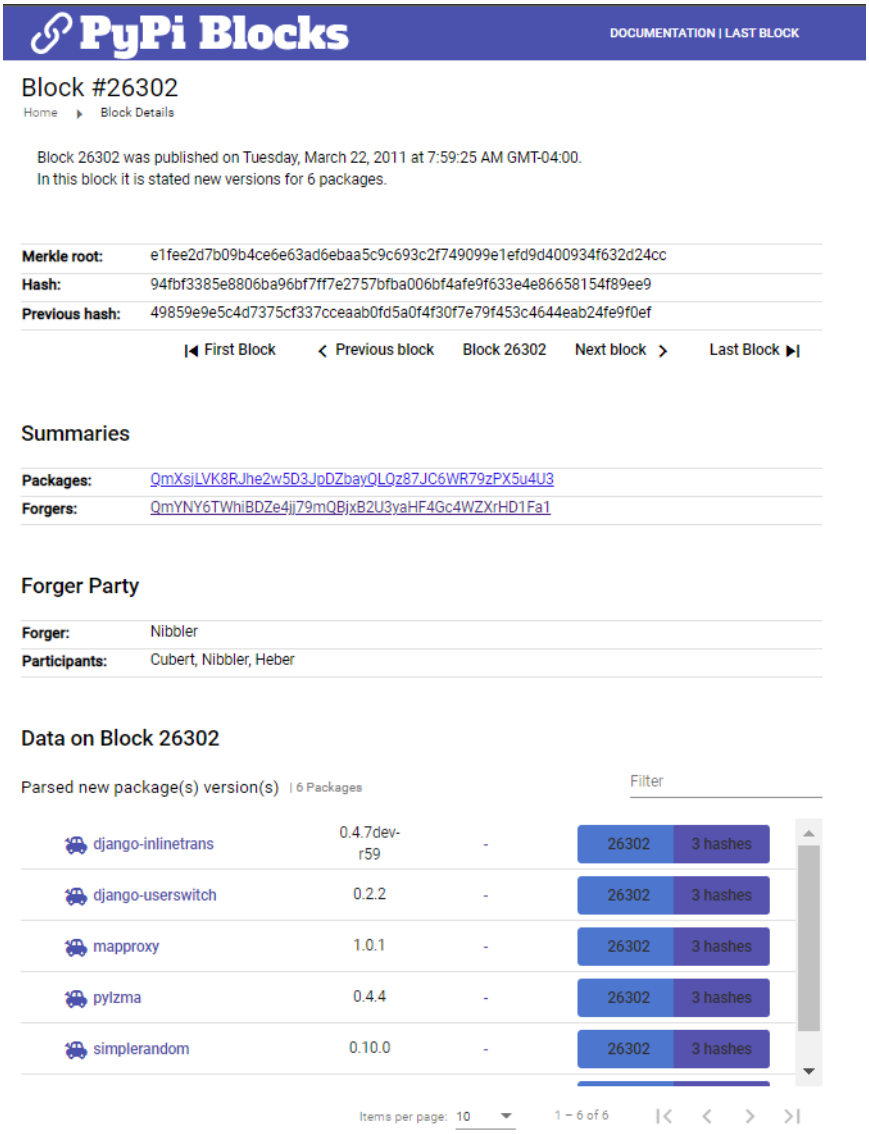
6.1.2.2 Blockchain explorer

As the name suggests, the Blockchain explorer provides ways for the user to navigate the Blockchain by either block or package name. The user can verify the data on every published package.

Fig. 28 shows a screenshot of the website where Block 71310 is displayed. Notice the list of packages published on the block. There is also the meta-data for the block, such as Merkle root and the previous block hash. Index by package is illustrated in **Fig. 29**, where the `heputils` package is listed. All versions of `heputils` are listed with the respective proof and storage

hash. An example of a proof is listed in **Fig. 29**. The figure shows a package meta-data together with its hash.

Figure 28 – Details on block number 26302. Block release on March 22, 2021 at 7:59:25 AM GMT-04:00, this block held 6 packages, as shown in the image.

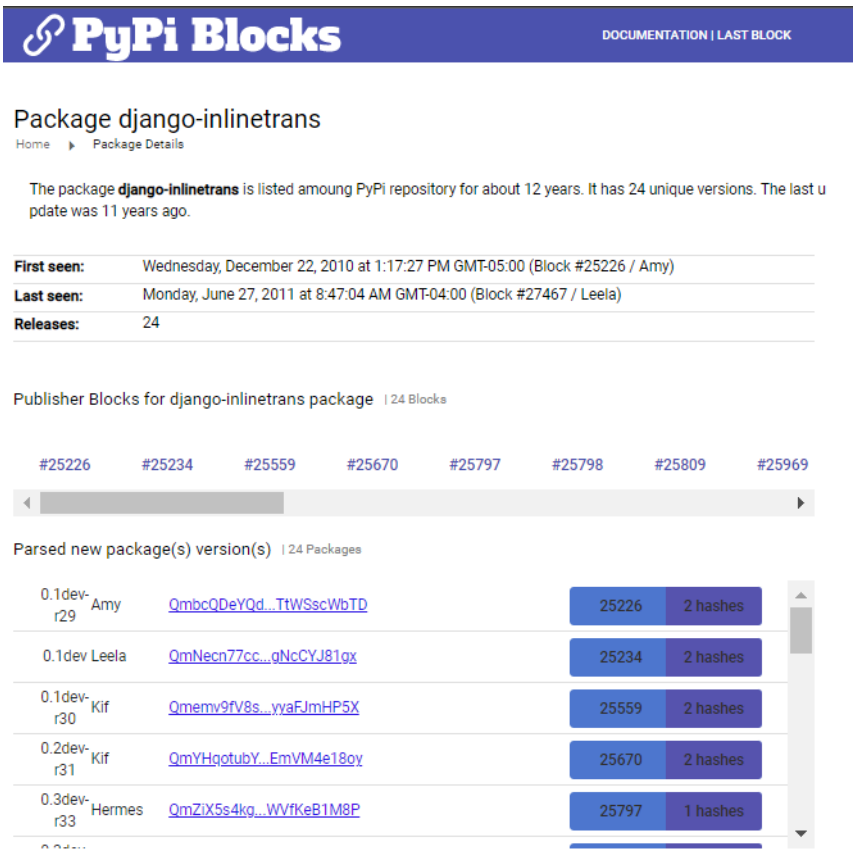


Source: Elaborated by the author (2022)

6.1.2.3 Storage retrieval

The storage retrieval is meant for checking only. Any CID can be extracted from the IPFS network. **Fig. 30** illustrates a IPFS client grabbing a CID for **django-inlinetrans** version 0.1dev-r24.

Figure 29 – Details on the package **django-inlinetrans**. This package has 24 versions published over 24 blocks, including: #25226, #25234, #25559, #25670, #25797, #25798, #25809 and #25969.



