University of Zurich[UZH]

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Kirti: Decentralized Reputation and SLA Enforcement for Cybersecurity

Amadeo Charlé
Zürich, Switzerland
Student ID: 14-943-708

Supervisor: Muriel Franco, Bruno Rodrigues
Date of Submission: September 17, 2020

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

The cybersecurity industry is growing at a rapid pace, with innumerable protection services available through a multitude of online marketplaces. To secure customer's trust in the offered protection services, platform operators offer reputation systems through which customers may rate said services. However these reputation systems are implemented in a centralized manner and thus platform owners have full control over reputation data [1]. As reputation data is stored in private databases, this paradigm is lacking transparency and verifiability.

This thesis breaks with the mentioned approach by implementing a decentralized reputation system utilizing peer-to-peer databases and blockchain technology. Additionally, the proposed solution combines the reputation system with automated SLA enforcement through smart contracts on Ethereum[1]. To handle automated SLA execution the proposed solution `Kirti` allows external monitoring solutions to report violations to the SLA terms via a RESTful API. While the practical feasibility of the proposed solution is clearly demonstrated, an evaluation shows that the implemented reputation system cannot fully mitigate fraudulent ratings. Further, the incurred costs during deployment and run time of the utilized smart contracts have been shown to be highly unpredictable to to large market swings of both *ether* and *gas* prices within the Ethereum network.

[1] https://ethereum.org/

Acknowledgments

Firstly I would like to express my gratitude to my supervisors, especially Muriel Franco who has continuously inspired me with his enthusiastic assistance and unwavering support in the process of writing this thesis.

Thanks be to Prof. Stiller whose Communication Systems Group provided the organizational frame and offered the opportunity to write this thesis.

Contents

Chapter 1

Introduction

In any commercial transaction of two or more parties, trust is a key factor. Only if there is a certain level of trust, individuals will engage in any kind of business relationship. In the setting of a cybersecurity marketplace, this trust is twofold: *(i)* trust of the buyer that the seller will deliver the product or service as promised and *(ii)* trust of the seller that the buyer will pay him appropriately for the delivered product. To enhance a customer's trust, online marketplaces have long relied on reputation systems in which buyers can rate sellers or service providers and thus sellers can boast a certain reputation visible to potential customers. In regards to the majority of today's reputation systems the following limitations and concerns can be observed however:

- **Truthfulness**: The reputation scores may not reflect the actual reputation of a seller, as some raters may leave biased, personally motivated ratings to promote their own reputation or defame their competition with an unfairly low rating.
- **Traceability**: On most e-commerce platforms anyone can leave a rating with a product or service, without actually having purchased said product or service. There exists a need to verify that only verified customers may leave a rating
- **Centralization**: Reputation is managed by platforms using a centralized system architecture. As a seller's reputation is linked to an account on an e-commerce platform, he is completely dependent on said platform and its existence. Further, the platform's operators may at any time alter one's reputation as all such information is stored in a private database.

1.1 Motivation

Reputation systems in general have been shown to enhance customers' trust and thus make for more sales on an online market place [2, 3]. In the jungle of cybersecurity solutions, marketplaces exist that implement a reputation system, however all such marketplaces make use of a centralized reputation system, as do all major online marketplaces [1], in which the reputation data is fully managed by the platform administrators. As such,

the storage, calculation and distribution of reputation data is liable to manipulation by potentially malicious platform operators.

Reputation systems built on a blockchain have been envisioned in the literature due to its inherent properties of security, immutability and decentralization of data [4, 5]. Such a system would for sure provide value for the cybersecurity market as both service providers and customers could profit from the mentioned characteristics.

In contrast with centralized systems, a reputation system which is decoupled from a marketplace in a decentralized manner, could allow protection service providers to build a reputation cross-platform, enabling especially new and less established players in the industry to build up a reputation and customers' trust, without being at the whims of a particular cybersecurity marketplace. In such a decentralized system reputation scores would ideally be immune to rating fraud and illicit modification. As regards rating fraud, blockchain based systems have been shown to effectively prevent certain attacks by design [6]. As a consequence, customers would have more confidence in the validity of a service's reputation and are more likely to purchase a solution which is satisfactory. Additionally, stakeholders entering the cybersecurity market would not have to develop their own reputation system but could simply integrated with an already existing system as has been proposed in the literature [5]. Further, a decentralized reputation system could be of relevance to protection service recommender systems such as *MENTOR* [7] to include a provider's reputation in their recommendation process.

Besides reputation systems, Service Level Agreements (SLAs), which provide agreed-upon performance targets of an IT solution [8] are used in the cybersecurity industry to enhance customers' trust in a protection service. An SLA is an agreement between a customer and a service provider defining the technical performance for a service as well as the responsibilities of the involved parties [9]. As such, the SLA of a cybersecurity protection service may include the details regarding a financial compensation for the customer in case the service performs below the benchmarks as specified in the SLA. While SLAs provide a well-defined catalogue of targets and responsibilities, in their practical application SLAs may add a lot of overhead for customers.

As such, the customer is usually required to file violation reports himself which raises the need for contracting a monitoring solution. A service provider may further make the process for reporting violations intentionally time and resource intensive, thus entirely discouraging a customer from filing claims. Additionally, a service provider may ignore the claims of the customer and not reward him accordingly. These issues which can currently be identified in the cybersecurity industry motivate the need for an automated SLA enforcement. Smart Contracts (SCs) as implemented by the Ethereum blockchain allow for automatic execution of arbitrary business logic and can thus also be applied to handle the automatic execution of SLA terms. In this way the compensation scheme and payment of the underlying service of an SLA can be handled in a transparent and decentralized manner, curbing the need for a trusted third party.

1.2 Description of Work

The main goal of this thesis is the design and prototypical development of a blockchain-based reputation system with application to the cybersecurity market, including automated SLA enforcement. In an initial stage, the thesis demands that relevant literature concerning existing reputation systems, especially those in relation to blockchain technology be assimilated. This research is to be undertaken with the aim of evaluating which characteristics a resilient reputation system must fulfill as per the requirements of the cybersecurity market. It is to be investigated if solutions proposed in the literature may be incorporated in the design stage of the thesis. Upon having gained sufficient insight into the current state of reputation systems and SLA enforcement using blockchain technology, the main use case scenarios shall be identified in order to appropriately address the nuances of the cybersecurity market. To showcase the system's usefulness in practice, a simple marketplace shall be developed, which allows service providers to upload protection services and customers to buy and rate said services. The underlying reputation system shall be designed under consideration of different attack vectors in regards to rating fraud. The SLA details of all services uploaded by providers shall be automatically encoded into SCs which handle the payment, compensation and termination of the underlying protection service. Further, the system shall allow for the integration with external parties such as *MENTOR* [7] by exposing reputation data via a RESTful API. Lastly a concluding section shall evaluate and discuss the efficacy of the proposed solution.

1.3 Thesis Outline

Chapter 2 provides a background of the relevant concepts and employed technologies necessary to fully understand the design and implementation decisions which led up to the final solution.

Chapter 3 examines the current state of related work in the academia with respect to reputation systems as well as SLA enforcement. It closes with a discussion and compares the proposed solution with other related works.

Chapter 4 presents an overview of the proposed system architecture first and afterwards examines each functional layer in more depth. It furthermore sheds light on design consideration and discusses how the components of the architecture are interconnected.

Chapter 5 showcases the implementation details of each of the components of the system architecture

Chapter 6 demonstrates the practicality of the proposed solution by means of a case study and follows up with a discussion on costs, decentralization and rating fraud

Chapter 7 concludes the thesis and hints at future work

Chapter 2

Background

This chapter introduces the relevant concepts and technologies forming the foundation of the proposed solution of the thesis. Firstly, the Cybersecurity Market and its providers are introduced, followed up by a brief discussion about reputation mechanisms. Next, the fundamental concepts of blockchain and SCs are discussed. Lastly, the InterPlanetary File System (IPFS) is introduced as a solid approach for decentralized storage to be used in the prototype implementation.

2.1 Cybersecurity Market

The digital revolution has found its way into virtually all aspects of our everyday life. Tools and technologies that were unheard of even a decade ago, now seem indispensable for most people. On a cautionary note however, with an increase in digitalization also comes an enhanced risk of cyber attacks. The cybersecurity market as a whole seeks to offer security minded corporations and individuals relief from the risk of falling victim to said attacks. The size of the cybersecurity market has been estimated to be in the billions of dollars [10], accordingly the number of service providers and protection services is enormous.

