Università di Genova
DIBRIS DIPARTIMENTO
DI INFORMATICA, BIOINGEGNERIA,
ROBOTICA E INGEGNERIA DEI SISTEMI

A Measurement Study of the Web3 Ecosystem

Federico Fontana

Master Thesis

Università di Genova, DIBRIS
Via Dodecaneso, 35 16146 Genova, Italy
https://www.dibris.unige.it/

MSc Computer Science
Software Security and Engineering Curriculum

A Measurement Study of the Web3 Ecosystem

Federico Fontana

Advisor:
Matteo Dell'Amico
Iskander Sanchez-Rola

Examiner: Marina Ribaudo

October, 2024

First, I thank my advisors and reviewers, Marina, Matteo, and Iskander, for allowing me to work on this thesis and sharing their knowledge.

I am extremely grateful for the friends I made in the past few years and the experiences I was so lucky to share with them. These have truly been some of the most exciting years of my life, and I am glad I could share them with you. Thank you to the members of "Tatan Tatan" and "i maghi"; having you around has made this time much brighter and easier.

Lastly, and surely not least, I want to thank my family for their unrelenting support in my academic endeavors.

Abstract

This work analyzes the structure, connectivity, and reliance on Web2 resources within a subset of the Web3 network. The study focuses on Web3 websites hosted on IPFS and accessed via ENS contenthash records. With this work, we reveal the current state of the Web3 network and uncover dependencies on Web2 infrastructure, particularly for dynamic content and user interactions.

Our findings show that Web3 websites frequently redirect users to Web2 websites or load assets from them, undermining their decentralized nature. Additionally, due to technical constraints like limited browser support for IPFS URIs and the immutability of IPFS, interlinking between Web3 websites is rare, hindering the development of a cohesive Web3 network. Additionally, we explore the adoption and usage of newer ENS features, which could offer insights into an evolving Web3 ecosystem.

Additionally, the thesis details the development process, including the scripts and methodologies used for data collection and analysis.

Table of Contents

Chapter 1 Introduction **8**

Chapter 2 Background **10**

2.1 Blockchains 10

2.1.1 Layer-2 Solutions 13

2.2 Ethereum 13

2.2.1 Smart Contracts 14

2.2.2 Events 15

2.2.3 Token Standards 15

2.2.4 Oracles 17

2.3 Ethereum Name Service 17

2.3.1 ENSIPs 18

2.3.2 Basic Concepts 18

2.3.3 Resolution Process 19

2.3.4 Architecture 20

2.4 Decentralized File Systems 28

2.4.1 IPFS 29

Chapter 3 Related Work **32**

Chapter 4 ENScraper **34**

4.1 Raw Data Source . . . 35
4.2 Overview . . . 36
4.3 Remote Service Interaction . . . 37
4.4 Features . . . 38
4.4.1 Rate Limit Compliance . . . 38
4.4.2 Caching . . . 39
4.4.3 Progress . . . 39
4.4.4 Confirmation . . . 40
4.4.5 Retry . . . 41
4.5 `Job` Interface Implementations . . . 42
4.5.1 ABI . . . 42
4.5.2 Etherscan Events . . . 43
4.5.3 Resolvers . . . 44
4.6 Example Execution . . . 45
4.7 Dataset Contents . . . 46
4.8 Future Improvements . . . 47

Chapter 5 Crawling Web3 Websites **48**
5.1 IPFS Data Access . . . 49
5.2 Preflight . . . 49
5.3 P2 . . . 50
5.3.1 Overview . . . 50
5.3.2 Notes on Security and Privacy Features of the Browser . . . 52
5.3.3 Task Pipeline . . . 53

Chapter 6 Results **58**
6.1 Parameters and Dataset Size . . . 58
6.2 Script Performance Analysis . . . 60

6.2.1 Preflight Performance with IPNS . . . 60
6.2.2 Cookie Banner Detection . . . 60
6.2.3 Crawling `iframe` Elements . . . 61
6.2.4 Database Structure . . . 62
6.3 ENS Feature Adoption . . . 62
6.3.1 Resolvers . . . 63
6.3.2 Contenthash . . . 65
6.3.3 Gasless DNS . . . 66
6.4 Crawling Results . . . 66
6.4.1 Page Redirects . . . 66
6.4.2 Portals to Web2 . . . 67
6.4.3 Data Loading . . . 69
6.4.4 Links . . . 70
6.4.5 Cookies . . . 72
6.4.6 Website Size . . . 72
6.4.7 Technologies Used . . . 74

Chapter 7 Conclusion **77**

Chapter 8 Future Work **79**

Bibliography **81**

Chapter 1

Introduction

Blockchains are a very interesting technology that has gained popularity in recent years. They are distributed ledgers that allow users to exchange money and information via fully trustless transactions in decentralized peer-to-peer networks. Furthermore, all the content that has ever been in the blockchain cannot be deleted, rendering it censorship-resistant. Their most important goals are privacy, security, and independence from a central authority. Blockchains have ignited the development of Web3, a new idea of the web that aims to be completely decentralized, hosting business logic and currencies on blockchains and files and resources on decentralized file systems. The ideals of Web3 echo those of blockchains, striving for a censorship-resistant web that does not rely on the current Web2 centralized infrastructure.

Ethereum [1] is a blockchain that specializes in smart contracts, which are programs that run on the blockchain and can be used to create decentralized applications (dApps). Ethereum's active community constantly develops new tools and applications that leverage the blockchain's capabilities. One of the most prominent and widely used projects in the Ethereum ecosystem is Ethereum Name Service (ENS) [2], a naming system for Ethereum accounts. It comprises a set of smart contracts that allow users to register and use human-readable names for their Ethereum accounts. It is commonly supported by many of the applications and libraries that interact with the Ethereum blockchain. ENS has evolved significantly since its first deployment in 2017, adding and evolving the uses for its many features. Starting with ENSIP-7 [3], ENS also supports standardized *contenthash* records that point to Web3 website hosted on decentralized file systems such as IPFS. Contenthashes are cryptographic hashes of file contents, guaranteeing the immutability of the linked resource.

In this Master's thesis, we analyze the current state of ENS and the adoption of each of its features. Additionally, we visited a subset of the Web3 websites referenced by user-

set contenthash records. We explore their content and discover the connections between them and traditional Web2 resources, their characteristics, and if and how they differ from conventional Web2 websites. Furthermore, we analyze and collect their security and privacy characteristics, discovering patterns and whether Web3 developers care about their users' privacy as one would expect on a platform with its users' privacy as one of its main objectives.

This thesis opens with an introduction to the foundational concepts needed to engage with the research. In particular, Chapter 2 discusses the main characteristics of blockchains, moving then to the Ethereum blockchain and its main feature, smart contracts, and how they interact with users and code off-chain. Then, the chapter outlines ENS' main features and the history of the various contracts governing its functionalities. The chapter closes by outlining the main features of decentralized file systems, with a particular focus on IPFS.

Building on this foundation, Chapter 3 reviews the existing literature, outlining the current state of research in the field and identifying gaps that this work aims to address.

The technical core of the thesis is developed in Chapters 4 and 5, where the focus shifts to the implementation of custom programs. These chapters detail the scraping of ENS data from the blockchain and the subsequent crawling of Web3 websites linked via contenthash records. They explore the technical decisions and challenges encountered along the way, offering insight into the process and the tools used.

Once the features and contents of the dataset have been defined, Chapter 6 presents the results, opening with a simple analysis of the quantity and quality of the dataset produced by the scripts and then moving to the analysis of its contents. The chapter begins by introducing statistics on the usage of various ENS subsystems and then transitions to an examination of the crawled Web3 websites. It first analyzes the actual locations where the websites' content is hosted, then scrutinizes their connections to the Web2 ecosystem. The chapter concludes with insights into the technologies used in developing these websites. Throughout these sections, we examine how the technical limitations of Web3 create challenges for developers and explore the strategies they employ to overcome them.

The thesis concludes with a synthesis of findings in Chapter 7, reflecting on the research results. Finally, Chapter 8 points toward potential future work, suggesting areas for further exploration and improvement based on the insights gained throughout this study.

Chapter 2

Background

This chapter introduces the main concepts needed to understand the work presented in this thesis. More precisely, the chapter opens with a brief introduction to blockchain technologies, what they are, a brief description of how they work, and the problems they try to solve. Then, Section 2.2 outlines the main features of the Ethereum blockchain, focusing on the essential information about smart contracts and how they interact with off-chain applications and vice versa, as well as some notes about smart contract storage.

The chapter then moves on to introduce the Ethereum Name Service (ENS) in Section 2.3, describing the problems it was designed to solve, its features and how it works, the types of smart contracts that implement its functionality, and a brief history section listing the changes and additions made to the overall system.

Finally, Section 2.4 introduces the main concepts behind decentralized file systems, focusing on the InterPlanetary File System (IPFS) and how it can store and retrieve data in a decentralized network.

2.1 Blockchains

Blockchains are distributed ledgers in which *transactions*, digitally signed records that transfer data or value between participants, are stored in blocks held and computed by nodes in a peer-to-peer network. They are immutable, meaning that once a transaction is added to the blockchain, it cannot be removed or modified. This is achieved through the use of cryptographic hashes, which link the constituent parts of the chain together, and consensus algorithms, which ensure agreement on the state of the blockchain between nodes in the network.

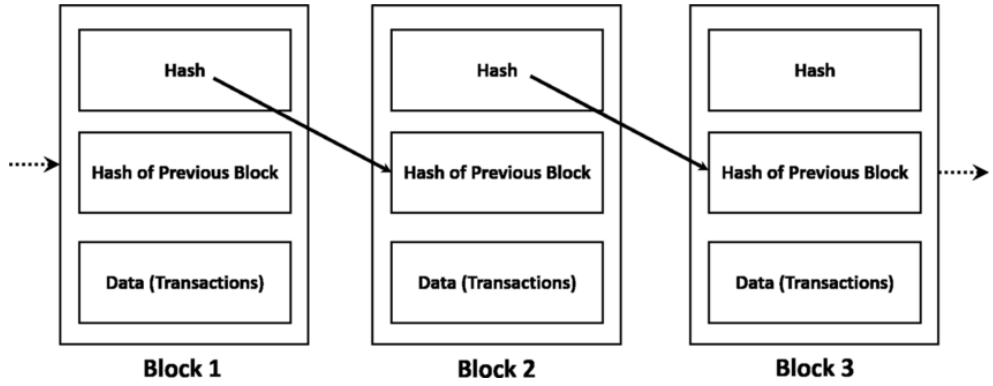

Figure 2.1: Simplified structure of a blockchain.
Source: Bosu et al.[4]

Blockchains take their name from their structure: transactions are stored in blocks linked together in a chain. Each block contains the previous block's hash, making it impossible for anyone to alter the history of the blockchain as long as the hashing algorithm is cryptographically secure. A simplified blockchain structure is shown in Figure 2.1.

Users who focus on creating blocks are called miners or validators, depending on the chain and the technologies used. Successfully adding a block to the chain awards the user that made the block with a prize, which is usually a combination of a base reward and the transaction fees of the transactions included in the block, included by the users to encourage miners to add their transactions to the block. In some cases, like in Bitcoin [5], the fixed reward is halved every few years and is the only way to add new currency to the network.

Miners must solve a cryptographic problem to create a valid block. The cryptographic problem solved by miners differs from chain to chain. Still, the characteristic they all share is that the challenge must pose rules that make creating incorrect blocks undesirable, and the validity of the resulting block must be easy to verify. The specific details about the challenges used change based on the type of blockchain and often dictate the speed characteristics of the chain, which also influences the overall economy of the tokens exchanged in the transactions. In chains like Bitcoin, where the difficulty of its cryptographic challenge automatically regulates the speed at which blocks are added to the chain to around one block every 10 minutes, users may be encouraged to add higher miner fees to have their transactions processed before others', while Ethereum has a much higher frequency of blocks being emitted, on average one every 12 seconds. Blockchains usually have further methods for discouraging the creation of incorrect blocks, along with the network's reaction to a malicious block, such as not including the malicious miners' transactions in following blocks or not accepting their blocks and rendering their future work useless.

As mentioned above, the consensus algorithm used differs from chain to chain. Take, for example, two of the most used blockchains: Bitcoin uses Proof-of-Work (PoW), where each miner is tasked with finding a hard-to-find nonce to make the first bits of the hash of the block be zero, while Ethereum uses Proof-of-Stake (PoS), where validators stake some of their ETH when they announce a block, and lose the stake if the block is incorrect. Ethereum used to task miners with similar hard-to-solve problems to Bitcoin, but since it completed its migration to PoS [6] on September 15, 2022, it saw a reduction in computing power used, carbon footprint, and emissions by up to 99.99%, as to a Crypto Carbon Ratings Institute report [7].

Blocks usually do not store the state of the blockchain at the time of their creation. Instead, the blockchain acts as a state machine with an initial state and uses the transactions stored in the block as instructions for updating the state. Traditionally, blocks only contain some metadata about themselves and the list of transactions ordered in time. Each node must apply the transactions in the order they appear to the initial state until they reach the state at the desired *block height*, the ordinal number of the block starting from the base state. Blockchains need to be closed systems to behave like deterministic state machines, meaning that they must store all the information that can cause a change in their state; otherwise, reconstructing the chain in different environments, or even different moments in time, may lead to different final states, which is inadmissible.

Since most blockchains are public and do not have a central authority overseeing them, their contents must be accessible to anyone for validation. Anyone with the necessary computing power and storage to fetch the blocks and decode the data can gather and aggregate the information they contain.

Depending on the chain, the data stored in the ledger may be mainly transactional, like in Bitcoin, or it may contain more complex data, even to the point of containing programs written in Turing-complete programming languages, like the smart contracts running on the Ethereum Virtual Machine (EVM).

Entering a blockchain network is usually straightforward and only requires creating an address, typically derived from the user's pair of public and private cryptographic keys. Addresses are used to send and receive transactions, and the only way to authenticate those transactions is by using the private key from the pair that generated the address. Since no central authority governs the blockchains, losing access to the private key means losing access to all the cryptocurrencies owned by the address.

Each peer can generate as many key pairs and addresses as they want. Users of blockchains such as Bitcoin may want to create a key for each transaction to anonymize them and then use ad-hoc solutions such as mixers [8] to reduce the chances of those addresses being linked to a single system user.

2.1.1 Layer-2 Solutions

As mentioned above, blockchains prioritize security at the cost of slow transaction throughput. Layer-2 (L2) solutions aim to solve this issue by operating on top of Layer-1 (L1) blockchains such as Bitcoin and Ethereum. This is usually done by performing most of the computational work on L2 while relying on L1 chains for storage and to settle consensus disputes. This allows L2s to offer faster and cheaper transactions while inheriting the security of the underlying L1 networks.

L2 blockchains achieve higher throughput by assuming that malicious behavior by their users is infrequent. Thus, they optimize for the usual case in which everyone plays by the rules while still being able to rely on L1 chains to solve disputes in case blocks are deemed incorrect by some part of the network. They also try to minimize the chances of false disputes by having all parties involved in the dispute stake some currency, slashing the losing party's staked currency.

In general, L2 blockchains show increased throughput and significantly lower fees, making blockchains more cost-effective for users than their L1 counterparts.

2.2 Ethereum

Ethereum is a decentralized and open-source blockchain that makes the execution of programmable smart contracts one of its main features. It was first described in 2014 by a whitepaper authored by Buterin [1], and later deployed on July 30, 2015.

The main currency used in the Ethereum blockchain is Ether (ETH), and its smallest unit is the Wei, but the blockchain supports the creation of custom tokens, as explained in Section 2.2.3.

Ethereum has an account concept, which can be either externally owned or a contract account. Externally owned accounts are controlled by private keys, while contract accounts are controlled by the code stored in the blockchain. Both account types have a balance, meaning they can receive, hold, and send Ether and tokens. Furthermore, both types can interact with smart contracts.

Only externally owned accounts can issue transactions, which are signed instructions from accounts. A transaction can be of one of three types: a transfer of ETH from one address to another, a contract creation, in which the externally owned account issues a transaction towards a specified address, passing in the compiled source code as data, or finally, the smart contract method call.

Ethereum is open-source, and the community drives its development. Changes to the

protocol are proposed and discussed via Ethereum Improvement Proposals (EIPs). They are later implemented into the Ethereum blockchain as smart contract standards or modifications to the tooling around the Ethereum nodes. Similarly, Ethereum Requests for Comment (ERCs) are proposed standards for smart contracts and tokens, respectively discussed in Section 2.2.1 and Section 2.2.3.

2.2.1 Smart Contracts

Ethereum expands on the base idea of a blockchain by focusing on monetary exchanges and allowing the creation, deployment, and interaction with smart contracts, which are the standout features of the Ethereum blockchain.

Smart contracts are pieces of code whose functionality is immutable and completely determined by the source code deployed on the blockchain. They are usually written in the Solidity programming language but can be written in any language that compiles to Ethereum Virtual Machine (EVM) byte code. Since EVM byte code is a Turing-complete language and all nodes in the network execute each transaction, the Ethereum blockchain can also be considered a "world computer" and "the slowest computer in the world."

Gas is the unit used to measure the computational work done by the EVM and is computed using the cost of each byte code operation to avoid infinite loops and spamming behavior. The *gas fee*, usually expressed in Wei, is calculated as the total gas used multiplied by the gas cost, which changes based on the computation performed in the previous block, regulating the amount of work done by Ethereum nodes while still retaining predictability for the users. Additionally, users can include a *priority fee* to the validators to encourage them to include the transaction in a block. Gas fees must be included in a transaction to be considered valid, and they are not returned to the transaction's initiator if the computations fail.

Smart contracts can expose functions that any blockchain user can call by issuing a transaction. Each smart contract function may be payable, meaning that ETH could be required to perform its operation. Even if a function is non-payable, calling it requires paying a gas fee unless the function is marked as `pure` or `view`, meaning that it does not use or change the smart contract storage, respectively. The output of a function call can change the chain's state or be returned to the caller, but only the inputs are stored in the blockchain, in the transaction's call data.

Each smart contract is isolated from all the others, meaning they do not share any storage. If a bug is found in a contract or a new feature needs to be added, a new contract must be created and uploaded to the blockchain, but the old one cannot be removed. This means some users may be led to use the old vulnerable contract by a malicious actor. This issue can be alleviated using naming systems such as ENS, described in Section 2.3.

```
event NewResolver(bytes32 indexed node, address resolver);
/* ... */
emit NewResolver(node, resolver);
```

Listing 1: Event definition and emission example in Solidity, extracted from the ENS registry contract.

Since the complete history of the blockchain is stored in its blocks, it is impossible to remove some information from it completely. A smart contract may offer an overwrite or delete functionality to change the current value of a field or flag in the current state, but the old values will always be accessible.

2.2.2 Events

Smart contracts accept inputs from the outside world via transactions and may return data via their response. While executing functions, smart contracts can emit events that enable them to communicate with frontends or other applications subscribed to them. Events are named data types that may contain data and topics. Events may contain up to 3 fields marked with the `indexed` attribute, adding them to the topics and allowing them to be used as filters while the other fields are stored in the data. Emitted events are stored in the blockchain, while their definition also appears in the contracts' ABI (discussed in Section 2.2.3.1), enabling the decoding of their contents by off-chain applications. Event data is ABI-encoded [9] and must first be decoded into other languages' native types.

In Solidity, events are defined with the `event` keyword and can be emitted with the `emit` keyword, as shown in Listing 1. They are usually defined in the first part of the smart contract's code before function definitions.

Events are identified by their signature, defined as the Keccak256 hash of a simplified version of their definition. The signature of the `NewResolver` event defined in Listing 1 is the Keccak256 hash of the string `"NewResolver(bytes32,address)"`.

2.2.3 Token Standards

The existence and focus on smart contracts have led to the standardization of interfaces that smart contracts must implement to create fungible and non-fungible tokens (NFTs), respectively defined in ERC-20 [10] and ERC-721 [11]. NFTs, unique by definition, collectible, and tradeable by standard, are known outside of the blockchain world because of

the high cost at which they are sometimes exchanged [12]. In general, ERC-721 tokens represent something unique, like a piece of art, music, or a special item in a decentralized videogame. Their uniqueness is guaranteed via a unique identifier stored on the blockchain. The NFT can also store metadata about the asset, such as a content identifier of the asset on IPFS.

On the other hand, each instance of an ERC-20 token is indistinguishable from the others and thus has the same value. Fungible tokens have many uses, like creating one's personalized coin or minting coins to be used internally in a decentralized application. One of the most common uses of personalized coins is their role as voting power inside Decentralized Autonomous Organizations (DAOs), which are member-owned communities. In these communities, each user can use the DAO-specific token to vote on the future of the DAO or the resources and systems it owns. DAOs are often controlled by smart contracts. A bug in one smart contract controlling the organization "The DAO" caused the infamous *Hack of The DAO* [13], in which a hacker was able to steal many of the DAO's resources by abusing an exploit, which ultimately led to the birth of Ethereum Classic after a hard fork [14] in the Ethereum network. The Ethereum Classic blockchain follows "Code is Law" [15] under all circumstances, while Ethereum decided to revert the hack and return the stolen funds to their previous owners.

Both standards have an issue in common: a smart contract that implements one of those standards can only control one type of token. This can be a problem when a developer wants to create many fungible tokens for different factions in their videogame or when they want to create many new ERC-721 art collections, each with its metadata attached to the tokens. ERC-1155 [16] is a standard introduced to address exactly the issue: it combines the functionality of the two types of tokens, allowing the developer to mint both fungible and non-fungible tokens in the same smart contract, allowing each of them to have their properties and metadata, while only deploying one smart contract.

2.2.3.1 ABI and Interaction Between Smart Contracts

When a smart contract is compiled, the compiler generates an Application Binary Interface (ABI). The ABI is a JSON-encoded standardized structure used to specify how off-chain applications can interact with smart contracts by listing, among other properties, the functions they expose with input and output parameter types, events, and the number and type of the data they carry. ABIs are used by off-chain callers to know how to encode the data sent and received when communicating with on-chain smart contracts.

As mentioned in Section 2.2.1, smart contracts can interact between themselves. This has led to the creation of many standards defining how smart contracts should be written and how they should interact. In particular, if a smart contract needs to interface with a dynamically defined contract, the standard defined in EIP-165 [17] allows the caller

to check if the callee implements a specific interface by invoking its `supportsInterface` function with a value that uniquely identifies the interface. This allows the caller smart contract to verify that others implement a given interface and safely call one or more of its required functions, even if the callee address is unknown at compile time.

2.2.4 Oracles

Smart contracts are purposefully isolated from off-chain resources and data, allowing the network to reach consensus while considering only immutable data stored in the blockchain. If off-chain data was allowed to dictate blockchain state changes, then different nodes could read different off-chain values and could not reach consensus. Thus, the only tools smart contracts can use to interact with the outside world are emitting events and receiving input from a transaction initiated by another account since those methods store all data on-chain.

Oracles are off-chain systems that provide on-chain smart contracts with vital information, such as the live ETH to USD conversion rates. They are usually split between an on-chain contract and an off-chain oracle node. The on-chain contract emits events with the requested data parameters once one of its functions is called; the off-chain oracle node listens to those events, gathers the data from off-chain services and resources, and then pushes the response on-chain by calling a function on the on-chain oracle contract. In this way, all data needed to update the chain's state is stored on the chain, and consensus can be reached while not altering its complexity by bringing in external data sources.

Both on-chain oracle contracts and off-chain oracle nodes can be of different types: the first can be of a simple request-response type, where data is pushed only once when requested, or publish-subscribe, where the oracle provides a feed of information from outside sources, while the latter can be centralized, or be organized in a network of independent data sources to avoid a single point of failure, and to align with the on-chain values of decentralization. This is the case for the oracles exposed by Chainlink [18].

2.3 Ethereum Name Service

The Ethereum Name Service (ENS) is a decentralized naming system deployed on the Ethereum blockchain. It was introduced with ERC-137 [2] by Johnson in April 2016 and officially deployed on the Ethereum blockchain in May 2017. Over the following years, it has seen many scope, logic, features, and governance changes.

While Ethereum addresses are great to keep one's identity private, they are hard to remember and manage. Smart contracts or users not interested in the anonymity of raw

addresses may prefer being reachable by using more memorable human-readable names. ENS, like DNS and all other naming systems in their respective scenarios, was born to solve the abovementioned issue. Particularly in the smart contracts case, this allows developers to deploy new refined smart contracts without the old bugs and vulnerabilities, swap the address the ENS name points to, and users will enjoy the new contract features and stability without actively doing anything.

The system's core relies on a set of Ethereum smart contracts deployed and controlled by the ENS DAO, which is responsible for the system's governance. Resolvers, user-defined smart contracts containing information about the registered names, complete the smart contracts needed to understand and use the system. The system allows users to register and resolve human-readable names from the ENS native `eth` top-level domain (TLD), set reverse name records, and use some DNS names they own off-chain in ENS. Each domain can be traded, renewed, and associated with extra information, such as content hashes, profile picture URLs, and even custom integrations with other systems, like saving the users' X (formerly Twitter) name.

Registered ENS names implement the ERC-721 standard, making them NFTs. In particular, they are compatible with NFT exchange portals, and name ownership can be easily transferred. Names can also be wrapped in ERC-1155 tokens, unlocking finer-grained control over them, as explained in Section 2.3.4.3.

2.3.1 ENSIPs

Like Ethereum with EIPs, ENS has its proposal system called ENS Improvement Proposals (ENSIPs). Many features described in previous sections have been proposed as ENSIPs and later implemented as features after the proposal was accepted.

There are currently many open proposals, mainly focused on the scalability, performance, cost of interaction with the system, and interaction with other blockchains. Specifically, the ENS team has recently shared their plan to move part of the ENS infrastructure to L2 blockchains in the ENSv2 [19] update, securing lower gas costs, faster system performance, and exploiting trustless communication via CCIP Read [20] to enable the use of ENS names on other blockchains. Furthermore, they plan on employing a hierarchical registry system, giving more granular control over domains, as exemplified by the implementation and adoption of `NameWrapper`, whose functionalities are outlined in Section 2.3.4.3.

2.3.2 Basic Concepts

ENS relies on some basic concepts and names that need to be explained before diving into its architecture and functionality:

```
def namehash(name: bytes) -> bytes:
  if name == '':
    return '\0' * 32
  else:
    label, _, remainder = name.partition('.')
    return keccak256(namehash(remainder) + keccak256(label))
```

Listing 2: Sample namehash implementation in Python, as shown in EIP-137 [2].

- **Label**: a string that conforms to the normalization standard defined in ENSIP-15 [21], giving more options to ENS users as opposed to the previously adopted UTS-46 [22] standard (dictated by ENSIP-1/EIP-137 [2]), while still disallowing common spoofing methods such as inserting zero-width UTF-8 characters.
- **Domain**: a dot-separated list of labels. It is the full string users employ to register and resolve names.
- **Namehash**: the hash function used to encode the domains for use in the system, as indicated in the "namehash algorithm" section of EIP-137 [2]. It is a cryptographic hash function, meaning the inputs cannot be recovered from the output. Its high-level algorithm is presented in Listing 2.
- **Node**: the namehash of a domain. All ENS contracts internally use nodes to store information about domains, providing a unified and standardized key for their internal data structures. Namehash produces strings of a fixed length while still retaining their uniqueness.

2.3.3 Resolution Process

Name resolution in ENS is a multi-step process: the user must first find the resolver of the name they want to resolve and then query the resolver for the address at which the name resolves, as shown in Figure 2.2.

The resolution process is initiated by calling the `resolve` function of the ENS registry contract with the node relative to an ENS name, and the registry will respond with the address of the resolver registered for that name. Since the standardization of wildcard resolution with ENSIP-10 [23], the resolution process has been expanded: if there is no resolver set for a node, the user may try looking for the resolver of the parent domain recursively until it finds one, or the root node is reached, which stops the resolution process.

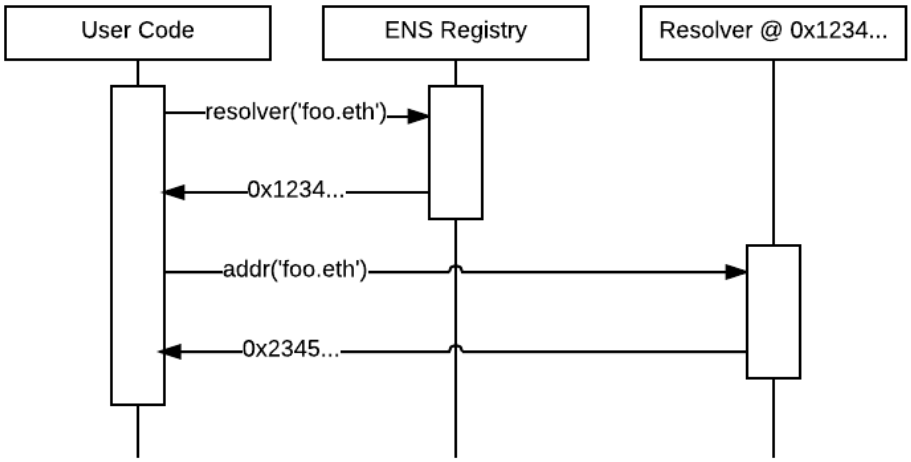

Figure 2.2: Example of ENS Resolution process.
Source: https://ens.readthedocs.io/en/stable/implementers.html

Suppose this step yields the address of a resolver. In that case, the user can invoke its `addr` function, at which the resolver will respond with the address of the address at which the name resolves if the name is valid. If users want to find other information about the name, they can invoke a different function on the resolver, such as `contenthash`.

2.3.4 Architecture

ENS relies on three main contract types to offer all its functionalities: the registry, resolvers, and registrars. The registrars are the most diverse and have three different architectures and implementations. The following sections describe each contract's exposed APIs, events, and changes over ENS's lifetime.

Note that all the tables presented in the following sections contain shortened addresses to fit in space constraints, but they link to their page on Etherscan, an Ethereum block explorer. Deployment dates are exact, as opposed to the month and year at which they stopped being commonly used, marked as *Stop*, that we extracted by roughly analyzing their usage trends. The *Stop* column should only be used to understand the transitions from one contract to its successor. Finally, the transaction count of each contract is shown in thousands under the *#TXN* column and can be used to roughly understand the feature and contract usage. All the data was collected when writing the work in September 2024.

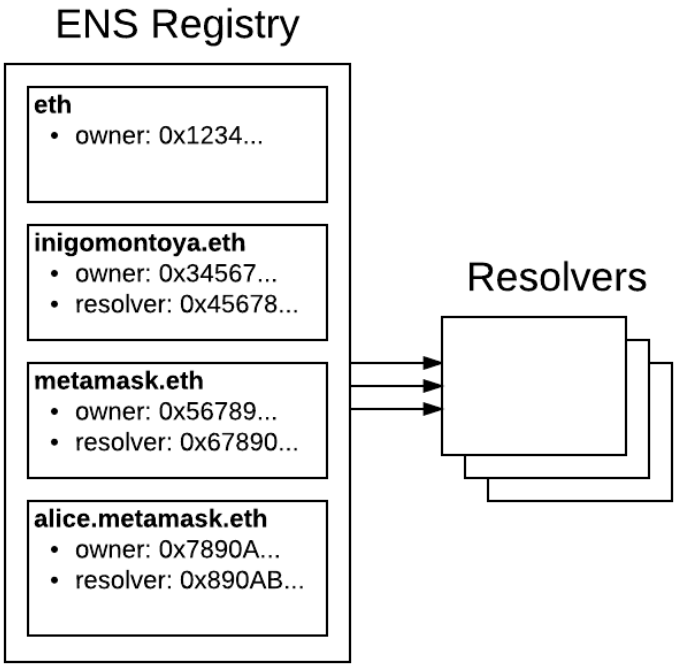

Figure 2.3: ENS Registry contents.
Source: https://ens.readthedocs.io/en/stable/introduction.html

2.3.4.1 Registry

The registry is the main contract at the heart of ENS, as it stores the mapping between nodes and the resolvers, owner, and caching Time To Live (TTL). It stores a large amount of data and is crucial both in the registration and in the resolution process, meaning that small changes to its API would bring breaking changes to the whole naming system, which is why it was designed to have the smallest and simplest interface possible, and the least abstractions, while still being able to store the mappings mentioned above regardless of changes to the overall system. This is in part allowed by the decision to use `namehash` as the hashing algorithm, allowing names to be stored as fixed-length cryptographic hashes regardless of the name normalization standard used in the system.

Its structure was first defined in EIP-137, and, as shown by Table 2.1, its implementation has been updated only once, when a wrapper that slightly changed the handling of the null address (i.e., `0x0000...000`) was deployed. As the name suggests, the wrapper contract only wraps the original contract with the changes to the aforementioned internal logic. A simple diagram of the registry and its contents can be seen in Figure 2.3.

The contract emits five events, four of which contain interesting information for our anal-

Address	Deployment	Stop	#TXN
0x314...59b	2017-3-10	2020-2	72
0x000...e1e	2020-1-30		220

Table 2.1: Smart contracts used as registry in ENS over time.

ysis: `NewOwner` and `Transfer` contain information about the node and the new owner, `NewResolver` carries the address of the resolver and the node it was assigned to, and lastly `ApprovalForAll` is emitted when a third party is allowed to manage **all** the records owned by an address. In that case, both addresses are contained in the event data.

2.3.4.2 Base Registrar

The base registrar controls the registration process of names that fall under the `eth` TLD. It is split into two contracts: the registrar and the controller. The former is responsible for name ownership-related operations like transfers, while the latter is responsible for registration and renewal. Since controllers can be dynamically added and removed, changes in the registration and renewal processes do not affect the registrar, which can be a smaller contract focused on exposing simple and basic APIs for the controllers to use.

The base registrar wasn't always split into two contracts. The first contract shown in Table 2.2 had no reference to controllers. It was meant to be used temporarily while other parts of the systems were being developed, as shown by the comments in the source code hinting at a future release in less than two years from its deployment. Furthermore, it implemented a method to transfer the current registrations to a future registrar implementation.

Name Registration and Renewal

The first registrar contract did not allow users to renew their names, and the registration used an auction system: when a user bid on a name, of which only names longer than seven characters were allowed, a *deed* would be created and stored on the chain. The deed represents the intent of the bidder's claim to the name and owns the staked Ether. At the end of the auction, the winner would be set as the owner of the name on the registry, while the other participants would receive most of their staked Ether back, and the rest would be *burned*. Burned Ether is sent to an invalid address and is thus removed from the system. In this context, this is done to disincentivize a monopoly over the bids by giving weight to each bid. When the name registration expired, the deed would return the Ether it contained to the old name owner.

The oldest registrar was deployed before the standardization of the NFT interface with

Address	Deployment	Stop	#TXN
0x609...8ef	2017-4-26	2020-2	1,997
0xfac...e04	2019-4-30	2020-1	12
0x57f...A85	2020-1-30		773

Table 2.2: Smart contracts used as base registrar in ENS over time.

ERC-721 in January 2018, while the second and the third comply with the new standard and wrap names in NFTs.

The second registrar, deployed in 2019, showed the two-contract structure mentioned above and implemented the functionality to accept transfers from the old registrar. Names transferred with that functionality were added to the current mapping, and the corresponding deeds were released, returning all Ether to the owner.

All the controllers use the new registration logic. First, names must be at least three characters long, and their price scales with their length, with three character names going for 640 USD for each registration year. The USD-ETH exchange rate is fetched at registration time via an off-chain oracle. The registration process is split into two phases: first, the user generates a commitment and stores it in the chain. A commitment is a hash of the name they want to register, their address, and a nonce they choose. After some time has passed, ideally after at least two blocks have been added to the blockchain, the user can reveal the commitment by providing the name and the nonce. If the name, nonce, and address match and the name is still available, the user is registered as its owner. The two-phase process is sometimes called a commit-reveal scheme and is used to prevent frontrunning: by storing an opaque commitment on the blockchain and then waiting for the chain to be built upon, the user ensures that their transaction is in a valid block. Even if the network does not accept the block, the user does not reveal any information about the name they want to acquire. If the hashing function used to create the commitment is cryptographically secure, malicious users cannot decode the name and the nonce used to generate the commitment and thus cannot submit a registration request for the same name.

The current contracts also allow renewals via a simple method. The address paying or requesting the renewal does not need to be the name's owner or an allowed third party. Furthermore, when a name registration expires, it enters a 90-day grace period in which only the old owner can reclaim it. When the grace period ends, anyone can buy the name. A rapidly decreasing premium is applied to the name for the next 21 days. The name will continue to resolve to the old owner until a new address claims the name.

Controllers

Table 2.3 shows some of the controllers used by the base registrar over time. Other than

Address	Deployment	Stop	#TXN
0xf0a...455	2019-4-30	2019-11	39
0xf0a...455	2019-11-4	2020-2	49
0x283...7f5	2020-1-30		4,942
0x253...03b	2023-3-28		1,069

Table 2.3: Smart contracts used as base registrar controller in ENS over time.

the `NameWrapper`, whose functionalities are outlined in Section 2.3.4.3, the controllers missing from the table below are the ones that do not contribute to our study, as they were only added for internal reasons, such as transferring name ownership from one registrar implementation to the next.