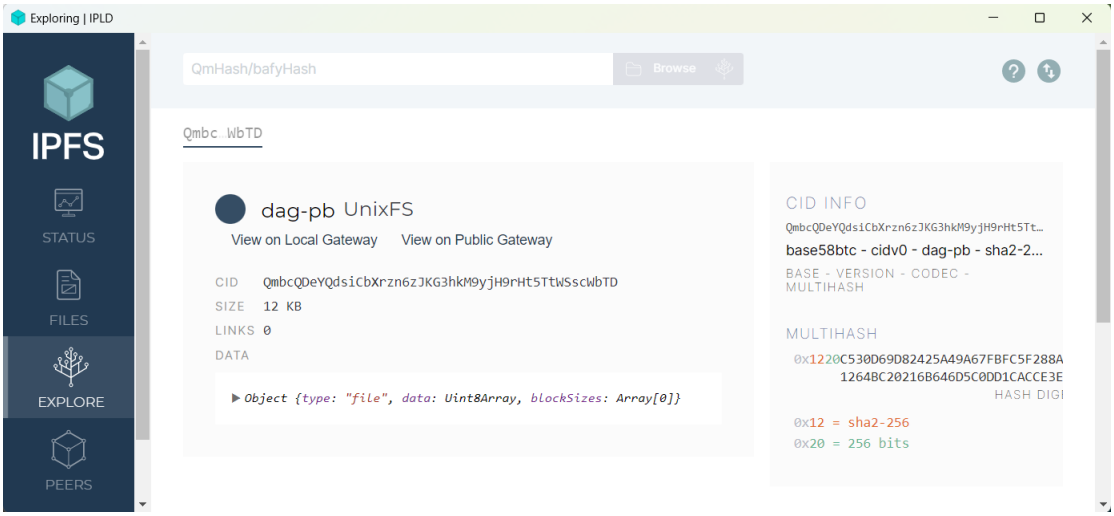
Source: Elaborated by the author (2022)

6.1.2.4 Statistics

This area is meant to allow users to track the health of the Blockchain. It also incentives discussion based on data captured live on the Blockchain. The chart includes information on the following.

- Proof size versus block publication;
- Amount of releases per block;
- Delay in block publication;
- Amount of packages per block.

Figure 30 – Example of data retrieval for the CID QmbcQDeYQdsiCbXrzn6zJKG3hkM9yjH9rHt5TtWSscWbTD on the IPFS client. On the screenshot there are all the details on the file storage.



Source: Elaborated by the author (2022)

6.1.2.5 Capi search on browser

The search on the browser is part of the website where a user cannot only search for packages but also share computer power, providing other users with search results as a node on the p2p network.

Once open, the site asks the browser to download the latest summary using the site API (**Section 6.1.1.3**). The summary is used to feed the local search on the browser, and the list is also used to address p2p requests. The browser connects to the p2p using libp2p javascript implementation (98), bootstrapping a node on the Blockchain p2p. The CPU sharing is light and should not drain the resources of the computer or the browser. Ultimately providing a full Search node (**Table 4-I**) .

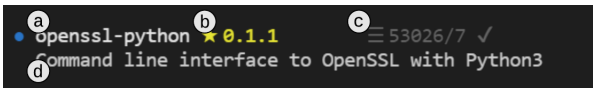
6.1.3 Capi command line utility

Capi is the command line utility for creating all nodes: search and download. Coded in *nodejs*, the application makes usage of libp2p (98) for establishing the p2p communication. Capi uses **pip** for package installation and version management. Capi does not manage package installation, only the downloads.

Package search via Blockchain introduces an extra step when identifying that the software

is verified to be part of the Blockchain. This verification is shown to the user in a User Interface element format. **Fig. 31** illustrates the output of a Capi search. Item **Fig. 31-c** presents the block number, the number of hashes in the proof, and lastly, the tick confirming that the information is real. Invalid or unreal search results are only shown in the **verbose** search output. At length, in non-verbose site search results, invalid search results are expected to be discarded without the user's knowledge.

Figure 31 – Package search results: (a) Package name; (b) Package version; (c) Block validation; (d) Package description.



Source: Elaborated by the author (2022)

7 Conclusion

In this work, we present a Blockchain without financial-based incentives, using distributed consensus based on the popularity of the participants. In the presented Blockchain, all the data used by the packages and Blockchain is publicly available for anyone to host, share, or consult. We also present a process for a semi-random forger selector. When combined, those features support anyone willing to share computing power, whether small or large, without friction, meeting the proposed objective of this work.

This proposal was evaluated under different aspects. On the first evalutation, it was observed that the Blockchain mechanics (**Section 4.2**) worked well while handling all of the packages for ArchLinux. The size of the Blockchain remained relatively small, considering that it held ten years of software package history from ArchLinux. Considering the example of the OpenSSL package, the delay in publications did not impede the adoption of the Blockchain mechanics. The limiting value on the Blocks for new trails was proved efficient, as it did not pose a limiting factor, and yet, prevented the Blockchain from indefinitely growing.

The equation for calculating popularity seems fair, as forgers got evenly elected to forge new blocks, and so is the download count. However, the massive adoption of PoDl will demand changes in the current available HTTP and IPFS servers and clients; As those amends eventually will be published as new versions, the adoption may not be immediate. Thus, compromising the adoption of Proof-of-Download (PoDl) in scale in the short term.

The second test shows that the Blockchain described is resilient to the presence of impostors and real-world network issues. None of the impostors were able to damage or delay the Blockchain operation. Still, the semi-random selection for block forgers was shown to be even. The whistleblower competition did not favor any particular node. Furthermore, given the monolithic nature of this test, the delay in package release was relatively small compared to test one.

Ultimately, the second test was able to provide evidence that untrustworthy peers could cooperate in the Blockchain with computer power, disk space, and bandwidth.

Ideally, test one and test two will cope with being fully operational Blockchain completely

autonomous for the adoption to whom may be interested. However, the adoption for HTTP clients and servers with support for PoDl as the adoption of IPFS implementation may pose a limiting factor. Therefore, for the adoption of the Blockchain, we suggest four milestones.

1. Having the Blockchain solely on Proof-of-Stake (PoS);
2. Adopting the Blockchain mechanics;
3. Adding the popularity factor computation;
4. Replacing Proof-of-Stake by Proof-of-Download.

Currently, the Capivara package manager is implementing the first milestone. Lastly, all materials used in this test, including the Capivara package manager, are published online (97). The results of this work have been published in two venues, detailed as follows.

- The proposal for the Blockchain using PoDl (**Section 3**) and the Blockchain mechanics (**Section 4.2**) have been published at the IEEE Blockchain Conference 2020 (99);
- The second test, which evaluates the distributed consensus (**Section 4.7**), the forger party (**Section 4.7.1**), and the package search (**Section 4.6**), has been published at the IEEE Access journal (100).

7.1 Future Work

We plan to continue studying how to apply Proof-of-Download on edge cases, such as when the challenger and challenged are hosted on a unique computer — mitigating the possibility of fake downloads. We also aim to continue evolving the Capivara package manager, overcoming the technical limitations, and expanding the third-party implementations to support our suggestion of improvements, ultimately deploying the four milestones cited in our conclusions.