While other markets, such as the one for commercial flights, offer a fairly homogeneous product, the same can in no way be said of the cybersecurity market. The multitude of protection services for different kinds of cyber attacks and technology stacks available seems endless. As such, although a large number of flight comparison websites exist, one is hard pressed to find an equivalent online resource to compare different cybersecurity solutions.

2.1.1 Services and Providers

Recently, efforts have been made to bring clarity to customers in light of the jungle-like vastness of available cybersecurity providers and services. Solutions such as MENTOR [7]

seek to facilitate decision-making in selecting an appropriate protection service. MENTOR allows a customer to enter details about budget constraints and the type of required service needed. With the help of its custom recommendation process, the customer is presented with those protection services which are most befitting his or her needs. Further, in an effort to bundle service providers, cybersecurity directories such as Cybertango [11] have appeared in recent years. Currently Cybertango displays a catalogue of more than 1300 different cybersecurity providers. It allows to filter cybersecurity providers by location and by category such as *Big Data Protection* or *DDoS Protection*.

2.1.2 Reputation Mechanisms

In many online interactions, the interacting parties don't have a history of mutual exchange and thus generally no foundation of mutual trust is in place. In commercial transactions in which money and resources are at stake, this raises the risk of uncertainty. This general observation holds even more true in case of the cybersecurity market, in which additionally a customer has no guarantee that any protection service will deliver the promised protection.

In the setting of online commercial transactions, reputation systems seek to close this gap of trust and uncertainty by introducing reputation scores. A reputation mechanism's main objective can be summarized as helping potential customers to evaluate sellers' trustworthiness and thus minimizing the risk of engaging in a transaction with a fraudulent seller [12]. Digital reputation mechanisms thus seek to attribute a reputation score to a target user, based on a collective opinion of its past behavior [13]. The issue with the majority of reputation systems in cybersecurity nowadays is that reputation data is centrally stored by the platform administrators who have full control over the data [14]. This empowers the platform owners to potentially manipulate reputation data.

In order to implement an effective reputation system, one must consider that a reputation system should be resilient against adversarial behavior in order to build a system which actually accomplishes the goal of increasing customer's trust [13]. The sore spot of reputation system is what the literature terms as rating fraud. Rating fraud is of two kinds [15]: *Ballot Stuffing* and *Bad Mouthing*. *Ballot Stuffing* means to provide unfairly high ratings while *Bad Mouthing* means injecting unfairly low ratings. Thus, in order to design a resilient system, both types of attack have to be taken into consideration. Due to its inherent properties, which shall be discussed in the next section, blockchain technology has been proposed in the literature to be used as an effective way to counter certain attacks such as *Bad Mouthing* [12] by design. In order to prevent rating fraud in traditional systems for instance, a customer's review may be marked as "verified", implying that he or she indeed purchased the service and is thus eligible to leave a rating for the service. In blockchain based systems however, it is trivially possible to ensure that only reviews by real customers are stored in the system, thus making the system more resilient to illegitimate reviews [6].

2.1.3 Service Level Agreements

SLAs are formal contracts between customers and service providers aimed at guaranteeing that customer expectations are met [16]. They may include various information about a service's technical performance levels as well as a compensation scheme, in case the service doesn't deliver the promised performance.

In the context of the cyber security market, SLAs give prospective customers a certain level of confidence in purchasing a protection service. Although they are no guarantee that the protection service will perform as promised, they act as a warrant that a customer may at least get a financial compensation in case the agreement's terms were violated.

2.2 Blockchain

A blockchain is a distributed ledger of records such as public transactions and other events executed by participating parties [17]. It derives its name by collecting network transactions in "blocks", which are linked to one another via cryptographic hashes, thus forming a "chain" [18].

Its core feature is that it enables untrusted peers to exchange transactions of monetary value in a secure peer-to-peer way without relying on an external trusted party [19].

Blockchain technology has attracted public and academic attention due to its inherent properties of ...

1. **Immutability** Once a block has been permanently added to the network, it is computationally infeasible to revert it.
2. **Decentralization** The blockchain network is not run by a central server, but instead by a multitude of nodes - programs that can validate network transactions and may be run by anyone in a permissionless way.
3. **Smart Consensus** Reaching a consensus in a distributed system is a challenge, however different consensus algorithms exist which guarantee that network peers reach an agreement about the current state of the blockchain [20].

2.2.1 Ethereum

Ethereum is a blockchain including a turing-complete language [21], allowing for the creation of SCs. Although using a digital currency *ether*, Ethereum is not intended to be merely a cryptocurrency such as Bitcoin, whose sole intentional use case is to transfer value between peers. Instead it has been envisioned to be a programmable blockchain, able to run arbitrarily complex code [22].

Ethereum features a set of high-level programming languages which allow anyone to de-

velop applications on top of the Ethereum Virtual Machine (EVM). While multiple programming languages are available, *Solidity* has gained wide-spread popularity amongst Ethereum developers. It is an object-oriented statically-typed programming language which has been influenced by C++, JavaScript and Python [23]. It supports inheritance, use of libraries and user defined types, making it the the programming language of choice for Ethereum developers [22].

In Ethereum, there exist two kinds of accounts, both of which are associated with an Ethereum address [22]:

- **Externally Owned Accounts (EOAs)**: controlled by real-world users
- **Contract Accounts**: controlled by code executed by the Ethereum Virtual Machine (EVM).

The majority of accounts are user-controlled EOAs. They simply enable users to store *ether* and send it to other EOAs or *Contract Accounts*. In case *ether* is transfered to a *Contract Account*, the associated code controlling that account may be triggered, in which case it executes its logic in the context of the EVM. Such code controlling *Contract Accounts* constitutes what is more commonly know as a SC.

2.2.2 Smart Contracts

Work on SCs has already been done in the 1990s [24], however at that time no technology had been envisioned for their implementation. In more pragmatic terms, SCs allow for the automatic execution and enforcement of the terms of an arbitrary agreement once the specified conditions are met [19]. In theory, this allows for lower overall business costs by reducing the need for a trusted third party as would be required in a traditional business setting.

An Ethereum smart contract has the following properties [25]:

1. It is triggered by transaction which may or not include any amount of *ether*
2. It is publicly visible as it is included in the blockchain
3. It is deterministic, meaning that the same input will always result in the same output

This property of being deterministic is what makes SC so attractive: The outcome of any input is entirely predictable. Arbitrary contract logic may thus be encoded into self-executing smart contracts without requiring a trusted third party. SCs on Ethereum can thus also receive and send *ether* according to the rules of the the underlying code. Based on those properties and benefits, SCs have been applied to various realms including financial services, healthcare and voting amongst others [26]. Still, there are many challenges and opportunities to explore the concept of SCs in regards to societal, economical and technical dimensions.

2.3 InterPlanetary File System (IPFS)

IPFS (InterPlanetaryFileSystem) is a file system allowing for decentralized storage and access of files, applications and other data. It combines and extends upon established technologies such as Distributed Hash Tables, BitTorrent and Git [27]. Further, it aims to outdo HTTP as the information transfer protocol to build a better World Wide Web (WWW). In IPFS files are downloaded from peers who have bits and pieces of the requested resource, while in HTTP content is downloaded from a single server. Peer-to-peer systems have been shown to account for significant bandwidth savings as in the case of a Video Delivery Network [28].

A striking difference between IPFS and HTTP is how data is accessed, which shall be illustrated below. Suppose a user wants to research information about aardvarks - a peculiar African animal fond of digging in the ground - and thus enters the following URL in his or her browser [29]:

```
https://en.wikipedia.org/wiki/Aardvark
```

When quering the WWW using an HTTP URL as in this case, the host server (Wikipedia) is asked to retrieve the requested resource (the page about Aardvarks), thus the *location* of the content to be retrieved is of primary importance.

The user could however also view the same content using IPFS via the following URL:

```
/ipfs/QmXoypizjW3WknFiJnKLwHCnL72vedxjQkDDP1mXWo6uco/wiki/Aardvark.html
```

In the above URL, the string beginning with `QmXo...` is called a content identifier (CID) [30]. CIDs are generated by cryptographically hashing the contents of a file when it is uploaded to IPFS. Files on IPFS can again be retrieved via the CID.

Thus in IPFS, not the *location* of a file is relevant for its retrieval, but its *contents*. The nature of the underlying most commonly used crypographic hash function *sha2-256* is such that if the contents of a file change even minutely, its CID will be completely different [30]. Note that this is vastly different from location-based content retrieval as compared to HTTP URLs and file paths, where the address of a file is uncorrelated with its content.