The first two controllers were used with the second registrar, as shown by the deployment and *Stop* dates shown in the table mentioned above, while the third and fourth ones were used along the latest registrar. Furthermore, the latest controller interfaces with the `NameWrapper` and was deployed on the same date while, thanks to the split into two separate contracts, the registrar contract did not need to be updated. Finally, the latest controller exposes an addition boolean flag on the `register` method that can be used to set the newly registered name as the addresses' primary name in the reverse registrar.

Events

The old registrar contract emitted some auction-related events. However, they are not worth exploring in detail as they do not provide more information about the names than what is logged by the registry with which the contracts interact. On the other hand, the new registrar contracts emit six events, all containing interesting information. First, `ControllerAdded` and `ControllerRemoved` are emitted each time a controller is added or removed. `Transfer`, `NameMigrated`, `NameRegistered`, and `NameRenewed` are each logged once the respective action is taken, and they contain the node on which the operation was performed, along with other useful information related to the action.

The controllers emit two events that carry interesting information: `NameRegistered` and `NameRenewed`. While the registrar contracts also emit these events, the controllers' version of the events contains the plain text name along with other data. Furthermore, the latest controller added the base cost and the premium used to register the name to the logs.

2.3.4.3 `NameWrapper`

The `NameWrapper` contract was first deployed on the blockchain on March 28, 2023. It was developed to improve the management of domains and subdomains by allowing greater

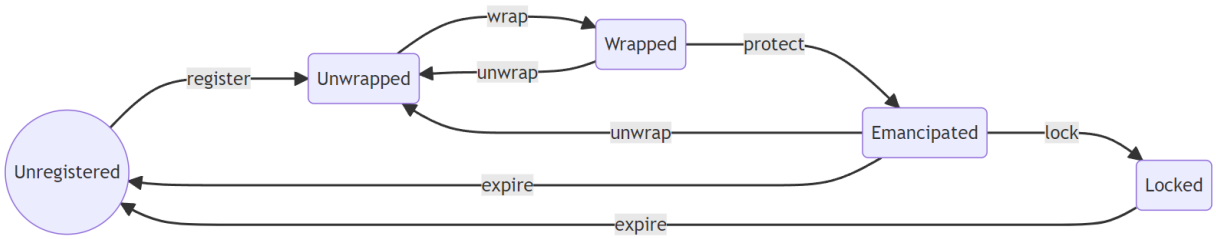

Figure 2.4: ENS name states.
Source: https://docs.ens.domains/wrapper/states

flexibility and control to the name owners.

Prior to the deployment of the `NameWrapper` contract, subdomains could be registered by calling the `setSubnodeRecord` on the registry, which would give the complete ownership of the subdomain to the new owner. This was a simple process, but it did not allow for fine-grained control over the permissions of the new owner on the subdomains. The `NameWrapper` builds on the ERC-1155 [16] standard, requiring system users to wrap their names in ERC-1155 contracts to unlock the new features. Specifically, wrapping a name or a subdomain allows its owner to lock certain actions and properties. This works by adding metadata to the name when wrapping it in an ERC-1155 token, encoding its *fuses*, flags that the owner can *burn* to disallow certain actions on the name before distributing it to its new owner. They control which perks, permissions, and guarantees to give the subname owners, like preventing further transfers, restricting the ability to change records, or the ability to create subdomains. In addition to fuses set by the standard, owners can create and burn up to 13 custom fuses. When the `PARENT_CANNOT_CONTROL` fuse is burned, the name is considered *emancipated*, meaning that its fuses can no longer be burned. The `CANNOT_UNWRAP` fuse must first be burned to burn the fuse mentioned above, disallowing subdomain owners from unwrapping the name and circumventing the restrictions posed by the burned fuses, rendering the name *locked*.

As the name suggests, burned fuses cannot be restored until the wrapped expiry is reached. For second-level domains of the `eth` TLD, the expiry is the end of the grace period discussed in Section 2.3.4.2, while for other names, there is no grace period, so the expiry coincides with the expiration date of the subdomain.

The contract was also added as a controller to the base registrar, allowing it to manage the names it wraps.

A comprehensive view of the states in which a name can transit is shown in Figure 2.4.

Finally, transitioning names to ERC-1155 tokens allows for a more efficient distribution of

Address	Deployment	Stop	#TXN
0x906...069	2017-5-29	2020-1	8
0x084...148	2020-1-30	2023-4	600
0xa58...7Cb	2023-3-28		87

Table 2.4: Smart contracts used as reverse registrar in ENS over time.

subdomains. The standard focuses on allowing multiple actions on different tokens in the same transaction while maintaining the wrapped name's uniqueness previously encoded by the ERC-721 standard.

2.3.4.4 Reverse Registrar

The reverse registrar, specified in December 2016 with EIP-181 [24], allows users to set up reverse resolution by claiming their subdomain of the special `addr.reverse` domain. Specifically, the address `0x<addr>` can claim the name `<addr>.addr.reverse` and optionally transfer its ownership to an address of their choosing. The reverse registrar also exposes a utility function that allows system users to claim their respective reverse name for themselves, setting its resolver to a default resolver and then setting the primary name record on that resolver to the ENS name passed as input to the function.

As Table 2.4 shows, the first reverse registrar was deployed in May 2017 in the initial release of ENS. The first version did not emit any events. Later, its code was updated at the start of 2020 with new Solidity language features, but the functionality remained the same. Lastly, the third contract deployed in 2023 added two events, and the logic of the registrar was updated to reflect the changes made to the overall ENS system. These changes are mainly reflected in reverse registrar methods allowing control of the names to contract owners, controllers of the name, and third-party addresses allowed to manage another address' names, and owners of the registrar being able to set the default resolver dynamically.

Even if the first two implementations did not emit events of their own, they interfaced and called methods on the registry, which emitted the events mentioned in Section 2.3.4.1. The latest version emits the `DefaultResolverChanged` and `ReverseClaimed` events, which contain the address of the new resolver, the address for which the reverse was claimed, and the node relative to the new name.

Address	Deployment	Stop	#TXN
0x587...ef2	2021-8-17	2024-1	1.4
0xB32...A2B	2024-1-17		1.4

Table 2.5: Smart contracts used as DNS registry in ENS over time.

2.3.4.5 DNS Registration in ENS

DNS names have been usable on ENS since 2021 when the first DNS registrar was deployed. Since then, a new registrar has been deployed, as shown by Table 2.5, solving the main drawback of the first one.

ENS does not support all the TLDs supported by DNS, and not all DNS TLDs use the registrars mentioned above, as shown by the official list.[1] All ENS-enabled DNS names must use DNS Security Extensions (DNSSEC), an added level of security that allows cryptographic verification of DNS records, as described in RFC-9364 [25].

Both implementations require the user to set a TXT record on the DNS name they want to claim with a specific format that lists the Ethereum address that will become the owner of the name on-chain. The first exposed an on-chain registration procedure via an on-chain DNSSEC oracle. While the claim itself was free, the verification of the claim on-chain proved to require much gas. Thanks to the advancements and the standardizations of wildcard resolution (ENSIP-10 [23]) and CCIP Read (EIP-3668 [20]), the second registrar interfaces with a new implementation of the DNSSEC oracle and can provide "Gasless DNSSEC": wildcard resolution is enabled at the DNS TLD level, and the DNSSEC proof is fetched off-chain thanks to CCIP Read, greatly decreasing the import prices, as the name suggests. It is worth noting that the old registrar can no longer be used, as the ownership of the TLDs has been moved to the new implementation.

Once a DNS name is registered on-chain, it behaves like all the other names and does not expire.

The main event emitted by both DNS registrars is `Claim`, which contains the plain text DNS name, the new owner's address, and the node. The updated DNS registrar's event also includes a field called `inception`, the Unix timestamp after which the DNSSEC signature used to verify the name can be used, as defined in RFC 4034 [26].

[1] https://docs.ens.domains/dns/tlds

2.3.4.6 Resolver

Resolvers are smart contracts that store the mapping between nodes, the addresses they resolve to, and the records set on them. Every resolver should implement the `addr` method, which returns the address to which the input node points. Resolvers have a concept of *profiles*, where each profile is an interface that defines one or more methods related to a record or a set of records and the events that resolvers should emit when their values are changed. Text and contenthash records, introduced respectively in ENSIP-5 [27] (and later extended in ENSIP-18 [28]) and ENSIP-7 [3], are examples of records controlled by profiles.

Resolvers can be freely implemented, deployed, and used by any ENS user, making them the most dynamic part of the system. This allows users to implement custom functionality or only the desired subset of profiles required for their use case. Even though the standard profiles defined by the ENS team indicate that implementors should emit events with a given shape and type, custom resolvers are not restricted to those shapes and event names. To this end, resolvers should implement EIP-165 [17], allowing interface implementation queries as described in Section 2.2.3.1.

Moreover, the introduction of ENSIP-10 [23], which standardizes wildcard resolution as discussed in Section 2.3.3, enables resolvers to handle subdomain resolution. Additionally, CCIP Read, outlined in EIP-3668 [20], establishes a secure and trustless method for retrieving data from off-chain resources, allowing resolvers to maintain records off-chain without compromising security.

The ENS development team deploys a public resolver that supports all the standard profiles each time one changes or is added. The public resolver stores data on-chain.

2.4 Decentralized File Systems

Decentralized file systems (DFS) are distributed file systems that allow users to store files in a decentralized manner. They are usually built on top of a fully decentralized peer-to-peer network, in which the responsibility of storing resources is distributed on all the nodes instead of relying on a central authority. Decentralized file systems strive to create a privacy-focused, resilient, and censorship-resistant system.

In these kinds of systems, data is owned by multiple nodes on the network and replicated among them to ensure its availability. Since no single entity controls one or a portion of the files in the network, it is much harder for a resource to be completely removed by the network, making DFSs particularly censorship-resistant. However, content on these networks is not usually guaranteed to remain on the network. Some resources may be

reachable only when their original publisher is online.

Furthermore, data on the network cannot be changed. This is usually achieved by addressing content not by location but by content. In particular, resources on these systems are addressed by content identifiers (CIDs), cryptographic hashes of the contents of the files, meaning that if a peer wants to change a resource, they need to upload a new copy, which will have a new CID by the properties of cryptographic hashes. Furthermore, addressing data by its content often allows decentralized file systems to deduplicate information stored on their network.

2.4.1 IPFS

The InterPlanetary File System (IPFS) is one of the best-known and most-used decentralized file systems. The IPFS protocol was introduced in 2014 by Benet with a whitepaper [29]. IPFS builds upon the ideas presented in Section 2.4: it is built on top of a peer-to-peer network, and it is designed to be a content-addressable, versioned, peer-to-peer file system, independent of a central authority or server.

More specifically, the concept of CID in the context of IPFS refers to a unique identifier derived from the data using cryptographic hashing functions. CIDs are encoded using multihash and multicodec standards, which ensure compatibility across different systems, versions, and cryptographic algorithms. These encoding standards, part of the Multiformats Project,[2] play a key role in IPFS's flexibility and extensibility, allowing different types of content to be referenced consistently within its ecosystem.

When a file is too large to be efficiently handled as a single unit, IPFS breaks it into more manageable chunks or *blocks*. Each block is independently hashed using a cryptographic hash function, and these hashes are used to identify and retrieve each block uniquely. The blocks' hashes are then organized in a Merkle DAG [30], and the hash of the overall resource is the root of that tree. When a user requests the original file, IPFS nodes can locate and reassemble the file by retrieving the corresponding blocks from across the network. This block-based architecture enables efficient data distribution, as different nodes can host and share different parts of a file without storing the entire file. Furthermore, Merkle DAGs easily translate to the storage of resource directories in IPFS, allowing complex file systems to be stored, accessed, and shared through the system: users can request an entire directory or individual files within it, and IPFS will retrieve the required blocks and reassemble them locally. This feature is useful for sharing and distributing large datasets, software repositories, or websites, where the structure and relationships between files must be preserved.

[2] https://multiformats.io/

IPFS nodes can require a lot of storage space to store the content they are responsible for. To address this issue, IPFS node implementations often allow users to request that all non-pinned data stored locally be deleted via garbage collection. IPFS nodes allow users to pin CIDs and their related content to their local storage, securing fast retrieval and ensuring the content remains available even if all the nodes that can serve those resources go offline, and stopping the garbage collector from freeing those resources. Pinning services, such as Pinata[3] and others, have emerged to allow users to outsource this responsibility, offering practicality at the cost of some centralization.

At the heart of IPFS lies its Kademlia Distributed Hash Table (DHT) [31], a decentralized, scalable data structure that allows peers to locate and retrieve content efficiently. When a peer requests specific content, the DHT maps the content's cryptographic hash to the peers that store it.

IPFS and blockchain technologies are naturally complementary, sharing similar goals of decentralization, security, and data integrity. Ethereum, a decentralized platform that supports smart contracts, benefits from IPFS by utilizing it to store large files off-chain while referencing these files within its smart contracts. One prominent use case of IPFS in Ethereum is through ENS. ENS records often reference IPFS content through contenthash records stored in resolvers, linking decentralized domains to files or applications stored on IPFS. This integration aligns with the decentralization goals of both systems: Ethereum provides a trustless, immutable ledger and an environment in which to store and execute the application logic, while IPFS ensures decentralized, efficient file distribution of both the dApps' web UI and blob storage. Accessing those dApps is made easier by IPFS gateways exposed by local IPFS node implementations or offered as online services, allowing users to visit dApps hosted on IPFS by treating them as Web2 websites.

2.4.1.1 IPNS and DNSLink

In some situations, providing an immutable reference to an immutable resource is not desirable, such as when the resource is highly dynamic and may change over time. In those cases, sharing a different IPFS CID each time the resource has been updated would be impractical. IPFS supports mutable addressing through the InterPlanetary Name System (IPNS) [32] and DNSLink.[4] Both were introduced to allow users to share mutable, secure pointers to immutable resources on IPFS.

IPNS is a mutable addressing system built on IPFS, allowing mutable references to immutable content in a fully decentralized environment. A name in IPNS is the multihash of a public key, and it is associated with an IPFS record. Records can be created at any time

[3] https://pinata.cloud/
[4] https://dnslink.dev/

by the holder of the corresponding key pair and contain the public key, a cryptographic signature signed by the private key, the content path they link to, and some metadata. Since they contain the public key and a signature, anyone can verify their authenticity, making them a self-certifying structure. This allows any node in the decentralized network to relay IPNS names, and the recipients can accept the names by verifying them independently.

ENS users often leverage IPNS to store mutable content behind their ENS name, avoiding updating the ENS contenthash record each time they update their resource and effectively avoiding the gas fees required to change the contenthash stored on the blockchain. However, this has some repercussions on the traceability of the resources linked to a name: while the blockchain stores each change in the contenthash record and each IPFS resource is immutable and always available (as long as there are online providers in the network), IPNS records do not store any historical data of the data ever pointed by them.

On the other hand, DNSLink bridges DNS and IPFS by exploiting DNS TXT records: setting the `dnslink` TXT record of the `_dnslink` subdomain of a DNS domain to a value formed by a namespace and a CID would allow IPFS clients that support it, such as Kubo,[5] to resolve `/ipns/<domain>` to the CID saved in the DNS TXT record. For example, using DNSLink to set the domain `example.com` to resolve to the IPFS CID `QmcniBv7UQ4gGPQQW2BwbD4ZZHzN3o3tPuNLZCbBchd1zh` would entail adding the DNS TXT record `dnslink=/ipfs/QmcniBv7UQ4gGPQQW2BwbD4ZZHzN3o3tPuNLZCbBchd1zh` on the `_dnslink.example.com` subdomain.

Although this is not inherently an issue, DNS cannot store a history of TXT records for each name. As a result, it becomes impossible to track all the records that a DNSLink name has previously referenced. This limitation is particularly significant in the context of our study, as it prevents us from observing how developers leverage the mutability of DNSLink and IPNS over time.

Both solutions allow IPFS content to be addressed securely by mutable pointers. However, while IPNS is a solution native to IPFS that only interacts with its decentralized network, DNSLink can be much faster by leveraging existing DNS infrastructure.

[5] https://github.com/ipfs/kubo

Chapter 3

Related Work

Decentralized naming systems have been the focus of many research papers published over the years, particularly after the advent of blockchain technologies. Zooko's Triangle [33], defined by Wilcox-O'Hearn in October 2001, describes an "impossible trilemma" that decentralized naming systems aim to solve but were unable to fully address before the advent of blockchains. Specifically, the problem lies in simultaneously offering security, decentralization, and human-meaningful names. Notably, works by Giamouridis, Kang, and Aniello [34] and Kalodner et al. [35] explore various blockchain-based naming systems, discussing their functionalities, benefits, and drawbacks. In cases like the Ethereum Name Service (ENS), security and decentralization are delegated to the Ethereum blockchain, while ENS only adds human-meaningful names. While the goal of a fully decentralized naming system is technically and functionally promising, our study focuses specifically on interacting with one instance, ENS, and analyzing a subset of the data it stores.

Xia et al. (2022) [36] provide an extensive overview of ENS, detailing its functions and smart contracts. Part of this thesis builds on their results by extending the historical analysis of the system, particularly in Section 2.3.4, while also updating the data through September 2024. However, outlining ENS history primarily frames the context for our study, which focuses on analyzing a specific subset of ENS data as discussed in Chapter 6.

Xia et al. also explored malicious Web3 websites associated with ENS contenthash records, identifying some through VirusTotal.[1] Additionally, they examined name-squatting behavior, a tactic involving the registration of names similar to well-known websites (e.g., "gogle" instead of "google") to deceive users. This behavior was further analyzed by Ke and Lau (2024) [37], who studied the prevalence of name-squatting and its correlation with domain resale prices. He et al. (2023) [38] expanded the research on malicious behavior in Web3 by developing a program that automatically detects phishing transactions, revealing

[1] https://virustotal.com

that phishing is indeed a commonly employed strategy. Although we are not concerned with the reasons behind users' domain purchases or the content of these websites, an interesting future research avenue could involve investigating whether malicious developers are more likely to establish connections with other dApps or Web2 services, expanding on both our and their work.

Pseudonymity is a key feature of blockchain systems, with many users valuing the ability to conduct transactions without linking their real-world identity to a blockchain address or one blockchain address to another. Maintaining this pseudonymity is important for users who wish to preserve their privacy or avoid unwanted attention. Deanonymization, however, remains an active area of research. As demonstrated by Jamwal et al. [39], deanonymization techniques can be applied to Ethereum even without introducing human-readable names.

The introduction of ENS names adds a layer of vulnerability to deanonymization efforts. For example, Béres et al. (2021) [40] showed how linking human-readable ENS names to addresses can be used to create heuristics to help compromise user privacy. This concerns many users, particularly those employing multiple addresses to avoid detection or correlation. In such cases, ENS names can reveal links between previously unconnected addresses, and transferring an ENS name between addresses can also lead to deanonymization. Furthermore, using multichain address records in ENS (introduced with ENSIP-9 [41]) may lead to deanonymization across multiple blockchains, which could be analyzed in future work.

While our work interacts with ENS ownership records, we aim not to deanonymize users. Rather, we focus on analyzing ENS records to study Web3 data without investigating the personal identities behind the addresses.

Web3 websites are a rapidly developing area of research, sparking numerous studies and technological innovations. Although the concept of Web3 is still evolving, many have attempted to define its scope and architecture rigorously. Prior studies have investigated the architectural characteristics of decentralized applications (dApps) deployed on Ethereum [42, 43]. While those works mainly focus on the definition of Web3 and the characterization of singular dApp architectures, we are interested in characterizing the network by analyzing a subsection of its contents.

Chapter 4

ENScraper

The information needed for this study can be gathered from two main sources: first, data about ENS names is collected from the Ethereum blockchain and inserted into a local relational database. Then, the contenthash information found in the result of the first phase is used to filter and crawl the web3 websites via an IPFS gateway.

This chapter includes information about the development process of the script that performs the first part of data collection. It opens with a survey on the data sources we considered using and the advantages and disadvantages of choosing them. Sections 4.2 to 4.5 then describe the choices, features, and problems encountered during the development of the program used to generate the output of the first data collection phase. Finally, Sections 4.6 to 4.8 close the chapter with a review of the overall structure of the program, an example execution flow, and possible future improvements.

The script, aptly called ENScraper, collects all the information related to ENS that can be found in the blockchain and that is needed for the analysis conducted in Chapter 6. This data includes registered names, associated records, and many more, as outlined in Section 4.7.

All the scripts used in this work were designed to be executed on a single computer with limited computing, storage, and bandwidth capabilities. Further, the scripts support being run multiple times to acquire historical data while considering the immutable nature of many resources to be gathered. Finally, they are designed to pick up where they were stopped in the previous run.

These constraints mainly influenced the data source used for the first phase of the data collection process, which shaped the implementation of the crawler, as described in Section 4.1.

4.1 Raw Data Source

As mentioned in Section 2.2.1, Ethereum smart contracts are immutable, and their storage is not stored in the blockchain as-is, but only the inputs to the functions that change it. Furthermore, each contract has storage, and updating its functionality entails deploying another contract with the updated business logic and storage. These facts make scraping the data we need from the blockchain hard: many versions of the smart contracts governing ENS have been deployed, as mentioned in Section 2.3, meaning that there is no central database used by all of them that we can scrape and map into our format. This led to the search for a suitable data source from which we could aggregate the needed information; in particular, we needed to find a service or dataset that would surface the required information or expose the essential calls for creating our own.

When deciding which data source to use, we identified four main strategies: using an existing dataset, downloading the blockchain and filtering data from the local copy, using an Ethereum light client, and using a service that surfaces the information we need. Using an existing dataset would have been the easiest option, as it would have eliminated the need for a custom program in this phase, if not a simple mapping script from the format of the dataset to our database schema. The main ENS Dataset[1] we found is the one used by Xia et al. in their work on ENS [36], and it was last updated in 2022, and thus we decided to consider other resources.

We ruled out the second option as soon as we conducted some preliminary research about the size of the blocks that we would need to download from the Ethereum blockchain, as the many terabytes of data from which we had to extract our dataset would not only overflow our storage capacity but would also take an enormous amount of time to download. Furthermore, computing the current state of the blockchain starting from the initial state would require much time and computational power.

The reasoning above led us to try Ethereum light clients, clients that are designed to be used by resource-constrained devices. This is achieved by only storing the headers of the blocks on the local device and requesting the information the client needs from other network nodes. Once and if a node responds with the data, the client can validate it using the information contained in the headers of the blocks. This seemed to be the perfect solution: we would have essentially unlimited access to **all** the raw data from the blockchain and could manipulate it however we wanted, without any request quotas or rate limits. Unfortunately, even though the official Ethereum documentation lists them as possible alternatives to full nodes [44], we found that they do not work after the merge [6].

Finally, the fourth option was analyzed. Upon a quick research, we found that Etherscan, the main Ethereum blockchain explorer on the web, offers a generous free tier for their web-

[1] https://github.com/ENSNames/ensnames

based API. The API surfaces many endpoints, including, but not limited to, endpoints used to invoke the RPC functions exposed by the common Ethereum node implementations. Specifically, the free API tier exposes an endpoint that can be used to get a paginated list of the events emitted by a smart contract, along with their data and some additional useful information, including the block in which they are stored and the timestamp at which it was published.

We decided to use the Etherscan API as our data source, as it was the most convenient and reliable source of information we could find. This approach is similar to the one used by Xia et al. (2022) [36], where they gathered information via event logs. However, before adopting this strategy, we ensured that all the smart contracts deployed after their work surfaced the information we needed through event records.

Furthermore, Etherscan allows users to verify the source code of the smart contracts on the blockchain by uploading the contract's source code, along with the compiler options and version used to compile it. Etherscan servers then compile the source with those settings. If the EVM byte code matches the one found in the blockchain, the source code is then published on their web UI and, most importantly for our use case, exposed via a dedicated API endpoint, along with the generated ABI of the contract. This feature is extensively used by `ENScraper`, as demonstrated in Sections 4.5.1 and 4.5.3.

Choosing to use the Etherscan API as our source of information did not come without its drawbacks, as technical issues in its implementation and outdated or outright wrong documentation sometimes led to a slowdown in the development of the scraper, as outlined in the next sections.

4.2 Overview

ENScraper is a program that scrapes data from the Etherscan API, collecting information about ENS via the events emitted by the smart contracts that make up the system. The program is designed to start from predefined addresses, event names, and logic to decide what to do when a result is returned from the remote API. This functionality is encoded in the `Job` interface, along with the logic required to interact with the many subsystems the scraper implements to ensure output correctness and completeness, resilience to unexpected errors, and compliance with Etherscan's API rate limits. Each instance of a `Job` can be uniquely identified by a string identifier and can dynamically create new jobs to be processed by the queue, making it so that each job's descendants build a tree.

A `Job` instance is responsible for exactly one request URL. Furthermore, the scraper discards jobs if a job with the same identifier has already been processed, which means that each URL should be contacted by exactly one job.

The scraper exposes a small public interface, namely composed of only two methods: the first, `AddJob`, takes a `Job` as input and schedules it to be run in the scraper's internal queue, and the other, `Run`, actually starts the scraper and a logging mechanism that periodically updates the user of the number of jobs active in the system.

The script is written in Golang,[2] a fast and lightweight language well-suited for highly concurrent programs, thanks to its simple yet effective concurrency primitives. Furthermore, the primary Ethereum client[3] and library[4] are written in Golang, allowing us to use stable implementations of the functions needed to decode data that originated from the blockchain, namely when decoding the data contained in the events, as mentioned in Section 4.5.2. Specifically, ENScraper implements a concurrent job queue, where each job runs on its goroutine, Go's native green thread implementation. The number of concurrent jobs is limited by a semaphore, used to avoid creating too many long-running goroutines at the expense of performance.

The output dataset is saved in an SQLite[5] database, a lightweight and easy-to-setup file-based relational database well suited for small to medium-sized datasets.

4.3 Remote Service Interaction

Understanding the APIs exposed by Etherscan, their parameters, and their outputs is important to understand better the decisions made during the development of the scraper. The Etherscan API[6] is a RESTful web service that accepts requests via HTTP GET and returns JSON responses. The API exposes many endpoints, but the ones used in this work are the ones that allow the retrieval of the events emitted by a smart contract and the ones that allow the retrieval of the ABI of a smart contract. The API is rate-limited, and the free tier allows for five requests per second, up to a maximum of 100k requests per day. The API requires the user to pass an API key via URL request parameters and always responds with HTTP status code 200 when it can process the request, while possible errors are encoded in the JSON body. To understand whether the body contains a result or an error, the user must check the `status` value in the decoded body.

The endpoint retrieves the events, accepts both the contract's address and the event's signature, and returns paginated data, ordered by ascending block number by default. The page returned by the API can be controlled with three parameters: `offset` controls the size of the page, `page` controls the page number in the results, with the constraint that

[2] https://go.dev/
[3] https://geth.ethereum.org/
[4] https://github.com/ethereum/go-ethereum
[5] https://www.sqlite.org/
[6] https://docs.etherscan.io/

`offset` $\times$ `page` $\leq 10,000$, and finally `fromBlock` dictates the first block from which the results will be returned. The successful responses sent from this endpoint only contain the response status and the list of events.

On the other hand, the endpoint that returns a smart contract's ABI takes a single parameter as input: the contract's address. It only returns the status of the response and the contract's ABI if it has been previously verified.

Each job implementation must provide a function that, given an Etherscan API key, returns the URL at which its corresponding request should be sent.

4.4 Features

As mentioned in Section 4.2, the scraper comprises subsystems that interact with the `Job` interface to ensure the result's correctness and performance of the overall system. The next sections outline the reasoning behind their existence, how they were implemented, how they interact with jobs, how their functionality can be customized by implementers of the `Job` interface, and finally, an analysis of the tradeoffs made in their implementation.

4.4.1 Rate Limit Compliance

As Section 4.3 reports, the Etherscan web API offers a generous free tier. In particular, it offers a limit of 5 requests per second. Semaphores are used to comply with these limits, allowing only five requests at a time for each API key in use. The slots in the semaphore are released after the reception of the API response and at least 1.5 seconds after their acquisition. The slots are released as soon as possible by making the critical section in which they are needed as small as possible to allow for maximum throughput while still complying with the limits. During development, we noticed that sending exactly five requests every second resulted in some spurious rate limit errors. The minimum ticket duration of 1.5 seconds in conjunction with the retry mechanism outlined in Section 4.4.5 ensures a low enough number of rate limit errors without degrading the program's performance and the output's completeness.

Furthermore, the scraper supports using multiple API keys to speed up the process and load-balances the jobs to always use the API key with the lowest number of active requests. All of the features mentioned in this section are implemented in the scraper logic and are completely opaque to the job implementations.

4.4.2 Caching

From the earliest stages of development, it was clear that a good caching system would drastically increase the speed at which the script was developed. Adding a caching system allows the script to almost completely forego the remote service and only read data from the local cache while still populating the database with correct data. The cache implementation used in this system is very simple thanks to the data immutability guarantees offered by the blockchain, making cache invalidation, which is usually the hardest part to implement in such systems, unnecessary.

The cache is implemented via a simple file cache, where each `Job` implementation can dictate the cache key and directory in which the response will be stored. Cache files contain the raw JSON response received from the API, meaning that changes to the internal representation of the scraped data would not invalidate the cache contents stored in a system created to shorten the time spent testing new changes to the code. This decision did not come without drawbacks, as storing the complete raw response meant that some storage space would be wasted on unnecessary data. The benefits of a caching system far outweighed the storage issues, as the script consumes most of the data in the responses, meaning that only a small part of the few gigabytes of cache files are wasted. Furthermore, saving the raw response means that `Job`s need to implement the ingestion logic for only one format, reducing the interface surface and simplifying the codebase and data pipeline through the system. Finally, reading requests from the cache does not require acquiring a semaphore ticket and thus allows other requests to contact the remote API, making the system more performant when the cache is available.

The cache is checked as one of the first steps in a job's data pipeline. If the response is not found in the cache or a spurious error stops the script from reading it, the scraper falls through to requesting the remote service. The cache is only written at the end of the pipeline. In particular, not all responses are saved to the cache: some are filtered out by the scraper itself, and the job implementations filter out others. The system filters out responses with HTTP status codes different from 200 and responses whose body contains an error. On the other hand, `Job` implementations can filter the responses in two ways: only a confirmed response, as defined in Section 4.4.4, can be saved to cache, but the implementer can also decide not to save the results, as explained for example in Section 4.5.2.

4.4.3 Progress

Thanks to the immutability of the data on the Ethereum blockchain, the scraper can be run multiple times to keep the dataset up to date while starting from where it left off in the previous run, thanks to the progress subsystem. It works when resuming a run stopped before the scraper could process all the jobs in the queue and when the program is run

multiple times to add new information to the dataset.

At the beginning of the data pipeline, each job has access to the database via the `ResumeProgress` interface function and can check whether previous instances of the same job have been run in previous program execution and resume the progress from where it was left off when the previous run was stopped. In the context of simple pages of contract events, this skips checks for files in the local cache. In contrast, when it is used with other events with more nuanced logic like `ResolverJob` (presented in Section 4.5.3), it can help to skip the considerable amount of requests to the remote service.

After the job's result has been added to the dataset, the job gains access to the database again in the `UpdateProgress` function, allowing it to save its current progress, if applicable.

Along with caching, this subsystem strives to minimize the requests made to the Etherscan API to increase throughput and decrease the quota of calls used. This allows other jobs to use the limited request semaphore slots.

4.4.4 Confirmation

During development, we noticed that some runs would end earlier than expected. Specifically, when checking the results in the database, we saw that sometimes the result contained around 3 million names, and others contained less than that. Running the script multiple times from scratch highlighted that this happened randomly: it would not happen all the time, and it would not always happen on the same request on the same events. Upon further inspection, we were able to pinpoint the problem: the Etherscan API sometimes returns fewer events than requested for each page, even if the service has access to them. Fortunately, we experimentally noticed that the bug in the service only happens with paginated results, such as events. Furthermore, we could also notice that while this issue causes the API to return incomplete data, it never caused it to return incorrect data.

This service implementation issue is problematic because the response sent from the remote API does not include a total page count that can be used to determine whether what was just returned is the last page or not. Instead, we rely on the number of returned events and stop scraping when we receive less than the required number of events. Randomly receiving fewer events than expected made the scraper stop early without trying to scrape the next page.

To solve this problem, we settled on a confirmation system where the individual `Job`s can define whether the result from a request is satisfactory or should be retried after some time. The timeout between requests is long enough for the remote service to respond with a freshly calculated result that hopefully contains all the events required. The job can decide whether the result is complete by comparing the last result with previous results

from the same request in the `CheckConfirmation` method on the `Job` interface.

While this approach showed great experimental results in detecting incomplete results and retrying the request when needed, we cannot be sure that the result returned by the API is complete. Periodically running the scraper helps reduce the chances of staying stuck on a specific false confirmation over time.

The addition of this subsystem has some drawbacks. Let n be the number of results needed for a job to decide whether a result is confirmed. First, the runtime of each job has increased by at least $(n-1)t$, with t defined as the timeout between confirmation requests in seconds. Second, in the worst case, adding this subsystem requires some jobs to make n times as many requests to the remote service as they would have made without it, with the best case being making $n-1$ more requests than without it. This price is most felt when scraping data from the ENS resolvers, as also explained in Section 4.5.3, as many of them may be used by only a couple of users, meaning that they may have emitted only a fraction of the logs that fit in a page, leading to n requests being made for a handful of logs of each kind of event. Unfortunately, we do not have any way of knowing how many events we should expect a contract to have emitted over its lifetime, making this blunt system the best solution we have found.

To avoid making all the extra requests this subsystem entails on each run, data must be confirmed before being stored in the cache so that subsequent program runs can assume cached data to be complete. Furthermore, since cached data must have been confirmed in some previous run, the system can skip confirmation checks when reading from the cache. Some optimizations were also made to have the highest throughput allowed by the system's requirements: since jobs do not need to be processed in a specific order, jobs can add their descendants even if the results still need to be confirmed. The scraper automatically keeps track of the jobs that have been started and stops them from being added after further confirmation requests are processed.

4.4.5 Retry

While `Jobs`' function to ingest data is invoked when data is found in the cache or a request to the remote service is successful, sometimes the Etherscan API may be unavailable, or the scarcity of bandwidth in the execution environment may lead to network errors, or as described in Section 4.4.1, the remote may respond with spurious rate limit errors. A retry subsystem was created to try to guarantee the program's resilience and the correctness and completeness of the output.

Since the Etherscan API always responds with HTTP status 200 when it can process a request, ENScraper automatically retries all requests that receive a non-200 status code. Furthermore, if the decoded body of the response contains an error, the resolver delegates

its handling to the `Job` implementation via the `OnRequestError` function. Through that function, the job implementation can opt out of the retry mechanism by handling the error with custom logic or letting the program retry the request. The job is retried multiple times up to a maximum number, after which the job is stopped, and the error is reported. During future scraper runs, the job will be resumed from where it left off via the other subsystems.

The timeout between retries of the same request follows an exponential curve that increases as the number of failed requests grows, plus an addition of a random number. The randomization decreases the chance that a group of requests that triggered a (spurious) rate limit error is retried at the same time, possibly triggering the same error again, while the exponential nature of the timeout decreases the number of requests being made, under the assumption that if the same request failed multiple times in a short amount of time, then it will probably fail again if it is retried after the same timeout as the previous.

4.5 `Job` Interface Implementations

We created three main `Job` implementations, designed to be mostly composable between each other and to be able to add and remove events to be scraped easily. The following is a brief description of their main uses and characteristics.

4.5.1 ABI

The `ABIThen` implementation retrieves the ABI of a smart contract and initiates the scraping of specified events by creating an instance of the `Events` job, described in Section 4.5.2, for each of them, provided those events are present in the ABI. It takes as input the contract address and a list of events to scrape. Once retrieved, the ABI is cached to avoid repeated API calls. However, the job never resumes progress as it must always create the `Events` jobs to ensure no data is missed in the final dataset.

While this job is not necessarily needed for contract addresses known at compile time, it is needed to correctly scrape events from base registrar controllers as they can be added and removed dynamically. We decided to use it for all contracts, as the number of extra calls made to the Etherscan API to retrieve the ABI of contracts whose address is known at compile time is low, and the requests are made only on the first run.

4.5.2 Etherscan Events

The `Events` job is responsible for scraping all events of a specified kind emitted by a contract by walking the pages of logs exposed by the Etherscan API. The job takes three inputs: the ABI of the contract, its address, and the event to be scraped. For each event collected from the remote service, the instance of `Events` may add other jobs to the ENScraper queue.

The job implementation delegates some event-specific functionalities to an implementer of the `EthereumEvent` interface, an in-memory Golang `struct` representation of the original emitted event. The interface requires three main methods to be implemented:

- `Save`: called when the wrapping `Events` job is tasked with saving data to the database.
- `Resumable` returns a boolean representing whether the event scraping process can be resumed. This is used to stop events whose scraping may result in adding other jobs to the queue to ensure the completeness of the produced dataset. Non-`Resumable` events may still be read from the cache.
- `FromRaw`: constructs an instance of the concrete event type using the topics and decoded data fields. The ABI-encoded data returned by the Etherscan API is transformed by a library that uses the ABI of the contract into native Golang types.

Furthermore, since the data contained in many ENS-related events changed with the evolution of the smart contracts that emit them, most of the implementations of `EthereumEvent` are built to be flexible and accept all the versions of the events that have been supported since their inception.

While the high-level functionality of the `Events` job is easy to understand, it is important to pay attention to edge cases and consider how this job interacts with the subsystems exposed by the scraper. First, results should only be cached if the length of the list of results returned by the API is equal to the `offset` parameter used in the corresponding request, whose usage is described in Section 4.3. Saving incomplete pages of events to cache would lead to the scraper always stopping at the same page without checking for new logs on the remote service. Second, `Events` jobs are resumable from the highest block encountered when scraping the same event on a previous run, but only if the `EthereumEvent` is resumable. This is mainly used by the `NewResolver` and `ControllerAdded` events, which add respectively an `ABIThen` and `ResolverJob` jobs to the queue for each returned event. If the events mentioned above were to be resumed from the previous progress, the jobs relative to some contracts would not be run. Thus, the output dataset would possibly be incomplete.

Then, since Etherscan poses a constraint on the product between parameters, as described in Section 4.3, walking the pages is not as easy as only incrementing the `page` API parameter. In our specific implementation, if the block height of the first and last events in the list are equal, then the implementation will add another `Events` job to the queue that maintains the same `offset` and increments the `page`. On the other hand, when the first and last events in the list are emitted in different blocks, the event resets the page number to one, and the `fromBlock` parameter is set to the block height of the last block in the list.

Lastly, this is the only job that meaningfully interacts with the confirmation system. Its `CheckConfirmation` deems a result confirmed if the length of the returned list of events is equal to the required `offset`. On the other hand, if the length is less than `offset`, the scraper tries the same job at least two more times. The latest result is confirmed if the length of the returned results is equal or monotonically increasing. If the list's length decreases at any point in the confirmation process, the list of previous results is cleared, and the confirmation process for the specific job instance starts again with only the current result (the one with the highest length) in the list.

4.5.3 Resolvers

A `ResolverJob` takes the address of an ENS resolver as input and is responsible for trying to collect its ABI and crawl the events emitted by standard resolver profiles if the resolver supports them. As this brief description suggested, its functionality is similar to `ABIThen`'s. The next paragraphs outline the reasoning behind the differences between the implementations of the jobs mentioned above.

As mentioned in Section 2.3.4.6, resolvers can be defined by users, meaning that their source code may not be verified on Etherscan, and events they emit may not conform to the standard profiles. `ResolverJob` implements custom error handling via the `OnRequestError` function described in Section 4.4.5, which allows it to assign the public resolver ABI to the resolver if its ABI is not verified on Etherscan. This allows the descendants of this job to gather the events emitted by the resolver if they conform to the standard or mark them as unrecognized if not. The resolver ABI is only cached if verified via Etherscan, allowing subsequent program runs to observe changes in its verification status.

When the job is tasked with adding its descendants to the ENScraper queue, it only adds the `Events` jobs for events supported by the ABI of the contract. Since the public resolver supports all standard profiles, `ResolverJob`s associated with resolvers with no verified ABI will try to get all the standard event logs. On the other hand, the job will not add descendants for unsupported events to avoid making unnecessary requests to the remote service. Furthermore, the `Events` jobs that are descendants of resolver jobs will be able to save whether the event emitted by the resolver contract conforms to the standard profiles.

Finally, resolvers are saved as completely scraped when their descendants are completed successfully. When a resolver job is tasked with checking if it should be run while resuming the process, it fetches the timestamp at which all its descendant jobs were completed in a previous run, and if they were completed recently, it does not run. While this means that the scraper could produce an incomplete dataset between runs of resolver jobs, this optimization also saves on many requests: most resolvers are rarely used and thus emit very few events, if any, meaning that at least three confirmation requests are made for each supported event each time a resolver job runs, just for the possibility of adding a handful of entries to the dataset.

4.6 Example Execution

The systems underlying the scraper's execution are relatively straightforward, but understanding how they interact can be challenging. To illustrate this process, we describe the execution of a single job and its descendants, highlighting only the subsystems with which they interact significantly. In practice, the system runs numerous jobs in parallel, many simpler than the example provided here.

Assume the crawler starts with an empty cache and dataset. The program starts from a set of hardcoded addresses and corresponding events to scrape. For example, consider a job tasked with scraping events emitted by the base registrar. The relevant hardcoded event is the `ABIThen` job, which includes one of the base registrars' addresses and the `ControllerAdded` `EtherscanEvent` implementation. When the crawler's `Run` function is invoked, the execution process begins.

The `ABIThen` job is placed in the queue and executed. It fetches the ABI of the base registrar from Etherscan, and subsequently, an `Events` job is added to the queue to handle the paginated `ControllerAdded` event logs. Finally, the ABI is saved in the cache. The confirmation system validates the ABI result immediately. Following this, the Events job retrieves event data via an API call since the cache is still empty for this kind of job, fetching up to 1,000 events from block 0 and page 1. Assume that the request is successful and that the response contains ten event logs. The job can immediately add subsequent jobs, as unconfirmed responses may be incomplete but not erroneous. Given that fewer than 1,000 logs were returned, the job is scheduled for a confirmation request after a delay to ensure the completeness of the dataset. Once the Etherscan API returns ten or more events multiple times, the result is confirmed, the `ControllerAdded` events are saved in the dataset, and the raw response is saved to the local cache. Once these tasks are complete, the job saves its progress to the database.

As described in Section 4.5.2, the `Events` job accepts a callback function to handle each scraped event. In this case, the function is used to add an `ABIThen` job for each controller

address found in `ControllerAdded` records, allowing the scraper to fetch the ABI of the controller to then scrape `NameRegistered` events on each one of them if they support it. These jobs are executed in parallel and are only added if the controller address has not already been scraped in this run of `ENScraper`. The `NameRegistered` events are fetched as described, and the process continues until all jobs are completed. After the scraping process, each `NameRegistered` event progress is saved, and when the program's queue is emptied, the program halts.

Consider a scenario where the scraper is terminated prematurely while scraping `NameRegistered` events for the first controller added to the blockchain. Assume each controller was added in separate blocks. Upon resuming, the scraper reads the base registrar's ABI from the cache and adds the `Events` job responsible for scraping `ControllerAdded` event logs to the queue. If the `ControllerAdded` event were configured as resumable, the scraper would skip the first controller's events, leading to an incomplete dataset. However, because `ControllerAdded` is marked as non-resumable, the scraper replays the process using cached requests, ensuring data completeness while avoiding redundant API calls to Etherscan. Progress data for non-resumable events like `ControllerAdded` is saved to the dataset for debugging purposes only and will not be read by the scraper.

In cases where a bulk restart of all `NameRegistered` events triggers a rate-limit error during the confirmation step for the latest page of results, the Events job employs the `OnRequestError` callback. This signals the scraper to retry the job after a specified timeout.

While the job described here is relatively complex, most other jobs involve simpler tasks, such as adding `ABIThen` and subsequent `Events` jobs. The most intricate cases involve `ABIThen` jobs combined with `ResolverAdded` and `ResolverJob` events, as discussed in Section 4.5.3, which include additional logic to handle specific cases.

4.7 Dataset Contents

ENScraper collects most of the data available on ENS. Specifically, it collects the ABI of all the contracts whose source code was verified through Etherscan and all the events that contain meaningful information about the system. These include names, their owners, their transfers, who can control which name, and all the records supported by all the resolvers used over the system's lifetime. Furthermore, each event in the dataset is associated with the metadata returned by Etherscan, namely the transaction that triggered the event, the block in which they are stored, the gas amount spent to execute the transaction, and the cost of gas at that time.

4.8 Future Improvements

`ENScraper`'s high-level functionality is easy to understand, as it only entails scraping information about ENS by creating a tree of jobs, where the most important jobs read paginated data. The scraper is designed to be easily extensible, and adding new jobs is straightforward. While the system was also deployed with simplicity in mind, and the current iteration of the program's implementation has already been changed multiple times to maximize the code's simplicity, performance, and readability, some changes would improve those aspects. First, the retry mechanism described in Section 4.4.5 could be greatly simplified by simply adding the job to the back of the queue when it needs to be retried, along with information about the number of the current retry, allowing for a more linear implementation of the business logic. Second, a similar change could be applied to the confirmation process overviewed in Section 4.4.4, rendering the worker thread implementation easier to follow. Lastly, as also described in Section 6.2.4, the database relational structure could be optimized to facilitate easier queries during the analysis of the results.

Note that while the changes to `ENScraper` outlined in this section would improve the codebase's readability and maintainability, some small sensible changes on the part of Etherscan would heavily simplify the logic of the scraper. In particular, allowing pages of events to be freely walked by increasing the `page` API parameter and adding a `totalPages` in the response body would provide us with vital information and ease of use, respectively, making the page-walking logic more linear and, most importantly, completely removing the need for the confirmation subsystem.

Chapter 5

Crawling Web3 Websites

This chapter focuses on the technical description of the second part of the dataset generation process. It begins by explaining the reasoning behind splitting this phase into two scripts, followed by a description of the data source in Section 5.1. Sections 5.2 and 5.3 detail the technical choices made and the challenges encountered during the development of the scripts.

Since resolving contenthashes can be time-consuming and the dataset produced by the first part of the data collection process is quite large, the second part is split into two programs. The first, called `preflight`, is a script that concurrently filters out unreachable or invalid contenthashes to avoid wasting resources trying to resolve unreachable CIDs in the more resource-intensive `P2` script, which takes as input the filtered contenthashes and recursively crawls the Web3 websites or resources they point to using multiple Google Chrome instances in parallel, producing the final part of the dataset that will be analyzed in Chapter 6.

We decided to split this part of the data collection process into two scripts after we tried running `P2` without pinning (described in Section 2.4.1) the resources first and noticed that most of the resources would not resolve before the processes timed out. One possible solution to that problem would have been to lengthen the timeout of the browser operations. However, the resolution process would have been limited to the number of browser instances running at that time, which in our environment is 15-to-20 browser instances running concurrently. Furthermore, resolving data from the gateway does not pin it to the local node, meaning that subsequent program runs would have possibly needed to wait for the resolution process. To maximize throughput in the resolution process and avoid wasting the limited resources available in our environment on a few browser instances, we implemented an additional script that could handle thousands of concurrent contenthash resolutions.

5.1 IPFS Data Access

Both `P2` and `preflight` need access to content stored on IPFS. The managed IPFS node hosting solutions we found did not completely fit the needs of the two programs, as some solutions would not offer a personal IPFS gateway to access the resources. Most would pose strict limits on pinned file storage, node bandwidth, and request rate limit in their free options, and the paid tiers were too costly for a service that is easily deployable in a local environment. After trying to run a local IPFS node, we found its resource usage manageable, even while running the other scripts.

A local IPFS node was thus needed to access the content stored on IPFS without any rate limits and to pin the content resolved by `preflight` to stop the garbage collection of the IPFS node from removing them from the local cache. Kubo[1] is the most used IPFS client, as mentioned in the IPFS documentation [45], and exactly fits our needs. It can easily be run locally and exposes a local IPFS gateway, allowing Web3 websites to be treated exactly as traditional Web2 websites by addressing them with contenthashes. The ability to access Web3 resources as traditional websites via a web browser allows the use of existing libraries and tools used in web testing in the `P2` codebase.

The ability to pin resources to a node is crucial, as it enables us to use a lightweight script for pinning. This allows later access from the cache without waiting for resource resolution again.

5.2 Preflight

`preflight` is a Golang script we wrote to map and filter the result of `ENScraper`, described in Section 4.7, to a list of contenthashes that contains only IPFS or IPNS CIDs that we were able to resolve and pin to our local Kubo node. The script is written in Golang to fully exploit the native and official IPFS Kubo libraries and Go's simple yet performant concurrency model while maintaining a low memory usage.

More precisely, `preflight` first filters out resources not stored in IPFS and invalid contenthash entries and then concurrently tries to resolve the remaining records. Resolution is performed in a highly parallel fashion to maximize the resources given to it for the whole program duration while limiting the concurrent requests made to the local IPFS node with a semaphore to avoid degrading performance. Each IPNS CID is first resolved to the corresponding IPFS record, and the association between the two CIDs is saved to the output dataset with the resolution timestamp. Then, if the script can resolve the contenthash, the corresponding resource is pinned to the local node. Pinning allows the resource to be

[1] https://github.com/ipfs/kubo

accessed instantly via the gateway by `P2` while preventing its removal from the local cache by the garbage collector when it runs to save some of our precious limited space.

By the nature of IPFS, many smaller Web3 websites' contents may be hard or even impossible to resolve and download since the small subset of nodes hosting their content could be offline at a given time. For this reason, the script has a configurable timeout for requests and can resume the script from where it left off in the previous run. This allowed us to retry to resolve names deemed unreachable by a previous run, building a more comprehensive dataset. Furthermore, the corresponding request is retried when an error is caught to avoid spurious errors caused by the flakiness of the networking capabilities available in our environment. The conjunction of these features results in a reliable and robust script that can run until a target dataset size has been reached while also allowing for stopping and restarting when needed.

5.3 P2

`P2` is the crawler we developed to visit the Web3 websites pinned by `preflight` and collect data about them. Thanks to the gateway exposed by Kubo, the websites can be visited as if they were traditional Web2 websites, allowing us to use the existing technology used for website testing and crawling. In particular, `P2` is TypeScript[2] program designed to be run on the NodeJS[3] JavaScript runtime. It uses Puppeteer,[4] a library used to control Google Chrome instances via the Chrome DevTools Protocol[5] (CDP), and Lighthouse,[6] a tool made to audit web page performance and security characteristics.

5.3.1 Overview

The program is built on a stripped-down and modified version of the `puppeteer-cluster` library,[7] which exposes a `Cluster` class that abstract a work queue where multiple tasks run concurrently on a configurable number of Chrome tabs. Our modified implementation mainly allows each task to be isolated on a different browser. Furthermore, during development, we found that many Web3 websites require users to install a wallet extension on their browsers to visualize their content fully. To this end, the modified library installs the

[2] https://www.typescriptlang.org/
[3] https://nodejs.org/en
[4] https://pptr.dev
[5] https://chromedevtools.github.io/devtools-protocol/
[6] https://github.com/GoogleChrome/lighthouse
[7] https://github.com/thomasdondorf/puppeteer-cluster

MetaMask[8] browser extension on each browser it creates and then proceeds to log into a MetaMask account that manages an Ethereum address before giving control of a browser to each task.

In particular, the modified library exposes a `Crawler` class that comprises a shared work pool containing information about the Web3 websites to be crawled, like the contenthash and codec associated with them, and a worker function. `Crawler` instances invoke the worker function with the parameters read from their internal queue and a reference to the Puppeteer `Browser` class, representing a Google Chrome browser instance. Since browser instances are quite computationally heavy, `Crawler` instances accept the number of max workers that can be active at any moment during their lifetime to avoid overloading the system. Each worker instance is given one browser instance to isolate each worker completely in case of browser failure.

Each worker function invocation is responsible for crawling a complete Web3 website. Each worker function has its internal queue that governs the breadth-first search (BFS) used to visit various website pages, building a graph of pages. The program has a set limit of pages to be crawled for each website and a depth limit. The worker first sets up the root database instance that groups all data about the current website and then starts the BFS loop. For each page to scrape in the queue, the worker starts a database transaction and creates a *task*, which collects information on that page, as described in Section 5.3.3. When the task terminates successfully, the worker adds the next pages to the queue, commits the transaction, and crawls the next page. If the task is completed with an error, it is retried up to three times. When a task is retried, the transaction is rolled back. In contrast, the transaction is committed if the last retry available for a task fails. The error is saved to the database.

Pages in the graph built by the crawler can be both pages found via links in the `a` elements in the page, of which only links pointing to the same domain currently being crawled are visited, and pages found as embeds via `iframe`s in the page. When an `iframe` is found on a page, extra information, such as its computed styles and bounding box, is collected and saved to the database for further inspection. Pages found in `iframe`s are treated as standalone pages and crawled in a separate task because of some technical limitations in the task pipeline: while Puppeteer allows developers to select and interact with elements in the embedded frame, parts of the crawling pipeline setup for tasks do not interact with `iframe`s, such as Lighthouse, whose functionality and use in the system is outlined in Section 5.3.3.3, which returns an error when running on pages that only contain embedded frames.

Note that the task depth is not incremented when adding `iframe`s to the internal worker queue because we want to crawl to the same depth on all websites if possible, and some

[8] https://metamask.io/

Web3 websites are just `iframe`s to another resource. A class of infinite `iframe` recursion is partly prevented by the HTML specification [46]; the rest of the possible issues are handled by the page limit per domain posed by the crawler.

Lastly, since the crawling process for the entire dataset is quite long, the program has a progress system that allows it to be stopped and started without losing the progress made in the previous runs. Specifically, once a task has completed all the pages in a domain, the domain is marked as completely crawled in the database, and the program skips the domain in the next run if it has been crawled recently. It is assumed that pages do not change in the small period between runs on the same domain. Technically, some domains addressed by IPFS CIDs could never change, as their source is completely stored on IPFS and does not load any Web2 websites via `iframe`s or redirects inside the HTML or JavaScript scripts. However, this progress system does not account for that optimization, as implementing it correctly would have been quite complex and would have taken more time than it would have saved.

5.3.2 Notes on Security and Privacy Features of the Browser

The crawler launches the browser instances with a custom set of flags that turn off most of the security and privacy features of the browser, allowing the websites to work mostly as they were designed to work, even if malicious, while disabling features like automatic downloads. However, as the official documentation[9] notes, there is no full list of all flags and switches that can be set on Chromium-based browsers, possibly because the project is very active, and many changes are made to the codebase each day. The lack of documentation mentioned above brought some problems during development: Google had announced [47] that they were going to deprecate third-party cookies on their browser, which led to the addition in Chrome v121 of a new Privacy Sandbox[10] switch to test the changes to the browser's cookie policy. We were aware of the planned changes, so we looked for a flag to turn off the feature on a newer Chrome version, in which the changes had already landed and were active by default. Any flag and switch we found that was supposed to turn off the feature did not work, maybe because of the high volatility of the command line interface surfaced by Chrome, so we resorted to using the latest version of Chrome For Testing v120, easily downloadable through the `@puppeteer/browsers` script.[11] Later, we found that Chrome was again changing the Privacy Sandbox and that third-party cookies would no longer be blocked by default [48]. However, the changes have not yet been implemented in the browser.

Note that modern browsers' security and privacy features have mostly been added for a

[9] https://www.chromium.org/developers/how-tos/run-chromium-with-flags/
[10] https://privacysandbox.com/
[11] https://pptr.dev/browsers-api/

good cause. All the scripts mentioned in this thesis were executed on a fresh installation of an operating system on one of our unused computers to avoid any personal information leak, or attack vector exploits while visiting websites without protection.

5.3.3 Task Pipeline

Each task in the pipeline collects information about a single page in a Web3 website. The task is composed of various steps, each responsible for collecting a specific type of information about the page. The following sections outline the main steps of a task in the order in which they are performed in the codebase. They highlight how they work and the data they collect.

5.3.3.1 Data Collection

The first part of the crawling process collects the data emitted as events by the Puppeteer `Page` class, an abstraction over a browser tab. Specifically, the task sets up the event handlers on the `Page`, responsible for collecting the data emitted by the `Page` instance. The events contain most of the information about dynamic events happening on the page, such as the requests made and the corresponding response received, the console messages and errors logged, the popups opened, and finally, the workers spawned. Each event stores information about the specific event, like URL, body, redirects, timing info for requests and responses, or the message in case of errors.

After event handlers have been set up, the page navigates to the URL of the Web3 website on the local gateway. The task waits for the page's Document Object Model (DOM)[12] to be stable, meaning that it has been loaded and client-side JavaScript code is not changing its structure. Then, the page's URL is saved again to account for redirects, and a screenshot of the page is added to the dataset. Timing information about the page load stages is also saved to the dataset, including the UNIX timestamps when the load started, when the DOM was first loaded, and finally, when the page was deemed stable by the crawler.

After that, the task collects the information in the page's storage, comprised of local storage,[13] session storage,[14] and IndexedDB.[15] Finally, the crawler collects information from the DOM, such as the `meta` tags and their contents, the URLs found in the anchor tags, and all resources loaded by the page and their respective source URLs, comprising `iframes`, `style` and `script` tags. Note that while only links in the same domain of the

[12] https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction
[13] https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage
[14] https://developer.mozilla.org/en-US/docs/Web/API/Window/sessionStorage
[15] https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API

initial or redirected URL are crawled, all of them are saved in the dataset.

5.3.3.2 Cookie Banner Detection

The study aims to determine if Web3 websites behave differently from Web2 websites, including whether they track users or follow the blockchain's privacy ideals. To this end, the crawler collects information about the cookies used in the websites and tries to understand whether they conform to General Data Protection Regulation (GDPR)[16] or they track the users without their consent.

To understand the cookies used by the websites, all the cookies set by the website are collected using the `Network.getCookies`[17] CDP call, allowing the collection of cookies with the `httponly` property set, which would not be possible by injecting JavaScript scripts in the page by the very nature of the property above.

First, all cookies set by the page are saved to the dataset. Later, different heuristics are used to find the cookie banner, if any, and to accept all cookies. To do this, all elements on the page with specified characteristics are enumerated and tagged with data attributes used to select them later in the process. Many different methods are used to collect and tag the elements on the page. Here are the methods and their associated name, listed from the most precise to the most blunt and generic:

1. **EnumCSS**: any element with common cookie accept button classes and IDs, such as `#onetrust-accept-btn-handler` and `.widget_eu_cookie_law_widget .accept`.
2. **EnumData**: buttons that have known cookie-related data attributes and contain specific text patterns, such as a button with the text "Accept" and a `[data-cookie]` attribute.
3. **AncestorData**: buttons that contain text that matches a specified pattern and has an ancestor with cookie-related data attributes.
4. **TextMatch**: buttons that contain text that matches a specified pattern.
5. **AncestorText**: buttons with an ancestor whose inner text matches a specified text pattern.

To avoid clicking on false positives, the search is limited to buttons only since they should be the most common elements used for such cases, and they should not lead the crawler to navigate to a different page in the process of trying to find the cookie button, as it would

[16] https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A32016R0679
[17] https://chromedevtools.github.io/devtools-protocol/tot/Network/#method-getCookies

likely happen with anchor elements. Furthermore, to avoid completely denying cookies with elements that match the selectors above, the text content of the elements identified by the heuristics above must not match a specified blocklist pattern that contains words commonly related to denying cookies, such as "deny" or "reject."

All elements from the second method onwards get discovered by running a slightly modified recursive depth-first search on the page. Specifically, this algorithm finds the deepest elements in each branch of the DOM tree of the page that match the CSS selector and the text filter, as shown in Listing 3.

Each tagged element on the page is then analyzed, starting from the ones in the most precise category and moving to the more generic ones. Specifically, first, the crawler gathers some information about the element, such as its computed styles, bounding box, selectors, and text content, and then takes a screenshot of the element. Puppeteer is then instructed to click on the element. Finally, some difference change detection algorithms are used to decide whether the element that was just clicked was the cookie accept button or not: first, the cookies of the page are once again collected and matched against the cookie set collected in a previous step. If the cookies have changed, the crawler assumes that the button was the cookie accept button. Otherwise, Puppeteer tries to find the element on the page, and if it can no longer be found in the DOM, then it is assumed to have been the element we were looking for. This is based on the assumption that most cookie buttons disappear from the page once clicked.

Lastly, suppose the candidate element is still on the page. In that case, the perceptual image hash [49, 50] of the page screenshot is compared with the perceptual hash of the previous screenshot. The perceptual hash is a hash of an image constructed so that the difference between two hashes can be used to measure how much the images differ. In this case, we look for a change in at least 10% of the hash's characters.

If the procedure mentioned above finds a cookie banner, its data is recorded in the final dataset, and the crawler proceeds to its next task. Otherwise, the next element is analyzed. The dataset contains information about the element clicked to accept the cookie policy, such as its text content, computed styles, classes, and attributes, along with its screenshot and the change detection algorithm that signaled the change in the page after clicking it. The screenshot of the page after clicking the element is also saved for future analysis. The cookies collected after the click are also saved, along with a flag indicating that they were collected after accepting the cookie notice.

Note that this step must run after the page's event collection has been stopped since clicking on elements in the page may trigger undesired events or navigate to other pages whose requests would pollute the dataset.

```
function findDeepestMatchingElements({
    selector,
    textRegex,
    avoidRegex,
    element,
}: {
    readonly selector: string;
    readonly textRegex: RegExp;
    readonly avoidRegex?: RegExp;
    readonly element: Element;
}): Element[] {
    // If the parent element does not match `textRegex`, then its
    // descendants will not either, so we can prune this part of the DOM.
    // Conversely, even if the parent element matches `avoidText`,
    // part of its descendants might not, so we only check that
    // condition before returning the element in the base case below.
    if (element.textContent === null
        || !textRegex.test(element.textContent)) {
        return [];
    }

    const result: Element[] = Array.from(element.children).flatMap(
        child => findDeepestMatchingElements({
            selector, textRegex, avoidRegex,
            element: child,
        })
    );

    if (
        result.length === 0 &&
        element.matches(params.selector) &&
        !avoidRegex?.test(element.textContent)
    ) {
        return [params.element];
    }

    return result;
}
```

Listing 3: The DFS algorithm used to find the deepest matching elements in P2.

5.3.3.3 Lighthouse

Lighthouse[18] is an automated tool and library built into Chromium-derived browsers designed to audit web pages and provide insights to developers for improving the performance, accessibility, search engine optimization (SEO), and best practice of their websites. The tool tries to mimic the user's behavior and environment as best as possible, generating a comprehensive report from which developers can extract precious information. Specifically, Lighthouse has an extensive built-in suite of audits evaluating different web page aspects. After an audit has run, the page is assigned a score for that specific audit, and the tool's insights are collected in an object that contains human-readable and machine-parsable information.

The crawler runs Lighthouse on each page and records the most important metrics and data the tool can extract. Lighthouse can find, infer, and aggregate information with audits that would be hard to implement from scratch: it has heuristics for performance indexes, it can measure the changes in UI over the load time of the page, it can analyze the source code to find the client-side libraries used by the page, whether it uses deprecated functions or libraries, security vulnerabilities, and more. The dataset contains the most important results extracted by its audits, such as some timing information about the page, its security vulnerabilities, and information about the providers of the resources loaded by the page.

Like the previous step, this part of the crawling process must run after page event collection has been stopped since Lighthouse tries to emulate the full user experience and thus loads the page from a blank page. If the event collection were still to be on, the dataset would include duplicated data such as duplicated navigation requests and redirects.

[18] https://developer.chrome.com/docs/lighthouse/overview

Chapter 6

Results

Our study aims to understand the state of the ENS ecosystem by studying the adoption of the features introduced in the last years and the quality and connectivity of the Web3 websites stored on IPFS behind an ENS name via contenthash records. This chapter presents an analysis of the dataset collected by `ENScraper` and `P2`, described respectively in Chapters 4 and 5. The chapter opens with a preliminary survey on the size of the dataset that will be analyzed in the next sections. In Section 6.2, a brief overview of the scripts' shortcomings and the errors encountered during the analysis process is provided. The chapter then continues with an analysis of some of ENS' key features in Section 6.3, concluding with our analysis of the data collected from crawling Web3 websites in Section 6.4.

All the data in this chapter is based on a dataset that contains data up to September 2024.

6.1 Parameters and Dataset Size

The first part of our dataset, collected by `ENScraper` from the Ethereum blockchain, contains all the information we could gather about ENS.

The dataset contains information about ENS names. Not all registrars used in the lifetime of the ENS system emit events or receive as transaction input the plain-text names, as some only operate on nodes, as mentioned in Section 2.3.4. In particular, we gathered the mapping between nodes and 3.2 million unique names under the `eth` TLD, 1.3k DNS names that were ported to ENS, and 375k reverse names. It also contains information about the transfers, which we will not analyze in this work. Finally, it contains information about the resolvers deployed on the Ethereum blockchain. Specifically, we scraped 4,950 unique resolvers and 6.8 million records set on them throughout the system's lifetime. Of those records, 53k refer to contenthashes for 36k unique hashes, as discussed in Section 6.3.2.

Parameter	Value
ENS names	3,221,226
DNS names	1,336
Reverse names	375,062
Resolvers	4,950
Resolver records	6,844,023
Contenthash records	53,441
Unique contenthashes	34,378
Resolved IPFS CIDs	3,543
Resolved IPNS CIDs	18
Crawled websites	3,543
Crawled pages	17,473

Table 6.1: Precise size information about the complete dataset.

The `preflight` script then tried to resolve 34k of them, of which 26k were IPFS CIDs, and 8k were IPNS CIDs. Over its runtime, the script resolved 3.5k IPFS CIDs, while we could not resolve almost all of the IPNS records, as described in Section 6.2.1. Note that the inherent nature of IPFS may introduce great variations in the dataset produced by the script. While we ran the script for multiple days to limit the issue, our bandwidth-constrained environment did not allow us to resolve more contenthashes. Nonetheless, being only able to resolve about 10% of the input dataset may also give us some information about the spread and popularity of some of the Web3 websites linked to by ENS names over the system's lifetime.

Lastly, the `P2` script was run on all contenthashes that `preflight` could resolve. It was run with a max depth of 3, meaning it was allowed to follow up to 2 links from the main page and a maximum number of pages per website of 20. The script ran for multiple days and produced a dataset containing data from about 17.5k pages across websites pointed to by 3,543 contenthashes. Note that the page count mentioned above could contain duplicated pages, as all websites are crawled independently. Thus, some pages may have been crawled multiple times if pointed to by multiple websites. In the next sections, we will remove duplicate information from those pages when aggregating data from different websites.

Precise information about the dataset size is shown in Table 6.1.

6.2 Script Performance Analysis

While the scripts we wrote to gather data mostly worked exactly as they were supposed to, we encountered some issues that we will discuss in this section. In particular, we will focus on the impact of the scripts' shortcomings on our analysis and how they can be alleviated or resolved.

6.2.1 Preflight Performance with IPNS

The `preflight` script, described in Section 5.2, worked perfectly with IPFS CIDs. However, we encountered some issues when resolving IPNS CIDs. Specifically, most attempts to resolve these records failed. After consulting the documentation[1] and reviewing the source code[2] of Kubo-the Golang library and local IPFS node we use for fetching, pinning, and accessing Web3 websites, as outlined in Section 5.1-we were unable to pinpoint the exact issue. This is because the error returned by the library is too generic and could be caused by a bug in either their or our code or an indication that the current nodes in the IPFS network could not resolve the IPNS records. We think that the error is caused by the last of those causes, as we could resolve some of the IPNS names we found, but further analysis must be done. If that is the error, running the script more often in an environment with better bandwidth may help.

Since the script could only resolve 18 IPNS CIDs of the 8k in total we found on the blockchain, we decided not to consider those in further analysis, as they would not give a good representation of the state of the Web3 websites stored on IPNS. The absence of such records in our dataset is particularly felt in the analysis done in Sections 6.4.1 and 6.4.2, where we cannot conclude in what measure Web3 websites behind IPNS records adopt the same techniques to show mutable data behind an immutable record in the Ethereum blockchain as IPFS Web3 websites.

6.2.2 Cookie Banner Detection

Detecting cookie banners was one of the most interesting challenges of this work. While not crucial to the main topic of our study, understanding the use of cookies in a network focused on privacy was an interesting addition. Unfortunately, our detection algorithm, described in Section 5.3.3.2, proved too generic with the definition of a cookie button, which resulted in many false positives. False positives were found by manually checking

[1] https://pkg.go.dev/github.com/ipfs/kubo
[2] https://github.com/ipfs/kubo

the metadata about cookie banners saved in our dataset and confirming their status by checking the screenshots we took of the pages and elements.

The algorithm detected 1,039 cookie buttons on the analyzed pages. While the `EnumCSS` element detection tactic proved effective in finding the correct elements, analyzing the results also shows that most of the pages in which those elements were detected belong to well-known Web2 websites such as linktr.ee[3] and Spotify.[4] This element detection strategy is the most precise. It produces no false positives, but it also misses many common cookie banners, meaning we must add more known CSS classes and IDs to make it more effective. The `AncestorData` detection strategy also produced no false positives. However, the same cannot be said about the `EnumData` and `TextMatch`. The main cause of false positives for those methods was the missed inclusion of more neutral terms like "manage" and "preferences" in the text patterns to avoid. Those words are usually paired with the word "cookie" on button elements, allowing users to set their cookie preferences more granularly. Lastly, no cookie banner button was found with the last element detection strategy, `AncestorText`.

While our cookie banner detection algorithm can be improved in numerous ways, there are three main ways in which we can increase its accuracy. The first is to rely on a more comprehensive dataset of known cookie banner button CSS selectors, while the second entails limiting the maximum change in the page via the image perception hash, assuming that usual cookie banners change the content of 10-to-30% of the page. Lastly, the page's screenshot could be analyzed with a detection algorithm powered by artificial intelligence to validate the presence of an element that fits our criteria or to check whether the bounding box of the candidate element contains a button with the sought-after characteristics.

Since the results show too many false positives, data about the cookie banners detected by our algorithm must be handled carefully. In particular, the analysis in which they are included must use data detected using a limited set of elements and change detection algorithms, as mentioned above. This means that the results may be incomplete. Furthermore, some analyses that fully relied on this system will not be shown in this work, such as the analysis on the number of cookie banners that are only used as a notice for the user to know that tracking cookies are being used, without the ability to deny their use and the difference of distribution of these between Web3 and Web2 websites.

6.2.3 Crawling `iframe` Elements

As mentioned in Section 5.3.3.1, the `P2` script was designed to crawl websites shown via `iframe` elements in the visited pages. This was done to analyze the content of the websites

[3] https://linktr.ee
[4] https://spotify.com

that store all or part of their content in a Web2 website and load its contents via an `iframe` element in one or more of the pages of the Web3 website. However, our implementation of such navigation is naive and does not filter the `iframe` elements it crawls. More specifically, we did not expect to find as many of such elements as we did, and this led to a dataset partially polluted by unwanted Web2 website information.

This problem can be solved in future implementations of the crawler by only navigating to `iframe` elements that are visible and have their computed width and height match those of the browser page while still saving the collected information about all other `iframe`s. This simple change allows the crawler to navigate on more Web3 pages while still storing which kind of `iframe` elements are found in the pages it visits.

Thankfully, in our search for a full and future-proof dataset, the crawler stores metadata about the `iframe` elements it finds in the pages it crawls, such as its computed CSS styles and visibility. These pieces of information are crucial to our analysis, as they allow us to filter the information about `iframes` that do not fit our more strict and updated criteria mentioned above. This does not fully solve the problem, as it renders analysis harder, as we have to filter out part of the data, and also makes the dataset smaller, as the pages found in websites pointed to by those elements occupy precious page-per-website slots.

6.2.4 Database Structure

While the data in the databases produced by the scripts is correct and complete, the database structure could be improved. In particular, even though some information can be derived by partially querying the databases and then processing its contents in a simple script, many of those transformations could be made easy with just SQL queries by altering the database structure. Furthermore, the P2 script could have benefitted from using a graph database such as Neo4j,[5] allowing more natural data representation. Adopting such systems would also have allowed us to conduct further analyses of the graph generated from the network of pages of the Web3 websites.

6.3 ENS Feature Adoption

The following sections cover the analysis of some of the additions to the ENS system over its lifetime, how they were adopted, and how they are used. We are particularly interested in the adoption of the new DNS registrar, how resolvers are deployed and used in the system, and lastly, the composition and distribution of the contenthash records saved in the resolvers.

[5] https://neo4j.com

	Address	Count
1	0x4976fb03c32e5b8cfe2b6ccb31c09ba78ebaba41	2,134,031
2	0x231b0ee14048e9dccd1d247744d114a4eb5e8e63	865,895
3	0xa2c122be93b0074270ebee7f6b7292c7deb45047	451,896
4	0xda1756bb923af5d1a05e277cb1e54f1d0a127890	70,696
5	0xf58d55f06bb92f083e78bb5063a2dd3544f9b6a3	27,653
6	0x5ffc014343cd971b7eb70732021e26c35b744cc4	15,922
7	0x1da022710df5002339274aadee8d58218e9d6ab5	12,340
8	0x0000000000000000000000000000000000000000	11,931
9	0x5913711b94f0e3c0fc933e400df94321436163fe	5,534
10	0x4da86a24e30a188608e1364a2d262166a87fcb7c	5,158

Table 6.2: Top 10 most used resolvers in our dataset. Public resolvers are highlighted in gray.

6.3.1 Resolvers

As mentioned in the introduction to this chapter, we collected data about the 4,950 unique ENS resolvers deployed on the Ethereum blockchain. In particular, Table 6.2 shows the top 10 resolvers associated with the most unique names at the time of writing, highlighting that the top 3 resolvers in use are public resolvers deployed by the ENS team, used by almost 3.5 million unique names.

The table also shows that many names use a null resolver. The names that still use it were deactivated using the `invalidateName` function on one of the first two base registrars discussed in Section 2.3.4.2 and were never re-claimed since. Other resolvers in the list are managed by Web3 projects such as Argent Wallet[6] and Loopring,[7] an L2 chain.

As mentioned in Section 2.3.4.6 and shown above, while most users adopt public resolvers, those resolvers make up a small percentage of the total number of resolvers. Most resolvers are thus user-controlled, and as such, they can choose not to implement standard profiles. Furthermore, resolver developers may not be interested in verifying the source code on Etherscan if their focus is not deploying the contract for public use. While we did not try to find custom resolver profiles, we crawled every event emitted by standard profiles and recorded whether the events were compatible with the ones used during ENS' lifetime.

In particular, we found that almost no resolvers support the now-deprecated profile that emitted the `ContentChanged`, superseded by `ContenthashChanged` introduced with ENSIP-7 [3], introducing a standardized contenthash format for the homonymous records. On the

[6] https://www.argent.xyz
[7] https://loopring.org

Profile	Implementers
Contenthash	4,388
Content	9
ETH Address	4,420
Multichain Address	4,388
Name	4,400
Text Record	4,409
ABI	4,388
Pubkey	4,396
DNS Record	4,366
Interface	4,374

Table 6.3: Resolver profiles and the number of their implementers.

other hand, while multichain address resolution was added in ENSIP-9 [41] and allowed users to set their address on multiple blockchains using a common encoding, many resolvers still decided to support the older resolver profile that only allowed for Ethereum addresses. Table 6.3 shows the number of resolvers supporting each standard profile.

Furthermore, we found that only two contracts emitted events that were present in some official profiles using an incompatible event signature. Both resolvers emitted a custom version of the `ContenthashChanged` event, which only contains the hash, while the address whose contenthash record changed is omitted. Upon further research, we found that three users only use those resolvers.

In general, the user base of ENS seems to have moved on from the deprecated content profile, and most of the resolvers in the system support most of the other standard profiles. Furthermore, most users rarely seem to need the features exposed by custom resolvers as they mainly use public resolvers, if not for the case of subdomains.

Even though most resolvers support most resolver profiles, the number of records ever set using them paints a different picture. Table 6.4 shows the number of records found for each profile in all the resolvers, highlighting that some profiles are rarely used while others have high use rates. Specifically, profiles like ABI and Interface, which provide standard ways to set the ABI of a contract and an alternative implementation address for an EIP-165 [17] compatible interface, respectively, are rarely used. In contrast, system users usually set addresses and text records for their ENS name. It is worth noting that some profiles allow the users to set more than one record for each type, such as text records and multichain addresses.

Profile	Record Count	Unique node users
ETH Address	2,958,407	2,647,901
Multichain Address	2,903,093	2,561,871
Text Record	464,426	239,777
Name	457,665	432,685
Contenthash	53,441	26,106
Pubkey	6,517	6,289
ABI	243	188
DNS Record	213	89
Interface	18	3

Table 6.4: Resolver records set using each of the standard profiles.

Codec	Count
ipfs	31,575
ipns	9,304
swarm	1,889
Skynet	67
onion3	52
onion	28

Table 6.5: Most used codecs in `ContenthashChanged` events over the lifetime of the system.

6.3.2 Contenthash

Contenthashes are crucial to this work, pointing to the Web3 websites crawled and analyzed in further sections. As mentioned before, and as defined in ENSIP-7 [3], the `ContenhashChanged` event supports multihashes, giving ENS users more flexibility when adding content identifiers linked to their names. In particular, as shown by Table 6.5, IPFS and IPNS are the most prominently used, while Swarm [51] and Skynet follow. Last are the `onion` and `onion3` codecs, which represent Tor websites, on which a job similar to ours has already been performed and published [52].

Swarm and Skynet are decentralized file systems, with the former focusing on integration with Ethereum dApps and the latter being a discontinued L2 built atop the Sia L1 chain [53]. In general, all contenthash records point to resources on systems focused on giving users privacy and providing censorship resistance for their content.

Address	Registrations	Deployment	Last	Avg TXN Price	
				Wei	USD
0x587...ef2	1,192	2021-8-17	2024-01-26	8.02×10^{16}	194
0xb32...a2b	144	2024-1-17	2024-09-21	3.21×10^{16}	102

Table 6.6: DNS registrar statistics. **Last** reports the last registration date in our dataset.

6.3.3 Gasless DNS

The newest DNS registrar was deployed in early 2024, and its goal was to greatly reduce gas fees by using a new registration process, as outlined in Section 2.3.4.5. The limited data we collected from the new registrar shows a 60% decrease in average transaction cost in Wei or about a 48% decrease in USD, as calculated using Yahoo Finance's ETH-USD historical data.[8] Note that the old registrar was released amidst the 2021 price hike [54], meaning that the decrease in USD price is not solely due to the change in registration method.

While this change was a net positive in making the system less expensive for its users, the data does not show the increase in registrations that we expected. In particular, as Table 6.6 shows, fewer DNS names have been registered on ENS per day since their inception.

6.4 Crawling Results

The next sections compose the heart of our study, in which we analyze some of the characteristics of Web3 websites, which services they make requests to, where they serve data from, whether they are connected, and lastly, some notes about their security, privacy, and performance. The results presented in these next sections are derived only from the dataset produced by `P2`.

6.4.1 Page Redirects

The spirit of Web3 is about breaking free from a centralized world and developing new applications and opportunities on decentralized platforms. We expected the Web3 websites we crawled to reflect those ideals, but unfortunately, what we found when analyzing the data was very different. In particular, we found that many websites use different methods

[8] https://finance.yahoo.com/quote/ETH-USD/history

Landing page	Content location	Count
True	Web2	960
True	Web3	2,207
False	Web2	9,383
False	Web3	2,992

Table 6.7: Distribution on the crawled pages between Web2 and Web3.

to show mutable data on their immutable website. As Table 6.7 shows, only 2,207 of the 3,167 pages to which the contenthashes resolve that responded with a 200 HTTP status code do not redirect to a Web2 website. If we expand our analysis to all the pages we crawled that responded with HTTP status 200, we notice that the situation is reversed, and only 30% of the pages visited are on IPFS due to the larger size of Web2 websites, as reported in Section 6.4.6.

Finally, we found that only 2,063 websites contain no pages that redirect to Web2. While this is a high percentage, we expected it to be higher, as the ENS system is built to allow users to point to their content and not to a centralized service.

6.4.2 Portals to Web2

HTML `iframe` elements enable developers to show content from another website. In the context of Web3, these elements often show content from a mutable source in an immutable website. As Table 6.8 shows, we found that the majority of `iframe`s (52%) were visible to the user. Visible `iframe` elements that cover the whole browser area are the last method used to use a Web2 website as part or whole website on Web3. In particular, we identified 17 such elements on a page pointed to by a content identifier, all found on the landing page of Web3 websites, meaning that they were serving solely Web2 content. This discovery reduces the number of Web3 websites that do not show full-page content from Web2 to 2,046 out of the 2,063 identified in Section 6.4.1.

While the most interesting use of `iframe`s in this context is their use as a way to show only Web2 content, we do not feel that other uses of such elements to show dynamic data from Web2 like embedding YouTube[9] videos or X[10] posts in page should be avoided at all costs by Web3 developers, as they allow to reference the source of the content they show in their websites, at the expense of interacting with a centralized censorship-prone network. In particular, Table 6.9 shows the ten most shown domains in visible `iframe` elements that do not cover the whole browser page in Web3 website pages that do not redirect to Web2.

[9] https://youtube.com
[10] https://x.com

Location	Size	Visibility	Count
Web3	Full-page	Visible	17
Web3	Partial	Visible	181
Web3	Partial	Hidden	120
Web2	Full-page	Visible	9
Web2	Partial	Visible	105
Web2	Partial	Hidden	171

Table 6.8: `iframe` element usage in different website contexts.

Domain	Count
youtube.com	109
soundcloud.com	20
google.com	14
app.uniswap.org	14
music.apple.com	11
lfg.news	9
twitter.com	7
audius.co	7
moca.show	7
spotify.com	4

Table 6.9: Most common `iframe` origins for Web3 pages.

Unfortunately, as mentioned in Section 6.2.1, we do not have enough data to conduct a similar study on IPNS records. However, we suspect that the usage of fullscreen `iframe` elements used to show Web2 content is much lower than on websites behind IPFS records.

6.4.3 Data Loading

As we try to understand the relationship between Web3 and Web2, we should also understand where Web3 pages load data from. Loading critical information from Web2 on Web3 websites means that while the Web3 websites themselves may be partially or completely controlled by code stored in an immutable decentralized file system and thus resistant to censorship and outages, the sources of information used to display its content may not be. This section mainly focuses on analyzing the requests made by Web3 web pages that do not redirect to Web2 or show fullscreen iframes to resources on Web2.

As Table 6.10 shows, the most prominent recipient of Web3 page requests is Google through its various services it provides like Google Fonts,[11] Google Tag Manager,[12] and Google Analytics.[13] The next two entries in the list are JavaScript Content Delivery Networks (CDNs). CDNs are services that offer sources for JavaScript libraries through their public pages and, as such, provide critical resources to Web3 websites. While Google Fonts and Google Analytics offer information that is not critical to the functionalities of websites, and not being able to load data from them may disrupt the expected styles of a website but leave its functionality untouched, an outage of the CDNs mentioned above may lead to websites not loading or working correctly. This is exacerbated by the reliance of modern websites on client-side JavaScript frameworks like React[14] to populate their pages' contents, as further discussed in Section 6.4.7.

The next reputable Web2 source worth noting in the table mentioned above is Pinata,[15] one of the most used services that provide IPFS RPC nodes and Web3 Ethereum nodes, along with Infura[16] and Alchemy.[17] Node providers are services that allow users who do not want or cannot run a local Ethereum or IPFS node to interact with the respective decentralized services by issuing RPC calls or HTTP requests. These services also allow users who do not have a wallet extension on their browser or cannot access a local node to access the website's content.

Further, if we restrict our analysis to the most critical information, removing requests for

[11] https://fonts.google.com
[12] https://tagmanager.google.com
[13] https://developers.google.com/analytics
[14] https://react.dev
[15] https://pinata.cloud/
[16] https://www.infura.io/
[17] https://www.alchemy.com/

Domain	Count
fonts.googleapis.com	495
fonts.gstatic.com	443
cdnjs.cloudflare.com	124
cdn.jsdelivr.net	115
www.googletagmanager.com	96
ipfs.io	93
pbs.twimg.com	72
gateway. Pinata.cloud	69
region1.google-analytics.com	67
www.google-analytics.com	64
www.google.com	54
www.gstatic.com	53
ajax.googleapis.com	49
play.google.com	43
www.youtube.com	42

Table 6.10: Top 15 most common remote URL domain Web3 pages load data from.

resources like CSS stylesheets and fonts, we find that out of the ones that do not show any content from Web2, only 1,516 pages and 445 of the domains load data from Web2.

Transferring data between Web3 websites should be fully trustless by reading and storing data on an Ethereum smart contract and leveraging blockchain nodes running on the client machine to interact with those contracts. Following the considerations made above about Ethereum node providers and keeping in mind that our crawler was run with a default installation of MetaMask, which internally interacts with the blockchain via a remote node provider, we can conclude that the websites that make no HTTP request do not read from the blockchain state without specific user input, as our program would catch those requests.

One possible improvement to better discover which Web3 websites do not read or write data from and to the blockchain would be to instruct the crawler to interact with websites' elements following certain heuristics, allowing the crawler to collect data about further requests made.

6.4.4 Links

Another important topic of our work is the study of the links between Web3 websites and Web3 and Web2. Furthermore, we aim to understand whether the Web3 ecosystem is a

network of websites or a collection of isolated islands.

While some projects are trying to integrate IPFS and similar DFS protocols with web browsers, allowing developers and users to reference data using URIs in the form of `ipfs://<cid>`, they have not been standardized yet. Furthermore, the temporary solutions adopted by users, such as a browser extension[18] that translates both the URIs above and common gateway URLs into URLs on the user's local node, are not common enough to allow developers to use `ipfs://` URIs assuming most of the user-base of their products will be able to resolve them.

The issue mentioned above makes linking Web3 website pages impractical, as it leads to the use of links to IPFS gateways hosted on Web2 infrastructure such as ipfs.io. The gateway mentioned above is the most used, and 56 links found in the pages we crawled use it to link to other Web3 websites. Only 34 of the links mentioned above are found on pages belonging to websites that do not show content from Web2. There are other similar gateways, but ipfs.io is the most used in our dataset, leading us to conclude that the Web3 ecosystem comprises many website islands that do not link to each other.

It is also worth noting that while it is very rare for a Web2 website to link to a Web3 website, it does sometimes happen. In particular, our dataset contains 15 links from Web2 pages to resources on IPFS via ipfs.io, spread across 4 websites. The main website using ipfs.io links was Foundation,[19] an NFT marketplace, which uses it to link to the NFT metadata and image.[20]

Further, as expected, we confirmed that the websites linked to by most Web3 website pages are Web2 websites, with the top 10 most linked websites containing social network giants like X, Telegram, and LinkedIn, as shown in Table 6.11. Special attention must be paid to Etherscan and Opensea,[21] the latter being the best-known digital marketplace for NFT collection and trading, two of the most important Web2 websites in the Ethereum ecosystem.

Finally, we found that 74 pages linked to websites such as archive.is[22] and web.archive.org,[23] projects that aim to preserve the content of the web, aligning with the Web3 ideals of censorship resistance and immutability.

[18] https://docs.ipfs.tech/install/ipfs-companion
[19] https://foundation.app
[20] https://foundation.app/mint/eth/0xC25a910324bbd6F85fa4C51A0fE92f7D04d68fB8/30
[21] https://opensea.io
[22] https://archive.is
[23] https://web.archive.org

Domain	Count
twitter.com	1,386
github.com	889
t.me	401
linkedin.com	398
youtube.com	375
etherscan.io	334
opensea.io	318
facebook.com	310
medium.com	268
instagram.com	261

Table 6.11: Most common remote URL domain Web3 pages link to.

6.4.5 Cookies

While our cookie banner detection algorithm described in Section 5.3.3.2 did not perform as expected, as reported in Section 6.2.2, we still managed to detect the presence of some cookie banners, and we have the data about the cookies present in the page before clicking on the accept button candidate elements.

We found that most used cookies started with the prefix `_ga`, signaling cookies set by Google Analytics.[24] Looking at the websites that use those cookies, we found that 66 are completely on Web3. Analyzing the dataset, we found that our cookie banner detection algorithm detected only false positives and no real cookie banners.

Finally, websites fully on Web3 use 60% fewer cookies than our dataset's average website. In particular, the average number of cookies set by a website fully on Web3 is 2.66, while the average number set by a website in our dataset is 6.32.

6.4.6 Website Size

Web3 developers must operate in an environment different from the one Web2 developers use. This poses challenges in communicating and linking pages, as exposed in Sections 6.4.3 and 6.4.4. This section analyzes whether the above restrictions change the websites' overall size.

In particular, Figure 6.1 shows that one-page websites are much more common between websites that contain only pages that do not show content via redirects or full page `iframe`

[24] https://developers.google.com/analytics

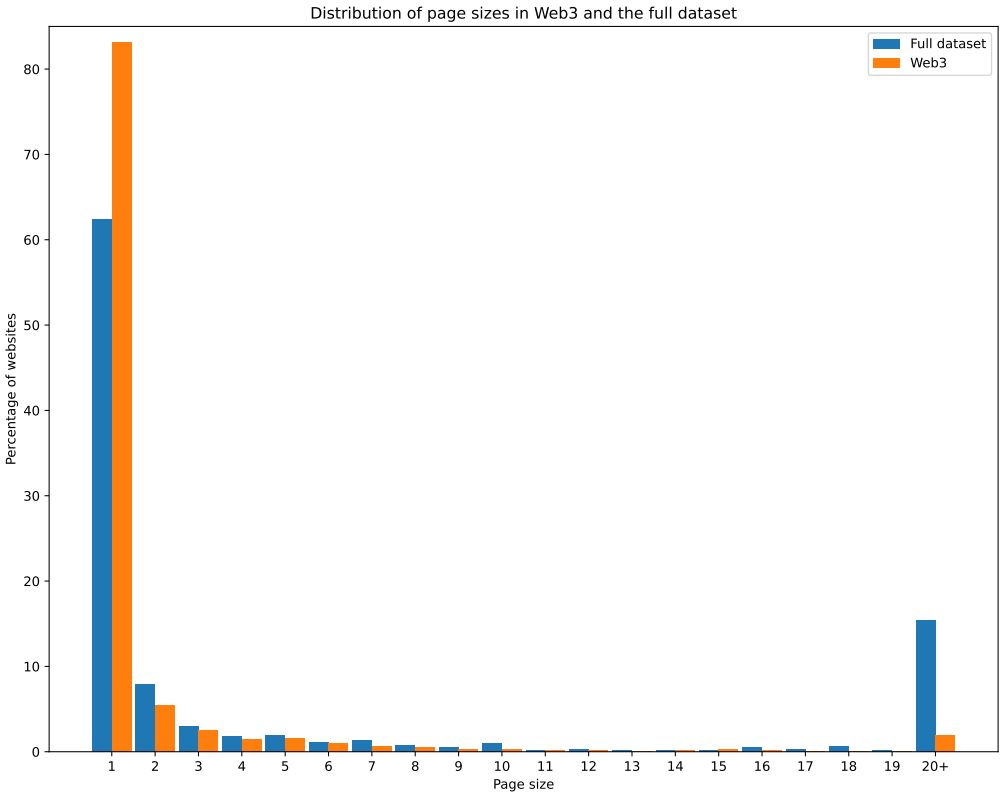

Figure 6.1: Size of the websites in the dataset.

Size	Web3		All	
	Count	Percentage	Count	Percentage
1	1,702	83.19	2,210	62.38
2	111	5.43	280	7.90
3	51	2.49	107	3.02
4	29	1.42	65	1.83
5	33	1.61	67	1.89
6	21	1.03	41	1.16
7	13	0.64	46	1.30
8	10	0.49	26	0.73
9	7	0.34	19	0.54
10	7	0.34	36	1.02
11	3	0.15	7	0.20
12	4	0.20	12	0.34
13	2	0.10	6	0.17
14	3	0.15	8	0.23
15	6	0.29	8	0.23
16	3	0.15	17	0.48
17	1	0.05	9	0.25
18	1	0.05	24	0.68
19	0	0.00	8	0.23
20+	39	1.91	547	15.44
Total	2,046		3,543	

Table 6.12: Page size distribution among different kinds of Web3 websites.

elements, signaling that Web3 developers seem to have adapted and started making smaller and more focused websites. While the trend is similar, the same analysis on our full dataset shows that allowing redirects and `iframe`s tends to produce bigger websites.

Furthermore, Table 6.12 shows that 90% of our dataset's websites containing 20 or more pages are at least partly on Web2. In contrast, almost 80% of one-page websites are fully hosted on Web3.

6.4.7 Technologies Used

Our last analysis focuses on the technologies used by the websites we crawled. In particular, we are interested in understanding if Web3 developers adopt the same JavaScript libraries when developing Web3 applications as they would with Web2 websites and whether and

Library	Count
Create React App	165
jQuery	161
React	93
Bootstrap	80
core-js	74
Lo-Dash	39
Vue	32
Next.js	19
lit-html	17
LitElement	12

Table 6.13: Most used client-side JavaScript libraries in Web3 websites that do not show Web2 content.

how much the immutability characteristics of the IPFS network impact the website's security and adherence to deprecations.

Tables 6.13 and 6.14 present the usage of client-side JavaScript libraries as detected by Lighthouse, as described in Section 5.3.3.3. The analysis indicates that Web3 developers use fewer JavaScript libraries detected by Lighthouse and favor modern technologies such as React. In contrast, the full dataset, including Web2 websites, still reflects a significant reliance on platforms like WordPress,[25] notably absent from the Web3 list. This discrepancy is likely due to the legacy nature of WordPress and the absence of server-side rendering capabilities in IPFS, where only static-site generation is feasible due to the lack of a centralized server. This limitation impacts WordPress and affects more contemporary alternatives such as Next.js[26] and other JavaScript meta-frameworks that appear further down in the library list.

Overall, the newer Web3 websites in our dataset tend to adopt more modern technologies, unlike Web2 websites, which frequently rely on older platforms such as WordPress. However, we cannot determine whether the immutability of IPFS is associated with higher usage of older library versions, as we lack sufficient data to draw meaningful conclusions on this aspect.

[25] https://wordpress.com
[26] https://nextjs.org

Library	Count
core-js	517
jQuery	485
React	412
Create React App	244
Lo-Dash	145
WordPress	135
lit-html	133
Next.js	131
Bootstrap	126
LitElement	119

Table 6.14: Most used client-side JavaScript libraries in all websites in our dataset.

Chapter 7

Conclusion

This Master's thesis focuses on developing programs designed to gather data on the Ethereum Name System (ENS) and the specific segment of Web3 to which its content identifiers point. In particular, we comprehensively analyzed the ENS system. By navigating through over three thousand Web3 websites, we were able to identify key characteristics of their content and examine their connections to the conventional web.

The results of our analysis highlight an evolving Web3 ecosystem, where developers are contending with various limitations imposed by both the underlying technologies and web browsers. A significant challenge faced by Web3 developers publishing websites on IPFS is the intrinsic immutability of the system. IPNS offers a partial solution, attempting to maintain the protocol's emphasis on security and decentralization. However, as discussed in Section 6.2.1, the lack of sufficient data on websites pointed to by IPNS records precludes a comprehensive study of how mutability in IPNS influences developer strategies for presenting mutable content within an immutable framework. One common approach we identified in IPFS is the redirection to Web2 via various methods, enabling developers to leverage Web2 infrastructure, which clashes with the decentralization ideals of both blockchains and IPFS. Additionally, we observed a lesser-used technique involving the use of `iframe` HTML elements. This method allows the display of content from external sources, functioning either as a means of redirecting users to Web2 content or as a way to incorporate sections of the original data hosted on Web2, typically from well-known platforms like YouTube or X.

The analysis also revealed challenges connecting Web3 websites, particularly due to the lack of a standardized method for referencing IPFS resources in web browsers. This results in a fragmented website network, where Web3 applications rarely link to one another. When they do, they are forced to use IPFS gateways. The reliance on Web2 infrastructure for interconnection undermines the potential for true decentralization. While efforts are

underway to standardize resource locators for IPFS and IPNS [55], the feature remains unfinalized. Even browsers such as Brave,[1] which have been prominent supporters of Web3, have withdrawn support for the feature due to low adoption [56]. However, users running a local IPFS node can use browser extensions to redirect URLs from public IPFS gateways to their local gateway, enabling a fully decentralized setup when paired with a local Ethereum node. For users unable to maintain a local Ethereum node, services such as Infura provide fallback mechanisms for retrieving data, allowing websites to function with Web2 infrastructure as a backup. Our findings suggest that while many Web3 websites avoid Web2 requests, some rely on node providers like Infura for data access.

Furthermore, while privacy is a key concern in Web3, a notable portion of Web3 developers still utilize Web2 tools such as Google Analytics to track users. Although imperfect, our cookie banner detection algorithm did not detect any cookie policy banners related to such tracking activities.

In terms of website architecture, our analysis indicates that Web3 websites are smaller than their Web2 counterparts, with most comprising single-page designs. This lack of large websites may be due to the constraints of the IPFS network, which complicates hosting large-scale websites. Additionally, we observed that Web3 websites frequently employ similar technologies to Web2 websites, with a notable reliance on client-side JavaScript frameworks like React. The lack of a traditional server concept on IPFS likely drives the preference for client-side frameworks.

Initially, our analysis suggested that many Web3 websites merely redirected to Web2 content, leading us to question the system's viability as a decentralized alternative. However, as we more thoroughly explored our dataset, we uncovered a complex ecosystem, exposing the various challenges faced by Web3 developers and the factors that have shaped the current state of Web3 applications.

[1] https://brave.com/

Chapter 8

Future Work

This study focuses on a subset of Web3 websites hosted on the IPFS. However, as demonstrated in Table 6.5, numerous records remain to be crawled within IPFS and other contenthash codecs. To address this, we plan to expand our analysis by incorporating all the contenthash codecs referenced in the above table. This will enable a more comprehensive examination of the overall structure of the Web3 ecosystem, the results of which we intend to publish in a subsequent scientific study.

A key aspect of this future work is to deepen our understanding of the relationships between Web2 and Web3 websites, including exploring their internal structures and identifying the differences between the Web2 subset we have crawled and a similarly sized dataset derived from the broader Web2 network. This comparative analysis will provide insights into how the limitations and opportunities of Web3, discussed in Chapter 6, shape the network.

Before progressing to this follow-up study, it is essential to reflect on the lessons learned throughout the development and analysis processes undertaken in this Master's Thesis. These insights will guide the refinement of our methodology and the expansion of our dataset. Improving both the size and completeness of our dataset will not only allow for a more detailed comparison of different codecs but will also facilitate a deeper exploration of the generational shifts between Web2 and Web3.

Moreover, while our study provides initial insights into adopting the latest ENS features, a dedicated analysis focused on this aspect of the ENS system could better assess the true effectiveness of these new additions. This could be achieved by examining key metrics such as the usage rates of the new features and the number of distinct addresses interacting with them.

The Web3 ecosystem is rapidly evolving, dealing with technological advancements that continually reshape its landscape. For instance, Ethereum and ENS consistently introduce

new features through their proposal processes. As discussed in Section 2.3.1, the ENS team plans to migrate portions of their system to an L2 solution, offering benefits outlined in Section 2.1.1. This transition is expected to unlock new features and improve user adoption. Once ENSv2 is fully implemented and has been in use for a sufficient period, analyzing the changes in its structure and usage patterns compared to the current version could yield valuable insights into the ongoing evolution of decentralized systems.

Bibliography

[1] Vitalik Buterin. *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform*. 2014. URL: https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf.

[2] Nick Johnson. *ERC-137: Ethereum Domain Name Service - Specification. Ethereum Improvement Proposals*, no. 137. April 2016. URL: https://eips.ethereum.org/EIPS/eip-137.

[3] Dean Eigenmann and Nick Johnson. *ENSIP-7: Contenthash field*. November 2018. URL: https://docs.ens.domains/ensip/7 (visited on September 10, 2024).

[4] Amiangshu Bosu et al. "Understanding the motivations, challenges and needs of Blockchain software developers: a survey". In: *Empirical Software Engineering* 24 (August 2019). DOI: 10.1007/s10664-019-09708-7.

[5] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". In: (2009). URL: http://www.bitcoin.org/bitcoin.pdf.

[6] *The Merge*. URL: https://ethereum.org/en/roadmap/merge/ (visited on September 10, 2024).

[7] CCRI. *The Merge - Implications on the Electricity Consumption and Carbon Footprint of the Ethereum Network*. September 2022. URL: https://carbon-ratings.com/dl/eth-report-2022 (visited on September 10, 2024).

[8] Usman W Chohan. "The cryptocurrency tumblers: risks, legality and oversight". In: (2017).

[9] *Contract ABI Specification - Argument Encoding*. URL: https://docs.soliditylang.org/en/develop/abi-spec.html#argument-encoding (visited on September 13, 2024).

[10] Fabian Vogelsteller and Vitalik Buterin. *ERC-20: Token Standard. Ethereum Improvement Proposals*, no. 20. November 2015. URL: https://eips.ethereum.org/EIPS/eip-20.

[11] William Entriken et al. *ERC-721: Non-Fungible Token Standard. Ethereum Improvement Proposals*, no. 721. January 2018. URL: https://eips.ethereum.org/EIPS/eip-721.

[12] Josie Thaddeus-Johns. *What Are NFTs, Anyway? One Just Sold for $69 Million.* March 2021. URL: https://www.nytimes.com/2021/03/11/arts/design/what-is-an-nft.html (visited on September 10, 2024).

[13] David Z. Morriz. *How The DAO Hack Changed Ethereum and Crypto.* May 2023. URL: https://www.coindesk.com/consensus-magazine/2023/05/09/coindesk-turns-10-how-the-dao-hack-changed-ethereum-and-crypto/ (visited on September 10, 2024).

[14] Vitalik Buterin. *Hard Fork Completed.* July 2016. URL: https://blog.ethereum.org/2016/07/20/hard-fork-completed (visited on September 10, 2024).

[15] *Code is Law.* URL: https://ethereumclassic.org/why-classic/code-is-law (visited on September 10, 2024).

[16] Witek Radomski et al. *ERC-1155: Multi Token Standard. Ethereum Improvement Proposals*, no. 1155. June 2018. URL: https://eips.ethereum.org/EIPS/eip-1155.

[17] Christian Reitwießner et al. *ERC-165: Standard Interface Detection. Ethereum Improvement Proposals*, no. 165. January 2018. URL: https://eips.ethereum.org/EIPS/eip-165.

[18] Steve Ellis, Ari Juels, and Sergey Nazarov. *Chainlink Whitepaper (v1).* September 2017. URL: https://research.chain.link/whitepaper-v1.pdf (visited on September 10, 2024).

[19] The ENS team. *Ethereum Name Service: ENS v2.* May 2024. URL: https://blog.ens.domains/post/ensv2 (visited on September 9, 2024).

[20] Nick Johnson. *ERC-3668: CCIP Read: Secure offchain data retrieval. Ethereum Improvement Proposals*, no. 3668. July 2020. URL: https://eips.ethereum.org/EIPS/eip-3668.

[21] *ENSIP-15: ENS Name Normalization Standard.* URL: https://docs.ens.domains/ensip/15 (visited on September 10, 2024).

[22] *Unicode Technical Standard 46: Unicode IDNA Compatibility Processing.* URL: https://www.unicode.org/reports/tr46/ (visited on September 10, 2024).

[23] Nick Johnson and @0age. *ENSIP-10: Wildcard Resolution.* URL: https://docs.ens.domains/ensip/10 (visited on September 11, 2024).

[24] Nick Johnson. *ERC-181: ENS support for reverse resolution of Ethereum addresses. Ethereum Improvement Proposals*, no. 181. December 2016. URL: https://eips.ethereum.org/EIPS/eip-181.

[25] Paul E. Hoffman. *DNS Security Extensions (DNSSEC)*. RFC 9364. February 2023. DOI: 10.17487/RFC9364. URL: https://www.rfc-editor.org/info/rfc9364.

[26] Scott Rose et al. *Resource Records for the DNS Security Extensions*. RFC 4034. March 2005. DOI: 10.17487/RFC4034. URL: https://www.rfc-editor.org/info/rfc4034.

[27] Richard Moore. *ENSIP-5: Text Records*. May 2017. URL: https://docs.ens.domains/ensip/5 (visited on September 9, 2024).

[28] *ENSIP-18: Profile Text Records*. August 2023. URL: https://docs.ens.domains/ensip/18 (visited on September 28, 2024).

[29] Juan Benet. *IPFS - Content Addressed, Versioned, P2P File System*. 2014. URL: https://github.com/ipfs/papers/raw/master/ipfs-cap2pfs/ipfs-p2p-file-system.pdf (visited on September 10, 2024).

[30] Ralph C. Merkle. "A Digital Signature Based on a Conventional Encryption Function". In: *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*. Ed. by Carl Pomerance. Vol. 293. Lecture Notes in Computer Science. Springer, 1987, pp. 369–378. DOI: 10.1007/3-540-48184-2_32. URL: https://doi.org/10.1007/3-540-48184-2%5C_32.

[31] Petar Maymounkov and David Mazieres. "Kademlia: A peer-to-peer information system based on the xor metric". In: *International workshop on peer-to-peer systems*. Springer. 2002, pp. 53–65.

[32] *IPNS Record and Protocol*. URL: https://specs.ipfs.tech/ipns/ipns-record/ (visited on September 10, 2024).

[33] Zooko Wilcox-O'Hearn. *Names: Distributed, Secure, Human-Readable: Choose Two*. October 2001. URL: https://web.archive.org/web/20011020191610/http://zooko.com/distnames.html (visited on September 24, 2024).

[34] George Giamouridis, Boojoong Kang, and Leonardo Aniello. *Blockchain-based DNS: Current Solutions and Challenges to Adoption*. 2024.

[35] Harry A Kalodner et al. "An Empirical Study of Namecoin and Lessons for Decentralized Namespace Design." In: *WEIS*. Vol. 1. 1. 2015, pp. 1–23.

[36] Pengcheng Xia et al. "Challenges in decentralized name management: the case of ENS". In: *Proceedings of the 22nd ACM Internet Measurement Conference*. 2022, pp. 65–82.

[37] Ping Fan Ke and Yi Meng Lau. *Exploring the Market Impact of Web3 Identity Imitation in Ethereum Name Service*. 2024.

[38] Bowen He et al. “TxPhishScope: Towards Detecting and Understanding Transaction-based Phishing on Ethereum”. In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2023, pp. 120–134.

[39] Shivani Jamwal et al. “A survey on ethereum pseudonymity: Techniques, challenges, and future directions”. In: *Journal of Network and Computer Applications* (2024), p. 104019.

[40] Ferenc Béres et al. “Blockchain is watching you: Profiling and deanonymizing ethereum users”. In: *2021 IEEE international conference on decentralized applications and infrastructures (DAPPS)*. IEEE. 2021, pp. 69–78.

[41] Nick Johnson. *ENSIP-9: Multichain Address resolution*. September 2019. URL: https://docs.ens.domains/ensip/9 (visited on September 19, 2024).

[42] Zhuotao Liu et al. “Make Web3.0 Connected”. In: *IEEE Trans. Dependable Secur. Comput.* 19.5 (2022), pp. 2965–2981. DOI: 10.1109/TDSC.2021.3079315. URL: https://doi.org/10.1109/TDSC.2021.3079315.

[43] Qin Wang et al. “Exploring Web3 From the View of Blockchain”. In: *CoRR* abs/2206.08821 (2022). DOI: 10.48550/ARXIV.2206.08821. arXiv: 2206.08821. URL: https://doi.org/10.48550/arXiv.2206.08821.

[44] *Ethereum: Light Clients*. URL: https://ethereum.org/en/developers/docs/nodes-and-clients/light-clients/ (visited on September 13, 2024).

[45] *IPSS Kubo for Go*. URL: https://docs.ipfs.tech/install/command-line/ (visited on September 14, 2024).

[46] Murray Maloney. *Implementing HTML Frames*. March 1997. URL: https://www.w3.org/TR/WD-frames-970331 (visited on September 14, 2024).

[47] *Third-Party Cookie Deprecation Testing and Debugging*. URL: https://www.chromium.org/Home/chromium-privacy/privacy-sandbox/third-party-cookie-phaseout/ (visited on September 14, 2024).

[48] *A new path for Privacy Sandbox on the web*. URL: https://privacysandbox.com/news/privacy-sandbox-update/ (visited on September 14, 2024).

[49] David Marr and Ellen Hildreth. “Theory of Edge Detection”. In: *Proceedings of the Royal Society of London. Series B, Biological Sciences* 207.1167 (February 1980), pp. 187–217. DOI: 10.1098/rspb.1980.0020.

[50] Christoph Zauner. *Implementation and Benchmarking of Perceptual Image Hash Functions*. Upper Austria University of Applied Sciences, Hagenberg Campus, July 2010. URL: https://www.phash.org/docs/pubs/thesis_zauner.pdf.

[51] The Swarm Team. *Swarm: Storage and Communication Infrastructure for a Self-Sovereign Digital Society*. June 2021. URL: https://www.ethswarm.org/swarm-whitepaper.pdf (visited on September 19, 2024).

[52] Iskander Sanchez-Rola, Davide Balzarotti, and Igor Santos. “The Onions Have Eyes: A Comprehensive Structure and Privacy Analysis of Tor Hidden Services”. In: *World Wide Web Conference (WWW)*. 2017.

[53] David Vorick and Luke Champine. *Sia: Simple Decentralized Storage*. November 2014. URL: https://sia.tech/sia.pdf (visited on September 19, 2024).

[54] James Royal. *Ethereum Price History: 2015 to 2024*. URL: https://www.bankrate.com/investing/ethereum-price-history/ (visited on September 21, 2024).

[55] Dietrich Ayala. *Adding IPFS Support to Chromium*. November 2022. URL: https://blog.ipfs.tech/14-11-2022-igalia-chromium/ (visited on September 22, 2024).

[56] Brian Bondy. *IPFS Support in Brave*. URL: https://brave.com/blog/ipfs-support/ (visited on September 22, 2024).