In further work, we also plan to craft an test to test the package search (**Section 4.6**) over the p2p network, including testing with the presence of an impostor. We plan to consider in the threat model not only poisoned responses, but valid though outdated responses that may imply on a user installing a valid, but outdated (maybe even vulnerable) package or dependency.

References

[1] IPFS-DEVS. *CID INSPECTOR*. 2020. Accessed: 2022-04-15. Available from Internet: <https://cid.ipfs.io/#QmRNAMFjhAfmq8NT2zLbSeGh3cMoQHg5ZwStbqf74he9aP>.

[2] FERREIRA, G. et al. Containing malicious package updates in npm with a lightweight permission system. *IEEEACM International Conference on Software Engineering (ICSE)*, p. 1334–1346, 2021.

[3] ABATE, P. et al. Mpm: A modular package manager. *CompArch'11 - Proceedings of the 2011 Federated Events on Component-Based Software Engineering and Software Architecture - CBSE'11*, p. 179–187, 2011.

[4] HINDLE, A. et al. Yarn: Animating software evolution. In: IEEE. *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. [S.l.], 2007. p. 129–136.

[5] PYPI. *pip search has been temporarily disabled*. 2020. Accessed: 2021-07-27. Available from Internet: <https://github.com/pypa/pip/issues/9312>.

[6] FOUNDATION, P. S. *Python Package Index*. 2018. Accessed: 2018-09-30. Available from Internet: <https://pypi.org/>.

[7] MAVEN, A. *Apache Maven Project*. 2021. Accessed: 2021-07-27. Available from Internet: <https://maven.apache.org/>.

[8] NPM. *NPM: Build amazing things*. 2021. Accessed: 2021-07-27. Available from Internet: <https://www.npmjs.com/>.

[9] CRATES. *Crates: The Rust community's crate registry*. 2021. Accessed: 2021-07-27. Available from Internet: <https://crates.io/>.

[10] YARN. *Yarn: Safe, stable, reproducible projects*. 2021. Accessed: 2021-07-27. Available from Internet: <https://yarnpkg.com/>.

[11] LEE, G. et al. eapt: enhancing apt with a mirror site resolver. In: IEEE. *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. [S.l.], 2020. p. 117–122.

[12] DALE, C.; LIU, J. apt-p2p: A peer-to-peer distribution system for software package releases and updates. In: IEEE. *IEEE INFOCOM 2009*. [S.l.], 2009. p. 864–872.

[13] HERRY, H. et al. Peer-to-peer secure updates for heterogeneous edge devices. In: IEEE. *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*. [S.l.], 2018. p. 1–5.

[14] ZHANG, Q. et al. An optimized dht for linux package distribution. In: IEEE. *2016 15th International Symposium on Parallel and Distributed Computing (ISPDC)*. [S.l.], 2016. p. 298–305.

[15] FEITELSON, D. G. *"We do not appreciate being experimented on": Developer and Researcher Views on the Ethics of Experiments on Open-Source Projects*. 2021.

[16] Wikipedia contributors. *Blockchain — Wikipedia, The Free Encyclopedia*. 2022. Accessed: 2022-11-10. Available from Internet: <https://en.wikipedia.org/w/index.php?title=Blockchain>.

[17] MUELLER, T. Let's attest! multi-modal certificate exchange for the web of trust. In: IEEE. *2021 International Conference on Information Networking (ICOIN)*. [S.l.], 2021. p. 758–763.

[18] ELLISON, R. J. et al. *Evaluating and mitigating software supply chain security risks*. [S.l.], 2010.

[19] SABBAGH, B. A.; KOWALSKI, S. A socio-technical framework for threat modeling a software supply chain. *IEEE Security & Privacy*, IEEE, v. 13, n. 4, p. 30–39, 2015.

[20] PROJECT, P. *PPIO / Proof-of-Download (PoD)*. 2019. Accessed: 2020-04-28. Available from Internet: <https://www.pp.io/docs/guide/>.

[21] KUMAR, S.; BHARTI, A. K.; AMIN, R. Decentralized secure storage of medical records using blockchain and ipfs: A comparative analysis with future directions. *SECURITY AND PRIVACY*, v. 4, n. 5, p. e162, 2021. Available from Internet: <https://onlinelibrary.wiley.com/doi/abs/10.1002/spy2.162>.

[22] NIZAMUDDIN, N. et al. Decentralized document version control using ethereum blockchain and ipfs. *Computers & Electrical Engineering*, v. 76, p. 183–197, 2019. ISSN 0045-7906. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S0045790618333093>.

[23] D'MELLO, G.; GONZÁLEZ-VÉLEZ, H. Distributed software dependency management using blockchain. In: IEEE. *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. [S.l.], 2019. p. 132–139.

[24] DOCUMENTATION, E. *Ethereum - GAS AND FEES*. 2022. Accessed 2022-11-10. Available from Internet: <https://ethereum.org/en/developers/docs/gas/>.

[25] INCE, M. N.; AK, M.; GUNAY, M. Blockchain based distributed package management architecture. In: IEEE. *2020 5th International Conference on Computer Science and Engineering (UBMK)*. [S.l.], 2020. p. 238–242.

[26] URQUHART, A. The inefficiency of bitcoin. *Economics Letters*, Elsevier, v. 148, p. 80–82, 2016.

[27] WHITE CHENGI JIMMY KUO, D. M. C. S. R. *Coping with Computer Viruses and Related Problems*. 1989. Accessed: 2022-06-19. Available from Internet: <https://web.archive.org/web/20071014212824/http://www.itmweb.com/essay503.htm>.

[28] MAURER, U. Modelling a public-key infrastructure. In: SPRINGER. *European Symposium on Research in Computer Security*. [S.l.], 1996. p. 325–350.

[29] CARONNI, G. Walking the web of trust. In: IEEE. *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2000.(WET ICE 2000). Proeedings. IEEE 9th International Workshops on*. [S.l.], 2000. p. 153–158.

[30] GOOGLE. *Google Play Store*. 2022. Accessed: 2022-04-09. Available from Internet: <https://play.google.com/store>.

[31] MICROSOFT. *Microsof Application Store*. 2022. Accessed: 2022-04-09. Available from Internet: <https://www.microsoft.com/en-us/store/apps/windows>.

[32] APPLE. *Apple Store*. 2022. Accessed: 2022-04-09. Available from Internet: <https://www.apple.com/store>.

[33] MONTECELO, M. A. F. *aptitude user's manual*. 2016. Accessed: 2022-06-19. Available from Internet: <https://www.debian.org/doc/manuals/aptitude/pr01s02.en.html>.

[34] KRAILETH. *The history of *nix package management*. 2017. Accessed: 2022-06-19. Available from Internet: <https://eerielinux.wordpress.com/2017/08/15/the-history-of-nix-package-management/>.

[35] WELSH, M. et al. *dpkg: Debian GNU/Linux package maintenance utility*. 1994. Accessed: 2022-06-19. Available from Internet: <https://web.archive.org/web/20150402141229/>.

[36] WALL, L. *The Timeline of Perl and its Culture*. 2001. Accessed: 2022-06-19. Available from Internet: <https://history.perl.org/PerlTimeline.html>.

[37] BORS, r. i. t. *Rust Releases History*. 2014. Accessed: 2022-06-19. Available from Internet: <https://github.com/rust-lang/rust/blob/master/RELEASES.md>.

[38] BORS, r. i. t. *Cargo Releases History*. 2014. Accessed: 2022-06-19. Available from Internet: <https://github.com/rust-lang/cargo/tags?after=0.6.0>.

[39] GOLANG, D. *Go Lang: Managing dependencies*. 2022. Accessed: 2022-06-19. Available from Internet: <https://go.dev/doc/modules/managing-dependencies>.

[40] LIBP2P. *What is libp2p?* 2022. Accessed: 2022-04-09. Available from Internet: <https://docs.libp2p.io/introduction/what-is-libp2p/>.

[41] ZHANG, C. et al. Unraveling the bittorrent ecosystem. *IEEE Transactions on Parallel and Distributed Systems*, IEEE, v. 22, n. 7, p. 1164–1177, 2010.

[42] DONENFELD, J. A. Wireguard: next generation kernel network tunnel. In: *NDSS*. [S.l.: s.n.], 2017. p. 1–12.

[43] MAYMOUNKOV, P.; MAZIERES, D. Kademlia: A peer-to-peer information system based on the xor metric. In: SPRINGER. *International Workshop on Peer-to-Peer Systems*. [S.l.], 2002. p. 53–65.

[44] GANESH, A. J.; KERMARREC, A.-M.; MASSOULIÉ, L. Peer-to-peer membership management for gossip-based protocols. *IEEE transactions on computers*, IEEE, v. 52, n. 2, p. 139–149, 2003.

[45] PAUL, E. What is digital signature-how it works, benefits, objectives, concept. *EMP Trust HR, Gaithersburg, MD, USA, Tech. Rep*, 2017.

[46] BIRYUKOV, A.; DINU, D.; KHOVRATOVICH, D. Argon2: new generation of memory-hard functions for password hashing and other applications. In: IEEE. *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. [S.l.], 2016. p. 292–302.

[47] KATZ, J.; LINDELL, Y. *Introduction to modern cryptography*. [S.l.]: CRC press, 2020.

[48] GIRAULT, M.; COHEN, R. et al. A generalized birthday attack. In: SPRINGER. *Workshop on the Theory and Application of of Cryptographic Techniques*. [S.l.], 1988. p. 129–156.

[49] YU, M. et al. Coded merkle tree: Solving data availability attacks in blockchains. In: SPRINGER. *International Conference on Financial Cryptography and Data Security*. [S.l.], 2020. p. 114–134.

[50] FRIEDENBACH, M.; ALM, K. *BIP98: Fast Merkle hash-tree*. 2017. Accessed: 2022-04-15. Available from Internet: <https://github.com/bitcoin/bips/blob/master/bip-0098.mediawiki>.

[51] BRTFS. *BrtFS*. 2022. Accessed: 2022-04-15. Available from Internet: <https://btrfs.wiki.kernel.org/index.php/Main_Page>.

[52] ZFS. *ZFS*. 2022. Accessed: 2022-04-15. Available from Internet: <https://zfsonlinux.org/>.

[53] NYALETEY, E. et al. Blockipfs-blockchain-enabled interplanetary file system for forensic and trusted data traceability. In: IEEE. *2019 IEEE International Conference on Blockchain (Blockchain)*. [S.l.], 2019. p. 18–25.

[54] TRAUTWEIN, D. et al. Design and evaluation of ipfs: a storage layer for the decentralized web. In: *Proceedings of the ACM SIGCOMM 2022 Conference*. [S.l.: s.n.], 2022. p. 739–752.

[55] TEAM ipfs developer. *ipfs | Docs*. 2022. Accessed: 2022-04-05. Available from Internet: <https://docs.ipfs.io/concepts/what-is-ipfs/#decentralization>.

[56] NAOR, M.; WIEDER, U. A simple fault tolerant distributed hash table. In: SPRINGER. *International Workshop on Peer-to-Peer Systems*. [S.l.], 2003. p. 88–97.

[57] XIA, R. L.; MUPPALA, J. K. A survey of bittorrent performance. *IEEE Communications surveys & tutorials*, IEEE, v. 12, n. 2, p. 140–158, 2010.

[58] PREISSE, R.; STACHMANN, B. *Git: Distributed Version Control–Fundamentals and Workflows*. [S.l.]: Brainy Software Inc, 2014.

[59] BENET, J. *IPFS-Content Addressed, Versioned, P2P File System*. 2020. Available from Internet: <https://ipfs.io/ipfs/QmR7GSQM93Cx5eAg6a6yRzNde1FQv7uL6X1o4k7zrJa3LX/>.

[60] INC, P. L. *A protocol for disambiguating the encoding*. 2016. Available from Internet: <https://github.com/multiformats/multibase>.

[61] INC, P. L. *Canonical table of of codecs used by various multiformats*. 2016. Available from Internet: <https://github.com/multiformats/multicodec>.

[62] INC, P. L. *Self identifying hashes*. 2016. Available from Internet: <https://github.com/multiformats/multihash>.

[63] IPFS-DEVS. *IPFS Gateway*. 2020. Accessed: 2022-04-15. Available from Internet: <https://docs.ipfs.io/concepts/ipfs-gateway/#overview>.

[64] NAKAMOTO, S. *Bitcoin: A Peer-to-Peer Electronic Cash System*. Working Paper, 2008. Available from Internet: <https://bitcoin.org/bitcoin.pdf>.

[65] ANTONOPOULOS, A. *Mastering Bitcoin*. 2. ed. [S.l.]: O'Reilly, Media, 2017. ISBN 978-1491954386.

[66] MUKHOPADHYAY, U. et al. A brief survey of cryptocurrency systems. In: IEEE. *Privacy, Security and Trust (PST), 2016 14th Annual Conference on*. [S.l.], 2016. p. 745–752.

[67] ZHENG, Z. et al. An overview of blockchain technology: Architecture, consensus, and future trends. In: IEEE. *Big Data (BigData Congress), 2017 IEEE International Congress on*. [S.l.], 2017. p. 557–564.

[68] TECH, P. *Proof-of-Authority Chains - Wiki*. 2018. Accessed: 2019-01-13. Available from Internet: <https://wiki.parity.io/Proof-of-Authority-Chains>.

[69] BALIGA, A. Understanding blockchain consensus models. *Persistent*, 2017.

[70] GILAD, Y. et al. Algorand: Scaling byzantine agreements for cryptocurrencies. In: *Proceedings of the 26th symposium on operating systems principles*. [S.l.: s.n.], 2017. p. 51–68.

[71] CIEśLA, K. *Average time to mine a block in minutes*. 2018. Accessed: 2019-01-05. Available from Internet: <https://data.bitcoinity.org/>.

[72] MALAN, D. J.; WELSH, M.; SMITH, M. D. A public-key infrastructure for key distribution in tinyos based on elliptic curve cryptography. In: IEEE. *2004 First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004.* [S.l.], 2004. p. 71–80.

[73] AMIR, A.; BENSON, G.; FARACH, M. Let sleeping files lie: Pattern matching in z-compressed files. *Journal of Computer and System Sciences*, Elsevier, v. 52, n. 2, p. 299–307, 1996.

[74] SUNAR, B. True random number generators for cryptography. In: *Cryptographic Engineering*. [S.l.]: Springer, 2009. p. 55–73.

[75] TSIMBALO, E.; FAFOUTIS, X.; PIECHOCKI, R. J. Crc error correction in iot applications. *IEEE Transactions on Industrial Informatics*, IEEE, v. 13, n. 1, p. 361–369, 2016.

[76] ZHENG, Q. et al. An innovative ipfs-based storage model for blockchain. In: IEEE. *2018 IEEE/WIC/ACM international conference on web intelligence (WI)*. [S.l.], 2018. p. 704–708.

[77] STEICHEN, M. et al. Blockchain-based, decentralized access control for ipfs. In: IEEE. *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. [S.l.], 2018. p. 1499–1506.

[78] CHEN, Y. et al. An improved p2p file system scheme based on ipfs and blockchain. In: IEEE. *2017 IEEE International Conference on Big Data (Big Data)*. [S.l.], 2017. p. 2652–2657.

[79] NIZAMUDDIN, N. et al. Decentralized document version control using ethereum blockchain and ipfs. *Computers & Electrical Engineering*, Elsevier, v. 76, p. 183–197, 2019.

[80] NIZAMUDDIN, N.; HASAN, H. R.; SALAH, K. Ipfs-blockchain-based authenticity of online publications. In: SPRINGER. *International Conference on Blockchain*. [S.l.], 2018. p. 199–212.

[81] FORMAT, C.; FILES, M. T. for C.-S. V. C. *Shafranovich*. 2005. Accessed: 2022-04-23. Available from Internet: <https://datatracker.ietf.org/doc/html/rfc4180>.

[82] COGO, F. R.; OLIVA, G. A.; HASSAN, A. E. Deprecation of packages and releases in software ecosystems: A case study on npm. *IEEE Transactions on Software Engineering*, IEEE, 2021.

[83] INTERNET, P. L. is building the next generation of the. *ProtoLabs*. 2022. Accessed: 2022-04-23. Available from Internet: <https://protocol.ai/>.

[84] HEUN, J. *JavaScript libp2p MulticastDNS discovery implementation.* 2021. Available from Internet: <https://github.com/libp2p/js-libp2p-mdns>.

[85] SANTOS, V. *JavaScript libp2p Implementation of the railing process of a Node through a bootstrap peer list.* 2021. Available from Internet: <https://github.com/libp2p/js-libp2p-bootstrap>.

[86] CLAVER, M. et al. Regis: Regular expression simplification via rewrite-guided synthesis. *arXiv preprint arXiv:2104.12039*, 2021.

[87] VINET, J.; GRIFFIN, A. *ArchLinux: A simple, lightweight distribution*. 2019. Accessed: 2019-01-13. Available from Internet: <https://www.archlinux.org/>.

[88] TEAM aurweb D. *AUR Home*. 2019. Accessed: 2019-01-13. Available from Internet: <https://aur.archlinux.org/>.

[89] POINTCHEVAL, D.; STERN, J. Security arguments for digital signatures and blind signatures. *Journal of cryptology*, Springer, v. 13, n. 3, p. 361–396, 2000.

[90] CONSERVANCY, S. F. *Git is a free and open source distributed version control system*. 2019. Accessed: 2019-01-13. Available from Internet: <https://git-scm.com/>.

[91] VINET, J.; GRIFFIN, A. *Package Search*. 2019. Accessed: 2019-01-13. Available from Internet: <https://www.archlinux.org/packages/>.

[92] FOUNDATION, I. F. S. *GNU Project's shell—the Bourne Again SHell.* 2022. Accessed: 2022-05-03. Available from Internet: <https://www.gnu.org/software/bash/>.

[93] CUREG, E.; MUKHERJEA, A. Numerical results on some generalized random fibonacci sequences. *Computers & Mathematics with Applications*, v. 59, n. 1, p. 233–246, 2010. ISSN 0898-1221. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S0898122109005136>.

[94] MATHUR, A. et al. The new ext4 filesystem: current status and future plans. In: CITESEER. *Proceedings of the Linux symposium*. [S.l.], 2007. v. 2, p. 21–33.

[95] WUSTEMAN, J. Rss: the latest feed. *Library hi tech*, Emerald Group Publishing Limited, 2004.

[96] MESHRAM, S. U. Evolution of modern web services–rest api with its architecture and design. *International Journal of Research in Engineering, Science and Management*, v. 4, n. 7, p. 83–86, 2021.

[97] ZIMMERLE, F. *Capivara repo*. 2020. Accessed: 2020-05-01. Available from Internet: <https://github.com/zimmerle/capivara>.

[98] LIBP2P. *A modular network stack.* 2021. Accessed: 2021-08-02. Available from Internet: <https://libp2p.io/>.

[99] COSTA, F. Z. d. N.; QUEIROZ, R. J. de. A blockchain using proof-of-download. In: IEEE. *2020 IEEE International Conference on Blockchain (Blockchain)*. [S.l.], 2020. p. 170–177.

[100] COSTA, F. Z. D. N. et al. Distributed repository for software packages using blockchain. *IEEE Access*, v. 10, p. 112502–112514, 2022.

Appendix A - Capavira JSON API

Capivara API: Response on block summary retrieval

Listing 5 – Capivara API: Response on block summary retrieval


```
{
  "summary":"QmXsjLVK8RJhe2w5D3JpDZbayQLQz87JC6WR79zPX5u4U3"
}
```


Source: Elaborated by the author (2022)

Capivara API: Response on forgers summary retrieval

Listing 6 – Capivara API: Response on block forger retrieval


```
{
  "forgers":"QmYNY6TWhiBDZe4jj79mQBjxB2U3yaHF4Gc4WZXrHD1Fa1"
}
```


Source: Elaborated by the author (2022)

Capivara API: Block retrieval response

Listing 7 – Capivara API: Block retrieval response.


```
{
  "info":{
    "version":"4",
    "height":26100,
    "merkle_root":"e72645be9d5c8498789eaacc4162733b4f71a328f1aba231a78bd0182177b869",
    "summary":"QmW5NMxSJjAvAd1mvEP83EFEee8viUH15D6a9SRSVmWijS",
    "forgers":"QmYNY6TWhiBDZe4jj79mQBjxB2U3yaHF4Gc4WZXrHD1Fa1",
    "previous_hash":"49859e9e5c4d7375cf337cceaab0fd5a0f4f30f7e79f453c4644eab24fe9f0ef",
    "time_stamp":"2011-03-05T15:59:25+00:00",
    "forger":"Cubert",
    "hash":"38f8e67ba368d774cdc87a9ef4b6ba16a0f1f25cc0ceba52b03aeea66bb5b5c4"
  },
  "payload":[
    "newpackage:django-flash:1.8:QmaqGBYFiC885vhdjQPrEavur4ku4PBVbzWko4gSwrBTrT",
    "newpackage:poly2tri:0.3.3:QmSHP9kPcroTXMWiSmVZYht3RgV2juaCWzqVZMzhLejx5H",
    "newpackage:sijax:0.1.6:QmQXVUUkHCdVn7KKRHiubWumkhEoYNSnyspBqomQtFWivJ"
  ],
  "status":{
    "packages_publishers":{
      "Hermes":2,
      "Kif":1
    },
    "forger_party":[
      "Cubert",
      "Heber",
      "Nibbler"
    ],
    "packageNames":[
      "django-flash",
      "poly2tri",
      "sijax"
    ]
  },
  "proofs":{
    "newpackage:django-flash:1.8:QmaqGBYFiC885vhdjQPrEavur4ku4PBVbzWko4gSwrBTrT":{
      "hashes":[
        {
          "type":-1,
          "hash":"c37b7043cc7f6861a46dfaf12fd6fe6d58dadc60dde681d15be42dbc28f22727"
        },
        {
          "type":1,
          "hash":"63d1f31de6a0341733e005a86c02b43f8c737ab72cd6c58a5e975fb5666a6d43"
        }
      ],
      "block":26100,
      "info":"newpackage:django-flash:1.8:QmaqGBYFiC885vhdjQPrEavur4ku4PBVbzWko4gSwrBTrT"
    },
    "newpackage:poly2tri:0.3.3:QmSHP9kPcroTXMWiSmVZYht3RgV2juaCWzqVZMzhLejx5H":{
      "hashes":[
        {
          "type":1,
          "hash":"aa34b75562ac1500ad37aebac35ee6dc04dfcecd924cf5b42b294cab48d49e4e"
        },
        {
          "type":-1,
```

```
          "hash":"bb51e003f6856cc635253c4a30644cee2b748f5c8577584ef7e16139cc6cefd1"
        }
      ],
      "block":26100,
      "info":"newpackage:poly2tri:0.3.3:QmSHP9kPcroTXMWiSmVZYht3RgV2juaCWzqVZMzhLejx5H"
    },
    "newpackage:sijax:0.1.6:QmQXVUUkHCdVn7KKRHiubWumkhEoYNSnyspBqomQtFWivJ":{
      "hashes":[
(...)
      ],
      "block":26100,
      "info":"newpackage:sijax:0.1.6:QmQXVUUkHCdVn7KKRHiubWumkhEoYNSnyspBqomQtFWivJ"
    }
  },
  "signarute":"..."
}
```

Source: Elaborated by the author (2022)

Capivara API: Response on package retrieval

Listing 8 – Capivara API: Response on package retrieval

```
{
   "name":"pygenx",
   "summary":"Wrapper for the genx lightweight canonical XML generation library.",
   "versions":[
      {
         "date":"2005-03-22T22:48:40Z",
         "version":"0.5.3",
         "whistleblower":"Amy",
         "_cid":"QmR7qbZ6ZNhPi3TNyTTe7N4GHoXxEng8wXGMr2KyLg1Lup",
         "proof":{
            "hashes":[
               {
                  "type":-1,
                  "hash":"66993f64c5216b38c0ec073f0996d2b5afdb3973c46f74345542ce4e5196142b"
               }
            ],
            "block":16,
            "info":"newpackage:roundup:0.8.2:QmVgxKkPCrRhnqTLPLdng8zvfbG9q7nJ11ZfAvFDVEVKk4"
         }
      },
      {
         "date":"2005-08-23T22:28:56Z",
         "version":"0.6",
         "whistleblower":"Kif",
         "_cid":"Qmajs3GftQEn6z9j1Njgu6ZjTTdu4ov4Xi2LCXZQhUy9HZ",
         "proof":{
            "hashes":[
               {
                  "type":-1,
                  "hash":"2f4a094db77eb6122ae91aba2cc3d3c6bfb6123c48b7bdb138b45035b01721f4"
               }
            ],
            "block":1864,
            "info":"newpackage:pygenx:0.6:Qmajs3GftQEn6z9j1Njgu6ZjTTdu4ov4Xi2LCXZQhUy9HZ"
```

```
        }
      }
    ]
}
```

Source: Elaborated by the author (2022)

Capivara API: Response on a block header request

Listing 9 – Capivara API: Response on a block header request


```
{
   "info":{
      "version":"4",
      "height":26302,
      "merkle_root":"e1fee2d7b09b4ce6e63ad6ebaa5c9c693c2f749099e1efd9d400934f632d24cc",
      "summary":"QmXsjLVK8RJhe2w5D3JpDZbayQLQz87JC6WR79zPX5u4U3",
      "forgers":"QmYNY6TWhiBDZe4jj79mQBjxB2U3yaHF4Gc4WZXrHD1Fa1",
      "previous_hash":"49859e9e5c4d7375cf337cceaab0fd5a0f4f30f7e79f453c4644eab24fe9f0ef",
      "time_stamp":"2011-03-22T11:59:25+00:00",
      "forger":"Nibbler",
      "hash":"94fbf3385e8806ba96bf7ff7e2757bfba006bf4afe9f633e4e86658154f89ee9"
   }
}
```


Source: Elaborated by the author (2022)

Capivara API: Response on package data retrieval

Listing 10 – Capivara API: Response on pakage data retrieval


```
{
  "publisher":"Amy",
  "allowed_publishers":[
    [
      "Amy",
      "QmayJQcbLx9whSNwnpZgWTFiL3GeRyN8ySGfRtsUKXLHUc"
    ],
    [
      "Hermes",
      "QmWrKz5Bqwi1VfFizYXn8T1ykTxMqmFrZa5sBqDQLBehUD"
    ],
    [
      "Kif",
      "QmYgtFZrhhP8NmbLxdHzX9teyuAAMDNW2e8YgzVHtV4aXc"
    ],
    [
      "Leela",
      "QmUYvS2CnzQW4h3Ntw9kTDFvmdF7Utzw9sVKCWN4u8JX76"
    ]
  ],
  "pypi":{
    "info":{
      "author":"Michael Twomey",
      "author_email":"mick@translucentcode.org",
      "bugtrack_url":null,
      "classifiers":[
        "Development Status :: 4 - Beta",
        "Intended Audience :: Developers",
        "License :: OSI Approved :: MIT License",
        "Natural Language :: English",
        "Operating System :: MacOS :: MacOS X",
        "Operating System :: Microsoft :: Windows",
        "Operating System :: POSIX :: Linux",
        "Programming Language :: C",
        "Programming Language :: Python",
        "Topic :: Software Development :: Libraries :: Python Modules",
        "Topic :: Text Processing :: Markup :: XML"
      ],
      "description":"Genx is a light weight C library for the sole purpose of generating\ncanonical XML. It is simple to
      ↪ use and gaurantees correctness. It operates\npurely in UTF-8 and is careful to escape strings when required by the
      ↪  XML or\ncanonical XML spec.",
      "description_content_type":null,
      "docs_url":null,
      "download_url":"http://software.translucentcode.org/pygenx/pygenx-0.6.tar.gz",
      "downloads":{
        "last_day":-1,
        "last_month":-1,
        "last_week":-1
      },
      "home_page":"http://software.translucentcode.org/pygenx",
      "keywords":null,
      "license":"http://software.translucentcode.org/pygenx/LICENSE",
      "maintainer":null,
      "maintainer_email":null,
      "name":"pygenx",
      "package_url":"https://pypi.org/project/pygenx/",
```

```
    "platform":"any",
    "project_url":"https://pypi.org/project/pygenx/",
    "project_urls":{
      "Download":"http://software.translucentcode.org/pygenx/pygenx-0.6.tar.gz",
      "Homepage":"http://software.translucentcode.org/pygenx"
    },
    "release_url":"https://pypi.org/project/pygenx/0.6/",
    "requires_dist":null,
    "requires_python":null,
    "summary":"Wrapper for the genx lightweight canonical XML generation library.",
    "version":"0.6",
    "yanked":false,
    "yanked_reason":null
  },
  "last_serial":803220,
  "releases":{
    "0.5.3":[
      {
        "comment_text":"",
        "digests":{
          "md5":"185e4ff91e0969e76641210d8334998e",
          "sha256":"473c10ea014b2da31405acfccc0ae141bf31dc148c2eb4fa3266cfc18954b2bb"
        },
        "downloads":-1,
        "filename":"pygenx-0.5.3.tar.gz",
        "has_sig":false,
        "md5_digest":"185e4ff91e0969e76641210d8334998e",
        "packagetype":"sdist",
        "python_version":"source",
        "requires_python":null,
        "size":50585,
        "upload_time":"2005-03-22T22:48:40",
        "upload_time_iso_8601":"2005-03-22T22:48:40Z",
        "url":"https://files.pythonhosted.org/packages/c4/1d/
    ↪ d23c5402d22922fb1aae8472d6dc8ead19176856c488450d6d8b555d1b70/pygenx-0.5.3.tar.gz",
        "yanked":false,
        "yanked_reason":null
      }
    ]
  },
  "urls":[
    {
      "comment_text":"Binary egg for use with setuptools",
      "digests":{
        "md5":"fad60570b22d94a9ec678cb697a48ff7",
        "sha256":"48128b8d4bc95bbb282bae53f0cb2b9f995979b5a5e94f2dcefb0f72dc931192"
      },
      "downloads":-1,
      "filename":"pygenx-0.6-py2.4-darwin-8.2.0-Power_Macintosh.egg.zip",
      "has_sig":false,
      "md5_digest":"fad60570b22d94a9ec678cb697a48ff7",
      "packagetype":"sdist",
      "python_version":"source",
      "requires_python":null,
      "size":33694,
      "upload_time":"2005-08-23T22:28:56",
      "upload_time_iso_8601":"2005-08-23T22:28:56Z",
      "url":"https://files.pythonhosted.org/packages/cc/05/90f827b9aa4b1c4aea9dec3b3e00e76339cda7afb6ee8ddbeac83745c642/
    ↪ pygenx-0.6-py2.4-darwin-8.2.0-Power_Macintosh.egg.zip",
      "yanked":false,
```

```
      "yanked_reason":null
    },
    {
      "comment_text":"Binary egg for use with setuptools",
      "digests":{
        "md5":"2c40f5f9ca1541520f07a3fd529434f8",
        "sha256":"d7a661aa2a8b4d798b94930bffe77ec1d23eeab3748962ddd7c2ac8cf65419ae"
      },
      "downloads":-1,
      "filename":"pygenx-0.6-py2.4-win32.egg.zip",
      "has_sig":false,
      "md5_digest":"2c40f5f9ca1541520f07a3fd529434f8",
      "packagetype":"sdist",
      "python_version":"source",
      "requires_python":null,
      "size":33608,
      "upload_time":"2005-08-24T09:16:33",
      "upload_time_iso_8601":"2005-08-24T09:16:33Z",
      "url":"https://files.pythonhosted.org/packages/93/50/40ab75af759029f96307194b1b0af1c162815bfdd63aefe5324f44d9bd9b/
    ↪ pygenx-0.6-py2.4-win32.egg.zip",
      "yanked":false,
      "yanked_reason":null
    },
    {
      "comment_text":"",
      "digests":{
        "md5":"085917e2be7e02a3ba53fb70850bf43e",
        "sha256":"467f418643604153dece7c674eebe3cac7124c7b7839c56397d0f3b4ba884fab"
      },
      "downloads":-1,
      "filename":"pygenx-0.6.tar.gz",
      "has_sig":false,
      "md5_digest":"085917e2be7e02a3ba53fb70850bf43e",
      "packagetype":"sdist",
      "python_version":"source",
      "requires_python":null,
      "size":60326,
      "upload_time":"2005-08-23T22:28:53",
      "upload_time_iso_8601":"2005-08-23T22:28:53Z",
      "url":"https://files.pythonhosted.org/packages/1f/9d/e780b0f44b619cf7af2e556b0ad754f64c130181d9227abee6587e02dc24/
    ↪ pygenx-0.6.tar.gz",
      "yanked":false,
      "yanked_reason":null
    },
    {
      "comment_text":"",
      "digests":{
        "md5":"0011f656541656fc32c0217fb92176d6",
        "sha256":"5da5d759404b61845289988999a65e8b92d5c8899f9af39f9972ba1e8b5c856d"
      },
      "downloads":-1,
      "filename":"pygenx-0.6.win32-py2.4.exe",
      "has_sig":false,
      "md5_digest":"0011f656541656fc32c0217fb92176d6",
      "packagetype":"bdist_wininst",
      "python_version":"2.4",
      "requires_python":null,
      "size":93899,
      "upload_time":"2005-08-24T09:18:49",
      "upload_time_iso_8601":"2005-08-24T09:18:49Z",
```

```
      "url":"https://files.pythonhosted.org/packages/51/78/92fdc3cef2ff710389b9007aaf659dbf106d90daf93fb6d22020ab450568/
    ↪ pygenx-0.6.win32-py2.4.exe",
      "yanked":false,
      "yanked_reason":null
     },
     {
      "comment_text":"",
      "digests":{
        "md5":"0f73e85c81299e597197c1c2f12670d9",
        "sha256":"448424e015aa838ddee9019de00196d2b44441412eda5ee6fdae157ed747b374"
      },
      "downloads":-1,
      "filename":"pygenx-0.6.zip",
      "has_sig":false,
      "md5_digest":"0f73e85c81299e597197c1c2f12670d9",
      "packagetype":"sdist",
      "python_version":"source",
      "requires_python":null,
      "size":82639,
      "upload_time":"2005-08-24T09:16:33",
      "upload_time_iso_8601":"2005-08-24T09:16:33Z",
      "url":"https://files.pythonhosted.org/packages/5b/b2/1334a8f98da0958f4cd15f9d72fa85c7b5fcfc43fdd78504afa4c894b0e2/
    ↪ pygenx-0.6.zip",
      "yanked":false,
      "yanked_reason":null
     }
    ],
    "vulnerabilities":[

    ]
  }
}
```


Source: Elaborated by the author (2022)

Appendix B - Example of Packages Summary

Listing 11 – Example of Packages Summary

```
zconfig,2.9.0,Structured Configuration Library,"26302,-1,9fd45b71e98346cc4dd7364daf1b829e378d9332d7894b501267544a1ce897bb
    ,-1,e6004b9a51ebc01804edc72eb49552ada781755ef769dd3eb0c7d4baa89e7205"
pygenx,0.6,Wrapper for the genx lightweight canonical XML generation library.,"1864,-1,2
    f4a094db77eb6122ae91aba2cc3d3c6bfb6123c48b7bdb138b45035b01721f4"
roundup,1.4.16,"A simple-to-use and -install issue-tracking system with command-line, web and e-mail interfaces. Highly
    customisable.","24319,-1,53924702cbcad16233c07e80602b9054e9f986a37082e49f7a0d707104c19b0a"
rlcompleter2,0.98,interactive tab-completion for python commandline,"20988,-1,
    c2f1b9529995972531d808e21b0ea760193f400bc162b3cf97d1b72231c837a0,-1,560
    c9279fb66ab43770f82d308118f1b9814f5f9d037aa9b33e70b72a3d3a5fb"
ll-core,1.11.1,"LivingLogic base package: ansistyle, color, make, sisyphus, xpit, url, xml_codec","12311,-1,
    b9e0dfcd317a46bda20577f0871ae93d10af1dc8c0b104eb521d75aa7976f1d9"
ll-orasql,1.27.1,Utilities for working with cx_Oracle,"17654,1,
    ac6bd318fd47d08b19b1128e9c3d4f1db384abfeef9f31b9e8c9f260b1646700,-1,
    a6267639888055eb8183a7a9b41283c49f88d1113a9865ed2e65c6cf12a17d52,1,252
    ba283d5efbee153f5e0fb5b203cadd24d2d231b54e9dd7b4ffc01b1be723f"
ll-toxic,0.10,Generate Oracle functions from PL/SQL embedded in XML.,"12265,-1,065
    c6203e06166c04c63d61bb4fe5c7155d17355c8a8224a412cc83727590caf,1,5084
    f6a9dd5104bbb8e22225a368e1ebfd4a1951ce738e2011ee25aa21299f42,-1,71475
    b16215e6e549df598ac57aa8541442159ec1e72102dd10f8701de222451"
ll-xist,3.17.3,"Extensible HTML/XML generator, cross-platform templating language, Oracle utilities and various other
    tools","26065,-1,7ac73da9629feae0263bec9973d446d7d35f6d58ad597be68c5489b0b25fa028,1,
    b0b01e55da4a6f49b5a6844ce69f19835c823df9e5790adaed18c1a54ea9c265,-1,16
    dad9966aab9102d898ed53ce01fd9a4eca199adae649202fc549c27ba943a6,1,7
    bbfe5964949f877508adf1f10785d8f5b94394c1a68e533bb9dea1b775d0e2b"
pywebsms,1.1,pyWebSMS is an graphical sms sender which supports many programmable providers.,"336,-1,
    e1b80b39e3bc45af77f899009ed76437c8d6186caf8a1016c2ad51d09559f16a"
webstack,1.2.7,Common API for Python Web applications,"11416,-1,649461
    ea6e20830dc395d311bb5c7e5998aa1a7d75a5d4ea455565792a941731"
jonpy,0.09,Jon's Python modules,"22801,-1,89b2f4b8a0c71e99d1e69cee0d762c8d3bd122a86398f9a4c0ff709f00953fb6,1,
    ed85be2577b1a775cac71805f993edc78798f15aba1fa02eca33b0480e89a8c6,-1,
    ed9371703a246df3622b824c7ebb4e801b8cdc242f7fcff556de7ad5748c3084,1,
    d007fc832d2a934d8f98a4190d5640ca05c89db2f973ef0efbb0a98e86c9dca8,1,
    fc49b0880352d5891c7838d8ba2deeea820cab0d35be69b6e9398f0af378d92b"
pyircserver,0.0.5.0-Alpha,"Powerful, very personalizable and multiplatform IRCd","933,-1,77
    c3a08fc1f0d3293125ff78488535587f58d92f851d32dd9f7cea93651bc125"
pytaskplan,1.3.3,Python task planner,"2225,-1,7cc2283cfb02061adec3d2f06a3aeb6d9320f98ea44fb20b9b78440ebd81c43d"
openpgp,0.2.3,OpenPGP implementation,"1228,-1,c94d11533cf3af106f42087cc005c0d6981deacdb2171a699b6430e1997d358a"
lsystem2,1.6,LSystem for Python,"4275,-1,fd2611beb817aa031f54a66e91d876c55f7b1bbcdf30334cfee0ebfe52f1e0e5"
fcgiapp,1.4,C implementation of the FastCGI application protocol,"1296,-1,
    f591908da73801fed35cb83ddac33513f6741cc56c792dea8490d2e14d7b8f90"
buildutils,0.3,Distutils extensions for developing Python libraries and applications.,"10945,-1,473
    c69a8f53a3413384ddd1f8b25b026da0543514817653684ca978c45aa3d91,1,5
    a163ad653f4232aa6c8dd615480c3933666f1b39eb82cdc5bf2116fa0b4d770"
pypedal,2.0.0b6,Tools for pedigree analysis,"3987,-1,14f41badbda9a6f8391aeccdb0108d5531dcb58267c53c851cd5396970200099"
typecheck,0.3.5,A runtime type-checking module for Python,"5188,-1,8
    f5de1a2649347f19bf6803852895a7542957cee9b32390f1451654e8e5ccef8"
pyipc,0.41,Python bindings to System V IPC,"5823,-1,35700718a52503f60dcc9c95b3d7dd013405a4164e74b44c78df0750a51c0bb8"
(...)
```

Source: Elaborated by the author (2022)

Appendix C - Example of Possible Forgers

Listing 12 – Example of Possible Forgers

```
1 Cubert,QmVgt6q6HmymmfL3UPMCyEWkT4DHc4fQhMEeTX2CwC8EXy
2 Nibbler,QmVhdk2xPnwhBspcwUmJ6pNHAuk5EN9thEm5FKvK7tHGkh
3 Heber,QmWMmXJM63Evt6f6wWoET8C87E3WacDKd6LM5CkawuQzUW
```

Source: Elaborated by the author (2022)