2.3.1 OrbitDB

OrbitDB is built on top of IPFS and allows for the creation of distributed peer-to-peer databases [31]. OrbitDB breaks with the relational model so common in traditional database systems and instead provides the following kinds of database implementations [32]:

- **log**: an immutable (append-only) log with traversable history

- **feed**: a mutable log with traversable history
- **keyvalue**: a key-value database
- **docs**: a document database to which JSON documents can be stored and indexed by a specified key
- **counter**: Useful for counting events separate from log/feed data

The concept of structuring data as envisioned in OrbitDB might appear non-intuitive: In an application resembling twitter for instance, "tweets" would not be stored in a single massive database storing tweets of all users, but instead, one is advised to generate a dedicated database for each user and his tweets [33]. Each tweet in and of itself may again have its own database, storing the comments left by other users for instance. In such an approach, one may freely combine and nest the above mentioned database.

All of the above databases implement *ipfs-log*[34], a Conflict-free Replicated Data Type (CRDT). As OrbitDB is a peer-to-peer network which enables peers to replicate another's database (assuming respective permissions), the question arises how to ensure synchronization between databases in a peer-to-peer distributed paradigm. CRDTs are the answer to this question. They have been formally shown to satisfy the property of *Strong Eventual Consistency*, which implies that "[...] replicas eventually reach the same final value if clients stop submitting updates", i.e "[...] correct replicas that have delivered the same updates have equivalent state [35]."

OrbitDB makes use of this convenient property of CRDTs in its implementation of *ipfs-log*, in which entries to the log are stored in IPFS and chained together by pointing to the previous entry, thus forming a graph [34].

Chapter 3

Related Work

In this chapter an overview of work related to this thesis shall be given. Section 3.1 discusses previous efforts of different kinds in the realm of reputation systems. In section 3.2, an overview of papers dealing with systems of SLA generation and enforcement is given. Lastly, section 3.3 shows how the system envisioned in this thesis related to previous works in the academia.

3.1 Reputation Systems

With the gain of popularity of peer-to-peer networks such as Gnutella and BitTorrent, significant research efforts were made to investigate the requirements and particulars of said systems. Pioneering work was done by Kamvar et. al in 2003, by presenting a method which may reduce the effect of malicious peers on the overall system by assigning trust values to peers [36]. These research efforts lay the foundation for further research. Followup work included the improvement of the trust model - how trustworthiness of a peer can be quantitatively measured [37], as well as the application of statistical methods to calculate trust scores and endeavors to increase scalability of the overall system [38]. Introducing reputation scores in a peer-to-peer network incentivizes peers to contribute to the networking by sharing resources [38]. By sharing resources with others peers increase their reputation, while peers who solely engage in "freeriding" - downloading files without contributing resources to the network - most likely incur a lower reputation score than their resource sharing peers.

Further the issue of anonymity in reputation systems has been raised in the literature in recent years[12, 39, 13]. The rationale behind considering the topic of anonymity is that a dissatisfied customer might incur disadvantages in future transactions by leaving a low review with a service provider. Most notably, Schaub et. al [12] describe a truly anonymous reputation system applicable to e-commerce settings. When paying a service provider in said system, a customer obtains a *token* from the service provider, which later acts as the proof of eligibility when leaving a review with the service provider. Cryptographic methods ensure that the service provider has no means to link tokens to transactions and thus customer reviews.

With the arrival and gain in popularity of blockchains, various systems built on blockchain technology have been envisioned. Dennis et al. propose a blockchain meant to record transactions of reputation, with the goal of both creating a system with an increased resilience to attacks as well as providing the basis of a generic reputation system which can be used in various kinds of networks [1]. Instead of contriving an entirely new blockchain, efforts were made to use the Bitcoin protocol as basis for a reputation system [40, 41]. In 2015, Carboni [40] proposed a most simplistic feedback system based on Bitcoin transactions in which a customer can signal that he was satisfied with a producer's product.

3.2 SLA Enforcement on Blockchain

While the concept of enforcing SLAs may not be novel, until very recently the technical means to do so in a decentralized manner were not available. This changed with the arrival of SCs which are capable of reliably executing terms as defined in a contract-like document, SLAs being no exception. A number of works use SCs deployed to Ethereum to enforce SLAs in a distributed manner [42, 43, 44]. The concept of encoding the terms of an SLA into a SC is common amongst all of the cited works. The papers differ however in their approach as how to ensure that the SC is aware of the current state of the SLA, especially whether its terms have been violated or not.

Uriarte et al. [42] propose a solution in which auditors monitor a service and reach consensus on the measurements by a consensus algorithm. These measurements are then feed to an oracle which in turn inserts the data into the blockchain, enabling the SLA SC to know the current state of the service. Zhou et al. [43] introduce a different concept to monitor the solution: They suggest that in a first step a witness committee of N blockchain users shall be formed. This N is initially agreed upon by both the service provider and the customer. The members of the witness committee monitor the service in exchange for a fee. Zhou et al. use concepts from game theory to ensure that for each witness truthfully reporting their observation as to whether a violation to the service agreements occurred or not is the most profitable strategy. Scheid et al. [44] also addresses the challenging issue of monitoring. The authors point to work done in [45] in which SLA violations are measured in a peer-to-peer manner, which could be used as the source of truth for their proposed solution. These measurements could then be provided to the SC encoding the SLA.

3.3 Discussion

While there has been significant work in the field of reputation systems, SLA enforcement using blockchains is a still a novelty. This is in part due to the only recent arrival of SCs, which may be deemed as not fully mature. Digital commerce nowadays can barely be imagined without reputation systems and as such, the are also used by multiple marketplaces in the cybersecurity industry [46, 47, 48]. However, marketplaces in the cybersecurity industry make use of centralized reputation systems, as do all major online

marketplaces [1], in which the reputation data is fully managed by the platform administrators. As such, the storage, calculation and distribution of reputation data is liable to manipulation by potentially malicious platform operators. Table 3.1 provides an overview of solutions which are related to this thesis in terms of functionality and compares the supported features with the proposed solution.

Table 3.1: Feature comparison of related work

Solution	**Reputation Mechanism**	**SLA enforcement**	**User-friendly Catalogue**	**Public API**
MENTOR [7]	✗	✗	✓	✗
BUNKER [49]	✓	✗	✓	✗
Scheid et al. [44]	✗	✓	✗	✗
Uriarte et al. [42]	✗	✓	✗	✗
Zhou et al. [43]	✗	✓	✗	✗
`Kirti`	✓	✓	✓	✓

MENTOR [7] is a recommender engine, helping customers to find the most suitable cybersecurity protection service suited to their needs. A partial goal of this thesis is to provide reputation data about protection services to external parties such as MENTOR, which is currently lacking thereof. Such systems could greatly benefit from incorporating this reputation data into their recommendation process. BUNKER [49] is a package repository allowing users to purchase Virtual Network Functions (VNFs) without relying on a trusted third party, by incorporating SCs running on Ethereum. It includes a reputation system allowing license holder to rate VNF packages, however it does not address SLA enforcement or creation. Previous work by Scheid et al, Uriarte et al. and Zhou et al. has already been discussed to include the generation and enforcement of SLAs using SCs, but none of the authors make the mention of integrating their solutions into a reputation system. The unique contribution of this thesis shall be the design of a system which features the generation and enforcement of SLAs, intertwined with a reputation system for protection service, while making reputation data available to external sources.

Chapter 4

Solution

This chapter discusses the design of the proposed solution `Kirti`[1] on a conceptual level. Section 4.1 presents a visual overview of the system architecture and mentions the main user flows and actors of the solution. The remaining sections of the chapter explore the layers of the architecture individually, elucidating their relevance within the context of the solution. The goal of `Kirti` is to implement a decentralized reputation system for cybersecurity providers, including the generation and enforcement of SLA SCs. `Kirti` allows the upload and purchase of cybersecurity solutions whose SLA terms are encoded into SCs running on a blockchain, providing automatic customer compensation in the event of agreement violations. Major events of the system such as the upload of customer reviews are fully auditable by notarizing them in the blockchain and storing a reference to the blockchain record. Its reputation system is designed with possible attacks such as *Ballot Stuffing* and *Bad Mouthing* in mind, following a decentralized approach. Additionally, this reputation data is available to external parties via a provided RESTful API. Figure 4.1 introduces the conceptual architecture of `Kirti` and describes its main components, thus showing the main functionalities and actors supported by the solution. Each of the layers of the architecture will be described in the rest of this chapter.

4.1 Architecture Overview

While Figure 4.1 gives an overview of the general architecture of `Kirti`, the main flows of component interaction shall be briefly discussed, before examining each of the layer in more detail. Additionally, the main actors in `Kirti` and how they interact with its different components is given in Table 4.1

Figure 4.1 visualizes the three main usage flows of `Kirti`, namely that of a service upload, a service purchase and a rating upload each of which shall be discussed in brief here, while covering them in more depth in later sections. When a service provider wants to make his protection service available on `Kirti`, he navigates to the appropriate section in the *User layer* where he may enter the details of his protection service, including

[1] Sanskrit term which translates to *reputation*, *fame* or *glory*

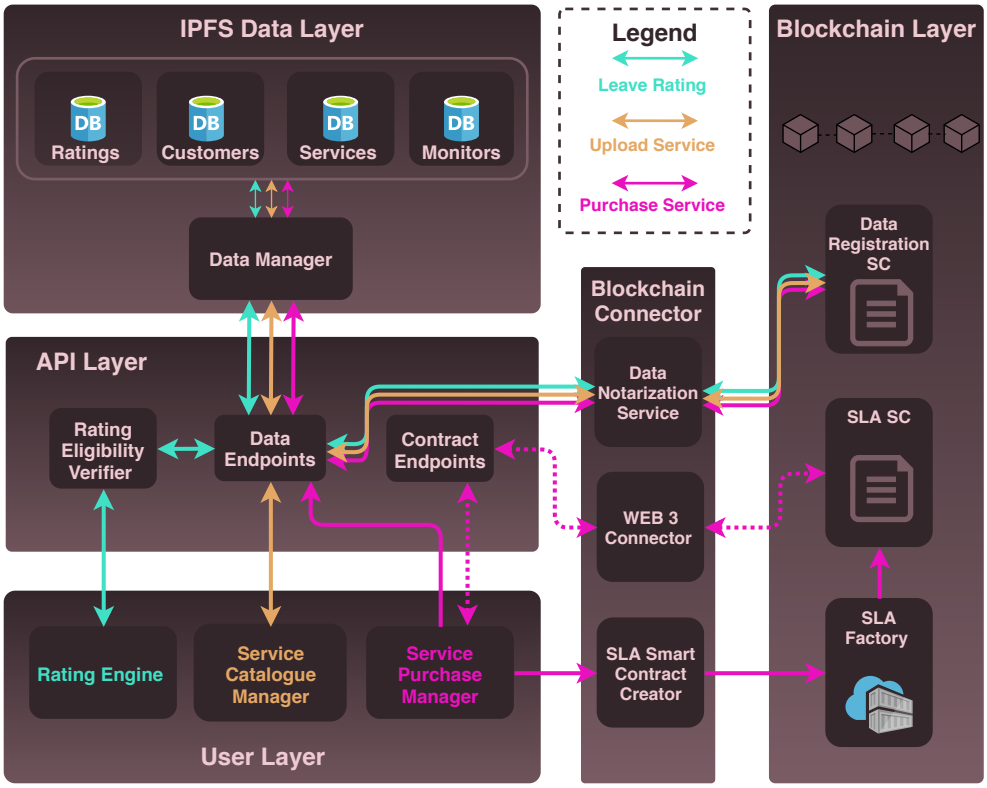


Figure 4.1: System Architecture including flows between main components

Table 4.1: Main actors of `Kirti` and their interactions with the components of the system architecture

Actor	**interacts with ...**	**to ...**
`Customer`	*Service Purchase Manager*	buy protection services
`Customer`	*Rating Engine*	rate protection services
`Provider`	*Service Catalogue Manager*	make services available for purchase
`Monitor`	*Contract Endpoints*	report violations to an SLA SC
`External Party`	*Data Endpoints*	query reputation data of services

the SLA specification. The *Service Catalogue Manager* passes the newly created service onward to the *API Layer*, which in turn invokes the *Data Notarization Service*. The *Data Registration SC* emits an event confirming the registration of the service and the newly created service is stored in the *IPFS Data Layer* together with that transaction hash.

Upon selecting a certain protection service, the *Service Purchase Manager* invokes the *SLA Smart Contract Creator* which in turn triggers the *SLA Factory* to generate a new SLA SC which encodes the SLA details of the underlying protection service. The user next makes a payment to the newly created contract, upon which the SLA SC emits a confirmation event and the *IPFS Data Layer* stores a record of that particular user having

purchased said service. Now that the SLA SC is active a monitor may report violations of the SLA agreement of the underlying service via the *Contract Endpoints*

Lastly, if a customer wants to leave a rating for a service which he has consumed, he first navigates to the according section in the *User Layer*, where he may fill out a rating form with the help of the *Rating Engine*. Next, the *Rating Eligibility Verifier* consults the *IPFS Data Manager* via the *Data Endpoints* whether this customer has purchased said service. In case the customer has indeed paid for the service at hand, the review is stored in the system.

4.2 User Layer

The *User Layer* provides a front-end, allowing users to interact with the system. Users can be divided into two groups: Those selling and those buying protection services, here referred to as service providers and customers, respectively. The *Service Catalogue Manager* displays the currently available services in the system and provides SPs with an interface to list a protection service up for sale. The *Service Purchase Manager* enables a customer of `Kirti` to purchase a protection service and informs him about the current state of his purchased SLAs. This information includes the current violation count as well as the time until the SLA expires.

4.3 IPFS Data Layer

In the endeavour to design `Kirti` in a decentralized manner, an appropriate data storage mechanism had to be identified. Storing data on the blockchain itself seems like an obvious solution, however the associated costs are enormous. All computations on Ethereum have an associated operational cost measured in terms of *gas*. The price for a unit of *gas* is termed as *gasPrice* and most commonly specified in units of *Wei*, where 10^{18} *Wei* equal one *ether*. A storage operation of 256 bits carries a computational cost of 20'000 *gas* [21]. Storing a mere kilobyte at a *gasPrice* of 50 *GWei* thus amounts to a fee of 12.5 CHF at a price of 400 CHF per *ether*. Evidently, storing data on the blockchain itself is vastly too costly, thus an alternative would have to be found.

IPFS is a suitable technology to circumvent the data storage problem blockchain developers are faced with. Regardless of the size of the uploaded contents, the cryptographic algorithm *sha2-256* most commonly employed in IPFS, always returns a hash of 32 bytes [30]. By first uploading data to IPFS and then storing the associated hash in the blockchain itself, the data storage overhead can be effectively reduced.

The question remains however how to effectively manipulate and manage this data, while making it available to external parties. In the approach of storing data on IPFS while keeping reference to such data in a SC, data modification and retrieval is an area of concern in the following two ways:

- **Costs**: To retrieve data, one would first have to invoke the SC to fetch the IPFS hash of the respective data. Note that when modifying data, as is necessary when a customer leaves a rating for a service, the *sha-256* hash of the new data is entirely different from the previous hash. Both data retrieval and modification thus involve SC function invocations which are associated with a *gas* cost.

- **Performance**: When supplying reputation data to external sources, transmission speed is of critical importance. To query data, first a SC function would to be invoked to return a storage hash, next the data pertaining to that hash would have to be retrieved from IPFS. As such, performance may deter for large data requests.

From the outset, `Kirti` was designed in such a way that reputation data be made available for external parties such as MENTOR [7] via a RESTful API, thus the potentially slow data retrieval makes the approach as mentioned above unsuitable. OrbitDB [31] was chosen as the storage medium of choice as it harnesses the power of decentralized file storage while allowing for quick data access. In an endeavour to maximize data verifiability in `Kirti`, each major event triggered by a customer is first recorded in the blockchain before being stored in the data layer. *Algorithm 1* demonstrates the generic flow of data upload in `Kirti`.

Algorithm 1 Generic Data Upload

```
procedure STOREINDATALAYER(EVENT)
    if not event already recorded then
        init transaction = smartContract.register(event)    ▷ record event in blockchain
        event.add(transaction.hash)                         ▷ add the transaction hash for reference
        store(event)
```

The service upload by a service provider, rating generation by a user as well as a SLA contract creation are all handled in the above manner. In this way, audibility and transparency are ensured. As such, each customer rating includes the transaction hash of the transaction triggered by the *Data Registration SC*. Assuming a production deployment to the Ethereum Mainnet, each rating could be audited by verifying the transaction details via the transaction hash.

4.4 Blockchain and SLA Layer

The *Blockchain Layer* encompasses the SCs deployed on the Ethereum chain. The other components of `Kirti` interact with the *Blockchain Layer* via the *Blockchain Connector* which handles the generation of SCs from SLA terms. Further it listens for events emitted from the SC.

Ethereum events are relied upon in `Kirti` to record crucial happenings in the system such as the upload of customer ratings, SLA contract creations or service purchases. They are a convenient and cost-effective way of recording small amounts of data in the blockchain.

On a more technical note, the EVM contains 5 different operational codes [21] to append log records to a transaction. These log records - more commonly referred to as *events* - are appended to a transaction, allowing for persisting data pertaining to that transaction in the blockchain.

4.5 API Layer

The main intent of the API Layer is to provide reputation data to external parties such as MENTOR [7]. Besides being accessible to external parties, it is used by the *User Layer* to interact with the *IPFS Data Layer*. It is capable of retrieving and storing data in the *IPFS Data Layer* as well as interacting with the SCs of the *Blockchain Layer*. Further, it allows monitors to report violations for a given SLA SC via a simple API call, mitigating the technical difficulties to invoke a SC function.

In `Kirti`, only those customers who have indeed paid for a protection service may leave a rating for it. Thus, before a review is submitted to the system, the *Rating Eligibility Engine* which forms part of the *API Layer* handles the logic to verify that a given customer may rate a particular protection service. To do so it consults the *IPFS Data layer*, which keeps track of all transactions within the system and can thus provide information whether a particular customer (i.e Ethereum address) has purchased a particular service. If any user could leave a rating for any service of the system, *Ballot Stuffing* and *Bad Mouthing* would be trivially possible, thus this verification of rating eligibility is required. In case a rater is ineligible, he is thus informed and denied to submit his review. Otherwise, the rating is recorded in the blockchain after which it is stored in *IPFS Data Layer*.

Chapter 5

Prototype Implementation

The present chapter offers insights into the implementation of the prototype. While section 5.1 visually showcases the *User Layer* from a user's perspective, the remaining sections are more concerned with the technicalities of the underlying implementation details of the respective layer within the context of the architecture.

5.1 User Layer

In this section the interaction with the *User layer* shall be discussed from a user's perspective, while the details regarding the underlying implementation shall be covered in the remaining sections of the chapter. The frontend of the solution is implemented using the Ionic framework[1], allowing for the creation of web and mobile, while providing a UI components library. It allows for integration with different frontend frameworks such as Angular[2], Vue[3] and React[4]. For the prototype implementation, Ionic Angular was chosen, one of the reasons being its support of TypeScript.

5.1.1 Service Upload

When a service provider wants to upload a protection service, he may navigate to the *Sell Page* panel of the dashboard to enter the relevant information about his service such as name, description, price and leasing period. The leasing period is relevant for the SLA SC derived from the specified service, as it marks the period for which a monitor may report violations of the SLA agreements to the SC. After this period the SLA SC automatically terminates and releases funds to the service provider as well as a compensation to the customer in case violations were reported. A service provider is also required to specify a valid Ethereum address to which funds from customers purchasing his service will be sent

[1] https://ionicframework.com/
[2] https://angular.io/
[3] https://vuejs.org/
[4] https://reactjs.org/

to, as can be seen in Figure 5.1. Unlike in a traditional cybersecurity market place, this transfer of customer funds is handled directly by a SC as will be seen in Section 5.2.

A peculiarity of `Kirti` is how a service provider can specify the SLA details which are later to be encoded into a SC. In order to demonstrate the feasibility of the proposed solution the metrics of *Time to Mitigate*, *Mitigation Efficiency* and *Service Availability* were selected as they may be attributed to a variety of different kinds of cybersecurity protection services. Additional parameters could be trivially added, as they are merely encoded into SCs, but serve no purpose in any computational logic in `Kirti`. Besides being of importance to users, the metrics of the SLA details are relevant to monitors who monitor whether any of the metrics of the underlying service of an active SLA SC has been violated.

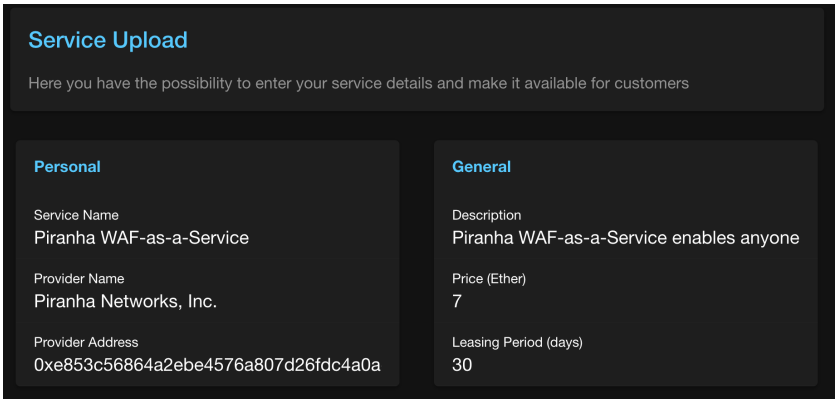
In addition to specifying values for the three mentioned metrics, a service provider is required to specify compensation percentages as well as thresholds for a violation of high, medium and low severity. Figure 5.1 shows an example of an *SLA Details Specification*. These values determine the compensation of a service's customer in the case of a violation of the SLA terms as calculated by an SLA SC. The details of how monitors report violation using these threshold values shall be discussed in 5.4.

5.1.2 Service Purchase

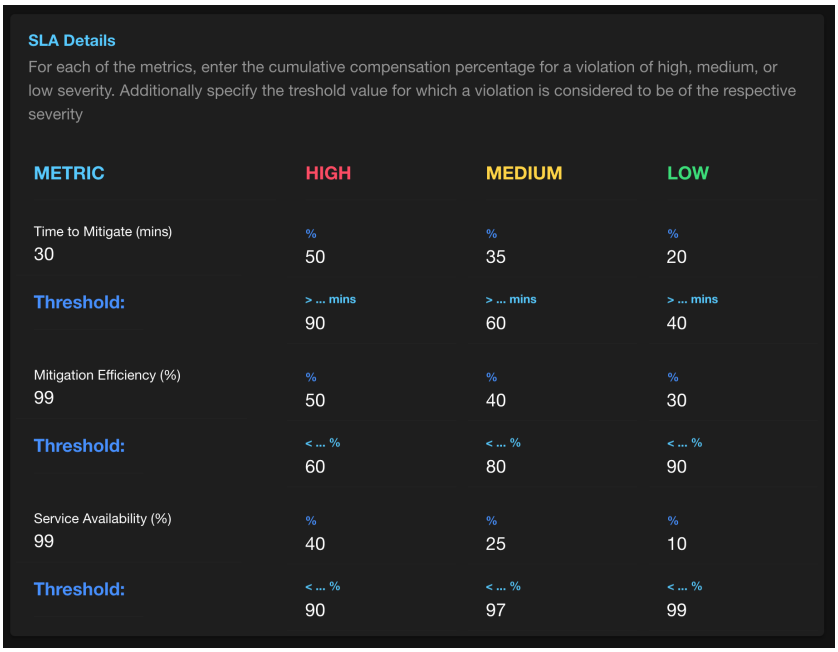
To purchase a protection service, the MetaMask[5] browser extension is required which acts as an Ethereum wallet and gateway to interact with SCs. MetaMask injects the *web3*[6] object, which acts as the Ethereum JavaScript API, making it accessible from the *User Layer*. When looking to purchase a protection service, a prospective customer may navigate to the *Buy Page*, giving him an overview of the available protection services, as shown in Figure 5.2. To inspect the SLA Details of a given service, a user may click the *show SLA details* button, providing him with a detailed view of the SLA details, including compensation percentages as well as threshold values for each of the three performance metrics as seen in Figure 5.2.

While the golden stars represent the average rating score of all ratings on a scale from zero to ten, a user may click the *show all ratings* button to have an in-depth look at all customer ratings of that particular service as shown in Figure 5.3. Upon clicking the green *purchase button* which includes the Ether price, the user is prompted to select a monitoring solution out of an available selection. While not within the scope of this thesis, it is assumed that such a monitor is in charge of monitoring and reporting possible violations of the SLA contract to be created. Upon selecting a monitoring solution, the MetaMask extension pops up, prompting the user to confirm the purchase. After a successful transaction, an SLA SC is created, the details of which shall be discussed in section 5.2.

[5]https://metamask.io/
[6]https://github.com/ethereum/web3.js/



(a) Specification of general information



(b) Specification of SLA details

Figure 5.1: User Interface to upload a service

5.1.3 Ratings Upload

A customer may rate a service based on the following criteria:

1. **Accuracy**: Indicator of the relationship between the service provider's claims and the subjective observed overall performance of the service.
2. **Usability**: Measures the service's ease of use regarding set up, maintenance and user interaction.
3. **Pricing**: Indicates the service's price-performance ratio.
4. **Support**: Measure of the quality of customer support in scenarios such as that of technical difficulties in regards to the service.

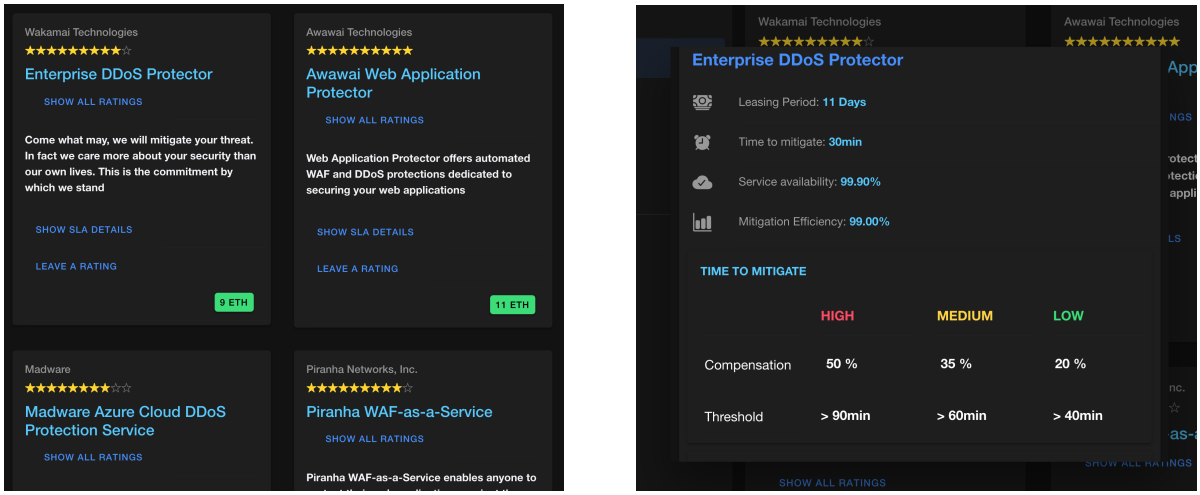
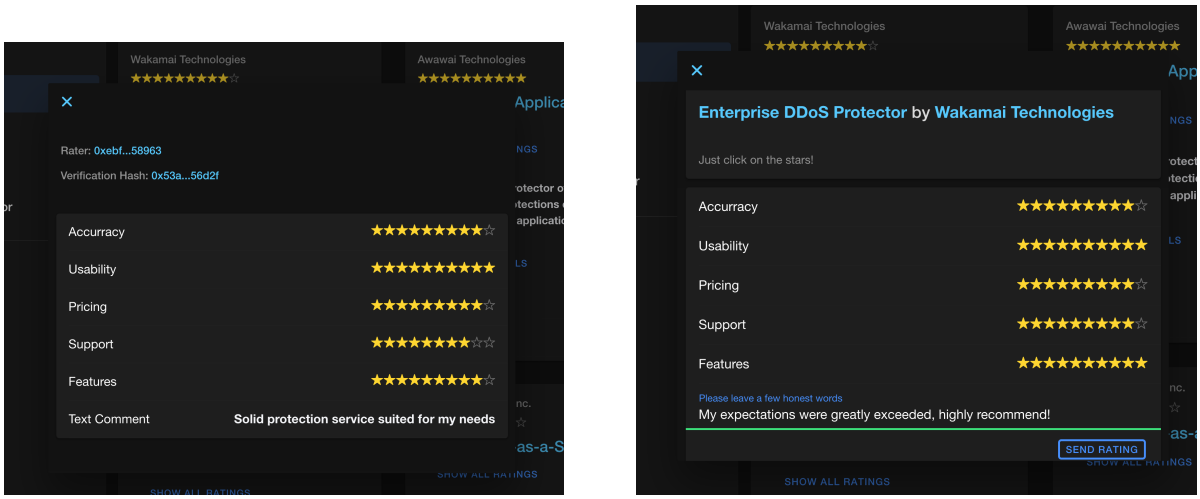


Figure 5.2: Overview of available services and SLA details of a particular service

5. **Features**: Assesses the service's level of sophistication and variety of delivered technical features

Rating scores range from zero to ten, ten being the best possible rating score. Additionally, a user is required to leave a comment about the service in plain text. To leave a rating for a service, a customer may click the *leave a rating* button of a particular service in the *Buy Page*, after which he is presented with a view as in 5.3. In case the user is eligible to leave a rating, his rating is submitted, otherwise the rating is rejected and the user is prompted an appropriate error message.



(a) View to inspect all ratings of a service

(b) View to leave a rating for a particular service

Figure 5.3: Inspecting ratings of and leaving ratings for a service in `Kirti`

Note that each rating object which is stored in the *IPFS Data Layer* includes the hash of the transaction which records the rating in the blockchain. As such, there exists the possibility to verify each rating by simply clicking on the *verification hash* as seen on

the left in Figure 5.3. The transaction details are then queried using *web3*'s `getTransactionReceipt()`, using the recorded transaction hash as input parameter and a popup displaying the transction details is displayed to the user.

5.2 Blockchain Layer

Connecting to the Blockchain Layer is achieved through the use of *web3js* , which acts as the Javascript API for the Ethereum JSON-RPC, thus providing all necessary functionality to generate and interact with SCs. The SC were developed using *Truffle* [50] and *Ganache* [51], which are popular development tools to compile, deploy and test SCs on a locally running blockchain. They are a great help to Ethereum developers as they abstract the intricacies of interacting with the blockchain, while allowing developers to recompile and deploy contracts in a local test environment within seconds.

Two kinds of SCs exist within `Kirti`, both of which are implemented in *Solidity*, which was mentioned in Section 2.2.1. The first, *Kirti.sol* corresponds to both *Data Registration SC* and *SLA Factory* as depicted in Figure 4.1. *SLA.sol* on the other hand corresponds to the *SLA SC* of the same figure. Both shall be discussed in the following section.

5.2.1 Data Registration Smart Contract

The role of the SC encoded in *Kirti.sol* is twofold. On one hand it permanently records all major events of `Kirti` in the blockchain, on the other hand it acts as a factory for SLA SCs as defined by *SLA.sol*. It thus corresponds to both *Data Registration SC* and *SLA Factory* of Figure 4.1. The main *events* recorded in the blockchain are `ProviderRegistered, ServiceRegistered` and `CustomerRatingRegistered`. These *events* are emitted upon evoking the respective registration functions `registerProvider()`, `registerService()` and `registerCustomerRating()` which are evoked from the *IPFS Data Layer*. Invocation access for these registration functions is limited by the `restricted()` modifier. Solidity natively provides the functionality to revoke a call to a contract's function via the `require()` variable, which reverts if the defined condition is not met. In our case, only the address which created the contract may invokes its functions.

5.2.2 SLA Smart Contract

SLA.sol defines the SC in charge of handling the enforcement of an SLA. Note that in contrast to *Kirti.sol*, of which only one contract instance shall be deployed, a new instance of *SLA.sol* is created for each purchase of a protection service. *SLA.sol* is then instantiated with the corresponding parameters of the underlying SLA agreement. While Table 5.1 gives an overview of its functions, the most prominent ones shall be discussed in more depth in the following section. Table 5.1 gives an insight into the functions of *SLA.sol*, which shall be discussed in the following section.

Table 5.1: Functions of *SLA.sol* by their role and intended caller in the context of `Kirti`

Identifier	**Called by**	**Role**
`init()`	*Kirti.sol*	initialize SC
`initOraclizeCallback()`	internal	set up automatic contract termination
`oraclize_query()`	internal	invokes *ORACLIZE_API* with validity period
`__callback()`	*Provable Oracle*	initiates contract termination
`payForService()`	Customer	activates SLA upon receiving funds
`reportViolationFromAPI()`	*Monitor*	lets specified monitor report a violation
`recoverAddress()`	internal	verifies identity of violation reporter
`reportViolation()`	internal	stores violation internally
`terminate()`	`__callback()`	self-destructs contract
`compensateCustomer()`	internal	transfers compensation to customer
`updateCompensation()`	*Kirti.sol*	recalculates compensation
`getViolations()`	*API Layer*	emits `LogViolations` *event*
`getState()`	*API Layer*	emits `LogState` *event*
`getValidityPeriod()`	*API Layer*	emits `LogValidity Period` *event*

Function `init()` marks the entrypoint of *SLA.sol* and is invoked by *Kirti.sol* upon which the contract is initialized with the correct parameters of the underlying SLA agreement. It is necessary that it be only callable once to ensure a correct contract termination. It should be mentioned that `init()` sets a variable `monitorAddress`, to ensure that a violation may only be reported by the the defined monitor. A discussion of the related function `reportViolationFromAPI()` is withheld here, but follows in section 5.4

After instantiating `currentSLA`, `init()` invokes `initOraclizedCallback()`, which makes a call to `oraclize_query()`, which in turn calls the *ORACLIZE_API* with the SLA's validity specified in seconds. In fact, *SLA.sol* inherits from *usingOraclize* imported from *oraclizeAPI-0.5.sol*, which exposes the *ORACLIZE_API*. Oraclize [52] (now called Provable) is an oracle service for multiple blockchains including Ethereum. The motivation behind oracle services is that SCs of today's blockchain can not fetch real world data such as exchange rate feeds or weather information. This need is filled by oracle services which provide such data to a SC. Provable allows clients to make calls to various resources such as IPFS, WolframAlpha or any URL as to be specified as input parameters to `oraclize_query()`. Provable then invokes `__callback()` with the result of the query. It is also possible to simply specify an empty URL and the amount of seconds after which `__callback()` shall be invoked, as in our case. This provides a clean solution to automatically terminate the contract as per the specified validity period of the underlying SLA, without having to periodically invoke the contract to check whether it should terminate. Thus, when `__callback()` is called, the logic to terminate the contract is executed, which

triggers a chain of contract function as illustrated in the following:

1. `terminate()` initiates the termination logic.
2. `updateCompensation()` calculates the current customer compensation based on the reported violations and the respective. compensation percentages for the type and severity of each violation
3. `compensateCustomer()` transfers the calculated compensation to the customer's address.
4. `selfdestruct()` removes the contract's code from the blockchain and transfers the remaining funds to the provider's address. Note that a `selfdestruct()` operation does not cost *gas*, but instead gives a refund of 24000 *gas* [21], thus making it more econonomical than transferring funds via `address(this).sendBalance()`.

5.3 IPFS Data Layer

Data in `Kirti` is stored using OrbitDB which has been discussed in Section 2.3.1. It has been mentioned that OrbitDB does not follow the traditional relational model common to most database systems, but provides five kinds of different data base types. OrbitDB allows for the creation of nested databases and as such one can think of `singleServiceRatingDb` as a nested database within `servicesDb` as seen in Table 5.2, which gives an overview of the kinds of database implementations which were used in the prototype implementation.

Table 5.2: Database implementations used in `Kirti`

Name	**Type**	**Role**
`servicesDb`	*docstore*	Stores all uploaded services
`singleServiceRatingDb`	*eventlog*	Stores the ratings of a particular service
`ratingsKeyVal`	*keyvalue*	Maps a service's name to its `singleServiceRatingDb`
`customersDb`	*docstore*	Stores all customers and their services

5.3.1 Rating Upload

To illustrate how data storage using OrbitDB is implemented, it suffices to give the example of how a user rating is stored, as the concepts are very similar for each of the database types native to OrbitDB. Firstly, we have to sure that only a single instance of OrbitDB is available throughout the codebase as demonstrated in Listing 5.1. As each

database within an OrbitDB instance is trivially available to only that instance, it has to be ensured that only one such instance of OrbitDB is instantiated. It is worthy to mention that OrbitDB does in fact allow to replicate databases across peers, however this is not directly relevant for the implementation at hand. During development of the RESTful API an issue with the global instance arose however. As the JavaScript code is executed both client-side as well as server-side, there is no way to ensure that both browser and server access the same globally defined variable. This ultimately lead to the design decision to access the global OrbitDB instance only server-side, while client-side the *API Layer* is used to connect to the *IPFS Data Layer*.

```
1  const IPFS = require("ipfs")
2  const OrbitDB = require("orbit-db")
3
4  class OrbitDBSingleton {
5    constructor() {}
6
7    async getInstance() {
8      if (!global.orbitdb) {
9        const node = await IPFS.create()
10       const orbitdb = await OrbitDB.createInstance(node)
11       global.orbitdb = orbitdb
12     }
13
14     return global.orbitdb
15   }
16 }
17 module.exports = OrbitDBSingleton
```

Listing 5.1: *OrbitDBSingle.js*, instantiation of a single global OrbitDB instance

While databases in the context of OrbitDB can be replicated amongst peers via IPFS, note that we restrict write access to only the local instance. When calling `addRatingForService()` in line 1 of Listing 5.2, we first invoke `fetchTxHashFromRegistration()`, which returns the transaction hash of the transaction corresponding to the invocation of `registerCustomerRating()` of *Kirti.sol*. This transaction hash is then added to the rating as reference before storing it in the *singleServiceRatingDb* as seen in 5.2. The procedure for storing protection services and customers is analogous. A particularity of OrbitDB is that each time one wishes to read the latest data from a given database, on first has to loads its contents into memory as intendend by `this.loadDb()` in line 16 of Listing 5.2.

```
1    async addRatingForService(rating) {
2      const registrationHash = await this.fetchTxHashFromRegistration([
3        rating.customerAddress,
4        rating.serviceName,
5        [
6          rating.ratingScore.accurracy,
7          rating.ratingScore.usability,
8          rating.ratingScore.pricing,
9          rating.ratingScore.support,
10         rating.ratingScore.features,
11         rating.ratingScore.ratingText,
12       ],
13     ])
```

```
    rating.registrationHash = registrationHash
    await this.writeToDb(rating.customerAddress, rating)
    await this.loadDb()
    return registrationHash
  }
```

Listing 5.2: excerpt from *RatingLib.js*, function to add a rating for a given service

5.4 API Layer

`Kirti`'s RESTful API is implemented using Express[7], a web framework for Node.js[8] which allows for the creation of APIs. The *API Layer* is used for all interactions between the *User Layer* and the *IPFS Data Layer* which gives rise to the necessity to define endpoints which should only be accessible internally and not be exposed to the public. An overview of the public endpoints is given in Table 5.4, while the private endpoints are listed in Table 5.4

Table 5.3: Public endpoints of `Kirti`'s RESTful API

Endpoint	**Method**	**Role**
`/contracts/:address/violation`	**POST**	Report a new violation
`/contracts/:address/violations`	**GET**	Fetch all violations of a contract
`/contracts/:address/state`	**GET**	Fetch the current state of a contract
`/contracts/:address/validity`	**GET**	Fetch validity period of a contract
`/contracts/:address/thresholds`	**GET**	Fetch violation thresholds of a contract
`/orbitdb/services/all`	**GET**	Fetch all currently available services
`/orbitdb/services/add`	**POST**	Upload a new service
`/orbitdb/ratings/service/:name`	**GET**	Fetch all ratings of a particular service
`/orbitdb/customers/all`	**GET**	Fetch all customers
`/orbitdb/customers/customer/:name`	**GET**	Fetch purchases of a customer
`/orbitdb/monitors/add`	**POST**	Add a new monitor
`/orbitdb/monitors/all`	**GET**	Fetch all monitors

The public endpoints beginning with `/contracts/` are primarily targeted to facilitate the reporting of violations by monitors. As the interaction with Ethereum SC functions is a nontrivial affair, the need arouse to allow monitors to report violations in a more simple manner. However this quickly lead to the following issue: How can the identity

[7] https://expressjs.com/
[8] https://nodejs.org/

of the caller of `/contracts/:address/violation` be assessed? In other words, how can it be ensured, that only the specified monitor may report a violation to the SC? The answer lies in Ethereum's elliptic curve cryptography. Solidity natively provides the function `ecrecover()`[9], which takes as input a hashed message and an elliptic curve signature of that message divided into chunks and returns the Ethereum address which created the signature. When reporting a violation, a monitor would thus send a request to `/contracts/:address/violation` including a message and the corresponding signature. The *API Layer* then correctly encodes both message and signature and thus invokes `reportViolationFromAPI()` of *SLA.sol* as shown in Listing 5.3 together with the reported violation. To prevent replay attacks, in which a malicious party gets hold of the signature which a monitor used to report a violation, it is required that each message hash (and thus each signature) be unique, see line 10 of Listing 5.3 for reference.

```
1     function reportViolationFromAPI(
2         bytes32 msgHash,
3         uint8 v,
4         bytes32 r,
5         bytes32 s,
6         Violation memory violation
7     ) public requireActivated {
8         address recoveredAddress = ecrecover(msgHash, v, r, s);
9         require(currentSla.monitorAddress == recoveredAddress);
10        require(uniqueMsgHashes[msgHash] == 0);
11        uniqueMsgHashes[msgHash] = 1;
12        reportViolation(violation);
13    }
```

Listing 5.3: excerpt from *SLA.sol*, showcasing monitor address verification

In this way we can ensure at the SC level, that a successful caller of the `/contracts/:address/violation` endpoint is indeed the verified monitor. Thus even if one is granted full access to the API, one can not successfully report a violation unless one has access to the private key of the specified monitor address. In order to correctly determine what determines a violation for an SLA SC, a monitor may call `/contracts/:address/thresholds`, which may return a response as shown in Listing 6.1. Values *high, medium* and *low* mark the threshold values for which a violation is considered to be of a specific severity. A monitor may thus compare the violation thresholds with his own monitoring measurements and thus assess the correct violation severity upon reporting a violation. The remaining endpoints beginning with `/contracts/` are targeted to be consumed by the *User Layer* to display the current state of an SLA to a customer.

```
1  {
2      "Time to Mitigate": {
3          "HIGH": "more than 90mins",
4          "MEDIUM": "more than 60mins",
5          "LOW": "more than 40mins"
6      },
7      "Mitigation Efficiency": {
8          "HIGH": "less than 60%",
9          "MEDIUM": "less than 80%",
10         "LOW": "less than 90%"
```

[9]https://solidity.readthedocs.io/en/v0.7.0/units-and-global-variables.html

```
    },
    "Service Availability": {
        "HIGH": "less than 95%",
        "MEDIUM": "less than 97",
        "LOW": "less than 99%"
    }
}
```

Listing 5.4: Examplary response from `/contracts/:address/thresholds`, marking threshold values of violation severities

Public API endpoints beginning with `/orbitdb/` are targeted at external parties including cybersecurity recommendation systems such as *MENTOR* [7]. While Table 5.4 is mostly self-explanatory, it should be mentioned that a monitoring solution can be added via the `/orbitdb/monitors/add` endpoint.

Table 5.4: Private endpoints of `Kirti`'s RESTful API

Endpoint	**Method**	**Role**
`/contracts/pay/:address/`	**POST**	Pays for contract via web3.js
`/orbitdb/ratings/add`	**POST**	Adds a new rating for a service
`/orbitdb/customers/service/add`	**POST**	Verifies a purchase for customer

The private API endpoints in `Kirti` are consumed by the *User Layer* and *IPFS Data Layer*. The first endpoint listed in Table 5.4 is used to ensure reliability of successful payments to a SC in a non-production environment and is merely listed here for sake of completeness. While the *User Layer* of `Kirti` needs to provide a reliable way of handling payments to an SLA SC, MetaMask cannot satisfy this reliabilty requirement in a non-production environment and as such `/contracts/pay/:address/` provides a way to ensure a reliable execution of Ethereum transactions via web3.js in case a MetaMask transaction fails.

While `/orbitdb/ratings/add` is not required to be a private endpoint, it is not exposed publicly. This is due to the fact that a customer may facilely submit a rating through the front-end and as such is unlikely to be willing to submit a rating through the API. The `/contracts/customers/service/add` is the only security-related endpoint which necessarily be inaccessible publicly. It is invoked by the *User Layer* upon the successful purchase of a service by a customer. The customer's purchased is then stored via the *IPFS Data Layer* and recorded in the blockchain. In the case said customer wants to leave a rating for a service, the *Rating Eligibility Verifier* uses this data to validate the customer's permission to rate this particular service. It is thus necessary that no external party temper with the data denoting which customer has purchased which protection service.

Chapter 6

Evaluation

In the chapter at hand, firstly a case study illustrating the fitness for purpose of `Kirti` is presented. Next up, the costs of the relevant SCs during deployment and runtime are discussed in Section 6.2, while Section 6.3 discusses the limitations of the solution in regards to true decentralization. Section 6.4 concludes the chapter by addressing the threat of rating fraud in the context of `Kirti`'s reputation system.

6.1 Case Study

To showcase the functionality of `Kirti` in a real-world setting its three major flows shall be demonstrated in this section. A user of `Kirti` is either a service provider, a customer, a monitor or an external party interested in reputation data of cybersecurity services.

To begin with, monitoring solution *ArgusEyed* is interested in acting as a monitoring solution in `Kirti`. He thus makes a POST request to the `/orbitdb/monitors/add` endpoint of its RESTful API as exemplified in Listing 6.1. Upon a successful registration of his details in the *IPFS Data Layer* he is thus informed and may now be selected as the monitoring solutions for a deployed protection service.

```
{
    "name": "ArgusEyed",
    "address": "0x9d8d840d00aa17e3f9adf03421a2b4dd43d06c3c"
}
```

Listing 6.1: Examplary request body as to be sent to `/orbitdb/monitors/add`

Next, assume provider *Piranha Networks, Inc.* wants to make his protection service *Piranha WAF-as-a-Service* available to customers via `Kirti`. Thus he navigates to the appropriate section in the front end to enter his service's general details as well as its SLA specification, which takes him no more than two minutes. Upon confirming the service upload, he is informed by a popup that his service upload was successful and is displayed the hash of the transaction which registered his service upload with `registerService()` of *Kirti.sol*, namely `0xccca4...eff56b7`.

Now let's suppose that a certain customer is interested in purchasing a web application firewall to secure his web application. He thus navigates to the overview of available protection services in the front end and after a short inspection and comparison of potential solution, he opts to purchase *Piranha WAF-as-a-Service* as the service matches his technical requirements, offers a generous SLA compensation and is well rated by fellow customers. The fact that he can verify each review on the Ethereum blockchain further increases his trust in the validity of customer ratings. After confirming the purchase, the front end displays a popup asking the customer to select a monitoring service which is in charge of surveilling the protection service and reporting violations to the SLA SC. After selecting *ArgusEyed* as the monitor, a popup shows up, stating that an SLA SC as encoded by *SLA.sol* has been deployed at address `0x545d8...4D91FCAc`. Next, MetaMask pops up, with the transaction details regarding the newly created contract's address as well its price already filled out. The user now only has to confirm the transaction. In case the transaction was successful, the user is thus informed and may also verify the transaction details. The user may now navigate to the *My Services* section of the front end's dashboard to inspect the current state of his newly purchased service as demonstrated in Figure 6.1. Of special importance is the field *current compensation* which displays the up-to-date value of the SLA's compensation as calculated by the number and severity of reported violations.

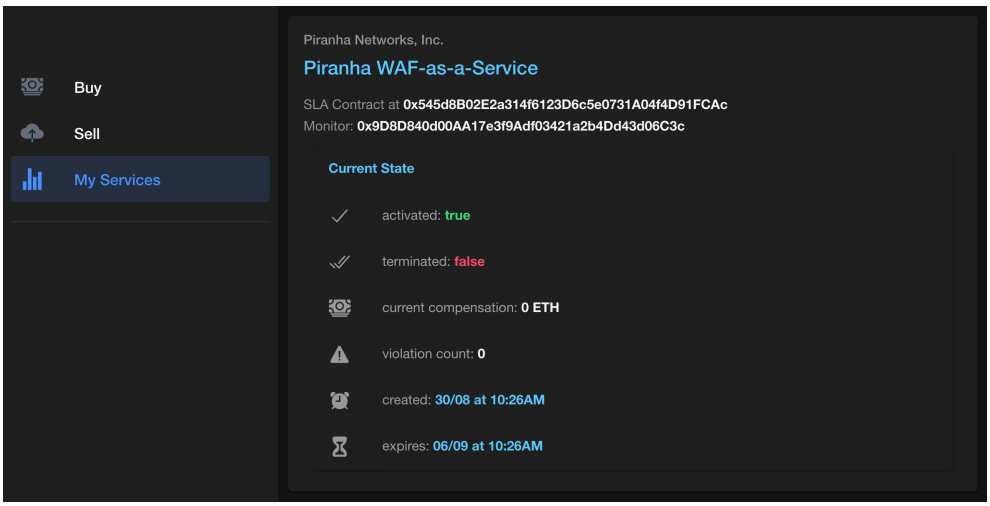


Figure 6.1: View displaying the state of a user's services

Now that the SLA SC is activated through the user's payment, it is the monitor's obligation to surveil the deployed protection service. Suppose now that a DDoS attack on the customer's web application takes place. Luckily *Piranha WAF-as-a-Service* includes DDoS protection and while the attack is successfully mitigated, the monitor notices that the mitigation took 62 minutes instead of the promised 30 minutes as specified in the SLA agreement. The monitor may now call the `/contracts/0x545d8...4D91FCAc/thresholds` endpoint, to receive information about the SLA's threshold values regarding the violation severities (see Listing .4]). