Aalto University
School of Science

Master's programme in Security and Cloud Computing - SECCLO

Securing the Bridges Between Two Worlds: a Systematic Literature Review of Blockchain Oracles Security

Alessandro Chiarelli

Master's Thesis
2023

Aalto University
School of Science

Author Alessandro Chiarelli

Title Securing the Bridges Between Two Worlds: a Systematic Literature Review of Blockchain Oracles Security

Degree programme Security and Cloud Computing - SECCLO

Major Security and Cloud computing

Supervisor and advisor Prof. Fabian Fagerholm, Prof. Raimundas Matulevičius, PhD. Mubashar Iqbal

Collaborative partner University of Tartu, Estonia

Date 9 May 2023 **Number of pages** 77+17 **Language** English

Abstract

Blockchain technology has paved the way for the decentralization of Internet services. It achieves this using a decentralized and distributed ledger that can withstand single points of failure. The ledger is secured through advanced cryptographic techniques and a decentralized consensus mechanism that ensures its resistance to tampering. Blockchain is a self-enclosed system, usually called *on-chain* world. To interact with the rest of the internet outside the blockchain (e.g., *off-chain* world), we need to set up interfaces to let the two worlds interact. These interfaces are called *oracles*. Given the role of the oracles in a blockchain system, it is paramount to design and implement them securely. We perform a systematic literature review that shows not much research is done into studying the security aspects of blockchain oracles. The research mostly focuses on the economic aspects of the oracles or on how to implement or design oracles that can benefit some specific use cases. In this thesis, we select two inbound oracles and implement them to evaluate and compare them from a security point of view. The contribution of this thesis consists of a literature review motivating the need for further research on the topic and comparing two inbound oracles, as the technique used to perform them can be extended and adjusted to other oracles as well. We also present the implementation of an outbound oracle for completeness and discuss its security properties. Furthermore, we present a novel approach that makes use of a decentralized oracle network (i.e., Chainlink) to build a system that fetches off-chain data to the blockchain and then securely retakes the data off-chain, such that there is no need to trust the oracle nodes. The technique we propose is thus blockchain and oracle agnostic and can be applied in various situations.

Keywords Blockchain, oracles, computer security

Preface

To my dear Nonna Maria

I would like to thank all of the people who helped me reach this point. First of all, my parents, Mamma Milena, and Papà Massimiliano, encouraged me to pursue my studies until the end. My sister Azzurra too, who has always been by my side. I would also like to thank my supervisors Raimundas Matulevičius, Fabian Fagerholm, and Mubashar Iqbal who were crucial in writing this work. My thanks go also to the SECCLO Consortium and more generally to the Erasmus+ program of the EU for the learning opportunity they have given me. Finally, I would also like to say grazie to my grandparents, both the ones who are still here to cherish the end of my studies - Nonno Giacomo and Nonna Rosa - and also those who could not make it - Nonno Carmelo and, especially, Nonna Maria who passed away last year.

Alessandro Chiarelli

With the support of the
Erasmus+ Programme
of the European Union

Contents

1 Introduction

Blockchain technology rose to popularity after the anonymous Satoshi Nakamoto launched Bitcoin in January 2009 [BTC]. Thirteen years later, the blockchain space evolved and changed with an explosive innovation that took over. There have been multiple Bitcoin forks (e.g., Bitcoin Cash, Bitcoin Original, Bitcoin Diamond) and new blockchains such as Ethereum, Polkadot, Litecoin, Monero, and many more. While Bitcoin is a blockchain implementation mainly designed for a *cryptocurrency*, it has some support for *smart-contracts* on its base layer (layer-0). There exist higher layers with better support (such as RGB [RGB] for layer-2 and -3), and smart contracts raised to popularity thanks to Ethereum. Ethereum was designed mainly as a smart contract platform, and thanks to its ecosystem, many different contracts have been popularized, in particular, ERC-20 [ERCa] tokens and ERC-721 [ERCb] tokens, more commonly known as NFTs (Non-Fungible Tokens).

Smart contracts [SMA] have great potential to automate and innovate many processes present in our daily life. Smart contracts are code that resides on the blockchain and run only when certain conditions are met. The most basic examples are exchanging money between different accounts. For example, every time an account receives some ETH, a certain amount can be sent to another as a commission. The next step in smart contract innovation is the execution of some actions when certain inputs coming out of the blockchain are received. An example is betting: two friends could use a smart contract where they put their bets in a third account for holding, and after the results of the bets, one of them will receive the win from the third account. The problem is that while smart contracts work almost flawlessly with *on-chain* data (that is, data on the blockchain), they need an interface with the *off-chain* world (that is, outside of the blockchain) in order to get the data they need to operate. In our bet example, someone or something needs to inform the smart contract of the result of the bet. This role is fulfilled by blockchain *oracles* [ORA].

Blockchain oracles can be decentralized or centralized, but the centralized ones are the most mature and widely used, so we will focus our research on them. Oracles are a peculiar type of smart contract with an initiator and a responder. The initiator is, as the name suggests, the entity that starts the interaction and either sends a request for information or sends information without another entity requesting it. The responder is the entity that reacts to the commands sent by the initiator. A blockchain oracle can be either *outbound*, where the information flows from the blockchain to the off-chain world, or *inbound*, where the information flows from an off-chain component to the blockchain. It can also either be *push-based*, where the initiator sends data to the responder, or *pull-based*, where the initiator requests data from the responder. There are four blockchain oracle models [Lev22]:

- **Push-based inbound**: in this case, the initiator is an off-chain entity that pushes information to be stored in the blockchain, which is the responder in this case.
- **Push-based outbound**: in this case, the initiator is an on-chain entity that pushes information out to an off-chain entity.

- **Pull-based inbound**: in this case, the initiator is an on-chain entity that requests data from an off-chain component.
- **Pull-based outbound**: in this case, an off-chain component requests data from the blockchain.

1.1 Related work

Blockchain oracles may be a new technology, but there already is some research conducted behind them. In Caldarelli literature review [Cal20b], the "Oracle problem" is presented as an apparent inconsistency between the original philosophy behind blockchain technology, that is, removing all middle-men and third parties between transaction for achieving a trustless system. The reason is that blockchain oracles are interfaces between the off-chain and on-chain worlds that are usually managed by central authorities, thus recreating the original problem that blockchain was designed to solve. The work then proceeds to analyze different use cases, including law, supply chain, energy, and healthcare. It is evident that solving the blockchain oracle problem is fundamental if this technology is to be implemented in the mentioned use cases, as they all require high-security standards. Williams and Peterson [WP19] provide a thorough formal analysis of economic incentives for blockchain oracles using game theory principles. In particular, they point out how the major security problem of decentralized blockchain oracles is the verification of *a posteriori* data coming from the off-chain world. A **decentralized** blockchain oracle is one where there are multiple oracle operators providing the required information to the oracle itself, instead of being just one operator/entity. In their work, Williams and Peterson propose a new oracle design to address the risk of multiple oracle operators providing false data to the oracle. Finally, another related work has been conducted by Albreiki *et al.* [AHuRSS20]. Their work focuses on establishing a common taxonomy for blockchain oracles as well as researching and comparing a different number of blockchain oracle implementations, with the goal of pointing towards a new direction in research. It also defines a few significant design patterns that are the most commonly implemented in most blockchain environments.

Our work provides a systematic literature review of the current state of the art of security for blockchain oracles, with no focus on a specific use case or scenario. To our knowledge, there is no work that provides a similar analysis, as they either focus on some specific scenario or industry or focus on presenting a single technique to be used when designing or implementing blockchain oracles. The other chapters of our work instead focus on the analysis of a system in which blockchain oracles are used, and the goal is to analyze and compare two different techniques for inbound oracles and on analyzing one technique for outbound blockchain oracles that is compatible with both of the previous techniques. Thus, this work serves as an introduction to the industry of blockchain oracles from a security point of view, starting from a general overview of the current state of the art and then switching to a more hands-on or practical approach for an analysis of different techniques and technologies that are currently used in the industry. Finally, our work also presents a novel technique that

allows using a decentralized oracle network (Chainlink) in a centralized way with no loss of security.

1.2 Research motivation and objectives

Blockchain oracles are already the interface between the off-chain and on-chain worlds. As more companies and institutions make moves towards implementing their own blockchain solutions, oracles will play a fundamental role in the coming years once this transition is completed. Especially in a future where smart contracts become widely used in our daily life in fields such as betting, finance, healthcare, and news, oracles will need to be secure to be trusted by the users of these applications, who can be the population of entire countries. Malfunctions and exploits in the oracles could be used by malevolent groups for their purposes, which could be of grave intensity, especially for critical services. Without a thorough knowledge of the security framework for blockchain oracles, the adoption of blockchain technologies, in general, will be slowed down and would never be viable for use cases that require high-security standards.

Our main research objective is thus to analyze if the security requirements are satisfied by the many different blockchain oracles, and in case they are, how they are satisfied. We will have particular attention to the different use cases found both in the industry and in the public sector. We are interested in their security threats and in discovering ways to mitigate them accordingly. We will restrict our research mostly to centralized oracles, as they are more mature and more widely used, and we will not consider the mechanisms that allow decentralized oracles to decide the data to send to the blockchain.

In other words, we first wish to discover what assets, whether data, financial assets, or of other nature, need to be protected when designing a blockchain oracle. To the knowledge of the authors of this work, there is no literature review nor surveys analyzing blockchain oracles technologies from a security point of view. There are many focusing on the economic and social aspects [Cal20b, Car20], performance properties [KSG+20] and consensus mechanisms for decentralized oracles [AHuRSS20]. Thus a review of the current state of the art is something that is needed in order to encourage and foster new research on the topic. Secondly, we aim to identify the threats and attack vectors against blockchain oracles. In particular, we focus on discovering potential vulnerabilities and exploits that could be made use of to attack an oracle. We remind here that our discussion will be limited to centralized oracles, as decentralized oracles are less mature and involve other types of threats due to their decentralized nature. Thirdly, we focus on developing techniques and strategies to protect against these threats that can be easily implemented when designing a new blockchain oracle. We are also interested in identifying the differences between the different blockchain oracle models and the scenarios that they were designed for. Finally, we will choose two oracle implementations and compare them from a security point of view. The goal is to understand the strengths and weaknesses of each implementation and their respective limitations, as well as the purpose they were designed for. We will also implement an outbound oracle compatible with our previous inbound oracles.

Here is a reformulation of our research goals as main research questions:

- **[RQ1]** What is the current state of the art for securing blockchain oracles?
- **[RQ2]** How to implement two widely used blockchain oracles securely?
- **[RQ3]** How can one compare the two blockchain oracles?

1.3 Contributions

This work contributes to research in two main ways. The first way is by presenting a systematic literature review of the current techniques to secure blockchain oracles in different situations and implementations. The goal of this review is to be the groundwork to encourage further research on the topic, as the security aspects of blockchain oracles are unfortunately often ignored or at least in the background, compared to economic properties, game theory strategies for decentralized oracles, or performance properties.

The second main contribution of this paper is by analyzing two similar and widely used blockchain oracle implementations from a security point of view, in order to see the strengths and weaknesses of both strategies and as a starting point for further analyses and comparisons in the future between other strategies and implementations. The contribution is completed by implementing an outbound oracle in order to have a full system.

1.4 Research Method

The method used for this thesis is as follows. This thesis follows a design science approach as in [HMPR04]. Our problem is to implement a system with on-chain and off-chain components that uses secure blockchain oracles, one inbound and one outbound. We will divide the process into multiple phases: technology survey, technology selection, implementation, and evaluation. The first task is to perform a Systematic Literature Review (SLR) using the Kitchenham method [KC07]. The SLR is the technology survey phase, as it allows us to learn the state of the art of current blockchain oracle security and to have an overview of many technologies. The SLR is performed by using a few publicly accessible libraries and performing queries using relevant keywords. Once the initial query is performed, relevant papers are selected according to inclusion and exclusion criteria and analyzed to extract specific information about blockchain oracles and their security. We will also be looking for papers discussing specific situations or use cases in which blockchain oracles are used as a part of the solution. The second task is to use the found information to choose the technologies and techniques needed to implement two inbound blockchain oracles and one outbound blockchain oracle. This task addresses the both the technoogy selection phase and the implementation phase. The final task is to evaluate the oracles after having defined evaluation criteria to compare the three oracles.

1.5 Structure of the Thesis

The remainder of the thesis is structured as follows. Chapter 2 presents related work previously conducted on the topic and presents the use cases and blockchain oracle models that will be the focus of our work. Chapter 3 provides an overview of the background technologies that will be used in the following chapters. Chapter 4 presents the implementation choices of the two chosen blockchain oracles. Chapter 5 defines evaluation criteria and evaluates the implemented oracles according to those criteria. The evaluation will be focused on the security aspects of the oracles. Finally, Chapter 6 concludes this thesis. An appendix is also added, with additional information for the thesis that was not necessary for the main work.

2 Systematic Literature Review

In this section, we conduct a Systematic Literature Review (SLR) to assess the current state of research on the topic of blockchain oracles security. The SLR acts as our technology survey in our design science approach. The SLR aims to identify relevant academic papers and articles and extract information on the security requirements of the main use cases that blockchain oracles are being used for, as well as how the most well-known practical implementations address these requirements. At the current time of writing, no previous SLR was conducted on the topic that the writers are aware of. In this section, we will be answering the first research question **[RQ1]: What is the current state of the art for securing blockchain oracles?** The question can be divided into the following subquestions, and in order to reply to **[RQ1]** we will first reply to the subquestions.

- **[RQ1.1]** What are the assets that need to be secured when designing a blockchain oracle?
- **[RQ1.2]** What are the most important attack vectors and threats against blockchain oracles?
- **[RQ1.3]** What are the most efficient techniques to secure the different blockchain oracles?

2.1 Literature Sources

The initial source for relevant work on the topic was conducted on the IEEE digital library, Scopus, ScienceDirect, and SpringerLink. References, citations, and related work sections of the selected papers also provided an additional source to be added to the initial search. Grey literature is considered and will be included in case the relevant results are not already covered in academic or scientific literature.

2.2 Search Terms

To perform the search, the following terms were used: *("blockchain" AND ("oracle" OR "oracles") AND ("use-cases" OR "security" OR "applications" OR "implementations"))*. Whenever appropriate, both plural and singular forms of the different words will be used to perform queries in the database. The terms *use-cases, applications* and *implementations* refer to similar concepts, but they are not perfect synonyms, as they all have different nuances. *Use-cases* is used to look for examples of areas where blockchain oracles can bring benefit; examples can be voting or healthcare. *Applications* and *implementations* will be used to look for examples where blockchain oracles have been practically implemented in the industry already.

2.3 Inclusion and Exclusion Criteria

The inclusion criteria are the following:

Table 1: Literature review screen

	Initial search	**Inclusion criteria**	**Snowballing**
IEEE	105	16	3
Science Direct	451	19	2
Scopus	193	9	0
SpringerLink	167	10	0

- Papers related to blockchain oracles
- Papers related to blockchain oracles security
- Papers related to blockchain oracles implementations
- Papers related to blockchain oracles applications

The exclusion criteria are the following:

- Papers not written in the English language
- Papers not accessible freely (ie not present in the main scientific databases)
- Papers older than 2015 (as smart contracts took off only after 2014)
- Short papers that do not take into account the security aspects of blockchain oracles

2.4 Papers Selection

The selection started with research in the digital libraries using the search terms as explained in **2.2 Search Terms**. As the blockchain field in general, and oracles in particular, is a vast field of research, numerous results were reported from the various queries and the articles were later analyzed manually to select the papers that presented the search terms either in the title, in the abstract or in the introduction. Subsequently, the sources of the papers will be analyzed as well, and relevant sources will be added to the search. Finally, the papers will be manually filtered according to the inclusion and exclusion criteria. The quantitative results of the queries are organized in Table 1, where the *snowballing* column refers to the sources found by analyzing the sources of the selected papers.

In total, 742 papers were found. After an initial screening using the inclusion and exclusion criteria, 54 papers were remaining. After reading these papers, 5 potential papers and pieces of grey literature were added through snowballing. Finally, the relevant papers for the literature review were selected for a total of 16 papers. The papers that were excluded were either redundant, since they were explaining the same concepts already present in other papers, or were not focusing on the security aspect of blockchain oracles or blockchain frameworks in general.

Table 2: Features for data extraction

Feature	Detail
ID	This is paper id.
Title	This field explains the paper title.
Authors	Who the authors are.
Oracle model	The oracle model or models presented.
Blockchain type	Blockchain framework and if the blockchain is public or permissioned.
Comm. protocol	Communication protocol used.
Data validation	Data validation techniques presented.
Threat	Type of threats presented.
Vulnerability	Vulnerabilities presented or found in the paper.
Impact	The impact of a potential attack.
Countermeasures	Countermeasures taken to prevent or limit threats.
Consensus mechanism	Consensus mechanism used in the blockchain.
Assets	The assets that can be targeted by an attack.
Rer.	References to the citations of relevant papers.

2.5 Information Extraction

The blockchain oracle models are known in advance and they are defined in the **2 Introduction** section. The first piece of information we aim to extract from the papers is the assets that need to be protected when designing a blockchain oracle. The vulnerabilities of the blockchain oracle models are not known in advance and thus they are the second kind of information that we aim to extract from the papers. Finally, we want to extract information about the techniques and security measures taken to address those vulnerabilities. Table 2 contains a detailed view of the information to be extracted from the papers along with an explanation of each piece of information.

2.6 Summary of Selected Articles

Y. Zhao et al. "Toward Trustworthy DeFi Oracles: Past, Present, and Future" [ZKL+22] present a thorough analysis of blockchain oracles for DeFi (Decentralized Finance). Centralized oracles are characterized by being fed information from a single trusted third party and then being more easily designed and implemented compared

to Decentralized oracles. Centralized oracles have high time efficiency and data throughput and are applicable when there is a low-risk tolerance as well as a fast response scenario. They introduce trust into the system, a single point of failure, and are not highly scalable. The authors presented *Provable* [Pro] as an example of a centralized oracle that could be used by institutions to interact with the blockchain. It later follows a discussion about decentralized oracles, which is not interesting to our purposes.

A. Al Sadawi et al. "On the Integration of Blockchain With IoT and the Role of Oracle in the Combined System" [SHN22] analyzes how blockchain technologies can benefit IoT. Of particular interest to us is the discussion about centralized oracles. In particular, oracles are necessary in order to satisfy the conditions of immutability and determinism of smart contracts. The oracles are the link between the IoT devices and the execution of their related smart contracts. The main problems in the IoT industry are that the entities involved in a system lack trusted relationships to guarantee the identities of the involved entities, the authenticity of the data provided by the IoT devices, and the reliability of data transmission in case any party fails to fulfill its services. The current solution to these problems is to use central authorities, but by combining blockchain and IoT this authority can be removed. The paper later presents a literature review discussing how different blockchain frameworks address the specific use case of IoT. The paper later proceeds with a description of the concept of blockchain oracles and of the oracle problem. It is relevant to us that the article explains that oracles collect data from IoT devices and then submit them to the blockchain alongside a certificate to attest to the authenticity of the data. The security of the data from the device to the oracle is guaranteed by security hardware instead. Later follows a discussion about different types of oracles, where the oracles are classified according to different parameters (number of nodes, type of data source, and the design pattern). The paper also presents a description of different oracles, of which some are decentralized and two are centralized. The centralized ones include: Provable, which was already mentioned in another article, and Town Crier [Towb], an oracle that uses hardware capabilities of Intel's Software Guard Extensions (SGX) to collect data and generate authenticity proofs. Finally, the paper analyzes a use case where the authors build a carbon pricing oracle. The oracle is built on the Ethereum blockchain using IoT devices capable of measuring the amount of carbon emissions being emitted by an entity. The authors propose two system designs, one using software solutions and one using hardware solutions. Both solutions are pull-inbound oracles. The security analysis yields that the oracles satisfy the requirements of data availability, authorization, accountability, confidentiality, integrity, and resilience to cyber attacks. In particular, the resilience to cyber attacks is guaranteed by the use of the Ethereum blockchain, and data integrity is guaranteed by cryptographic primitives that immutably save CO2 emissions in the blockchain along with information to verify their integrity.

A. El Fezzazi et al. "Towards a Blockchain-based Intelligent and Secure Voting" [FAB21] discusses the case of combining blockchain and machine learning (ML) technologies for secure e-voting practices. In the specific case of e-voting, there are two additional requirements that need to be addressed: the first one is identifying

the voter and ensuring that their vote is not tampered with, and the second one is guaranteeing the secrecy of the vote. The authors discussed several approaches and finally presented their own. Among the processes discussed, in most solutions, voting is implemented as a blockchain transaction where tokens are exchanged and the voter identification is either based on the asymmetric cryptography infrastructure or using a centralized identification service. The authors instead proposed a system that uses a Machine Learning Blockchain Oracle (MLBO) to use face recognition for voters. An MLBO is divided into two parts, one is the oracle script itself that interacts with the blockchain platform and the other one is the ML system. The oracle script focuses on generating face recognition API requests to the ML validators that are in charge of performing the face recognition.

S. Kuang et al. "Reliability analysis for blockchain oracles" [LXSY20] performs a reliability analysis using Fault Tree Analysis. The discussion begins with a comparison between different oracles, some centralized and some decentralized. In order not to overlap with previous papers, we will only be reporting the relevant results of the reliability analysis. In particular, Provable uses TLS-Notary to get data from off-chain sources and to provide cryptographic proof about this data. The data is obtained from websites using HTTPS protocol. Town Crier, Corda, and MS Bletchley instead use Intel's Software Guard Extensions as hardware modules to protect their environments. All of the aforementioned oracles have a reliability higher than 98% and this is due to their reliance on software and hardware components. Furthermore, the oracle designs present an approach similar to fault-tolerance design patterns. Town Crier and Corda use Intel SGX technology to contain errors and faulty data. MS Bletchley uses multiple oracles, which guarantees higher reliability. Finally, the authors highlight the oracle problem (ie, the reliability of the data source feeding information to the oracle).

M. Moussa et al. "Blockchain for Giving Patients Control Over Their Medical Records" [MBY+20] in their study provide and evaluate a patient-centric framework to allow patients, doctors, hospitals, regulator agencies, and insurance to interact on the blockchain. Their proposed solution is based on using a reputation system for oracles that are in charge of fetching data from a decentralized database network and sharing it among the patient and the other entities involved. The reputation system gives each oracle a score based on its interactions with both smart contracts and doctors. Regulatory agencies are public authorities that register patients, hospitals, and doctors. Hospitals are in charge of preparing the medical records and sharing them with the patients. Finally, the doctors request access to a patient's data, and once approved, an oracle will decrypt said data. The encryption of medical records is based on asymmetric cryptography, with symmetric keys being used to encrypt medical records; the symmetric keys are encrypted using the patient public key instead. According to the authors' security analysis, their framework allows sharing of data securely, as all data is encrypted and can be decrypted only with patient approval. The private keys used for the decryption of symmetric keys are not exposed, so the encryption is secure. The framework is secure against other types of attack (such as DDoS) thanks to the use of decentralized databases that remove vulnerabilities related to a single point of failure. Additional security measures are the use of the reputation system for the oracles as well as ensuring that only the patients can access functions

that modify their medical records. The authors remark that this framework can be generalized to other use cases, in general where one would need a data source and consumers of data from the data source.

Z. Song and Y. Yu in their study "The Digital Identity Management System Model Based on Blockchain" [SY22] propose a digital identity management system based on blockchain. Their system relies on four smart contracts. The first one is the *identifier management* smart contract, which is constructed by the user who wants to get a way to prove their identity using a Digital Identity Device (DID) and its respective public key. The second smart contract is used by the attribute provider (an institution in charge of releasing the digital identities) and binds the digital certificates with the DID of a user. The binding is performed using hashes of the certificates and by keeping track of the status of their certificates. The third smart contract is used by the validation service provider, which is an institution in charge of verifying the validity and status of the digital certificates provided by the user. Oracles are used for this smart contract to perform the related costly cryptographic computations. If the certificate is valid, the validation service provider will feed it to the fourth and final smart contract. This last smart contract is called identity proxy and it is used by the validation service provider to store the information of valid certificates it received. This allows binding a user DID with a digital certificate on the validation service provider side and in the future, only the DID will be needed to authenticate a user. The four smart contracts are executed in order and this allows to efficiently manage the digital identity of users using blockchain. The first two smart contracts are used when a user wants to register a new DID, the third is used when a user wants to register their DID to a new validation service provider and the fourth and last smart contract uses the DID to authenticate users by different service providers.

S. Wang et al. in "A Novel Blockchain Oracle Implementation Scheme Based on Application Specific Knowledge Engines" [WLS+19] proposes using Application Specific Knowledge Engines (ASKE) in conjunction with blockchain oracles to provide authoritative sources of information. An ASKE is a framework that allows one to fetch information from many different sources and reduces the possibility of single points of failure, for the same information one can use multiple sources. At the same time, using multiple sources allows one to verify the authenticity and truthfulness of the information. An ASKE is implemented as a set of web crawlers that process queries related to a specific domain. The authors recognize many limitations in this system; in particular, the sources of information are still centralized (and thus do not solve the oracle problem) and in their original implementations there is no way to verify that the data being fed to an oracle is authentic. The solutions to these limitations that the authors proposed are respectively the employment of a decentralized oracle system to have a fully decentralized source of information and the use of authenticity proofs, such as the one employed by TLS notary [TLS].

S. Cao et al. in "Hybrid Smart Contracts for Privacy-Preserving-Aware Insurance Compensation" proposes an original strategy to manage the medical records of patients and to process insurance claims. In their work, there are three entities interacting, the patient, the insurance provider, and the hospital. The medical records are stored on a private blockchain that can be accessed only by the hospital and the patient through

the use of a private smart contract. The private smart contract can be considered as a pull-outbound oracle, as it is invoked from entities outside of the blockchain and the source of data is the blockchain itself. The processing of the claims is instead performed through the use of a public smart contract that is invoked by the hospital and the insurance provider. The hospital will send the required information about a patient's medical record, such as the medicines used and the medical procedures performed on the patient. The list of prices for the medicines and medical procedures is available on the public blockchain and thus the oracle that sources this data is pull-outbound as well. The paper finally goes into detail in how the process for an insurance claim is conducted, and provides a security analysis and performance evaluation. The security analysis points out that zero-knowledge proofs are used to verify and protect the privacy and personally identifiable information of patients in insurance claims. The analysis also shows that the oracles perform the computations in a secure environment with dedicated hardware using Intel SGX so that cryptographic secrets cannot be leaked. The paper does not specify any specific cryptographic strategy implemented to verify and validate the data by the oracle. From the security analysis, it can be inferred that the blockchain oracle makes use of asymmetric cryptography and digital signatures to verify the data, and especially in the case of a private blockchain it can be inferred that digital certificates are also needed to certify the identities of the different entities involved.

2.7 Summary of Results

Table 3: Blockchain framework presented in the papers

Ref.	Oracle model	Blockchain type	Consensus mechanism	Data Source
[ZKL+22]	pull-inbound, push-inbound	Ethereum (public)	PoW (originally), PoS	Trusted centralized source
[Pro]	pull-inbound, push-inbound	Ethereum (public), HLF (permissioned), Corda (permissioned)	PoW, PoS, PBFT	Website, IPFS, Wolfram Alpha, random bytes, computation from TEE
[SHN22]	pull-inbound, push-inbound	Ethereum (public)	PoW (originally), PoS	Centralized source or IoT device
[LXSY20]	pull-inbound, push-inbound	Ethereum (public), HLF (permissioned), Corda (permissioned)	PoW, PoS, PBFT	Centralized source or IoT device
[WLS+19]	pull-inbound, push-inbound	N/A	N/A	ASKE (multiple centralized sources)
[Towb]	pull-inbound	Ethereum (public)	PoW (originally), PoS	Website
[FAB21]	pull-inbound, push-outbound	Ethereum (public), permissioned (unspecified) blockchain	PoW (originally), PoS, unspecified	Blockchain, Face recognition device, online server
[CZW+22]	pull-outbund	(unspecified) private blockchain, Ethereum	PoW (originally), PoS, unspecified	Blockchain
[SY22]	push-inbound, pull-outbound	Unspecified	Unspecified	User device, Blockchain, Attribute provider
[MCK21]	push-inbound, pull-inbound	Unspecified (public)	STB (variant of PoS)	IoT devices
[MCK19]	push-inbound	public/private	unspecified	IoT devices
[BHP20]	unspecified, inbound	Ethereum (public)	PoW (originally), PoS	external
[HCL+22]	pull-inbound	public	unspecified	smart devices for healthcare
[CYX21]	pull-inbound	Ethereum, public	PoW (originally), PoS	external web-api
[ABAQS+19]	pull-inbound	Ethereum, public	PoW (originally), PoS	External (unspecified)
[WZSK21]	pull-inbound	Ethereum (public)	PoW (originally), PoS	Decentralized storage

In this section, the results of the literature review are presented using tables 3 and 4. Let us focus on the first table.

Table 3 focuses on presenting information about the blockchain frameworks used in the different oracles. We were able to find detailed information for all blockchain models. Thus we will present the information for each one of them. We will first explain each characteristic in the table:

Oracle model refers to the blockchain oracle model being used to implement or design the current oracle.

Blockchain Type refers to whether the blockchain is public (such as Ethereum), or permissioned (such as Hyperledger Fabric).

Consensus mechanism refers to the consensus mechanism used by the blockchain. The ones that were found are Proof of Work (PoW), Proof of Stake (PoS), Practical Byzantine Fault Tolerance (PBFT), Delegated PoS (DPoS), and Asynchronous Byzantine Fault Tolerance (aBFT). The differences between the different consensus mechanisms are outside of the scope of this work.

Data source refers to the source of information for the oracle. In particular, we found that it can be a Website (or online server), an InterPlanetary File System (IPFS), Wolfram Alpha (an answer engine developed by Wolfram Research), random bytes, the result of a computation in a Trusted Execution Environment (TEE) which usually are ad-hoc devices, a blockchain or some other kind of device, like in the example of Face recognition devices.

Ref. reports the references concerning the row being discussed.

Table 3 reports the information about the security techniques found in the different blockchain oracles.

Comm. protocol refers to the communication protocol being used between the oracle and the off-chain entity.

Data Validation refers to where the data is validated, whether in a decentralized manner (on the device) or in a centralized manner.

Threat refers to the possible threats identified in the specific scenario being discussed.

Vuln. refers to the vulnerabilities identified in the specific scenario being discussed.

Impact refers to the consequences of a successful attack on the oracle.

Countermeasures refers to the countermeasures taken to limit the attacks.

Assets refers to the funds or data that need to be protected from attackers.

Ref. reports the references concerning the row being discussed.

2.8 Presentation of Results

After performing the survey, it is clear that there exist common practices and techniques used over multiple solutions. Thus we will now be presenting a summary of the results that were found in a content-centric way, topic by topic.

Blockchain types: Most of the surveyed papers focused on oracles based on the Ethereum blockchain, which is indeed a public blockchain. Some papers discussed general techniques that could be applied to multiple blockchain types, whether public or private, in some cases specifying some implementations, in particular Corda and Hyperledger Fabric (HLF).

Consensus mechanisms: Since the majority of the papers focused on pre-2022 Ethereum, the main consensus mechanism that was used for the blockchain was Proof of Work (PoW). It is worth noting that since Ethereum updated [POS] in late 2022, now the same analyses are valid with the new Proof of Stake (PoS) consensus mechanism, as what changed in Ethereum was the consensus and not the Ethereum

Virtual Machine (EVM) that is in charge of executing the smart contracts [ethc]. Other papers focusing on other blockchains had different consensus mechanisms, in particular PoS and PBFT (respectively for HLF and Corda). What is remarkable is that since the oracles interact with smart contracts and are not directly involved in the consensus mechanism as the Ethereum upgrade in particular shows, the consensus mechanism itself does not influence the workings of the oracle. It is worth noting, that in some cases the papers were analyzing decentralized oracles, and in such cases, it can happen that the expression *consensus mechanism* referred to the internal consensus of the oracle network to come to a decision of the values of the oracle.

Table 4: Security information of the blockchain oracles

Ref.	Comm. protocol	Data Validation	Threat	Vulnerbility	Impact	Countermeasures	Assets
[ZKL+22]	HTTPS	No data validation	Attacker feeding false information to the blockchain	There is no default channel of information between the source and the blockchain	Impacts funds and normal operation of smart contracts	Authenticity Proofs	Data and funds in the smart contracts
[Pro]	HTTPS	No data validation	Attacker feeding false information to the blockchain	There is no default channel of information between the source and the blockchain	Impacts funds and normal operation of smart contracts	Authenticity Proofs (optional)	Data and funds in the smart contracts
[SHN22]	Unspec.	On device	Attacker feeding false information to the blockchain	There is no default channel of information between the source and the blockchain	Impacts funds and normal operation of smart contracts	Authenticity Proofs, Secure module	Data and integrity of the IoT ecosystem
[LXSY20]	Unspec.	Unspec.	Attacker feeding false information to the blockchain, Denial of Service	There is no default channel of information between the source and the blockchain; single point of failure	Impacts funds and normal operation of smart contracts	Authenticity Proofs, Secure module	Data and reliability of the system
[WLS+19]	Unspec.	Performed by ASKE	Attacker feeding false information to the blockchain, Denial of Service, Trusted source misbehaving	There is no default channel of information between the source and the blockchain	Impacts funds and normal operation of smart contracts	Authenticity Proofs, Updating of ASKE source	Data and reliability of the system, reputation of trusted source
[Towb]	HTTPS	Digital Signatures	Attacker feeding false information to the blockchain	There is no default channel of information between the source and the blockchain	Impacts funds and normal operation of smart contracts	Secure module	Data and funds in the smart contracts
[FAB21]	Unspec.	Centralized server validates API requests	Attacker tries to influence the voting process or steal the facial information of the user	No encryption process specified, no secure channel between blockchain and MLBO	The impact extends to the theft of personally identifiable information and to influence the result of the vote	Reputation system for ML providers and definition of clear roles with different levels of privileged operations	Personally identifiable information and voting result
[CZW+22]	Unspec.	Comparison between ciphertexts and Zero-knowledge proofs	Attacker tampering with medical records or fabricating false insurance claims	There is no secure channel between the two blockchains involved. Need to manage CAs	Medical records, personally identifiable information, insurance claims, reputation of institutions.	Private blockchain for private data and a public one for public data. Role-based access control.	Medical records, insurance funds, and personally identifiable information
[SY22]	Unspec.	Digital certificates and signatures	Attacker impersonating someone else	Multiple attribute and service providers	Access to online services	Delegation of identity management to attribute providers and storing of digital certificates on the blockchain	Digital Identity and access to the digital services
[MCK21]	Unspec.	Digital signatures and weighted P2P verification	Malicious oracle node feeding false data, attacker feeding false information to the blockchain	The oracle is made by decentralized nodes, there is an insecure network between oracle and blockchain nodes	Impacts transactions in the blockchain and the integrity of the off-chain database	Multiple oracle nodes needed to validate information and reward system for truthful oracle nodes	Data in the database, IoT ecosystem
[MCK19]	Unspec.	Oracle compares sensor data to acceptable data	Attacker feeds false information to the blockchain and breaks food supply chain	There are multiple organizations involved, each with different standards	Resilience of supply chain, products, and consumers	Quorum system for approval of new members, each member has a list of public keys it accepts	products in the food supply chain
[BHP20]	Unspec.	No data validation	Malicious oracle, attacker compromises oracle	Oracles are a single point of failure	Data to be stored on the blockchain	Oracles linker as interface between consumer smart contracts and oracles; Oracles selected by Random Oracle Provider and Oracles store; reputation system	Data in the blockchain
[HCL+22]	Unspec.	Done by TPM	Data poisoning to the smart health devices	There are multiple devices and oracles in the network	Reliability of data of patients	Whitelist with unique identifiers of smart devices and use of TPMs to sign data sent to the blockchain; multiple oracles need to verify the data	The health of the patients involved

[CYX21]	Unspec.	On-device	Attacker feeding false data to the blockchain or compromising an oracle	There is no default channel of information between the source and the blockchain; multiple oracle nodes	Smart contracts in blockchain	Trusted Executed Environments in TPMs	Data and funds in the smart contracts
[ABAQS+19]	Unspec.	Unspecified	Attacker trying to get access to get unauthorized access to Data	IoT devices cannot store Access control lists (ACL); ACL need to be managed	Data confidentiality	Access control managed by smart contract; ACL on blockchain	Data in the external sources
[WZSK21]	Unspec.	Validation of metadata of files by smart contract	Attacker tries to access confidential data	Decentralized storage does not hold access control policies	Confidentiality of data	Access control policies stored either on Blockchain, Oracle or Storage	Data on the decentralized storage

Data source: There are many possible data sources presented in the surveyed papers, and they can be all included in two main categories, external (for inbound oracles) and internal or blockchain (for outbound oracles). The first category can be further divided into three categories: the first one being an IoT device, for example, a medical device or a sensor in a supply chain; the second one being a web source, either a website or an IPFS; the third and last one being the result of a computation performed in a trusted environment, or by a trusted device.

Communication protocol: There is not much variety in the communication protocols being used. Only HTTPS/TLS is sometimes mentioned.

Data validation: on this side, there is a bigger variety of techniques. In many cases, we either have no data validation or if there is it is still unspecified. A common technique is to perform data validation on devices, especially common for IoT devices. Another common solution is to use cryptographic techniques more or less sophisticated, such as digital signatures or zero-knowledge proofs. Finally, there are some implementation-specific techniques.

Threats: the most common threat analyzed in the papers is that of an attacker feeding false information to the blockchain or trying to get access to confidential information, whether it be personally identifiable information, such as in the case of voting or healthcare, or other information such as in a more general access control situation. Other threats that are considered are data tampering, impersonations, denial of service, and in a few cases the case of oracles being compromised and/or malicious is also considered.

Vulnerabilities: the most common vulnerability is the lack of a default channel between the data source and the blockchain. In some cases, this is mitigated by the use of HTTPS/TLS. Another common vulnerability is the fact that in many cases a centralized oracle is also a single point of failure, which is vulnerable to denial of service or can be more easily compromised compared to a network of oracles. In other cases, there can be multiple organizations involved in an oracle, and thus one has the need to manage identities through the use of CAs or other identity management solutions.

Impact: the impact is not thoroughly analyzed in the papers. As most of the papers dealt with more general situations, they limited their analysis of the impact on the funds and the normal operations of smart contracts. Papers that analyzed more closely a specific use case, introduced a more specific description of the impact. In particular, in the case of healthcare, personally identifiable information, personal data, and the reputation of medical professionals and institutions, such as hospitals and insurance companies. The reputation is impacted also in all cases where a user or

a smart contract needs to make a choice, either between different data sources or institutions with which to interact. Resilience and strength of supply chains or access to data are also impacted by attacks.

Assets: the most common asset that the papers present of a blockchain are the data funds and the smart contracts themselves. Then, similarly to the case of the analysis of **impact** as above, papers analyzing some specific scenarios tend to have a more specific description of the assets. In some cases, the assets are the personally identifiable information of people, or reputations of either software, professionals, or institutions. For example, in the case of voting, the outcome of an election is to be considered a major asset, just as important as the information of the voters and the secrecy of their vote. Or in the case of IoT ecosystems, the integrity and resilience of the ecosystem are to be considered the main asset to be protected.

Countermeasures: there are many different countermeasures taken, which in many cases depend on the scenario taken into consideration. Authenticity proofs are a common one, as they allow one to prove that data was indeed obtained by a specific web source. Another common strategy, especially in the case of the IoT ecosystem is the use of secure modules to generate signatures that prove the authenticity of the data, and in these cases, there is a list with all of the acceptable public keys or digital certificates accepted by the IoT blockchain ecosystem. In other cases, especially where there can be multiple data sources, there are reputation systems being used, where if a source is consistently providing truthful data it will be rewarded and used more frequently; conversely, it will be demoted and used less frequently if it holds a bad reputation. There are also more sophisticated techniques, involving multiple different technologies for some specific use cases, such as the use of access control lists (either on-chain or off-chain), or even the use of multiple blockchains interacting with each other, both public and private, for different kinds of data, depending on whether the data is public or private.

2.9 Answers to Research Questions

The results of the literature review have been presented in **Section 2.7**. Now we will use those results to answer the research questions that were asked at the beginning of this chapter. We will answer each question in order.

[RQ1.1] What are the assets that need to be secured when designing a blockchain oracle?

The assets that need to be secured when designing a blockchain oracle depend on the specific scenario we are discussing. In general, in all cases, the data itself and the normal operation of the smart contracts involved is to be considered an asset. This is particularly important in the context of blockchain, as transactions are irreversible and data, once stored, cannot be deleted. Other assets to be considered rely on the specific scenario being considered. For example, in the case of voting [FAB21], personally identifiable information related to the voter is an asset to be protected, and it is also subject to regulations in different countries, such as the EU's GDPR regulations. In the same example, the result of the election is to be considered an asset in and of itself,

as it can have drastic consequences on people's everyday lives, especially if it is to be adopted for political elections. In the case of Healthcare, [CZW+22], we also have additional assets such as the patient's medical records and the insurance claims, as we do not want an attacker to tamper with the blockchain, nor false insurance claims or unjustly considering a legitimate insurance claim as false.

[RQ1.2] What are the most important attack vectors and threats against blockchain oracles?

The most important threat is an attacker feeding false information to the blockchain. The reason is that if the blockchain is fed false information, then the normal operation of smart contracts will be tampered with. Furthermore, depending on the scenario, this can have more important consequences. In particular, in healthcare [CZW+22] it can impact the databases of hospitals or insurance companies, resulting in claims not being processed, false claims being processed instead, or the loss and tampering of important medical data related to the patient's health. In this case, the attackers can be competitors that try to damage the reputation of legitimate companies. Other attackers can instead try to process false claims or modify medical records, either to fabricate claims or to tamper with someone else's information. In the case of voting [FAB21], the main attack vectors can be either foreign adversaries, terrorist groups, or other organizations trying to influence the outcome of elections so that they get a favorable one.

[RQ1.3] What are the most efficient techniques to secure the different blockchain oracles?

The most efficient techniques in securing blockchain oracles reside in the use of strong cryptographic primitives. In particular, the use of TEE or other secure chips such as SGX is commonly used, as it manages to secure the cryptographic secrets in a way that they can not be tampered with by attackers. It will be the role of the oracles or the off-chain entities to verify the signatures of the data being exchanged.

2.10 Limitations

There are threats to the validity of the presented SLR. The first one is that the review focused on centralized blockchain oracles. Thus, the discussion ignored decentralized blockchain oracles. The current landscape includes many different decentralized oracles and thus any discussion about the security of blockchain oracles that does not include decentralized oracles is excluding a part of the landscape that is already in use. The second threat to validity is due to the sources used, as the literature review relied on the sources that were found. There cannot be an SLR fully comprehensive of all possible sources, whether they are scientific or grey literature. The third threat to validity is due to the fact that the SLR was based on looking for existing or proposed solutions to specific use cases. This excludes all work that focuses on general discussions about blockchain oracles, without a specific scenario in mind. Finally, the last threat to validity is related to the economic aspect that was considered out of scope for this work. Given the nature of blockchain applications, the economic aspect should always be kept in consideration whenever designing a solution for any scenario.

3 Background Technologies

This chapter focuses on the technologies that will be used in implementing blockchain oracles in our experiments. There are multiple technologies and multiple implementations of the same one; for this reason, it is needed to take a step back and focus on the technologies and explain both the reasoning why a certain choice was made and how it works. This chapter acts as the technology selection phase of our design science approach.

3.1 Oracle Services

Provable [Pro] makes use of TLSNotary [TLS] in order to generate proof of authenticity. The proofs are digitally signed by a Notary and can be used to generate proofs of content on a web application. There is a form of trust, in the sense that one needs to trust that the Notary does not provide malicious proof by colluding with the web application. One way to mitigate that risk is to require multiple Notaries to provide multiple proofs, but we will not be considering the possibility that a Notary is colluding with the web application. The security is entirely software-based and relies on the TLS protocol.

TownCrier [Towb] instead makes use of Trusted Execution Environments (TEE) in order to generate proofs. A TEE provides both integrity and confidentiality of all of the operations being performed in the TEE. The TEE is executed on an Intel secure chip called SGX, but in our simulation, we will be using a docker container. Comparing TownCrier and Provable allows one to see how hardware-based and software-based solutions might differ. There is a major hurdle though, which is that TownCrier was acquired by Chainlink [Towa] in 2019. Today, the TownCrier project is deprecated, and the TownCrier TEE technology is used inside the Chainlink nodes to make them more secure. To our knowledge, it is not possible to leverage the TownCrier technology to make oracles anymore. Even if it were, this project is deprecated, and thus for research purposes, it is more important and productive to focus on new technologies that are still being used and developed today over deprecated ones.

Outbound blockchain oracles differ in nature from inbound oracles. The two previous solutions deal with inbound oracles. To the knowledge of the writer, there is not much research done on the security of centralized outbound oracles for public blockchains, as the research on security aspects focuses mostly on decentralized oracles or proposes the usage of private/permissioned blockchains. The solution that we propose is to design the blockchain smart contract in such a way that it can send information only to a specific address. In this way, the integrity of the data and the authenticity are provided by the blockchain itself, whereas confidentiality is not provided. The reason why this design was chosen is that most of the oracle technologies and research focuses on inbound oracles, as the data in the blockchain cannot be modified and it may influence the operation of other smart contracts or other applications. When we are instead making an outbound oracle, we control both the sending address, which will be the oracle address and the receiving address. The oracle sends data as part of a transaction between the addresses, and thus in most

circumstances, one only needs to set up the receiving address. The situation is different in case the receiving address cannot be set up priorly and needs to change in function of who is querying the oracle. In this case, there can be multiple solutions, with the trivial one being that the receiving address is the same as the one querying the oracle. More sophisticated solutions are reliant on the specific scenario that is being considered, and as we have seen in the systematic literature review, in many situations a permissioned or private blockchain may be the better solution as it allows more flexibility in managing transactions and access to data on the blockchain.

Chainlink [Chaa] is an ecosystem that provides a decentralized oracle. Chainlink has an architecture relatively similar to that of blockchain, as its oracle is composed of multiple different oracle nodes that validate the information. There is a consensus mechanism that allows one to settle for one specific value based on the number of nodes agreeing on the value of the data. Different oracle nodes can perform different jobs, which means that they can perform different operations and query different data. This oracle was originally intended to be excluded from our work, as the thesis focuses on centralized oracles and not decentralized ones since the security considerations differ greatly. But since we discovered that TownCrier is a deprecated project, we decided to use Chainlink in order to implement our smart contract proposing a novel approach that we will be explaining in detail in a dedicated section.

Augur [Aug] is a decentralized protocol on the Ethereum blockchain that provides the ability to create prediction markets that can be used to bet on world events, financial markets, or sports events. The protocol also provides a decentralized oracle with the peculiarity that the source of data for each oracle node is a human. Each user has to stake an amount of a cryptocurrency (either REP or REPv2) that corresponds to his or her reputation. The event that will be considered true is the one that has the most reputation staked in. If the user stakes REP correctly, then they will gain more REP, otherwise, they will lose their stake. Augur allows to create prediction markets on events that cannot really be queried automatically by computers, such as "Did X betray his country?" or "Was X the worst thing to happen in year Y?". In these situations, data is not enough to simply answer those questions and thus the so-called "wisdom of the crowd" is required. Again, the reason why Augur was excluded is that it is out of the scope of this work, as it is a decentralized oracle network. Furthermore, it adds human and economic issues due to the way that the oracle settles for "true" data.

Tellor [Tel] is another decentralized blockchain oracle similar to Augur that allows users to report data to the oracle. The Reporter has to submit data and stake some of their tokens. The data can be disputed for 12 hours by any Disputer who has to match one-tenth of the Reporter's stake. During the dispute, multiple rounds can be performed where other users vote on the data to settle the dispute. After the dispute is settled, the winner gets the other party's stake. The reason why Tellor was excluded is that it is a decentralized oracle and its security mostly relies on the stake-and-dispute mechanism.

Ultimately, the chosen technologies are Provable, Chainlink, and a custom outbound oracle. The reason is that they all meet our criteria of being centralized oracle technologies that were found in the systematic literature review.

3.2 Ethereum Endpoint and Web3.js

When building an outbound oracle, an important piece in our system is the Ethereum endpoint. An Ethereum endpoint [Ethb] is a URL address that allows one to interact with the Ethereum blockchain using a specific protocol. The main operation that an endpoint allows that interests us is that endpoints allow us to query data from the blockchain network. This action is key to building our outbound oracle, as it will need to be able to query the blockchain in order to get data that is stored in smart contracts. It is possible to run your own endpoint, which involves running an Ethereum node with all of the associated costs, with the main ones being bandwidth and storage [Etha]. There are multiple different companies that offer Ethereum endpoints, and we will be using Infura [Inf] as it provides a free option that satisfies our requirements; furthermore Infura also interacts with both the Sepolia and Goerli testnets.

One of the ways to interact with an Ethereum endpoint when developing a web application is to use Web3.js [Web]. Web3.js is a popular JavaScript module that is used by developers to create a web application that is able to interact with the Ethereum network. It provides an interface to perform most actions that one would need to interact with Ethereum, such as sending transactions and executing smart contracts. Web3.js is an essential tool and for this reason, it has become the standard for Ethereum development. We will make use of Web3.js to build our outbound oracle.

3.3 Difference between Testnet and Mainnet

Mainnet and Testnet are commonly used terms in Blockchain [Tes]. *Mainnet* refers to the main network that a blockchain is being run on. It is the chain that is in charge of transferring values between different addresses and that is used by the public. A *testnet* is instead a network that is used to develop and implement projects before they are deployed in the mainnet. There can be private testnets and public testnets, the first ones are usually managed by the individuals or the team developing a specific project and the second one is managed by a network of nodes in a similar way to the mainnet. A private testnet allows to test smart contracts in a secure and private environment and allows the developers to test the contracts without having to get test Ether, which can sometimes be a challenge as one can get test Ether either by someone else who does not need their test Ether anymore, or from *faucets*, tools that allow one to get free test Ether in their address. A public testnet cannot transfer valuable tokens, which means that it cannot transfer value between two addresses and should only be considered as a test environment to deploy and test the different projects without incurring the risks of deploying an unsafe project into the mainnet. There usually are multiple public testnet for each mainnet, usually because each testnet can have slight differences that can make it more useful for development, such as a higher block throughput or a different block size. Other testnets can have the same parameters as the mainnet and serve as a faithful reproduction of the mechanisms of the mainnet.

3.4 Remix IDE

The chosen IDE is Remix IDE as it is the most commonly used one for Ethereum. It allows us to program smart contracts, and deploy and test them, both locally and on test clients. For ease of development and to have the most general setting possible, the IDE is accessed through this URL: https://remix.ethereum.org/. Remix IDE allows one to perform multiple operations, in particular, one can create a workspace that will act as its own root directory with multiple sub-directories that can be organized as the developer wishes. The contracts need to be written in Solidity and the extension of a Solidity file is *.sol*. Once the contract is fully developed, one needs to first compile it and later deploy it. When it comes to deployment, there are multiple options that either use local VMs or chains, public testnets, or the Ethereum mainnet itself. Whenever deploying on a public network, no matter whether it is a test network or a public network, one needs to have enough Ether (test or main ether) to fund the contract and pay the related gas fees.

3.5 Web Development

We will be building inbound and outbound oracles. In both cases, we will need to build a web app that is able to interact with the blockchain, either directly or indirectly. In the case of inbound oracles, the web app will act as a data source, whereas in the case of the outbound oracle, the web application will act as the destination. Web development techniques and technologies are outside the scope of this work, but it is imperative to explain the technologies used for our implementation of oracles. In order to make use of Web3.js, we will be using Node.js to create our web apps. We will be using some specific modules for the Node.js framework. In particular, we will be using crypto.js, which is included in the standard Node.js package, as our library to use the cryptographic primitives we will be needing. We will also use Express.js as a module to create the web app and manage routes, middleware, and HTTP requests and responses. The module Cors.js will be needed as well to make the app accessible from any IP and we will also use Body-parser.js to parse the HTTP requests that we will be receiving. We will also use Nodemon.js to run the application in a test environment for ease of development and logging and dotenv.js is needed to interact with the execution environment and store the secrets that are needed when the application is running, such as the database password and the private key used to generate digital signatures.

As our data source for our inbound oracles will be needing a database, we will be using MongoDB Atlas, as it provides a free option that suits our needs. Furthermore, MongoDB Atlas offers an instance of MongoDB which is a NoSQL database that is well-suited for JSON objects. In order to interact with MongoDB, our Node.js app needs Mongoose.js, which is a module designed to interact with it.

Finally, for ease of development, we will use Render.com, which at the time of writing offers a free option, in order to deploy and run the web app once its development has been completed. Furthermore, Render.com natively offers an HTTPS/TLS certificate, which will be needed for TLSNotary [TLS] and Chainlink [Chaa]. Finally, Render.com supports Continuous Integration/Continuous Deployment

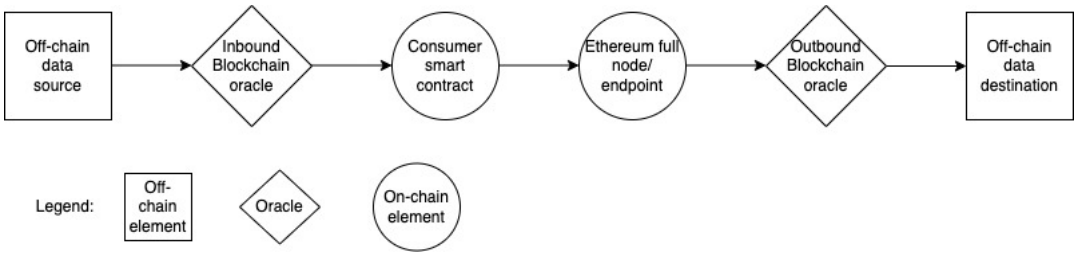

Figure 1: System using blockchain oracles

(CI/CD) development using GitHub repositories, thus for each web app, we will also be setting up a CI/CD pipeline with a GitHub repository.

3.6 Components of Blockchain Oracles

A system that makes use of blockchain oracles is fairly involved, as it has multiple interacting components. Figure 1 shows a small graph that at a glance explains the flow of information between the different interacting components. Squares are used to show elements that are off-chain, rhombuses are used for blockchain oracles, and circles are used for on-chain elements.

In our experiments, the off-chain components are always web apps, we have one app that acts as the data source and one app that acts as the data destination. The two blockchain oracles are instead different in all three experiments we performed. The blockchain oracles act as bridges between the on-chain and off-chain world and are at the same time on-chain and off-chain components. The first blockchain oracle (the one on the left) is either the Provable oracle service or a Chainlink oracle node. We call Provable an oracle service as it has an off-chain component that interacts with our data source and sends the information to a smart contract that interacts with our consumer smart contract. We instead use a Chainlink oracle node as Chainlink is a network of nodes, where each node is an oracle service similar to Provable. The consumer smart contract is the smart contract that queries the oracle to get information about the off-chain world. The contract will then use the information for its purposes according to its programming instructions. The smart contract resides on the Ethereum blockchain, which resides in the distributed network made up of Ethereum nodes. In order to interact with the blockchain from the off-chain world, an endpoint is needed, but ultimately an endpoint is just an Ethereum node accessible from a URL. This is still considered an on-chain element. The second blockchain oracle (the one on the right in our figure 1) is the bridge that allows querying the blockchain. In our experiment, the oracle is a role performed by our destination web app, as it uses the Web3.js library to interact with the Ethereum endpoint and bring on-chain data to the off-chain world. Finally, the off-chain data destination is where the on-chain data will be manipulated off-chain. In our experiment, this is our second web app.

The security properties of each of these components influence the security of the whole system. We will discuss them one by one. Let us start with the on-chain components, in particular, we will discuss the consumer smart contract first and later

the Ethereum endpoint. The blockchain provides data integrity and data authenticity by design. In particular, once a transaction is made on the blockchain it is irreversible and the data stored on the blockchain cannot be modified, and this is also true for smart contracts, as they are code that is executed on the EVM. This means that the smart contract code is a potential vulnerability. Once the smart contract is deployed, its code cannot be modified and thus if there are any bugs or vulnerabilities in the smart contract code, then we have a major vulnerability. The most common ones are the operational and business logic of the smart contract, the funds that are stored on it, and the ownership of the contract. Data authenticity in the Ethereum blockchain is provided through the concept of accounts, as each account can perform transactions, and the information on those transactions is stored on-chain and no transaction can be reversed. The transactions are signed using the account's private key in the case of user accounts, or just by running the code if it is a smart contract account.

The Ethereum node and its relative endpoint operate along the rules of the Ethereum protocol, which can be considered secure for our standards. When the node acts as an Ethereum node, it acts as a peer-to-peer node in a network regulated by Ethereum rules. Again, data integrity and authenticity for the data, which is entirely on-chain in this case, is guaranteed by Ethereum's properties. When the node acts as an endpoint, it needs to act as a server that listens to clients' requests and prepares and sends responses. If we assume the endpoint to be trustworthy, then this means that the endpoint always queries the data from the blockchain and prepares the right response. The vulnerabilities here are related to Denial of Service attacks, as a node acts as a server and it can be overrun by a number of requests that is higher than what the network can handle, and to the communication between the Endpoint and its clients. There are multiple protocols that have been proposed for endpoints [Ethb] and all of them make use of HTTPS/TLS to establish an encrypted and secure channel between the client and the endpoint. The main vulnerability thus relies upon the management of the TLS certificates and of the TLS private key used to establish the connection. Thus, if an endpoint gets its private key compromised, it cannot be trusted to establish secure channels between its clients.

If the endpoint is not trustworthy, then the above vulnerabilities still apply, with the addition that even if the communication happens flawlessly and securely, the endpoint can send false information. There is no way to protect against this situation, as the client cannot query the blockchain directly. For this reason, it is necessary to take countermeasures against this situation or to run your own node/endpoint, which can be trusted as it is self-managed.

The off-chain components need to be secure as well. The main vulnerabilities that they can bring are related to the communication between the off-chain component and the oracle. Thus the goal is to establish a secure channel between the two. This is the reason why the oracles that were used in the experiments take advantage of HTTPS/TLS properties to establish secure channels. This means that if the private key used for TLS is compromised, the entire web app is compromised and so is the communication between the oracle and the web app, and thus we cannot trust the information we get from it. There can be other security vulnerabilities. For example, the off-chain data source makes use of a database, which can be attacked, and is a

centralized data source, which means that it can be victim to Denial of Service attacks and other vulnerabilities related to centralized infrastructure. These vulnerabilities, though, do not affect the way that the oracles and this system operate directly. In case of Denial of Service, the consequence will be that the oracle is not able to fetch the data and feed it to the blockchain, which means that the consumer smart contract may not be able to operate as intended, and some countermeasures need to be taken. If the smart contract receives false data, then it will be stored and it will permanently reside on the blockchain. If instead the database is successfully attacked, this means that the web app will be malfunctioning and that it may provide false data, as its database is compromised, and the consequences for our system are similar to the other case we just discussed.

Finally, the last piece we need to discuss is the off-chain data destination. The security of the data destination does not influence in any way the security of the other elements of the system. As the flow of information goes from left to right, this means that the security of an earlier step influences the security of a later step. But there is no other component that follows the off-chain data destination, which means that its security does not influence the rest of the system. The data destination is the client of the Ethereum endpoint and can use different protocols to interact with it, but all rely upon TLS. Because of TLS properties, if the private key of the Ethereum endpoint is compromised, then the off-chain data destination cannot trust the data it receives from the endpoint. The off-chain data destination technically does not even need to have a permanent TLS private key and can use a simple ephemeral one. The security of the off-chain data destination thus only influences its users and its business logic, but not the security of the rest of the system as a whole.

3.7 Summary

As a summary of this chapter, it is important to mention that there are multiple technologies involved in making a blockchain oracle, as we have a system with multiple components interacting with each other. We need to make use of web3 technologies, in particular of endpoints to interact with the Ethereum blockchain, and of web development technologies to have an interface to interact with the blockchain itself. The technologies that we will be using are thus:

- Ethereum blockchain, in particular, the Goerli and Sepolia testnets
- Node.js with some modules
- Infura ethereum endpoint
- Provable
- Chainlink
- MongoDB
- Render.com and Github for CI/CD pipeline

4 Implementing Blockchain Oracles

This section of our work addresses the second research question **[RQ2] How to implement two widely used blockchain oracles securely?**. This chapter is the technology implementation phase of our design science approach. In order to answer this question, we need to divide it into subquestions:

- **[RQ2.1]** What are the assets to implement for a blockchain oracle?
- **[RQ2.2]** What are the threats to be taken into consideration?
- **[RQ2.3]** What are the countermeasures to be implemented?

As part of our work, we implemented three blockchain oracles and we will later evaluate them to see how well they satisfy the security requirements. Since there is more material available on Solidity and Ethereum and since it is the framework that we are most familiar with, the oracles will be implemented using Solidity for the Ethereum blockchain. The first two are inbound oracles using similar technologies, and the last one is a custom outbound oracle. This is to see how off-chain data can get included in the blockchain as well as how to get on-chain data into the off-chain world.

We will be using *Provable* [Pro] and *Chainlink* [Chaa] for the inbound oracles as they have been used or mentioned quite frequently in the papers surveyed for the literature review. The custom oracle instead will not make use of any specific technology and will make use of specific smart contract properties to function. The reason is that research around outbound oracles either focuses on decentralized oracles or on private/permissionless blockchains.

As was mentioned in the **Introduction**, blockchain oracles are a special type of smart contract where we have an initiator and a responder. Thus one could model the oracle as an interface between the *off-chain* and *on-chain* world. Let us assume we have a network with three nodes, one being the entity using the oracle, the oracle itself, and then the rest of the blockchain. We have two edges in this network, one between the oracle and the rest of the blockchain and one between the oracle and the *off-chain* entity. These edges represent channels of communication. We can reasonably assume that the channel between the blockchain and the oracle is secure. This is because the oracle is simply a smart contract running over the blockchain, and thus it simply is some code being executed over the multiple nodes of the blockchain. The same smart contract will always produce the same output even when run on different nodes. We will, instead, assume that the channel between the oracle and the entity is not secure, and thus we can model it using the traditional Dolev-Yao model. Thus, while implementing our oracles, we want to evaluate whether or not the communication between the oracle and the *off-chain* entity can be successfully attacked. The definition of "successful attack" will depend on the oracle model being used. Here is a thorough explanation:

1. **Push-based inbound**: in this case, an attack is successful if an attacker can either tamper the data being sent from the off-chain entity or if it can send data to the oracle in some way.

2. **Push-based outbound**: in this case, an attack is successful if an attacker can either tamper the request being sent from the on-chain entity or fabricate a false request.

3. **Pull-based outbound**: in this case, an attack is successful if an attacker can either tamper or fabricate a request or if it can fabricate or tamper the response.

4. **Pull-based outbound**: in this case, an attack is successful if an attacker can either tamper or fabricate a request or if it can fabricate or tamper the response.

Another problem, commonly known in the research as *The oracle problem* [Cal20a], is that blockchains are decentralized by design, and oracles introduce some centralization to the system unless they are decentralized oracles, which are outside the scope of this research. The oracles introduce centralization either as a single source or destination of information. Especially in the case where they are a source of information for the blockchain, there is no guarantee that the entity sending information to the oracle is sending truthful information. Thus the oracle is introducing an element of trust in the blockchain, as one has to trust that the entity is providing truthful information through the oracle. This problem is not of easy solution, and for this reason in our analysis, we will assume that the entity providing data to the oracle is providing "true" or "truthful" data. The expression "false" data will refer to data sent by an attacker to the oracle, either through fabrication or tampering with originally truthful data.

We will be explaining our implementation choices in the following subsections. The first one is about the Data source of our oracles and then the following two subsections will discuss the two oracles that have been chosen, Provable and Chainlink. We then proceed with the fourth subsection which focuses on the outbound oracle and the last two subsections will answer the research questions and address the limitations of this chapter.

4.1 Data Source

The data source for our inbound oracles will be a custom made web-application. The assumption is that the web app is secure, as we are not interested in the security of the web application. Furthermore, the web app will provide publicly available data. The web app can be thought of as a data source such as a weather website, a newspaper, or a repository of data for statistical analysis such as information about a country's economy.

This work does not focus on web development techniques, but for reproducibility reasons, it is important to outline the deployment mechanisms and the overall architecture of the web app. In particular, the app is available at this URL: `https://two-oracle-thesis-web-app.onrender.com/posts`. The app was made using the technologies listed below:

- Node.js / npm to create and manage the app (version 9.3.1)

- crypto.js library which is installed along with Node.js. It is used for generating an asymmetric key pair and to generate the digital signatures

- MongoDB Atlas is used as a Database
- Mongoose.js to interact with MongoDB from the app
- Express.js to manage Middlewares and Routes
- Cors.js to allow interacting with the app from any IP
- Body-parser.js to parse the HTTP requests and responses
- Dotenv.js to deal with environment variables and hide run-time secrets; in this case it is the database login information
- Nodemon.js to run the app in a test environment safely
- Render.com to deploy the app and manage CI/CD and to make it accessible from the web at a specific HTTPS-enabled URL
- a Github repository for CI/CD available at this URL: https://github.com/alexcarchiar/thesisWebApp

All of the chosen technologies are either open source, or proprietary but still offered a free option; the main reason behind the choice of these technologies was that it should be easy to reproduce and free. Furthermore, Render.com also offered at the time of testing an HTTPS certificate, which is needed for our purposes, in particular for Provable/TLSNotary.

The oracles are designed to interact with the app through a REST API. The reason is that the oracles that were chosen require interaction with an interface in order to get back data. All of the requests either provide or need JSON data. In particular, there are only three possible requests to the API:

- GET /posts - Provides all of the posts that are on the database
- POST /posts - Allows one to submit a new article to the WebApp
- GET /posts/latest - Allows one to get the latest article that was submitted to the WebApp

The inbound oracles perform a GET /posts/latest request; the other two requests were made for ease of development and testing. A similar app is the destination of the outbound oracle. The app queries the blockchain in order to get information about the data submitted to it. For simplicity reasons, the outbound oracle provides the same data that is submitted to the blockchain from one of the previously designed inbound blockchain oracles.

There are some main development files that are used to implement the web app and define the required functionality. The main files are app.js which acts as the main execution file; routes/posts.js which contains the routes used for the API requests and finally models/Post.js which contains the database schema to define database objects in the web app. The code for each of these files is available in the appendix and here you can find an explanation of the code. We will be referring to the corresponding listing.

app.js

The code is reported in Listing 4. The first few lines import the required modules, for example, Mongoose or Express. The third line `require('dotenv/config')` specifies the file containing the environment variables, in this case, they are the login information to connect to the database. `app.use(input)` allows to add middlewares to the server. In particular, the `app.use('/posts',postsRoute)` allows one to define that all of the requests being made to /posts, need to use the middleware `postsRoute`, which is a custom-made module defined in `routes/Posts.js` that we will explain later. We define a get request with `app.get('/')` for the homepage of our web server. The response is just the string `'We are on home'` followed by the public key used to verify the signatures of the various different `Post` objects that are dealt with by the web app. The public key is an environment variable. This exists to check that the website is up and running as well as to let a theoretical user know the public key to verify the digital signatures. The last lines are used for the database connection and to open a listening port so that the web app can receive requests.

posts.js

The code is reported in Listing 5. The first two lines allow us to import the `Router` object from the Express module. The third line imports the `/models/Post` custom module to obtain the object schema of a post used to interact with the database. The `Router` is used to define the different API requests. As one can see, we call two methods of the router to generate three functions: the `get` and `post` methods, which are used to deal with either GET or POST HTTP requests. In both cases, we first define the relative path of the request and then we define the function dealing with the request. The two `get` methods return respectively all of the posts or just the latest one; the `post` method instead allows to create a new Post. In particular, it obtains the title and the description from the request body and generates the signature. Finally, the file defines the export object itself.

Post.js

The code is reported in Listing 6. The first line imports the Mongoose package which is needed in order to create the object schema to interact with the database. The schema is then defined in the following lines: it is a `PostSchema` which contains three required strings, one called `title`, one called `description`, and one called `signature`. In our simple experiment, the public data source acts as a "flash news media repository", and each piece of flash news is defined by a title, which is used to captivate the reader, and by a description, which is made up by a couple of lines that are needed to give some basic context to the title. The signature is a digital signature generated using a private key that is kept by the web app. The signature is generated by appending the `title` and `description` strings, and the goal is for the web app to provide an attestation of this data so that it can be verified using the public key available on the home page. The last line defines the export for this file, in this case, we use the Mongoose package to explain that we just created a model named `Post` for our schema named `PostSchema`.

4.1.1 Generating Key Pair

The generation of the key pair was performed once, and then the generated public and private keys were added to the environment variables. In order to make this proof of concept reproducible, you can find in Listing 1 that reports the code that was used to generate the keys and later to import them and check that the whole process was working as intended. The file is not intended to be run as is, rather you should first run the part related to the generation of the keys, then add the keys to the execution environment. Finally, you can run separately the last lines to check that the keys are imported in Node.js.

```
% importing the cryptography module
const crypto = require('crypto')

% generating the key pair for elliptic curve cryptography
var keyPair = crypto.generateKeyPairSync('ec', {
  namedCurve: 'secp256k1',   % specifying the curve
  publicKeyEncoding: {
    type: 'spki',
    format: 'pem'
  },
  privateKeyEncoding: {
    type: 'pkcs8',
    format: 'pem'
  }
});

console.log(keyPair.privateKey)
console.log(keyPair.publicKey)

% the following lines check that it is possible to get the private
    key from the execution environment and generates a private key
    object
private_Key = process.env.PRIVATE_KEY
console.log(private_Key)
console.log(crypto.createPrivateKey({
    key: private_Key,
    type: 'pkcs8',
    format: 'pem'
}))
console.log(crypto.createPrivateKey({
    key: private_Key,
    type: 'pkcs8',
    format: 'pem'
}).export({
    type: 'pkcs8',
    format: 'pem'
}))
```

Listing 1: generateKey.js is the file to generate private and public key pair

The first line imports the crypto.js method. Then, the following lines are assigned to the variable keyPair. These lines are the calling of a function in the crypto.js module

named generateKeyPairSync(), which as the name suggests, generates a key pair in a synchronous way. It takes two inputs. The first input is the string 'ec' specifies the key type, which in this case is identified by the following OID: 1.2.840.10045.2.1 . The second input is an object that specifies the parameters of the key pair we are generating. In this case, we use the elliptic curve secp256k1 and for both private and public keys we specify the encoding as a PEM file. The PEM file will be of type spki for the public key and of type pkcs8 for the private one.

Afterward, the console outputs are used to print the keys to the console, so that they can be manually added as variables to the execution environment. The last lines simply import the private key from the execution environment and later use console messages to test that the key was correctly imported as a string. Subsequently, the console messages take as input the output of crypto.createPrivateKey(). This function takes as input an object that specifies the key and the key parameters in a similar way as above. The first time this function is called, it is directly given to the console, which will print the key object. The second time, instead, we give to the console logger the output of the export method of the key object. The export method takes as input parameters that specify how the key should be exported as a string, and again we use pkcs8 type and PEM format like above.

4.1.2 Generating and Verifying Signatures

The generation and verification of digital signatures are explained in Listing 2. The same lines are present in the routes/posts.js file, but here we will go more in detail. This file can be run as is if you have already set the private and public key pair as environment variables in your machine.

```
% importing the cryptography module
const crypto = require('crypto')

% getting the private key from the environment
const private_key = process.env.PRIVATE_KEY

% setting the parameters for the digital signature
const signing_algorithm = 'sha256'
const format_string = 'hex'

% getting the public key from the environment
const public_key = process.env.PUBLIC_KEY

% setting two example strings for generating and checking the
    signature
const first_string = "example1"
const second_string = "example2"

% generating the digital signature by appending the two strings
let signer = crypto.createSign(signing_algorithm)
    signer.update(first_string)
    signer.update(second_string)
    signer.end()
let signature = signer.sign(private_key,format_string)
```

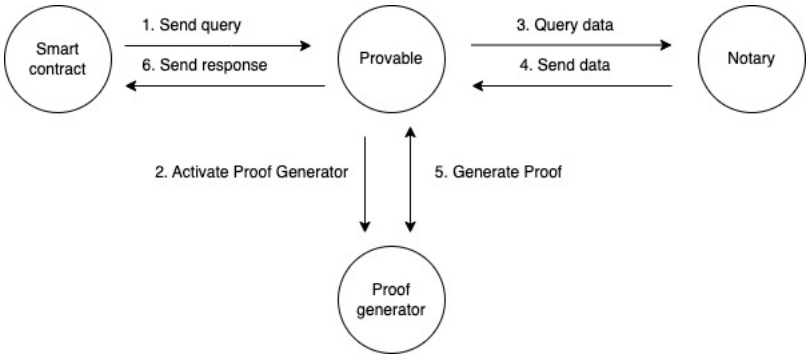

Figure 2: Graph showing the design of Provable oracle

```
% verifying the signature
let verifier = crypto.createVerify(signing_algorithm)
    verifier.update(req.body.title)
    verifier.update(req.body.description)
    verifier.end()
    console.log(verifier.verify(public_key, signature,
        format_string))
```

Listing 2: verifySignature.js file to verify a digital signature

The first lines define constant strings that will be used for the signing and verification process. The first one imports the `crypto.js` module. The names of the constants are self-explanatory. The private key and the public key are obtained from the execution environment. The hashing algorithm that was chosen was SHA-256 as it is secure and commonly used in the industry. The `format_string` constant is used to represent the signature as a hexadecimal string. Finally, the last two constants are two strings that are simply used as examples to show how to use the signing and verification procedures.

The second group of lines is used to generate a signature. We first have to create an object `signer = crypto.createSign(signing_algorithm)`. This object is used to first append the two example strings using its `update()` method and finally, the signature is generated using its `sign()` method.

The third and last group of lines is used to verify the signature that was just generated. A `verifier` object is created using `crypto.createVerify()` and the strings are appended using its `update()` method. Finally, the signature is verified using the `verify()` method. The output of this call is given to the console logger which will print either `True` or `False` depending on the result of the signature verification.

4.2 Provable

Provable [Pro] is an inbound blockchain oracle that can have multiple different types of data sources. Figure 2 shows how the Provable oracle works at a glance. The data sources include URLs, the WolframAlpha computational engine, IPFS, a random byte generator, and the result of a computation. Provable is to be used as a trusted oracle, as it does not provide any security measures by default. Provable allows requesting

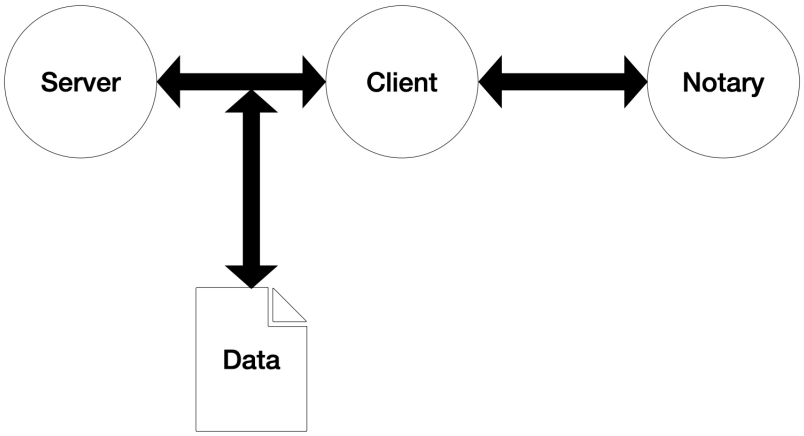

Figure 3: Graph showing TLSNotary in the general scenario [TLS]

authenticity proofs of the data Provable is submitting. The proofs are generated using TLSNotary [TLS] or there are other solutions, but we will focus on TLSNotary since it was the one mentioned in the papers found during the literature review.

TLSNotary is explained at a glance in figure 3. TLSNotary proofs make use of TLS to generate a transcript of communications that took place between a web server and a Client-Notary pair. The Client-Notary pair have one common TLS key, but none of them have access to it. They have access to their share of the key and thus the communication between the Client-Notary pair and the web server needs the pair to perform 2-party computations in order to encrypt and decrypt the communications between the pair and the server. This allows to generate a transcript of the communications and the transcript itself is the authenticity proof. It is important to also talk about other functions that Provable offers and that were not used for this work [Pro]. In particular, the other types of authenticity proofs offered are Ledger proofs and Android Proofs.

Android proofs make use of Google's SafetyNet, which is a remote attestation technology used to turn an Android device into a secure hardware device. The only way to generate a false proof would be to gain access to the physical device or exploit the Android OS itself with a vulnerability that is unknown to Google. Furthermore, there is a system in place to automatically update the device and generate valid proofs only if the device is running the latest version of SafetyNet. Ledger proofs make use of a Ledger Nano S, which is a hardware cryptocurrency wallet that is used as a secure environment to either generate random numbers or generate authenticity proofs. The advantage of using a Ledger is that the private key used to generate the authenticity proofs is stored on a secure chip in the device.

Finally, the last function that Provable provides is encryption, of both the request of data and the response. The encryption is based on Elliptic curve cryptography

and on Elliptic curve Diffie-Hellman key exchange. The reason why this was not used is that the function is currently available only on the mainnet and the scenario that we designed does not have a need for encryption, as the data is assumed to be publicly available. Another reason for the exclusion of this function is that no paper in the systematic literature review mentioned the use of it. Our smart contract was deployed and tested for the Goerli testnet. Below is the code of our smart contract using Provable oracle.

```
pragma solidity ^0.4.22; % specifying compiler version: at least
    0.4.22 and lower thatn 0.5

% importing the Provable API smart contract
import "https://github.com/provable-things/ethereum-api/blob/master
    /contracts/solc-v0.4.25/provableAPI.sol";


contract ExampleContract is usingProvable {

    % this string stores the information related to the JSON object
        obtained from the web app
    string public latestArticle;

    % this stirng stores the authenticity proof generated by
        Provable
    bytes public AuthenticityProof;

    % validIds is used to store the request IDs of the requests we
        make to Provable; it is used to avoid someone sending a
        response to a request that does not exist
    mapping(bytes32=>bool) validIds;

    % defining the smart contract events, their signatures are self
        -explanatory
    event LogConstructorInitiated(string nextStep);
    event LogArticleUpdated(string article);
    event LogNewProvableQuery(string description);

    % this is the constructor function that creates the smart
        contract and emits an event announcing its creation
    function ExampleContract() payable {
        provable_setProof(proofType_TLSNotary | proofStorage_IPFS);
        LogConstructorInitiated("Constructor was initiated. Call '
            updateArticle() to send the Provable Query.");
    }

    % the __callback function is run once the smart contract
        receives a response from Provable
    function __callback(bytes32 myid, string result, bytes proof) {
        if (!validIds[myid]) revert(); % checking that the ID
            matches a previously made request
        if (msg.sender != provable_cbAddress()) revert(); %
            checking that the sender of the response is Provable
            smart contract address
```

```
        % updating the string variables
        latestArticle = result;
        AuthenticityProof = proof;

        % emitting an event to log the obtained result
        LogArticleUpdated(result);
    }

    % the updateArticle function is called by the user whenever
        they wish to update the smart contract with the information
        about the latest article available on the web app
    function updateArticle() payable {
        % we first need to check that we have enough funds to pay
            Provable for the query
        if (provable_getPrice("URL") > this.balance){
            LogNewProvableQuery("Provable query was NOT sent,
                please add some ETH to cover for the query fee");
        } else {
            % if we have enough funds, then we are sending the
                request as a Provable query and we emit an event for
                 logging purposes
            LogNewProvableQuery("Provable query was sent, standing
                by for the answer..");
            bytes32 queryId = provable_query("URL", "json(https://
                two-oracle-thesis-web-app.onrender.com/posts/latest)
                .0");
            validIds[queryId] = true; % here we save the request ID
                 for a later check
        }
    }
}
```

Listing 3: Provable smart contract

The first line specifies the compiler version; in this case, it is higher than 0.4.22. The second line instead imports the Provable API that contains the functions definitions and extensions that we need in order to implement the contract.

The rest of the code is inside a structure called contract which is conceptually the same as a class in Java or Python. The keywords contract ExampleContract is usingProvable mean that we are creating a new contract named ExampleContract which extends the previously created contract usingProvable, which is imported at the second line of the code.

We then define our data structures. We have two publicly accessible strings, one to store the latest article and one to store the relative Authenticity proof. Each request to the Provable oracle generates a unique id which is used to request information about it and for security reasons, all messages relating to the same request need to have the same request ID. The mapping validIds is used to store these IDs.

We then define three events that are used when running the smart contract. Each event is used at the end of the execution of a function to log the current state for ease of debugging. The name of each event is self-explanatory.

We finally have three functions: ExampleContract, __callback and updateArticle. The first one is the constructor and it has the attribute payable which is used to add funds to the contract. This is needed as in order to make requests to Provable, one needs to pay some ETH. It is also used to pay for the authenticity proofs and encryption criteria; in this case, we are only setting the authenticity proof and we are using TLS Notary. The second function is the __callback which is called by the Provable oracle in order to send the response to the request it received. It has three inputs: myid which is the request ID, result which is the result of the request, and finally proof which is the authenticity proof. The functions perform some safety checks, in particular, it checks the validity of the ID and that the address of the contract calling __callback is valid as well. Finally, the contract variables are updated. The last function is updateArticle which is also payable in case one needs to fill up the funds of the smart contract. The function checks the contract has enough funds to pay Provable for the request using provable_getPrice ("URL". The input of this call is "URL" as the data source is a URL. If there are not enough funds, the smart contract logs it; if there are enough funds then the smart contract sends the query and logs it. Furthermore, the query ID is kept.

Analysis of Provable oracle

We can now analyze the Provable oracle through the lens of our research questions. This oracle is a pull-inbound oracle, as it fetches off-chain data and is activated from the blockchain. In particular, when it comes to the assets to implement for a blockchain oracle that uses Provable, we can start by mentioning that we do not need to develop an oracle service by ourselves. Furthermore, we do not need to design nor implement any kind of security measures from the web server side, as they will be dealt with by Provable and by the chosen authenticity-proof system. The assets that we need to implement for the Provable oracle are the business and operational logic of the smart contract, and if we are also in control of the data source, then we need to implement the business and operational logic of the data source as well. In our case, the assets from the data source are the server itself, the information that is stored in the different posts, and the REST API itself. The assets for the smart contract, in our case, reflect the data source, as we need to implement the data of the object we get. We need to implement a way to store the authenticity proof, as we will ask for it from Provable itself. We do not need to manage the Ether in the smart contract, as it is handled by Ethereum.

There are no additional threats to be taken into consideration in this case compared to what we already discussed in the **2 Systematic Literature Review** chapter. We can assume the on-chain data to be secure and that the transactions between our smart contract and Provable's smart contract are secure, as they are backed by the Ethereum blockchain. Without an authenticity proof, we would have to trust Provable to always provide the correct information, which leads us to the oracle problem. There is another threat, specific to Provable, which is denial of service, as Provable relies on centralized private servers, that can be deprecated, attacked, or fail at any point.

The countermeasures that are to be taken in this case are related to making sure that Provable cannot provide false data. Thus Provable offers authenticity proofs. The

authenticity proofs can be verified off-chain using any software that is able to perform the required cryptographic functionality. A big limitation of smart contracts is that they cannot perform cryptographic computations as they are computationally heavy, thus one would need to perform the verification off-chain. If it needs to be integrated with the smart contract, one would therefore need their own server, or some other oracle service, that is able to perform the computation and verify the validity of the authenticity proofs. This adds new layers of complexity and possible failure points, as the oracle service that is used to verify the authenticity proof is also a victim of the oracle problem. For this reason, the best way to use Provable would be to either have your own oracle that verifies the authenticity proofs or perform them off-chain. The latter is suited for scenarios that allow for high latency but is not acceptable for low-latency scenarios.

4.3 Chainlink

Chainlinik [Chaa] is a decentralized oracle network that allows one to get data from the off-chain world to the on-chain world. Chainlink is made up of three main components: the nodes and node operators, the smart contracts and the developers, and the LINK token. The nodes [Chad] are run by node operators. Each node can be considered a node in a blockchain ecosystem, in the sense that it can take part in the Chainlink ecosystem by performing jobs and satisfying user requests to get data from the off-chain world into the on-chain world. Each node is run independently and can run one or multiple jobs. Each job offers a publicly available specification explaining its capabilities and functionalities. Furthermore, each node has a reputation system and a cost per request, which can be verified so that users can always choose the best combination of price and reputation needed for their situation. A job can perform a limited amount of actions, and therefore they are not very flexible.

The inflexibility of the jobs means that developers may not always find a job that performs what they need. Thus, they either have to make their own node and become node operators as well, and define their own job, or design their smart contracts in a way that manages to use other existing jobs giving the intended result. The latter costs more on-chain resources, as one needs to pay for each request done to other nodes, but it may be simpler and more cost-efficient than designing, deploying, and running a job and a Chainlink node. If one were to run a Chainlink node, some costs can be recouped by accepting requests from other users. The smart contracts are usually written in solidity and for the Ethereum Virtual Machine (EVM), but Chainlink offers compatibility for non-EVM blockchains, such as Solana [Chaa]. We only focus on Ethereum.

The last component of the Chainlink is the LINK token. The LINK token is an ERC677 token [ERCc] which is an extension of the ERC20 token [ERCa]. The difference between them as reported by the ERC677 standard is the following: "This adds a new function to ERC20 token contracts, `transferAndCall` which can be called to transfer tokens to a contract and then call the contract with the additional data provided. Once the token is transferred, the token contract calls the receiving contract's function `onTokenTransfer(address,uint256,bytes)` and triggers an event `Transfer(address,address,uint`

, bytes)" [ERCc]. This was needed in order to simplify the operation of the contracts. From a more practical perspective, a developer working on a smart contract or a node operator on the Chainlink ecosystem needs to know that they will transact in LINK in order to request data or fulfill said requests. Node operators can set their own price, this allows more flexibility and a market that is based on both the reputation and the cost of each node. It is important to note, though, that each time a smart contract is called, the user needs to first pay some Ether as gas to call the contract. Later, the contract needs to have enough LINK to pay for its requests and finally, it will have gotten its off-chain data. In order to fund the contract, a user needs to send LINK to the contract, and each transaction requires some Ether to be paid as gas. Thus at the bare minimum, a user needs at least two assets, Ether and LINK, each for their respective network.

Chainlink is a decentralized oracle network, and in our scenario, we have a centralized data source and we wish to have a centralized oracle as well. Furthermore, Chainlink does not provide any sort of proof of authenticity in a way similar to Provable. For this reason, any data needs to be independently checked off-chain, and one cannot blindly trust a single oracle node. As mentioned above, there is a reputation system, which means that some nodes behave correctly and others misbehave. We propose a novel approach to leverage the decentralized properties of Chainlink and use it as if it were a centralized oracle, without the need of running your own node. There are two reasons why we do not want to run our own node. The first one is that it would defeat the purpose of treating Chainlink as a decentralized oracle network, and thus we would be simply implementing a Chainlink node that acts as a server and a smart contract that interacts with it. The second one is that this approach is more involved in terms of skill and time, and it may not be the solution that most companies in the industry would like to leverage, as each company would have to have a team working on the smart contracts and a team working on their Chainlink nodes, thus increasing overhead and costs in a way that often may not be acceptable.

We designed our web app in a way that is suitable for this purpose. In particular, as seen in section **3.2 Data source and destination**, when you fetch from the API the latest object, you would get three strings, each one referring to either the title, the description, or the signature of the object. The signature can be verified off-chain using the public key which is accessible on the home page of the web app. This means that the signature is acting as an authenticity proof. Thus we can query any node that is capable of processing a job that provides the ability to fetch a JSON object from a publicly available API and process it to obtain strings from it. To the knowledge of the writer, there is no job that allows one to perform all of the three operations. Therefore, the more straightforward option is to make our own node, make it run our own custom job, and then make the smart contract that interacts with it. But this goes in contrast with our goal from earlier. After performing a thorough research, the solution that seems the most reasonable is to use the job with the following ID: 7d80a6386ef543a3abb52817f6707e3b [Chac]. This job allows one to perform a GET request from HTTPS-enabled servers and to obtain JSON objects. The obtained objects can be parsed and one can obtain a string from one of the JSON object properties. The idea is to call this job multiple times, and each time to change the property that we are

parsing from the JSON object. This will provide us with the three parameters that we need, including the signature that can be checked off-chain. Since we are working on the Sepolia testnet, we only have access to one and only one testnet oracle node that is able to perform this job. In the mainnet, there are multiple nodes that are able to run this job and satisfy our request. Below is the code of our smart contract and follows an explanation.

```
% specifying compiler version: at least 0.8.7 but lower than 0.9
pragma solidity ^0.8.7;

 % importing smart contracts from Chainlink for inheritance
import "@chainlink/contracts/src/v0.8/ChainlinkClient.sol";
import "@chainlink/contracts/src/v0.8/ConfirmedOwner.sol";

contract GetLatestPost is ChainlinkClient, ConfirmedOwner {
    using Chainlink for Chainlink.Request;

    % defining the strings that keep the queried information from
        the web app
    string public title;
    string public description;
    string public signature;

    % the counter is used for the operation of the smart contract
        as we need to perform 3 queries in succession
    uint256 public counter;

    % the jobId is used to select the right job from the oracle
        node in the Chainlin network
    bytes32 private jobId;

    % this variable is used to store the fee the smart contract is
        willing to pay for the services of an oracle node
    uint256 private fee;

    % this event is used to log the result of each query
    event RequestLatestPost(bytes32 indexed requestId, string
        currString);

    % this function is the smart contract constructor
    constructor() ConfirmedOwner(msg.sender) {

        % this is used to set the Chainlink token as the currency
            to pay for in the Chainlink network, the address is
            specific for the Sepolia network
        setChainlinkToken(0
            x779877A7B0D9E8603169DdbD7836e478b4624789);

        % this is used to specify the address of one specific
            oracle node that the smart contract interacts with
        setChainlinkOracle(0
            x6090149792dAAeE9D1D568c9f9a6F6B46AA29eFD);
```

```
    % this sets the jobID to a specific value
    jobId = "7d80a6386ef543a3abb52817f6707e3b";

    % this sets the fee that we wish to pay to 0.1 LINK, which
        is computed by 1 * LINK_DIVISIBILITY (which is 1e18) and
         then divided by 10. The fee is expressed as a number of
         Juel, which is the 1e-18 of a LINK token
    fee = (1 * LINK_DIVISIBILITY) / 10;
    counter = 0;
}

% this function requests the latest post from the web app
function requestLatestPost() public returns (bytes32 requestId)
    {
    % This is used to build up the chainlink request by
        specifying the jobID, the address of this smart contract
         and the callback function
    Chainlink.Request memory req = buildChainlinkRequest(
        jobId,
        address(this),
        this.fulfill.selector % this is the callback function
    );

    % here we add to the request the URL of the HTTP GET
        request we wish to perform
    req.add(
            "get",
            "https://two-oracle-thesis-web-app.onrender.com/
               posts/latest"
    );

    % this checks the value of the counter; depending on it it
        will then parse the JSON object obtained to get either
        the title, the description or the signature field of the
         object
    if (counter == 0) {
        req.add("path", "title");
    } else if (counter == 1) {
        req.add("path", "description");
    } else if (counter == 2) {
        req.add("path", "signature");
    }

    % here the Chainlink request is being sent
    return sendChainlinkRequest(req, fee);
}

% this function is run once the response from the oracle node
    is received
% _currString is a string that is received from as part of the
    resonse, and depending on what was asked it referes to
    either the title, the description or the signature
function fulfill(
    bytes32 _requestId,
```

```
        string memory _currString
    ) public recordChainlinkFulfillment(_requestId) {
        % an event is emitted to log the response that was gotten
        emit RequestLatestPost(_requestId, _currString);
        % depending on the value of the counter we update the
            respective string
        if (counter == 0) {
            title = _currString;
            counter = 1; % the counter is set to 1
            requestLatestPost(); % we perform a new request with
                counter==1, thus asking for the description
        } else if (counter == 1) {
            description = _currString;
            counter = 2; % the counter is set to 2
            requestLatestPost(); % we perform a new request with
                counter==1, thus asking for the signature
        } else if (counter == 2) {
            signature = _currString;
            counter = 0; % the counter is set to 3
            % we do not call requestLatestPost() again to break the
                 recursive execution of the smart contract as we
                have just updated all of the strings
        }
    }

    % this function is used to withdraw the LINK tokens that are in
        the smart contract should the owner wish so
    function withdrawLink() public onlyOwner {
        LinkTokenInterface link = LinkTokenInterface(
            chainlinkTokenAddress());
        require(
            link.transfer(msg.sender, link.balanceOf(address(this))
                ),
            "Unable to transfer"
        );
    }
}
```

We first have to import two other contracts that our contract inherits. After that, we specify the contract name GetLatestPost and the inheritance, as we are inheriting ChainlinkClient, ConfirmedOwner. Once inside the definition of the contract, we first make use of the directive using Chainlink for Chainlink.Request to attach to Chainlink the Request member for ease of use.

Finally, we can define our variables. Our four public variables are three strings, each referring to one of the JSON object properties we are interested in and an unsigned integer called counter. The counter is used to allow the recursive execution of our contract three times. This is needed because we have three JSON properties. Thus, when the counter is zero, we will get the title; when it is one, we will get the description; when it is two we will get the signature. After that, we reset it to zero so that the next time we call the contract, the whole process can restart again. The two private variables are the jobId and the fee. The first one is used to identify the job that

the oracle node needs to run in order to satisfy our request and the second one is the amount of LINK we are willing to pay in order to get our request satisfied. We also define an event, called RequestLatestPost which is used to log to console each response we get from the Chainlink oracle node, by associating the requestId and the current string received (currString). After the variables, there are four functions. We will see them in order.

The first function is the constructor method which is called when the contract is deployed. The first two lines set that we want to use the LINK token and the oracle node that we want to interact with. They are hardcoded for our proof of concept as there only is one testnet Chainlink node that can satisfy our request. It is used also to set both the job ID and the fee we are willing to pay. The fee is 0.1 LINK tokens. Finally, we set the counter to 0.

The second function is the requestLatestPost and it is the one that a user calls to get information about the latest post. We first create a request object using the information we have, that is, the job ID, the address of the current smart contract which is needed to get the reply, and the action we want to be performed, in this case this . fulfill . selector which allows parsing the received JSON object and obtain one string from one of the properties. We proceed by adding an action to the request, in this case, we are performing a GET request to our API available at the address https://twoâĹŠoracleâĹŠthesisâĹŠwebâĹŠapp.onrender.com/posts/latest. We have a couple of conditionals, which all depend on the value of the counter. In all cases, we specify the property we want to get from the JSON object. In particular, if the counter is zero, we get the title; if it is one we get the description; if it is two we get the signature. Finally, the function ends by calling a method that sends the request to the Chainlink oracle node along with the related fee.

The third function is the fulfill function. It allows us to receive the response from the Chainlink node. It has two input values, the first one is the ID of the request we performed and the second one is the string that we just obtained. The function first emits an event to log the values we just received. Then, there are conditionals that allow us to update the value of the string variables to the latest value we obtained. The value we update depends on the current value of the counter. After we update the value, the counter is assigned a new value in order to update the next string. If the counter value is 0 or 1, we end the function by calling the requestLatestPost function. Thus our contract calls itself recursively to perform multiple requests and update all of the values it needs.

The fourth and last function is a standard one, which is used to withdraw the LINK tokens that are still on the contract in case one wants to empty the contract's balance.

Analysis of Chainlink oracle

We can now analyze the Chainlink oracle through the lens of our research questions. This oracle is a pull-inbound oracle as in the previous case, as it fetches off-chain data and is activated from the blockchain. In the case of a Chainlink oracle, the assets to be implemented vary depending on whether one wants to leverage the decentralized network as intended; make their own oracle node; or finally on creating a smart

contract that interacts with the Chainlink network as if it were a centralized oracle, but with the right measures. If we are using a decentralized oracle, then we are in a similar situation as in the Provable oracle. We just have to implement some functionality that allows the smart contract to choose the right oracle network and job ID to be run. In case we make our own oracle node, we need to implement the functionality to fetch the data from the off-chain world as well as the node and the server themselves. We also need to choose whether we want to open the oracle node to everyone or not. In the last case, we are again in a situation similar to Provable, but in this case, we also need to implement a logic that allows us to bypass the problems inherent to the decentralized nature of the oracle. This approach is best suited when we are also in control of the data source, as we would need to have some system to verify that the data did indeed come from the real source and it was not manipulated by a rogue oracle node. The network assets such as the Ether and the LINK tokens are managed by the Ethereum blockchain, thus they do not need to be implemented by us.

The threats to be taken into consideration are again similar to what was found in the SLR. There are two additional threats. If we are using the decentralized network as intended, then the additional threat is that the oracle node that we are interacting with can misbehave or reject our requests. This can be for multiple reasons: some are legitimate, like in the case we do not have enough funds to pay for the transaction or in the case that the oracle network was designed to only interact with a limited number of smart contracts; others are not legitimate like in the case an oracle node tries to provide false data or is unable to satisfy all of the requests it receives. Just because an oracle node is misbehaving, it does not mean that the entirety of the network will be misbehaving. The other threat is related to the case we are running our own Chainlink oracle. This opens up threats related to denial of service and to a centralized point of failure.

The countermeasures to be implemented again depend on the way that we decide to leverage Chainlink's technologies. If we run our own Chainlink oracle node, then we need to decide whether we want to accept requests from all contracts in the network or not. In the first case, the oracle node would end up earning LINK tokens, and thus recouping part of its costs, if not outright earning more. But by using this approach, we may run into the problem of the oracle receiving more requests than it can handle. If instead, we do not open the node, we reduce the risk of denial of service to near zero, as the oracle node would only accept requests from our smart contracts. In case we want to avoid creating and deploying our own oracle node and related jobs, we need to use already available oracle nodes. In this situation, if we do not own the data source, the only way to avoid accepting faulty data is to define our risk level acceptance and use Chainlink as intended. This means that our smart contract needs to query multiple different oracle nodes for the same information and accept it once it has received enough confirmation. The smart contract needs also to have some functionality to modify the set of oracle nodes it interacts with so that they can be updated according to our own risk level, the cost of querying data from the nodes, and the reputation of the nodes. Sybil attacks are not a concern, as it is the smart contract developer who chooses which oracle nodes to use, so a malicious attacker would not be able to simply create multiple oracle nodes and influence the decision process, as

they would have no reputation at the beginning and they would need to be actively added to the smart contract. If instead we own the data source, then the best way to secure the data would be to add some authenticity proof directly into the information being sent to the oracle. The way we propose is through the use of digital signatures.

4.4 Outbound Oracle

The outbound oracle is implemented in a different way. In particular, in order to read the blockchain one needs to have an endpoint that acts as a bridge between the blockchain itself and all of the other pieces of software that wish to interact with it. To be more precise, an endpoint in Ethereum is an Ethereum full node that can be accessed over the internet by other software.

A company can easily create and manage its own endpoint, as the work required is not substantially different from implementing, running, and maintaining a node. For individuals or even some small-scale companies, this option may not be the best one, as there are costs associated with running an Ethereum node, especially when it comes to storage, as the Ethereum blockchain has a significant size and it is increasing: at the time of writing, it already is over 13 TB [Etha].

In our work, we make use of Infura [Inf], which is a service that provides an Ethereum endpoint that is accessible for free for our testing purposes. Infura was chosen as it provides a free option and it is compatible with both the Sepolia and Goerli testnets. They provide an API key which is to be used to query all of the necessary data. The requests and responses are sent over HTTPS. A sample of the app is available at this URL: https://outbound-oracle.onrender.com/.

The app interacts with the Chainlink oracle in order to obtain information about the three strings stored in the smart contract. A very similar app can be made with some trivial modifications that interact with the Provable oracle instead. The app queries the information about the strings stored in the oracle and then checks that the obtained digital signature is valid, and shows the result to the user. This is done for two reasons. The first one is that in this way the app is both "oracle agnostic" and "endpoint agnostic"; which means that the security is provided directly by the Data source (as described previously in this **Chapter**). Should there be a need to change either the endpoint or the smart contract, one will need to make small and trivial modifications to the code. The second reason is that this removes completely the necessity for trust in the oracle and in the endpoint, and adds an additional layer of difficulty, as an attacker that wishes to attack our system needs to successfully create a fake signature about the Post that was received and successfully bypass HTTPS, as the communication between the app and the endpoint is over HTTPS.

In a similar vein to **Section 3.2**, we created a small web app with the following technologies:

- Infura's endpoint, in particular, the one for the Sepolia network available at https://sepolia.infura.io/v3/API-KEY
- Node.js / npm to create and manage the app (version 9.3.1)

- Web3.js which is used to interact with Infura's endpoint
- crypto.js library which is installed along with Node.js. It is used for verifying the digital signatures given by the oracle
- Express.js which is used to manage the ports for the server to listen
- Dotenv.js to deal with the environment variables and hide run-time secrets; in this case, it is the API key
- Nodemon.js to run the app in a test environment safely
- Render.com to deploy the app and manage CI/CD and to make it accessible from the web at a specific HTTPS-enabled URL
- a GitHub repository for CI/CD available at this URL: https://github.com/alexcarchiar/thesisOutboundOracleWebApp

All of the chosen technologies are either open source, or proprietary but still offered a free option; the main reason behind the choice of these technologies is that the reproducibility of this experiment should be easy and accessible to the broadest possible crowd. There only is one request to the app, which is an HTTPS GET request to the homepage, that is, the path of the request is empty. This is because, in our scenario, this app is just a consumer of blockchain data. The development of this app consists of only one file `app.js`, as we have no need for routes or middleware. The code is reported in the appendix in Listing 7. Below is an explanation of the code.

Let us now explain the code. The first few lines import the required packages, in particular Dotenv.js, Express.js, Web3.js, Crypto.js, and Buffer.js. One line also creates a web3 object that acts as an interface between this code and Infura's Ethereum endpoint for the Sepolia testnet.

After that, we create a variable to store the Ethereum address of the smart contract we want to interact with. The address is the real address of our inbound Chainlink oracle.

The following object called `contractABI` contains the ABI code of the smart contract. This code is generated by the Remix IDE. It is encoded as a JSON object and it contains the most lines of this file, as it specifies each variable and each method of the contract. The code is reported in the appendix, as it is long, and reporting it does not add any insight into our implementation.

Once we are done with the ABI code, we create a contract interface `const contract new web3.eth.Contract(contractABI, contractAddress)`. It takes both the contract address and its ABI code to create the object and we use a method of the Web3.js library.

We then use the Express.js library to create an object `app` which we will use to deal with the HTTP GET requests.

We define two functions. The first one is an asynchronous function called `getInfo` whose purpose is to interact with the smart contract interface and get the required information. As one can see, we query the contract three times in order to get the

`title`, the `description`, and the `signature` of each post from the contract. The function then combines the three obtained strings in an array and then returns it.

We define three variables, one is the public key, which comes from the original data source of the data before it was sent to the smart contract that is queried by this web app. The other two set the parameters needed for verifying the digital signature, that is, the algorithm and the string format.

We now define the second function. The second function is `verifySignature` and its goal is to get the information from a post, that is, the title and the description, and verify the respective signature given the public key and the other parameters. The function returns a boolean, with a value of true if the signature is valid or false otherwise.

We finally define the HTTP GET request to the homepage, in particular the function that deals with it. We first call the `getInfo` function to get the array of the strings and we later parse them to obtain three different strings, one for each element of the array. We now call the `verifySignature` function with the right parameters and assign the returned value to a boolean `isValid`. We can start to build the response to the HTTP GET request. The response will be a string containing the information that was taken from the smart contract. The last piece of information that is added is whether the signature was valid or not, and we use a conditional for that. We then send the response.

The last few lines of code are needed to open up a listening port so that the server can operate with the rest of the internet.

Analysis of the outbound oracle

We can now analyze the outbound oracle through the lens of our research questions. This oracle is a pull-outbound oracle, as it fetches on-chain data and is activated from the off-chain world. In particular, when it comes to the assets to implement for an outbound blockchain oracle, we only really have the endpoint as a critical element, as it is the interface between our web app and the blockchain. We are making use of an already existing endpoint, so in this case, we need to make our API key secure, as otherwise one would be able to perform requests to Infura's API using our own key. If this happened, this would be a security problem, especially when it comes to billing, as the free option only supports a limited amount of requests, and any company would be paying Infura in function of the number of requests they need. Other assets to be implemented are the off-chain business logic and the functionality required by the app.

There are three threats to be taken into consideration. The first one is that of a malicious endpoint that provides false data. The second one is that of an attacker that tampers with the communication between the web app and the Ethereum endpoint. The third and last one is that of an attacker trying to obtain the API key. There still exists another threat, which is a denial of service (DoS). There can be two services that can be a victim of a DoS attack: the web app and the endpoint, as they are both centralized.

The countermeasures to be taken in these situations are different and each addresses one different threat. In particular, in order to protect against DoS attacks, we cannot really do much to protect against a successful DoS attack on Infura. There are two ways

to mitigate this threat: the first one is to run your own Ethereum node/endpoint, which incurs higher costs and higher complexity needed to implement, run and maintain the node; the second one is to use multiple endpoints provided by different companies so that if one is not working, we have another one to fall back on. If the DoS attack is on our web app, then the countermeasures are outside the scope of this work, as they are related to security practices for developing web applications. In order to protect against an attacker tampering with the communication between the web app and the endpoint, we can assume the connection to be secure, as we are using the HTTPS protocol, thus there is no need to take additional measures. The only way to assure that an endpoint is providing true data is to run the Ethereum node directly instead of relying on some other company. By using the digital signatures strategy defined earlier in this chapter, we are taking a countermeasure to mitigate this risk as the signature is generated off-chain and is checked again on the destination web app, thus if the endpoint were to provide false data, it would be detected. The endpoint cannot generate new posts and new signatures, but it can provide an old one, that is, it can decide not to provide up-to-date information, and there really is not any way to mitigate this risk; but it is important nonetheless to remark that the consequences of this risk are not as bad as in the case of an endpoint that is able to provide outright false data. The countermeasure to be taken to protect the API key are related to good software development security practices and are outside the scope of this work, but in general, we mention that we encourage the use of environment variables and self-contained environments to run the server on.

4.5 Answer to Research Questions

The results of the work of this chapter allow us to answer the research questions that were presented in the first section of this chapter. We will now be addressing these questions one by one.

[RQ2.1] What are the assets to implement for a blockchain oracle?

The assets to be implemented when designing a blockchain oracle are the logic and information related to the normal operation of smart contracts. In the vast majority of cases, blockchain oracles are just an interface, and for this reason, the main asset is information. The specifics may vary in function of the scenario we are facing. According to the strategy chosen by the institution or company operating the oracle, one may need to design the data source in a way that provides the security property or may need to design, implement and maintain oracle nodes and servers to maintain the normal operation of the smart contracts. This consideration needs to be evaluated from both a technical perspective and an economical one, as developing, maintaining, and running oracle nodes has related costs and is a challenge in itself. A similar consideration can be made about running and maintaining an Ethereum node/endpoint, which is needed in case one wishes to implement an outbound blockchain oracle. In the case it is preferred to take advantage of a service that offers an endpoint, then the API key to interact with it becomes an asset itself, as it is the way that allows to interact with the endpoint and establish the costs for using it.

[RQ2.2] What are the threats to be taken into consideration?

It is safe to assume that the on-chain data is secure thanks to the security properties that blockchains provide. At the same time, the security of the data of the off-chain data source/destination is blockchain-agnostic, as it does not rely on blockchains themselves, and the security of said systems is out of the scope of this work. There are two main threats to be taken into consideration. The first one considers that if you are using an oracle service or a third-party endpoint, such as Provable or Infura, you need to trust and ensure that said service is providing the right information, and that it is not providing false information. If instead, you are using a custom-made oracle, then you need to ensure that nobody can send transactions to the smart contract of the oracle that is being used, otherwise, anyone can send false information. If you are using your own endpoint, then you need to maintain an Ethereum node and the threat against it is denial of service, as a single Ethereum node ultimately acts as a server that can be attacked and overrun by a very large amount of requests.
Especially in the case of someone using a decentralized oracle as if it were a centralized one, additional threats to be considered are related to the reputation of the decentralized oracle nodes that one chooses to use and their respective costs. As costs fluctuate over time, one may need to consider the case of your smart contract not working due to an increase in costs. Another threat is that of the oracle node denying service, either due to it being deprecated or somehow attacked or being overwhelmed with requests. And the last threat to be considered is the case of a job that is not performed by many different oracle nodes and is thus subject to the threat of not having any kind of oracle node that is able to perform it thus rendering the smart contract unusable.

[RQ2.3] What are the countermeasures to be implemented?

One important detail to be taken into consideration is whether the oracle and the off-chain data source/destination are designed separately or jointly. If they are designed jointly, one can take advantage of the blockchain properties to guarantee the integrity and authenticity of data. The reason is that the off-chain component can have a blockchain address used to interact with the oracle on the blockchain, which will also have a specific address. This means that one can program the oracle smart contract to only interact with a specific address, thus guaranteeing authenticity through the use of one specific address and integrity via the blockchain transactions.
If the blockchain and the off-chain component are designed separately, which is the case we have analyzed in our work, the situation is more complicated, as one needs to have a component scraping the data and feeding it to the blockchain oracle in a secure manner, and one cannot simply rely on the blockchain properties. A solution to this is to implement some authenticity proofs in a way that is cryptographically secure and sound. There is though a big limitation in the smart contracts, at least Ethereum based, as the computation and verification of cryptographically secure authenticity proofs are too computationally heavy to be performed on-chain. Thus one needs to perform said verification off-chain, which can be a big limitation for smart contracts, especially real-time ones. A possible solution could be to use a custom trusted smart contract, but the implementation becomes more complicated as one cannot take advantage of ready-made technologies such as Provable.
In the case of Chainlink, a possible solution could be to run your own oracle node. This results in higher costs, but it would be able to have total control over your node,

thus reducing or removing completely some risks, especially the ones related to node deprecation and cost increases for requests. You would also be able to mitigate denial of service threats by making the oracle node only able to reply to requests coming from your own smart contracts. Another possible solution is to design the smart contract so that it is possible to change the oracle node and job that it interacts with, possibly adding a system to choose between multiple of them. It is also paramount to choose a job that is performed by multiple oracle nodes, as it would decrease the possibility of it being deprecated from multiple nodes and ultimately not having any node able to run it.
In the case of our outbound oracle, instead, our solution relies on the fact that data integrity is provided by the blockchain itself and we trust our chosen endpoint to provide true information. If we own the endpoint, then we can reasonably trust it to provide truthful information, otherwise, the countermeasure to be taken in most situations is to use multiple endpoints and check the information that they all provide so that the probability of all endpoints providing false information is decreased and eventually becomes negligible. Another countermeasure, which is the one we proposed in our work, relies on the design of both the original data source and the inbound oracle that fetches its data. The original data source provides a digital signature to check the validity of the data off-chain so that any data consumer can check the validity without worrying about who or what relayed the data. This removes the need for trust in oracles and can be closer to the original philosophy of blockchain, but it cannot be applied to all cases as it requires the data source to provide these digital signatures.

4.6 Limitations

The main limitation of this chapter is the choice of the scenario. The chosen scenario is that of a centralized source that feeds the oracle publicly available information. This immediately removes a constraint of confidentiality from the environment and the necessity to manage certificates or a public key infrastructure. Furthermore, the simplicity of the Proof of Concept may not be enough to picture a system where one might have multiple data sources, or multiple different types of data coming from the same source.

Furthermore, the smart contract were designed to only have a couple of functions and state variables. The functions update the data and the state variables keep track of the latest values that were given to the oracle and the related authenticity proof. The limited set of allowed interactions with the oracle does not correctly picture a condition where one might have a more sophisticated design or requirements. We limited the use of Chainlink to only a single node, and thus we are not taking advantage of the decentralized properties of the oracle. We are using Chainlink as if it was a centralized oracle, which it was not really intended for. This means that many of the aspects involved in using Chainlink - the choice of reputable nodes, the choice of the right job, or the implementation of one - are not taken into consideration, unlike a real-life scenario.

Another threat to validity is related to the chosen technologies, as this research only focused on a few very specific technologies, and thus all of the technologies that

were excluded from the analysis are not considered. It is also important to remember that the work focused on using Ethereum technologies and established solutions or techniques that are common for other public blockchains, thus the discussion may not apply trivially to permissioned or private blockchains, given the differences that exist between the different kinds of blockchains.

Our outbound oracle only consisted in a web application that possesses a web3 interface, thus it does not take into consideration many other aspects related to web3 applications, in particular decentralization and data access. Our outbound oracle is also limited by our design, in the sense that it is intended to work in tandem with our inbound oracles/smart contracts. This means that our approach is really applicable as long as one has access to a description of the smart contract it needs to interact with.

5 Evaluation of the Models

In this section of our work, we will be addressing the third and final research question **[RQ3] How can one compare the two blockchain oracles?**. In this case, the blockchain oracles are the ones discussed in the previous **Implementation** section. In order to reply to the research question, we need to divide it into three subquestions:

- **[RQ3.1]** What are the evaluation criteria to be considered?
- **[RQ3.2]** How well does each oracle satisfy the evaluation criteria?
- **[RQ3.3]** How do the oracles compare to each other?

This chapter is the evaluation phase of our design science approach. This chapter will be organized into multiple subsections, each one addressing one question. The first subsection will motivate why blockchain oracles are needed for our scenario. The second subsection will present the chosen evaluation criteria; the third one will evaluate the Provable oracle while the fourth one will evaluate the TownCrier oracle. Then, the following subsection will compare the evaluations of the two oracles. Finally, the last subsection will reply to the research questions.

5.1 Motivating Blockchain Oracles

In our simple scenario, we have a public data source that is accessible to anyone on the internet. It is supposed to simulate a common situation, such as the one of a news media company website, a weather website, or even many others, such as a trading platform providing prices of publicly traded financial products and assets. There are many more similar situations that fit into this category that we are not listing here. In many cases, one might have smart contracts that need data input from one of these sources to operate. Some decentralized oracles such as Augur [Aug] or Tellor [Tel] provide examples in their whitepapers. These two oracles are mostly focused on the betting market, even for real-life events such as elections, sports matches, and other more exotic bets such as betting on a movie's box office result. Bets in a blockchain environment can be regulated as a smart contract: two or more people deposit money to the contract address which listens to the oracle. Once the oracle provides the result of the bet, the smart contract pays out the winner.

Aside from betting, another common issue in modern times is that of online misinformation and disinformation, and while this topic is vast and of difficult resolution, one common practice is that of *Stealth editing* [Lee18], performed even by reputable and trusted sources such as *The New York Times* or *The Washington Post*. Stealth Editing is the practice of editing articles, either their content, the headline, or both, without mentioning how the article was modified, why it was modified, and without keeping track of older versions. One possible way to combat this practice is to archive articles on the blockchain since the blockchain is immutable and it would be possible to store all versions of the same article. This would also make it impossible for a media company to repudiate an article they published without acknowledging it

and issuing corrections. In order to store the articles on the blockchain, it is imperative to use blockchain oracles, as they are the interface between the off-chain and on-chain worlds. If articles or other publicly available data is stored on the blockchain, then one needs an outbound oracle in order to retrieve all of the on-chain information. Our discussion about articles can be applied to other pieces of information, such as weather and scientific papers.

The same scenario can be extended to non-publicly available data if one were to encrypt the data before submitting it to the oracle. But in this case, the security considerations do not differ much from our case, as the main difference is that the data is encrypted, and thus it would need to be decrypted before it can be accessed. In this case, in our SLR we have seen how the common practice is to use asymmetric cryptography, and thus the additional security considerations are related to the safe storage of the key pair as well as the choice of a cryptographically secure algorithm.

Thus, it is now evident that by using blockchain oracles one can improve the public trust in publicly available information by allowing for safe storage and versioning, as well as allowing the public to use the data safely on smart contract applications. Otherwise, websites can easily repudiate the data they submitted and the smart contracts would be very limited in scope.

5.2 Evaluation Criteria

The evaluation criteria rely on the scenario being considered. For example, in the case of someone implementing an access control mechanism, the goals of such a system are confidentiality and integrity of data as well as the integrity of the access control lists, as one only wants the admin to modify them. In the case of insurance in the healthcare domain, there are different concerns, such as the integrity and confidentiality of patient data, as well as the funds being transferred between insurance companies and patients.

We remind that, using the results of our SLR (**Chapter 2**), the main threats that we need to take into account can be grouped into three big categories: the first one is the impersonation of the data source; the second one is data poisoning of the data being sent to the blockchain or outside the blockchain; the third one is Denial of Service. In case we have multiple data sources, such as in the case of decentralized networks like Chainlink [Chaa] or ASKE [WLS+19], we have an additional threat which consists of the trusted data source misbehaving. These threats need to always be considered when designing, implementing, and evaluating blockchain oracles, and thus our evaluation criteria will be based on them. But in order to have meaningful criteria, we need to first understand the specificities of our use case.

For this work, we will be considering only our relatively simple case. We have a centralized data source (which is a web app) that sends data to the blockchain through an oracle. We assume that the source of data is secure. There are three goals that need to be satisfied: we want data integrity, as we do not want an attacker to be able to modify the data being sent into the blockchain; we want to authenticate the source, as we do not want an attacker to impersonate the source; finally, we want to measure the performances of the oracle, as we want to be able to send data multiple times. With performance, in this case, we measure by the gas cost and the number of requests that

Table 5: Evaluation criteria by group

Identifier	Criterion	Group
EV1	How is data integrity provided and satisfied?	Security
EV2	How can an attacker impersonate the source of the data?	Security
EV3	How can an attacker impersonate the oracle service?	Security
EV4	How can a malicious web application repudiate data it has submitted to the oracle?	Security
EV5	How many requests per unit of time can be satisfied?	Performance
EV6	How much ether is spent to use the oracles?	Performance
EV7	Are there any further limitations?	General

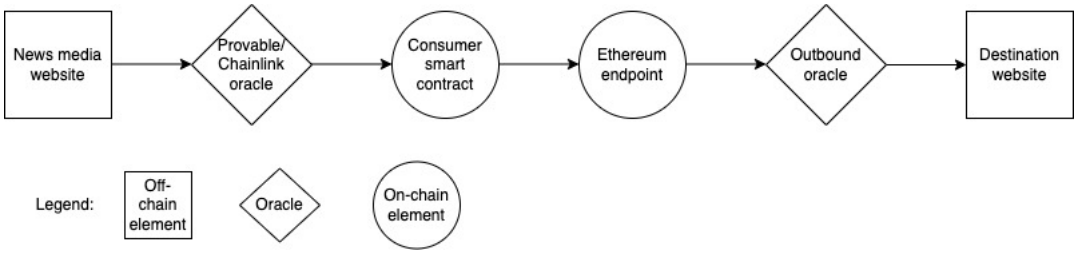

Figure 4: Scenario graph presenting the use case implemented by our oracles

can be satisfied in a determined unit of time. We do not need data confidentiality, as in our case the data source has public data. The evaluation criteria are shown in Table 5.

We will group the evaluation criteria into three groups in order to better refer to them. The groups are: **Security ([EV1], [EV2], [EV3], [EV4])**, **Performance ([EV5], [EV6])** and **General ([EV7])**. They are divided into these categories as each category addresses one major aspect of an oracle design and implementation. The first category is the one that is the most important to our work, as it addresses the specific security properties that we are focusing on. The second category addresses the performance aspect, which cannot be excluded from a complete discussion of an oracle. The cost per request is also considered as part of the performance aspect. Finally, the last category addresses miscellaneous aspects that cannot be easily categorized together.

5.3 Evaluation of Provable Oracle

In this section, each paragraph will address one evaluation criterion.

[EV1] How is data integrity provided and satisfied? The data integrity is provided by using authenticity proofs. In particular, in our example we chose TLSNotary [TLS], the proof is given by creating a transcript of the TLS communication between the source of data and the pair Notary-Provable. Figure 3 shows at a glance how the TLSNotary authenticity proof is generated. The TLSNotary [TLS] proof makes use of the TLS properties in order to generate the proof. The proof is generated during the execution of a standard TLS protocol between a server and a client. The peculiarity is

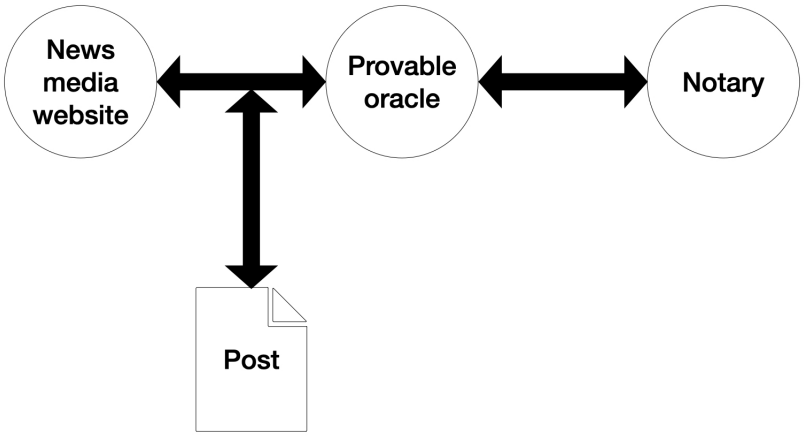

Figure 5: TLS Notary graph [TLS] adjusted for our scenario

that the TLS secret share of the client is divided into two parts, one is owned by the client itself and the other is owned by the Notary. The authenticity proof is a record of the complete TLS protocol, which can only be completed if the client and the Notary cooperate, as neither of them holds the entire private TLS key. The assumption is that the Notary is trustworthy, or at least is considered so by the Verifier, who will verify the obtained authenticity proof. This puts the trust in the Notary. This problem can be mitigated by using multiple Notaries and obtaining multiple proofs, or even running your own Notary.

[EV2] How can an attacker impersonate the source of the data? In order for an attacker to impersonate the source of data, they would need to get the TLS private key of the data source. Thus this aspect of security is strictly dependent on the security of the TLS private key. Different data sources will adopt different security measures, so the security is not easily measurable. The biggest vulnerability here is the human factor, as humans can be socially engineered to give up secrets.

[EV3] How can an attacker impersonate the oracle service? In order for an attacker to impersonate the Provable oracle service, he or she would need to have access to the private key of the addresses that Provable uses to satisfy queries and requests from the different smart contracts. We are in a situation similar to the previous one, where security is not directly measurable, and the main vulnerability is the human factor.

[EV4] How can a malicious web application repudiate data it has submitted to the oracle? A malicious web app in order to repudiate the data has two challenges to overcome. The first one is that a web app does not know whether it is interacting with a single TLS client or with a Notary-Provable pair, as the pair shares a private/public key pair. In the case of public blockchain, it is possible for a malicious web app to query the blockchain and learn whether it was queried by a smart contract or not. This

can only happen *a posteriori*; that is, the web app cannot learn it is being queried by a Notary-Provable pair while it is communicating with it. If the web app were able to discern whether it is interacting with a Notary-Provable proof, it would technically be able to send fake information in order to generate a valid proof. Since we can assume that the web app will find out about a proof only afterward, in order to repudiate the data it would need to generate a new TLSNotary proof by getting access to the private/public key shares of both the Provable client and the TLS Notary. Furthermore, once the proof is generated and stored in the blockchain, it cannot be modified, thus in order to repudiate the data, the web app would need to generate the same exact proof, but with different data and without having access to the secrets managed by the TLS Notary-Provable pair. The difficulty would also increase if multiple authenticity proofs are generated, either by the same contract (as TLSNotary [TLS] recommends) or by multiple contracts querying the same information.

[EV5] How many requests per unit of time can be satisfied? The number of requests per time that can be satisfied is dependent on the amount of Ethereum transactions that can be sent. Provable [Pro] provides two main ways to send Provable queries and they both rely on two strategies: either calling the smart contract multiple times or making recursive requests. In the first case, the upper limit of the request throughput will be dependent on how many Ethereum transactions are being performed by the network, how congested it is, and the gas fee the contract is willing to pay. In the second case, the throughput is dependent, as always, on gas fees and the congestion of the network, but there is a further constraint: since the smart contract is sending a request to Provable and waiting for a reply, we need at least two transactions per each request, one from the smart contract and one from Provable. Thus in an ideal situation, there can only be one query for every two Ethereum blocks mined. Furthermore, after testing, we discovered that Provable only satisfies one query at a time. This means that, if the smart contract makes one query and then generates a new one before receiving Provable's response, only the first query will be satisfied, and the second one will never receive a response.

[EV6] How much ether is spent to use the oracles? The Ethereum cost is made up of two parts, which are both variable. The first one is the ether needed to pay for the transaction fees in the Ethereum network, both for sending the request and receiving the response. The other part is instead dependent on the fee that Provable requires for its services. The first part is dependent on the network congestion and the second one is dependent on Provable itself and how it chooses the pricing for its services [Pro]. In our case, the cost is 0.05$ for querying a URL (0.01$) and a TLSNotary proof (0.04$).

[EV7] Are there any further limitations? An additional limitation of Provable is that the verification of the authenticity proof is too resource expensive, whether using TLSNotary or other proof types. Thus the smart contract cannot verify the proof it receives, and they need to be stored on-chain and externally verified in some other way. Provable provides some tools available at this URL: `https://github.com/provable-things/proof-verification-tool` with guides on how to perform the verification. The challenge now is to verify the proof from the smart contract automatically, as the manual verification is relatively simple and the retrieval of the proof from the blockchain is as well. Furthermore, since the proof

is stored on-chain, one can assume that the integrity of the proof is satisfied as it is provided by the chain.

Another limitation of Provable is related to the query function. As it can only return strings as results. Also, parsing the result strings is usually a task that is too computationally heavy for a smart contract, thus in order to get the information that the contract need there are two possible solutions: either perform one query and store the result as is, including the pieces of data that are not needed and thus incurring in higher storage costs; or perform multiple queries where we make use of the Provable parser, thus reducing the storage costs but increasing the number of transactions needed. In case we need to also store the proofs, a mechanism needs to be put in place to store them too. Since the proof stores the communications between the Provable-Notary pair and the Web source, saving multiple proofs will lead to an increase in storage costs. But the proofs will always be attesting to the authenticity of similar communications since the messages from the web source will always include the same data. Thus there can be differences in the cost-effectiveness of the two strategies in case the excess data to be stored is bigger than storing multiple authenticity proofs, but in our use case, it was more cost-efficient to store the entirety of the data received from the API and just one proof.

Addressing our scenario

The Provable oracle addresses many of the aspects of our scenario. We recommend revising **Section 5.1** to see a description of our scenario. The Provable oracle can be used as a general way to create an archive of articles or other pieces of data. In our simple smart contract, we simply store the content of the entire article in a variable as is, with no processing. Every time a new article is submitted to the oracle, the smart contract replaces the old article with the new one. Since Provable provides query identifiers for each query, by using them it would be possible to develop more sophisticated storage techniques that allow storing all of the articles and even multiple versions of the same article. By pairing the storage of articles with the respective authenticity proofs, no source would be able to repudiate the articles or any other piece of information. This would address the Stealth editing practice and increase public trust.

Given that the Provable oracle does not provide any sort of processing of the data it obtains, the data obtained by the smart contract may not be ready to be used as is immediately. There are two cases: the data obtained from the Data source API does not need any parsing or processing, or it does. In the first case, then the smart contract can utilize the data, but this situation requires the data source API to be designed as such, which will very rarely be the case. In the second, more common, case, the data will need to be processed or parsed. This operation cannot be usually performed on-chain, and thus one would need to be performed so off-chain. In order to perform the off-chain computations, one needs other oracles, both outbound to send the data to be processed, and inbound to send back the processed data. This raises the complexity level but may be needed depending on the scenario. In particular, it would be needed if the data source provides multiple pieces of data and we only need one for the operation

of the smart contract. An example could be a weather station that provides data from multiple sensors such as temperature, humidity, wind direction and speed, and more, but the smart contract only needs to know the temperature. In case we are storing articles, then we may not need to process data as we would be storing the article in its entirety.

The final consideration is related to the validity of the authenticity proof from a legal standpoint. While the cryptographic primitives make the authenticity proofs cryptographically secure, in Provable's case they are generated by a Notary chosen by Provable and not by the data source, thus unless the authenticity proofs are legally recognized, there could be legal implications in using them, and claiming that a website has indeed repudiated some data or an article. The legal framework of oracles and cryptographic proofs in general is a novel field and is outside the scope of this work.

5.4 Evaluation of Chainlink Oracle

In this section, each paragraph will address one evaluation criterion.

[EV1] How is data integrity provided and satisfied? The Chainlink oracle network does not provide data integrity. As Chainlink is a network of oracle nodes, it provides more flexibility in the design of the smart contract that interacts with it. Thus while data integrity can not be provided by a single oracle node, Chainlink offers a reputation system and the possibility of interacting with multiple oracle nodes in order to settle for the right and truthful information. Thus we can consider that the network properties provide data integrity as long as multiple nodes are being used. Another solution is to run our own node that we can trust; in this case, data integrity is provided by the blockchain itself as the data will be sent through blockchain transactions, which by our assumptions are secure and have their integrity satisfied. Since our node is trusted, we do not need to have authenticity proof. In the case that one uses authenticity proofs through digital signatures at the data source layer, he or she would be using the Chainlink network as if it were a centralized oracle, with the data integrity being provided by the data source.

[EV2] How can an attacker impersonate the source of the data? In the case of Chainlink, an attacker can only be present at the oracle node layer. It can either be acting as a Man-In-The-Middle (MITM) between the oracle node and the API and replace the information being sent from the API to the node, or the attacker can be the oracle node itself. In the MITM case, the use of HTTPS and multiple nodes is able to mitigate the occurrence of this attack. In the latter case, the oracle node is the source of the data that will be given to the smart contract, and thus can simply provide false data at this stage. If the oracle node is assumed to provide truthful data than an attacker would need to gain control of it in order to make it misbehave, as the oracle node has an interface that interacts directly with the blockchain and possesses its own blockchain key pair. If the attacker gains control of this key pair, then it can easily impersonate the source of data.

[EV3] How can an attacker impersonate the oracle service? An attacker can impersonate the oracle service by either taking over the oracle node that is being used by the oracle or by getting access to the private key the oracle node uses to sign its

transactions. In the first case, the security depends on the practices and measures taken by the maintainers of the node; in the second case, it depends on how the private key is stored, as Chainlink relies on the Ethereum blockchain which uses 256-bit keys.

[EV4] How can a malicious web application repudiate data it has submitted to the oracle? In case someone uses Chainlink as intended, the web app can repudiate the data easily, as Chainlink does not provide any way to bind the data to its source. The web app just needs to delete the entry in its database. If we are using our own oracle node, then we would need to design some functionality to provide non-repudiation, otherwise, the web app could delete the database entry all the same. In the case of our suggested approach, instead, data non-repudiation is provided through the use of digital signatures. Assuming that a private key stays indeed private, the only way that one can generate a digital signature is if they own it. Thus, even if the web app were to delete some data entries related to some of its posts, the smart contract keeps the signature. As long as the web app does not change the key pair and keeps the same public key, it is possible to verify that a signature was generated using the private key of the web app, even though the web app might delete the related data entry. Using our setup with Render.com and Node.js, there are two possible ways that the key can be changed so that the web app can repudiate the data it has sent. The first one is by updating and redeploying the entire website. In this case, with our measurements, it takes 5 minutes for our web app which is a minimal proof of concept. This means that in this case the repudiation cannot be performed in real time. The second way is for the app to already have multiple key pairs deployed and just changing the key that is used by changing a state variable; in this case, the attack could be performed in real-time. In a real-life scenario, especially for news companies that have high audiences, it is safe to assume that many users would notice a change in the public key, thus the attack would not be successful. Furthermore, in our proof of concept, the public key is just shown as a textual string, whereas in a real-life scenario the web app would show a certificate signed by a CA, and it would be expensive and traceable for the web app to obtain multiple certificates for different public keys.

[EV5] How many requests per unit of time can be satisfied? The number of requests per time that can be satisfied is again dependent on the amount of Ethereum transactions that can be sent, as in order to activate the oracle one needs to perform a transaction. Furthermore, one needs to take into account additional transactions that need to be made to fund the smart contract with enough LINK tokens to provide funding to pay for the queries. In addition to that, with our approach, we query an oracle node recursively three times. Since for each query, we need at least two transactions, one to query the oracle and one to get the reply, we need at least six transactions in order to perform a full update of the smart contract state. In the end, we need at least seven Ethereum transactions, one to call the contract and six to update its state, to update the smart contract state. To these, one needs to add at least two overhead transactions, one to deploy the contract and one to fund it. As the smart contract consumes its LINK, one would need to perform other transactions to fund it again. In any case, the upper limit of the throughput will be dependent on the congestion of the Ethereum network, and given the high number of transactions required, this approach is not suited for low-latency purposes, but it is suitable in situations where we can

allow for some latency to pass. In addition to that, if one calls the smart contract while it is being updated by the different recursive calls, the smart contract will behave in unexpected ways and could be broken; thus in a production environment, this contract should only be called in a safe manner. The unexpected behavior happens because if two requests are made to Chainlink at the same time, then Chainlink will attempt to satisfy both of them.

[EV6] How much ether is spent to use the oracles? Ether is only spent when deploying the contract, when funding it with LINK tokens, and when calling it. The amount of Ether needed for the operation of the contract is related only to the gas required to send small transactions that carry relatively light information, in particular, a transaction of LINK tokens between two addresses and the information to call the contract. In the testnet, each oracle request requires 0.1 LINK.

[EV7] Are there any further limitations? There are further limitations to the Chainlink oracle. In particular, there is an additional asset to be consumed, which is LINK tokens. LINK tokens on the mainnet have a cost in real currency and need to be purchased, earned, or acquired in some way in order to interact with the Chainlink network. Another limitation is that when interacting with the off-chain world, we are limited by the work that existing Chainlink jobs can perform. This means that we either have to increase our costs by designing our own oracle node and job or design our smart contracts to leverage existing jobs in an efficient way, which may mean that we incur a higher number of transactions and thus higher nodes. Finally, we are limited by the job we are using. In our case, the job we are using does not allow us to store a JSON object as is, and we forcibly need to parse it and obtain the information we require, which means that we need to perform more transactions and incur a higher cost.

Addressing our scenario

The Chainlink oracle addresses many of the aspects of our scenario. We recommend revising **Section 5.1** to see a description of our scenario. The Chainlink oracle can be used as a general way to create an archive of articles or other pieces of data. Our simple smart contract stores the entire information of an article but it is able to parse it into three variables: title, description (or content), and digital signatures. This means that with a similar approach, the smart contract could store and parse multiple pieces of information coming from a data source, which would be useful in case the data source is a weather station providing multiple data points and we only really need one or two of them. The on-chain parsing reduces the complexity level compared to the Provable oracle as we do not have any need to create a more complex system in order to parse data off-chain. It is important to remind though that the Chainlink oracle relies on the jobs that can be performed by different oracle nodes, and thus it may be needed to implement a custom job or even run a Chainlink oracle should the processing of incoming information require complex operations that are not already offered by already running Chainlink nodes.

Since the data is already parsed, it means that smart contracts that use it can operate without the need for additional off-chain processing. It also means that the storage

of articles is of easier implementation, as that information can be more easily and efficiently paired up with the Chainlink request identifier that is associated with each request. The Chainlink oracle does not provide natively any sort of authenticity proofs, and thus with our implementation, it is limited in its use by the websites that digitally sign their data on their own. Since the signatures are generated by the data source itself, the legal implications of this oracle may be different compared to the Provable case, but again, the legal framework for blockchain and digital signatures is still in its infancy in most countries in the world.

5.5 Comparison Between Provable and Chainlink

Chainlink and Provable are different technologies. Furthermore, Chainlink can be implemented in three ways, including the one we propose. **Table 5** is a summary that compares these two oracles at a glance.

5.5.1 Security

The first difference between the two is that Provable offers data integrity through authenticity proofs. It is possible to choose the type of authenticity proof that one prefers, but they all work in a similar way. Chainlink does not offer data integrity instead. Provable and Chainlink are similar when it comes to attackers. In particular, if the attacker wants to impersonate the oracle service, they would need to obtain the private key used to sign the blockchain transactions between the oracle smart contract and the consumer smart contract that queries the oracle. If the attacker instead wants to impersonate the source of data, both technologies rely on TLS and HTTPS to provide protection against that threat. It needs to be mentioned, though, that since Chainlink is decentralized, a node can misbehave and the node itself can be the attacker, which means that the attacker can impersonate the source of data if it controls the queried oracle node. This cannot happen in the case of Provable as one always interacts with the same Provable interface. In the case of Chainlink, one can either interact with always the same oracle nodes, or change them in function of their reputation, costs, or other considerations the developers of the smart contracts might have.

This discussion leads us to compare how an attacker might impersonate the oracle services. This is actually a similarity between the two, as in both cases an attacker can be successful in impersonating an oracle only if it has access to the private key used to sign the transactions between the oracle and the consumer smart contract. In Chainlink, unless some special measures are taken similar to our proposed solution, there are no ways to allow for non-repudiation of data that is submitted to the oracle. Our solution makes it impossible for a web app to repudiate data, but it needs the web app to generate the authenticity proofs by itself. Provable instead through the use of TLSNotary authenticity proofs is able to provide non-repudiation of data.

5.5.2 Performance

The number of requests per unit of time that can be performed in a unit of time depends in both cases on the Ethereum network congestion. In our scenario, in all cases, Provable has a higher throughput, as it requires fewer transactions to satisfy a single query. Provable only requires two transactions, whereas Chainlink requires at least seven using our own proposed solution. If one runs their own oracle node, they would be able to create a job that would perform the request in less than seven nodes, but at the very least it would be two transactions, one to request data and one to fulfill the request, with an additional one when needed to fund the consumer smart contract. It is also important to note that, due to limitations in the Provable API, Provable can only provide strings as results of queries and they cannot be parsed. An advantage of Chainlink, though, especially if one chooses a job that can be performed by multiple different oracle nodes, is that it is not limited by a single service. This means that if Provable is overwhelmed with requests, it may not be able to fulfill them, whereas in the case of Chainlink, the requests are made to many different oracle nodes and in case a node does not fulfill the request, one can query another node. The Ether that is spent in both cases is similar, but it is higher for Provable, especially when demanding authenticity proofs. This is due to the fact that Provable queries are paid for in Ether, whereas Chainlink queries are paid for in LINK tokens.

5.5.3 General

The main difference is that Provable is a centralized oracle service, whereas Chainlink is a decentralized oracle service or network. This means that when we are interacting with Provable, we are always interacting with it in the same way and with the same parameters. With Chainlink we have more flexibility as when developing a smart contract, we can choose to interact with different oracle nodes.

Finally, both Chainlink and Provable are limited in similar ways by the computational resources available to smart contracts when it comes to the verification of authenticity proofs, digital signatures, and the limitations of their APIs. We argue that Provable is more limited, as it offers one API as is. In Chainlink, instead, one can change the oracle nodes their smart contract is interacting with and can even create their own oracle node with the job specifications and custom API. This means that we have more flexibility with Chainlink. Furthermore, Provable is a centralized service whereas Chainlink is a network. Provable is offered by a company, which means that it can be shut down more easily, in case the company behind Provable wants to deprecate it. It also means that if Provable starts misbehaving, one needs to make entirely new smart contracts. Chainlink is instead a network, and should the Chainlink foundation cease to exist, the network can still continue to function and develop as the protocol would still continue to exist. The same reasoning applies to whether a specific oracle node or more nodes are taken offline. And finally, Chainlink always offers a developer the possibility to make their own oracle node, which is a great advantage over Provable.

After having seen how the two different inbound oracles operate, we can have

Table 6: Comparison between Provable and Chainlink oracles

Criteria	Data integrity	Data source impersonation	Oracle service impersonation	Data repudiation	Requests per unit of time	Ether spent	Further limitations
Provable	Native authenticity proofs	Get TLS private key of data source	Get blockchain private key of oracle service	Get shared private key for TLSNotary proof	Dependent on Ethereum network congestion; 1 request per two blocks	Fees for 2 transactions + Provable fee	Centralized third-party oracle; query function limited in functionality; authenticity proofs to be verified off-chain
Chainlink	No native data integrity; proposed from data source	Get TLS private key of data source	Get blockchain private key of oracle node	No native countermeasure; proposed through digital signatures	Dependent on Ethereum network congestion; multiple blocks per request	Fees for 1 Ethereum transaction	LINK token spent for oracle node services; decentralized network; functionality limited by oracle node jobs

a discussion about the use cases that they are both better suited. As our analysis had a specific use case in mind, there are many similarities between the use cases, but there are some differences as well that are worth noting. The Provable oracle is suited for a situation where we have an external data source and we wish to import its data into the blockchain. It is better suited when we do not possess the data source, as Provable is able to generate authenticity proofs natively that can be later verified off-chain. Provable, though, is only able to return strings and is not able to parse them, thus it should be used either when the API that Provable is interacting with either returns simple pieces of information or when we need a full object that does not need any parsing. Examples of simple pieces of information are the price for an exchange pair between currencies; data points for weather stations such as temperature or humidity; dates or numbers and other similar situations. The other case is explained by our experiment, as our data source returns a JSON object that contains all of the information needed to identify it, in particular the title and the description. The main drawbacks related to Provable are two. The first one is that the authenticity proofs can only be verified off-chain, thus Provable is not suited in case we need our smart contract to operate automatically without any sort of human input unless there is an additional trusted oracle that is set up so that it interacts with the smart contract in order to provide a boolean value depending on whether the authenticity proof is valid or not. This adds a difficulty layer and the need to run and implement an oracle and possibly an Ethereum endpoint/node, thus removing most of the advantages of using an oracle service such as Provable. The second main drawback of Provable is that it is a fully centralized oracle service, thus it is maintained by a single company that sets the prices and can go out of business. Thus, should Provable modify its functionality, go out of business, or set prices that are higher than what the smart contract is willing to pay, the entire system breaks down, mainly because the consumer smart contract will not be able to be updated due to limitations in the blockchain.

The Chainlink oracle is again well suited for a situation in which we have an external data source and we want to fetch its data into the blockchain. Chainlink does not provide any kind of authenticity proof natively. This means that this oracle is well suited when we own the data source and thus we can generate an authenticity proof or a digital signature on it directly. This also applies to the case in which we do not own the data source, but it still provides data authenticity and integrity through some other means. The main drawback to digital signatures and authenticity proofs is that, as in the previous case, these cryptographic objects are too computationally heavy to be verified on-chain, therefore the verification is to be set up off-chain. A solution could be, as in the previous case, to build an additional oracle with the purpose of verifying digital signatures and authenticity proofs. Given the decentralized nature of Chainlink, though, one can make use of different oracle nodes as long as they perform the same jobs, and `market.link` provides a Chainlink node market with information about both the reputation of a node (ie, how likely it is to provide true information) and the cost of its services. Should one prefer, one can set up a Chainlink node [Chab] and a relative job to perform the exact operation that is needed for the specific scenario. In this case, we have a trusted oracle and thus we remove the need of having authenticity proofs or digital signatures. A Chainlink node needs to interact

with an Ethereum endpoint, but apart from that it requires fewer resources than running an Ethereum node. The main drawback of this approach is that we need to trust a specific endpoint to operate truthfully unless we wish to build our own Ethereum node. Another advantage of having a proprietary Chainlink node is that you do not need to pay for the services of the node, and if it is open to the rest of the network other users may pay for your services, and thus it can become a source of income. Since Chainlink is a decentralized network, another advantage is that if the consumer smart contract is built to interact with multiple oracle nodes depending on costs and reputation, as well as the jobs that they can perform, should a node lose its reputation or go offline, there will be other nodes to fall back on, thus granting more resilience to the consumer smart contract over time. And finally, for the same reason, Chainlink network does not depend on a centralized institution that can go out of business, so even if the Chainlink foundation ceases to exist, the network can still continue operating without it. The last drawback of Chainlink is that the smart contract and its developers and users need to interact with both Ether and Link tokens, as Ether is needed when a user interacts with the smart contract and LINK is needed when the smart contract interacts with the Chainlink network.

5.6 Evaluation of Outbound Oracle

In this section, each paragraph will address one evaluation criterion.

[EV1] How is data integrity provided and satisfied? The data integrity is provided by the blockchain itself once the data is on-chain, as the blockchain leverages cryptographic primitives to build a secure system. In our experiment, the original data source of the data, though, is not the blockchain itself, but it is an off-chain data source. In general, it is safe to assume that data integrity is provided by the blockchain once the data is sent on-chain, but in our experiment, our original data source provides data integrity as well through the use of digital signatures. This also provides data integrity in the case of a malicious or faulty Ethereum endpoint or in the case an attacker was able to tamper with the communications between the destination web app and the Ethereum endpoint. Furthermore, the communication between the web server and the Ethereum endpoint relies on HTTPS, which provides a level of both encryption and integrity.

[EV2] How can an attacker impersonate the source of the data? An attacker cannot impersonate the source of data, as the source of data is the blockchain and in order to be successful, the attacker would need to either modify on-chain data, which we can consider impossible, or tamper with all HTTPS communications between the endpoint (or endpoints, in case multiple are being used) and the web app. In order for an attacker to provide false data, it would need to be the endpoint itself. Furthermore, our technique makes it so that even if an endpoint is providing false data, its validity can still be checked and thus an attacker cannot impersonate the source of data.

[EV3] How can an attacker impersonate the oracle service? An attacker cannot impersonate the oracle service, as the oracle service is a web app that is controlled by the owner and it limits its operation to query the blockchain.

[EV4] How can a malicious web application repudiate data it has submitted

to the oracle? In this case, this criterion does not apply, as it is meant for inbound oracles.

[EV5] How many requests per unit of time can be satisfied? The requests per unit of time for this oracle are not limited by the blockchain properties but are limited by the endpoint. This means that the number of requests per time is dependent only on the endpoint's capacity to serve a high number of requests. With the free tier option, we can perform a maximum of 100 thousand requests per day. Through testing, the endpoint was able to successfully satisfy up to 10 thousand requests per second.

[EV6] How much ether is spent to use the oracles? In this case, our oracle is not performing any on-chain transaction, as its operation is limited to querying the data from the blockchain. Thus there is no ether being spent for the operation of the oracle.

[EV7] Are there any further limitations? There are some further limitations. The first one is the need to have an Ethereum endpoint/node in order for this oracle to operate. This means higher costs, measured in sovereign currency such as euros or dollars for the deployment and maintenance of the endpoint. An alternative is to use a third-party endpoint, and in this case, the costs are related to the payment for the usage of the third-party endpoint. In this case, the API key used to interact with the endpoint needs to be stored safely, as otherwise it may be stolen and malicious individuals may perform requests to the endpoint in our place. Finally, the last limitation is that the trust is shifted from the oracle to the endpoint, which means that in order to fully trust the endpoint, one needs to either run the endpoint themselves or use multiple endpoints; both alternatives end up rising the costs. Our solution, which does not result in higher costs relies on the fact that the data has a digital signature used to check its validity, and while this approach may be useful, it requires that the original data source implements it, thus this solution may not apply to all scenarios.

This outbound oracle is suited for a broad scenario in which we have an external web application that interacts with a smart contract. It is designed to interact with a smart contract that stores some data as variables publicly accessible and does not need to make transactions or other operations. The security aspects are provided by the secure communication between the web app and the Ethereum endpoint that happen under the execution of a TLS protocol. Data integrity is provided by the blockchain itself. With our novel approach, the security of our web app is made endpoint agnostic and does not need a TLS protocol, as the data integrity and data authenticity are provided by the use of digital signatures. This approach though is applicable only in case the original off-chain data source already provides an authenticity proof or digital signatures, which is our situation, or in case the smart contract obtains an authenticity proof in some other way, as in the case of Provable. Given that there are no costs associated with the blockchain network, this approach is suited in case the blockchain needs to be queried with high frequency, with the limitation being the cost to interact with the endpoint, if the endpoint is a third-party one, or in case the endpoint is a first-party one then the costs are related to the running and maintenance of one or multiple Ethereum nodes. Finally, this oracle will query the blockchain on its own any time it needs to update its information. This means that it is not suited in case one needs an outbound oracle that updates only when the queried smart contract is updated. In that case, we would need a different approach.

5.7 Answer to the Research Questions

The results of the work of this chapter allow us to answer the research questions that were presented in the first section of this chapter. We will now be addressing these questions one by one.

[RQ3.1] What are the evaluation criteria to be considered?

The evaluation criteria to be considered are shown in **Section 5.1**. In particular, they are related to two broad categories: data security and costs. Thus the criteria focus on how data integrity is provided and satisfied; how an attacker can impersonate the data source or the oracle; how data can be repudiated once it is submitted to the blockchain; how many requests per unit of time can be satisfied; how much ether is spent for the operation of the oracle. The last criterion is broad and questions whether there are other further limitations.

[RQ3.2] How well does each oracle satisfy the evaluation criteria?

We will reply to this question by talking about each oracle individually. The Provable oracle satisfies successfully most of the criteria that were considered. In particular, it natively provides data integrity through the use of cryptographically secure authenticity proofs such as TLSNotary and leverages industry standard protocols such as HTTPS/TLS and the blockchain public/private key cryptosystem to make sure that no attacker can impersonate either the data source or the oracle service itself while guaranteeing that the data source cannot repudiate the data it has submitted to the oracle. The cost of the oracle is not fixed and varies and it is set up by the Provable company. The main problem related to Provable is the query function that can only return strings and cannot parse data, which means that the information that can be gotten from the data source cannot easily be manipulated by the consumer smart contract.

The Chainlink oracle is a different story compared to Provable. It does satisfy successfully many of the criteria that were considered, but the main one that is not satisfied is data integrity, as Chainlink does not provide any way to check data integrity natively. We proposed a technique to circumvent his problem, but it requires that the data source implements it, which may not be possible in all situations. Chainlink relies on two tokens to pay for its operations and each oracle node may require a different amount of LINK token to be paid for its services. The impersonation of both the data source and the oracle node is similar to the case of Provable, in the sense that an attacker would need to obtain access to either the HTTPS/TLS private key or the blockchain private/public key pair.

The outbound oracle satisfies successfully all criteria, with the exception of data integrity. It does not rely on blockchain transactions for its operations, leading to avoiding the use of ether to pay for operations. The costs are instead dependent on the Ethereum endpoint used, which can depend on the chosen solution, whether it is a third-party endpoint or a first-party one. Data integrity is not provided at all, but in our proposed solution it is provided thanks to the digital signatures that are generated off-chain before the data was submitted on-chain. This solution satisfies the criteria, but it may not be applicable to all situations. If the endpoint can be trusted, then the data integrity is provided by the blockchain itself, and not by the oracle.

[RQ3.3] How do the oracles compare to each other?
In order to reply to this question, we will first compare the outbound oracle to the other two, and then we will compare the last two.
The outbound oracle is different from the other ones because of the different functions they perform. The outbound oracle queries the blockchain for data, whereas the inbound oracles fetch the off-chain world for data to submit to the blockchain. The oracle problem relates to the trust of the endpoint in the case of the outbound oracle, whereas it actually relates to the oracle in the inbound case. In the outbound case, the oracle problem can be removed entirely by relying on a first-party Ethereum node but it will result in higher cost. Furthermore, the outbound oracle can be made entirely on the web without the need for smart contracts, which means that it can be updated as time goes on and it does not perform on-chain transactions. This is compared to the inbound oracles which essentially are smart contracts, and thus they cannot be modified once deployed and rely on on-chain transactions. The number of requests per time in both cases has an upper limit, which is higher in the case of the outbound oracle as it is limited by its endpoint.
The inbound oracles have some similarities and some differences. The similarities consist in the functionality; in the fact that they are smart contracts that perform on-chain transactions and thus have a limit on the number of transactions they can perform that depends on the blockchain network congestion; they both rely on an oracle service in order to fetch off-chain data. Furthermore, in both cases, the validity of the data can be checked either via authenticity proofs (Provable) or digital signatures (Chainlink), but this operation cannot be performed on-chain and can only be performed off-chain. A difference is that Provable is able to generate the authenticity proofs by itself for a fee, whereas in the Chainlink case, the proof is generated off-chain by the data source itself, but it is this way only because we implemented it so, otherwise, it would not happen. While they both need ether to pay for their transaction fees, Provable requires further payment for their operations in Ethereum and Chainlink requires further payment in LINK, which is the Chainlink ERC-677 token for the Chainlink oracle nodes network. Another similarity between the two oracles is that an attacker can impersonate the oracle service or the data source only if they gain access to either the TLS private key (for the data source) or the private key of the account that manages the Ethereum smart contracts. There are further limitations that are different between the two oracles. In the case of Provable, the query function used to fetch off-chain data only operates with strings and the parsing operation is too computationally heavy to be performed on-chain. In the case of Chainlink, it is important to keep in mind that Chainlink is a decentralized network of oracle nodes and that each node can perform different jobs, thus it is important to choose a reputable node as well as one that can perform the job that we need. The oracle node that we use can only be interacted with through the jobs it can perform, and thus it may happen that there is no job able to perform the exact operation we need, as it was in our case, or that the only oracle node (or nodes) able to perform it is not reputable.

5.8 Lessons Learned

Implementing and evaluating three oracles gave us insight into the process. In particular, given the limited computational availability of smart contracts, it is important to devise a security strategy that leverages both off-chain and on-chain strengths. The best way to mitigate data poisoning from an attacker is to use authenticity proofs or digital signatures that leverage cryptographic primitives to attest to the truthfulness of data. The challenge is that these proofs need to be verified off-chain, thus there is a need to have an Ethereum node or endpoint to read the blockchain and verify the proofs. The main limitation of the suggestion above is that oftentimes the data source does not provide authenticity proofs, thus one needs to sue third-party solutions, such as Provable [Pro] and TLSNotary [TLS]; another solution is to instead deploy your own oracle or oracle node as in Chainlink [Chad] and then hardcode the requests of the smart contract to it. The security of both the smart contract and the oracle will then reside in the private keys used for managing the contracts and the web servers. The best security practice here is to use dedicated hardware modules to store cryptographic secrets. Man-in-the-middle attacks are a worrying threat, but in our design, we mitigated it using TLS, which is also the industry standard according to our SLR. MITM attacks can happen at any point during the communication of different elements of our process, in particular between off-chain and on-chain ones. Finally, the oracles we deployed are not suited for either real-time applications, as they require multiple blocks to get all of their data; nor for high-frequency applications as the cost per each request is moderately high, over 5 cents per request. These oracles are instead useful for high-latency applications and for low-frequency ones.

5.9 Limitations

The main threat to validity is that this work only focused on a few very specific technologies with a very specific scenario in mind. Different scenarios and different technologies may yield different results. Furthermore, we did not take into account the management of identities or public key infrastructure, as confidentiality was not taken into account given that the scenario envisioned the sharing of publicly available information. Another threat to validity is that our research was not able to test the validity of authenticity proofs for Provable. The reason is that the generation of authenticity proofs and their verification is resource intensive, and thus the generation is available only on the mainnet. For this reason, the authenticity proofs were not properly analyzed in this work. When it comes to the Chainlink oracle, our evaluation only takes into account the case of a data source generating its own digital signatures, and it implicitly assumes that the signatures are verified off-chain. In our case, they are verified off-chain by our outbound oracle. The evaluation of the Chainlink oracle is also limited by the centralized approach, as we are not using Chainlink as intended.

6 Conclusion

In this work, we explored the security properties of blockchain oracles. We conducted a systematic literature review in order to expand our knowledge on the topic and learn the state of the art in both industry and academia when it comes to the security practices, requirements, and scenarios of applications of blockchain oracles. We then chose one of these scenarios and implemented it using different blockchain oracle technologies in order to compare them. The scenario we chose was that of a media outlet that acts as a public data source accessible to any party. The media company publishes articles that need to be stored on the blockchain in order to take advantage of different blockchain properties offered by it; in particular data integrity and immutability, which are needed to combat the practice of stealth editing. We used different technologies, in particular some blockchain-specific technologies such as oracle services, Ethereum endpoints, and the Remix IDE. We used web development technologies to develop our data source and destination, in particular Node.js and Javascript, along with Github and Render.com. We implemented two pull-inbound blockchain oracles using Provable and Chainlink that fetch data from a website and store it on-chain. Furthermore, we presented a novel way of using the Chainlink technology in order to use a decentralized oracle without the need of trusting any single node, as the data integrity is provided by the original data source and can be verified off-chain. We also implemented an outbound blockchain oracle to work in pairs with our inbound oracles, in particular with our Chainlink oracle, in order to show a complete process, with a flow that starts off-chain, goes on-chain and finally comes full circle off-chain again. We then defined several evaluation criteria for our oracles. This provided us with information to objectively compare the oracles. The strategy that we used can be applied to other pairs of oracles and to scenarios different from the one we considered. We also used the same criteria to analyze our outbound oracle in order to have a full understanding of our scenario. The last two sections of this chapter answer the research questions that motivated this work and the last one provides leads for future work.

6.1 Answer to Research Questions

This section reiterates our main research questions and objectives. Given the importance that blockchain oracles already have and will have in the coming years, they play and will play an ever-increasing role in many different scenarios as they are the interface between the off-chain and on-chain worlds. Due to their critical aspect, our goals were to first analyze whether and how the security requirements are satisfied by the different blockchain oracles; then to identify threats against blockchain oracles and systems using them in a more general sense; finally, the last goal was to develop technique and strategies to protect against the threats and implement them. Our research questions were:

1. **[RQ1]** What is the current state of the art for securing blockchain oracles?
2. **[RQ2]** How to implement two widely used blockchain oracles securely?

3. **[RQ3]** How can one compare the two blockchain oracles?

6.1.1 Answer to [RQ1]

The answers to the four subquestions in **Chapter 2** are the answer to the research question **[RQ1] What is the current state of the art for securing blockchain oracles?**

In particular, we saw that there is not much research done on blockchain oracles from a security point of view. Whenever security aspects are mentioned, their scope is limited to a specific situation or use case. There are different techniques, either software-based or hardware-based and they all rely on digital signatures. The main threat is data poisoning by an attacker feeding false information to the blockchain; other threats are more specific to the scenario the blockchain oracle was designed for. The main assets to be taken into consideration are the normal operation of the smart contracts, the data in the blockchain, and the normal execution of business operations, which depend on the scenario being considered.

6.1.2 Answer to [RQ2]

We can now use the answers to our sub-research questions in **Chapter 4** to answer the main research question of this subject: **[RQ2] How to implement two widely used blockchain oracles securely?**

In order to implement our blockchain oracles, we needed to first choose the technologies to use, which was done using the results of our SLR. After that, it is important to define the assets to implement for the blockchain oracle. This operation requires a complete understanding of the purpose why the oracle is being implemented, with particular attention to the business logic of the system. The second step is to analyze what threats are the most probable to happen to our system and the last one is to implement countermeasures to mitigate them. The threats and the respective countermeasures are dependent on both the scenario and the chosen technologies, thus we recommend reading this chapter thoroughly to go in depth.

6.1.3 Answer to [RQ3]

We can now use the answers to our sub-research questions in **Chapter 5** to answer the main research question of this subject: **[RQ3] How can one compare the two blockchain oracles?**

The way that we suggest comparing two blockchain oracles involves multiple steps. The first one is to define evaluation criteria in order to have a definite way to establish metrics and considerations that enable us to have a qualitative or even quantitative way to compare the two oracles with each other. In **Section 5.1** we propose the evaluation criteria that we used for our analysis. The evaluation criteria that we chose are relatively general and can be applied to a wide variety of use cases. We also did not assign different weights to the criteria, but this is an operation that can be performed in case some criteria are more important than others.

The second step is to gather information about the oracles and analyze them through the eyes of the chosen evaluation criteria. In this way, it is possible to gather objective information tailored to our scenario about our oracles. In our analysis, we analyzed two inbound oracles and one outbound oracle. We recommend reading the relative **Sections 5.2, 5.3, and 5.5** to have the full information about them.

The last step is the actual comparison between the oracles, which is performed by seeing how differently and how similarly the many different oracles satisfy or do not satisfy the different criteria. We recommend reading **Section 5.4** and the reply to the sub-research question **[RQ3.3]**.

6.2 Future work

This section contains our final remarks as well as our consideration for future research that should be conducted in this field.

This work shows that there are limitations in the current state-of-the-art research about the security of blockchain oracles. A good starting point for future work is to reproduce a similar analysis and implementation of oracles to what was conducted here, but taking into account different scenarios, as our work only focused on one specific scenario. It is also paramount to perform similar analyses on other blockchain frameworks, as our work mostly focused on Ethereum, but most industries are or may be more interested in similar analyses for permissioned or private blockchains. Our research focused on a scenario where the off-chain data source provides publicly available data; there needs to be research done on similar techniques but where the data is confidential and not publicly available. In addition to that, our off-chain data destination was again a web app, but this may not always be the case, as it can be any kind of device that is connected to the internet depending on the situation at hand.

Our systematic literature review showed that the research is mostly focused on performance improvements and economic incentives when using blockchain oracles or is focused on decision algorithms for decentralized blockchain oracles. Future work should also include a systematic literature review of decentralized blockchain oracles, as they have different properties and characteristics compared to centralized ones. We made use of a decentralized oracle network in our implementation phase, but the way we used it was more similar to a centralized paradigm than a decentralized one, as we hardcoded the query to one specific oracle node and did not implement any kind of decision algorithm to choose among different oracle nodes.

Another future research lead is to find a systematic way of defining when a centralized oracle is preferred over a decentralized one and vice versa, especially for the scenarios that are the most common in the industry, such as supply chain and healthcare. We have seen that centralized oracles have some advantages and disadvantages during the SLR, and we briefly discussed some differences with decentralized oracles in our implementation phase. This aspect needs to be expanded upon for decentralized oracle networks, and there needs to be a way to discern whether centralized oracles are still a useful option given the existence of decentralized ones. In particular, if one can make a decentralized oracle node and interact with it as if it was centralized, as we did in our implementation work, then the question becomes to

investigate the difference between this approach and a fully centralized oracle.

Our work also showed that this field is still going through a period of high-velocity innovation and development, and thus research written only a few years ago using certain technologies may already be out of date, as in the case of the TownCrier oracle which now is a deprecated project. Thus it is important for research to be conducted frequently as long as the field continues to go through this rapid developmental phase. Furthermore, it would be beneficial for any security research to focus on the more theoretic and implementation-agnostic aspects so that the results can stay relevant for longer periods of time and is not bound to a few very specific technologies.

Our work also proposed a novel approach to make use of a decentralized oracle network to build secure oracles without the need to trust the different nodes. The approach we showed is blockchain agnostic and can be applied to many different situations, as long as the original data source provides authenticity proofs or digital signatures for the data that is fetched. A future research lead is to replicate the same approach on other blockchains and networks and to overcome the limitation that the data source needs to provide an authenticity proof or a digital signature so that this approach can be applied more broadly and easily with a less restrictive set of conditions.

References

[ABAQS+19] Hamda Al Breiki, Lamees Al Qassem, Khaled Salah, Muhammad Habib Ur Rehman, and Davor Sevtinovic. Decentralized access control for iot data using blockchain and trusted oracles. In *2019 IEEE International Conference on Industrial Internet (ICII)*, pages 248–257, 2019.

[AHuRSS20] Hamda Albreiki, Muhammad Habib ur Rehman, Khaled Salah, and Davor Svetinovic. Trustworthy blockchain oracles: Review, comparison, and open research challenges. *IEEE Access*, PP:1–1, 01 2020.

[Aug] Augur homepage. Augur https://augur.net/.

[BHP20] Benedikt Berger, Stefan Huber, and Simon Pfeifhofer. Oracleslink: An architecture for secure oracle usage. In *2020 Second International Conference on Blockchain Computing and Applications (BCCA)*, pages 66–72, 2020.

[BTC] Bitcoin whitepaper. Bitcoin: A Peer-to-Peer Electronic Cash System, Nakamoto, S. https://bitcoin.org/bitcoin.pdf.

[Cal20a] Giulio Caldarelli. Real-world blockchain applications under the lens of the oracle problem. a systematic literature review. In *2020 IEEE International Conference on Technology Management, Operations and Decisions (ICTMOD)*, pages 1–6, 2020.

[Cal20b] Giulio Caldarelli. Understanding the blockchain oracle problem: A call for action. *Information*, 11(11), 2020.

[Car20] Toni Caradonna. Blockchain and society. *Informatik Spektrum*, 43(1):40–52, Feb 2020.

[Chaa] Chainlink homepage. Chainlink https://chain.link/.

[Chab] Running a Chainlink Node. Running a Chainlink Node. Chainlink Foundation. https://docs.chain.link/chainlink-nodes/v1/running-a-chainlink-node.

[Chac] Testnet Oracles. Chainlink https://docs.chain.link/any-api/testnet-oracles/.

[Chad] What is a Chainlink node operator? Chainlink https://blog.chain.link/what-is-a-chainlink-node-operator/.

[CYX21] Lili Chen, Rui Yuan, and Yubin Xia. Tora: A trusted blockchain oracle based on a decentralized tee network. In *2021 IEEE International Conference on Joint Cloud Computing (JCC)*, pages 28–33, 2021.

[CZW+22] Sheng Cao, Qian Zhang, Dongdong Wang, Peng Xiangli, and Xiaosong Zhang. Hybrid smart contracts for privacy-preserving-aware insurance compensation. In *2022 IEEE Wireless Communications and Networking Conference (WCNC)*, page 1533–1538. IEEE Press, 2022.

[ERCa] ERC-20 token standard. Ethereum foundation, multiple authors https://ethereum.org/en/developers/docs/standards/tokens/erc-20/.

[ERCb] ERC-721 non-fungible token standard. Ethereum foundation, multiple authors https://ethereum.org/en/developers/docs/standards/tokens/erc-721/.

[ERCc] ERC677. Github, Ethereum community https://github.com/ethereum/EIPs/issues/677.

[Etha] Ethereum Full Node Sync (Archive) Chart. Etherscan https://etherscan.io/chartsync/chainarchive.

[Ethb] JSON-RPC API. JSON-RPC API https://ethereum.org/en/developers/docs/apis/json-rpc/.

[ethc] Proof-of-stake (pos). Ethereum community https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/.

[FAB21] Asmae El Fezzazi, Amina Adadi, and Mohammed Berrada. Towards a blockchain based intelligent and secure voting. In *2021 Fifth International Conference On Intelligent Computing in Data Sciences (ICDS)*, pages 1–8, 2021.

[HCL+22] Mingyuan Huang, Sheng Cao, Xiong Li, Ke Huang, and Xiaosong Zhang. Defending data poisoning attack via trusted platform module and blockchain oracle. In *ICC 2022 - IEEE International Conference on Communications*, pages 1245–1250, 2022.

[HMPR04] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, 2004.

[Inf] Infura Homepage. Infura https://www.infura.io/.

[KC07] Barbara Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. 2, 01 2007.

[KSG+20] Petar Kochovski, Vlado Stankovski, Sandi Gec, Francescomaria Faticanti, Marco Savi, Domenico Siracusa, and Seungwoo Kum. Smart contracts for service-level agreements in edge-to-cloud computing. *Journal of Grid Computing*, 18(4):673–690, Dec 2020.

[Lee18] Kalev Leetaru. Stealth Editing and Transparency: Why Archiving Fact Checks Is Vital. *RealClearPolitics*, Jun 2018.

[Lev22] Olivier Levasseur. Model driven engineering of blockchain oracles, master thesis, 2022.

[LXSY20] Sin Kuang Lo, Xiwei Xu, Mark Staples, and Lina Yao. Reliability analysis for blockchain oracles. *Computers & Electrical Engineering*, 83:106582, 2020.

[MBY+20] Mohammad Moussa Madine, Ammar Ayman Battah, Ibrar Yaqoob, Khaled Salah, Raja Jayaraman, Yousof Al-Hammadi, Sasa Pesic, and Samer Ellahham. Blockchain for giving patients control over their medical records. *IEEE Access*, 8:193102–193115, 2020.

[MCK19] Hajar Moudoud, Soumaya Cherkaoui, and Lyes Khoukhi. An iot blockchain architecture using oracles and smart contracts: the use-case of a food supply chain. In *2019 IEEE 30th Annual International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, pages 1–6, 2019.

[MCK21] Hajar Moudoud, Soumaya Cherkaoui, and Lyes Khoukhi. Towards a scalable and trustworthy blockchain: Iot use case. In *ICC 2021 - IEEE International Conference on Communications*, pages 1–6, 2021.

[ORA] What is a blockchain oracle? What is a blockchain oracle?, Chainlink, multiple authors https://chain.link/education/blockchain-oracles.

[POS] MIT Technology Review. The Merge is here: Ethereum has switched to proof of stake. Rebecca Ackermann https://www.technologyreview.com/2022/09/15/1059520/the-merge-is-here-ethereum-has-switched-to-proof-of-stake/.

[Pro] Official provable documentation. The documentation is available here : https://docs.provable.xyz/.

[RGB] What is RGB? RGB organization, multiple authors https://www.rgbfaq.com/faq/what-is-rgb.

[SHN22] Alia Al Sadawi, Mohamed S. Hassan, and Malick Ndiaye. On the integration of blockchain with iot and the role of oracle in the combined system: The full picture. *IEEE Access*, 10:92532–92558, 2022.

[SMA] Introduction to smart contract. Introduction to smart contracts, Ethereum foundation, multiple authors https://ethereum.org/en/developers/docs/smart-contracts/.

[SY22] Zhiming Song and Yimin Yu. The digital identity management system model based on blockchain. In *2022 International Conference on Blockchain Technology and Information Security (ICBCTIS)*, pages 131–137, 2022.

[Tel] Tellor homepage. Tellor https://tellor.io/.

[Tes] What are MainNet and TestNet: Key Crypto Development Stages. Phemex https://phemex.com/academy/what-are-mainnet-and-testnet.

[TLS] Tlsnotary website. https://tlsnotary.org/.

[Towa] Cornell's Town Crier Acquired By Chainlink To Expand Decentralized Oracle Network. Forbes https://www.forbes.com/sites/darrynpollock/2018/11/01/cornells-town-crier-acquired-by-chainlink-to-expand-decentralized-oracle-network/.

[Towb] Town crier documentation. https://www.town-crier.org/.

[Web] Web3.js. Web3.js. Ethereum Foundation, 2021. https://web3js.readthedocs.io/.

[WLS+19] Shuai Wang, Hao Lu, Xingkai Sun, Yong Yuan, and Fei-Yue Wang. A novel blockchain oracle implementation scheme based on application specific knowledge engines. In *2019 IEEE International Conference on Service Operations and Logistics, and Informatics (SOLI)*, pages 258–262, 2019.

[WP19] Austin K. Williams and Jack Peterson. Decentralized common knowledge oracles. *Ledger*, 4, Dec. 2019.

[WZSK21] Christopher Wiraatmaja, Yuanyu Zhang, Masahiro Sasabe, and Shoji Kasahara. Cost-efficient blockchain-based access control for the internet of things. In *2021 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, 2021.

[ZKL+22] Yinjie Zhao, Xin Kang, Tieyan Li, Cheng-Kang Chu, and Haiguang Wang. Toward trustworthy defi oracles: Past, present, and future. *IEEE Access*, 10:60914–60928, 2022.

I. Glossary

Blockchain
Smart-contract or smart contract
Cryptocurrency
BTC - Bitcoin ticker
Ether
ETH - Ethereum ticker
NFT - Non-fungible token
Off-chain
On-chain
Oracle
Centralized
Decentralized
Provable
TownCrier
TEE - Trusted Execution Environment
Oracle problem
PoW - Proof of Work
PoS - Proof of Stake
PBFT - Proof of Byzantine Fault Tolerance
IPFS - InterPlanetary File System
HLF - Hyperledger Fabric
IoT - Internet of Things
Decentralized storage
Token
Chainlink
Provable
Infura
Endpoint
Node
Node.js
Web3.js
Cryptography
Digital signatures
Authenticity proof

II. Code

app.js

```
% Importing all of the modules that are needed: Express, Mongoose,
    dotenv, Body Parser, Cors and custom Routes module
const express = require('express')
const mongoose = require('mongoose')
require('dotenv/config')
const bodyParser = require('body-parser')
const postsRoute = require('./Routes/posts')
const cors = require('cors')

% creating the app to add middlewares
const app = express()

% adding the following middlewares: cors, Body Parser for JSON
    objects, custom Router for the /posts path
app.use(cors())
app.use(bodyParser.json())

app.use('/posts', postsRoute)

% defining response for HTTP GET / request
app.get('/', (req,res) =>
    res.send("We are on home\nPublic key is: \n"+process.env.
        PUBLIC_KEY)
)

% connecting web app to the database
mongoose.set('strictQuery', false)
try {
    mongoose.connect(
        process.env.DB_CONNECTION,
    () => console.log('connected to db'))
} catch(e) {
    console.log(e)
}

% opening port to be listened to
const PORT = process.env.PORT || 3000
app.listen(PORT, () => {
    console.log(`server started on port ${PORT}`)
})
```

Listing 4: app.js is the main file of our data source

routes/posts.js

```
% importing the required modules: Express and its Router object,
    the custom Post model to interact with the database, the
    cryptography module to generate digital signatures and the
    dotenv module to interact with the execution environment
const express = require('express')
const router = express.Router();
const Post = require('../models/Post')
const crypto = require('crypto')
require('dotenv/config')

% defining our cryptographic parameters
const private_Key = process.env.PRIVATE_KEY
const signing_algorithm = 'sha256'
const format_string = 'hex'
const public_key = process.env.PUBLIC_KEY

% this function gets all posts that are present in the database
router.get('/', async (req, res) => {
    try {
        const posts = await Post.find() % this is used to obtain
            all of the posts from the database. The find() method
            with no parameters selects all posts
        res.json(posts)
    } catch(err) {
        console.log(err)
        res.json({message: err})
    }
})

% this function allows for an HTTP POST request to be made to
    create a new post and submit it to the database
router.post('/', async (req, res) => {
    console.log("received")
    console.log(req.body)
    % generating the signature which is generated by signing a
        string which is the result of appending the title and
        description strings together
    let signer = crypto.createSign(signing_algorithm)
    signer.update(req.body.title)
    signer.update(req.body.description)
    signer.end()
    let signature = signer.sign(private_Key,format_string)

    % creating a new post object with the required parameters
    const post = new Post({
        title: req.body.title,
        description: req.body.description,
        signature: signature
    })

    % verification of the signature that was just generated and
        printing to console, this is for logging purposes and can be
        removed in production environment
```

```
    let verifier = crypto.createVerify(signing_algorithm)
    verifier.update(req.body.title)
    verifier.update(req.body.description)
    verifier.end()
    console.log(verifier.verify(public_key, signature,
        format_string))

    % the following block tries to save save a post to the database
        and logs the response, both in the successful case or in a
        failing case
    try {
        const saved = await post.save()
        console.log("saving")
        console.log(saved)
        res.json(saved)
    } catch (err) {
        res.json({message: err})
        console.log(err)
    }
})

% this function gets the latest post to be submitted to the
    database
router.get('/latest', async (req, res) => {
    console.log("Getting latest")
    % the following block gets the latest post and returns it the
        user and logs it to console; otherwise it logs the error
    try {
        % here the find method is still empty to select all posts;
            the sort method contains a paramter that specifies that
            the obtained array needs to be sorted in reverse order
            and the limit method selects the first element in the
            array, which in this case is the last post to be
            submitted
        const latestPost = await Post.find().sort({ _id: -1 }).
            limit(1)
        res.json(latestPost)
        console.log(latestPost)
    } catch(err) {
        console.log(err)
        res.json({message:err})
    }
})

module.exports = router
```

Listing 5: routes/posts.js is the file that manages API requests

models/Post.js

```
% importing mongoose, which is required to define the Post object
    to interact with the database
const mongoose = require('mongoose')
```

```
% definition of the Post object as a schema for the database
const PostSchema = mongoose.Schema({
    title: {
        type: String,
        required: true
    },
    description: {
        type: String,
        required: true
    },
    signature: {
        type: String,
        required: true
    }
})

module.exports = mongoose.model('Posts', PostSchema)
```

Listing 6: models/Post.js is the file that creates the Post object

Data destination

app.js

```
    % importing the required modules: dotenv, Express, Web3js,
        cryptography and buffer module
    require('dotenv/config')
    const express = require('express')
    const Web3 = require('web3');
    const web3 = new Web3('https://sepolia.infura.io/v3/API-KEY');
        % it connects our Web3 client to the Infura endpoint for the
          sepolia testnet; the API key is not specified here as it is
          to be kept a secret
    const crypto = require('crypto')
    const buffer = require('buffer')

    % this is the contract address of the smart contract we want to
          interact with
    const contractAddress = '0
        x75aF7066bbB0e791E4424228f94d552274422f6c'

    % the contract ABI code is generated by Remix and it specifies
        the functionalities of the contract for Web3JS
    const contractABI = YOU CAN FIND IT IN THE APPENDIX

    % here we create an interface to interact with the oracle using
          the contract address and the contract ABI code
    const contract = new web3.eth.Contract(contractABI,
        contractAddress)

    % here we create an express app to deal with the routes
```

```
const app = express()

% we define a function whose goal is to get the information
    from the smart contract
async function getInfo() {
    % each of the following lines interacts with the contract
        interface and gets one of the strings stored in the
        contract
    const title = await contract.methods.title().call()
    const description = await contract.methods.description().
        call()
    const signature = await contract.methods.signature().call()

    % we create an array where we push the three strings we
        just obtained
    const arr = new Array()
    arr.push(title)
    arr.push(description)
    arr.push(signature)

    % the array is returned so that it can be manipulated by
        the function that calls it
    return arr
}

% the public key is taken from the execution environment, in a
    real world scenario this is gotten from the website that is
    the original data source
const public_key = process.env.PUBLIC_KEY

% we are setting the parameters for the cryptographic
    algorithms
const signing_algorithm = 'sha256'
const format_string = 'hex'

% the verify signature function takes as input the title, the
    description and uses them to verify the validity of the
    third input, their signature
function verifySignature(title, description, signature,
    public_key, signing_algorithm, format_string) {
    % a verifier object is created and then we append the two
        strings together
    let verifier = crypto.createVerify(signing_algorithm)

    verifier.update(Buffer.from(title)) % we use the Buffer
        because the verifier requires a Buffer object
    verifier.update(Buffer.from(description))
    verifier.end()

    % we call the verifier verify method to verify the
        signature and return a boolean value
    const isValid = verifier.verify(public_key, signature,
        format_string)
```

```
    % the obtained boolean value is returned for manipulation
        from the function that calls this verifySignature
        function
    return isValid

}

% we define a function to deal with GET requests coming to the
    homepage
app.get('/', async (req,res) => {
        % we call the getInfo function to obtain an array
            containing three strings, which host the information
             about the post
        const strings = await getInfo()

        % we parse the array to three different strings with
            meaningful names
        const title = strings[0]
        const description = strings[1]
        const signature = strings[2]

        % we call the verifySignature function by passing it
            the required arguments and store the value in a
            variable
        const isValid = verifySignature(title,description,
            signature,public_key,signing_algorithm,format_string
            )

        % we build a response to be shown in the homepage
        % the response contains the title
        let response = "Title: " + title
        % then we add the description
        response+= " Description: " + description
        % and then we add the signature
        response+= " Signature: " + signature

        % we then check the value of the boolean that tells us
            whether the signature is valid or not
        if (isValid) {
        % if the signature is valid, we mention it in the
            response
            response+=" Valid Signature"
            res.send(response)
        } else {
        % if the signature is not valid, we mention it in the
            resopnse
            response+=" False signature"
            res.send(response)
        }
    }
)

% opening up a port to listen to requests
const PORT = process.env.PORT || 3000
```

```
app.listen(PORT, () => {
    console.log(`server started on port ${PORT}`)
})
```

Listing 7: app.js This is the main file of the data destination server

ABI Code

```
% the contract ABI code is generated by Remix and it specifies
    the functionalities of the contract for Web3JS
const contractABI = [
    {
        "inputs": [],
        "name": "acceptOwnership",
        "outputs": [],
        "stateMutability": "nonpayable",
        "type": "function"
    },
    {
        "inputs": [],
        "stateMutability": "nonpayable",
        "type": "constructor"
    },
    {
        "anonymous": false,
        "inputs": [
            {
                "indexed": true,
                "internalType": "bytes32",
                "name": "id",
                "type": "bytes32"
            }
        ],
        "name": "ChainlinkCancelled",
        "type": "event"
    },
    {
        "anonymous": false,
        "inputs": [
            {
                "indexed": true,
                "internalType": "bytes32",
                "name": "id",
                "type": "bytes32"
            }
        ],
        "name": "ChainlinkFulfilled",
        "type": "event"
    },
    {
        "anonymous": false,
        "inputs": [
            {
```

```
            "indexed": true,
            "internalType": "bytes32",
            "name": "id",
            "type": "bytes32"
        }
    ],
    "name": "ChainlinkRequested",
    "type": "event"
},
{
    "inputs": [
        {
            "internalType": "bytes32",
            "name": "_requestId",
            "type": "bytes32"
        },
        {
            "internalType": "string",
            "name": "_currString",
            "type": "string"
        }
    ],
    "name": "fulfill",
    "outputs": [],
    "stateMutability": "nonpayable",
    "type": "function"
},
{
    "anonymous": false,
    "inputs": [
        {
            "indexed": true,
            "internalType": "address",
            "name": "from",
            "type": "address"
        },
        {
            "indexed": true,
            "internalType": "address",
            "name": "to",
            "type": "address"
        }
    ],
    "name": "OwnershipTransferRequested",
    "type": "event"
},
{
    "anonymous": false,
    "inputs": [
        {
            "indexed": true,
            "internalType": "address",
            "name": "from",
            "type": "address"
```

```
            },
            {
                "indexed": true,
                "internalType": "address",
                "name": "to",
                "type": "address"
            }
        ],
        "name": "OwnershipTransferred",
        "type": "event"
    },
    {
        "inputs": [],
        "name": "requestLatestPost",
        "outputs": [
            {
                "internalType": "bytes32",
                "name": "requestId",
                "type": "bytes32"
            }
        ],
        "stateMutability": "nonpayable",
        "type": "function"
    },
    {
        "anonymous": false,
        "inputs": [
            {
                "indexed": true,
                "internalType": "bytes32",
                "name": "requestId",
                "type": "bytes32"
            },
            {
                "indexed": false,
                "internalType": "string",
                "name": "title",
                "type": "string"
            }
        ],
        "name": "RequestLatestPost",
        "type": "event"
    },
    {
        "inputs": [
            {
                "internalType": "address",
                "name": "to",
                "type": "address"
            }
        ],
        "name": "transferOwnership",
        "outputs": [],
        "stateMutability": "nonpayable",
```

```
    "type": "function"
},
{
    "inputs": [],
    "name": "withdrawLink",
    "outputs": [],
    "stateMutability": "nonpayable",
    "type": "function"
},
{
    "inputs": [],
    "name": "counter",
    "outputs": [
        {
            "internalType": "uint256",
            "name": "",
            "type": "uint256"
        }
    ],
    "stateMutability": "view",
    "type": "function"
},
{
    "inputs": [],
    "name": "description",
    "outputs": [
        {
            "internalType": "string",
            "name": "",
            "type": "string"
        }
    ],
    "stateMutability": "view",
    "type": "function"
},
{
    "inputs": [],
    "name": "owner",
    "outputs": [
        {
            "internalType": "address",
            "name": "",
            "type": "address"
        }
    ],
    "stateMutability": "view",
    "type": "function"
},
{
    "inputs": [],
    "name": "signature",
    "outputs": [
        {
            "internalType": "string",
```

```
                "name": "",
                "type": "string"
            }
        ],
        "stateMutability": "view",
        "type": "function"
    },
    {
        "inputs": [],
        "name": "title",
        "outputs": [
            {
                "internalType": "string",
                "name": "",
                "type": "string"
            }
        ],
        "stateMutability": "view",
        "type": "function"
    }
]
```

Listing 8: ABI Code for Chainlink oracle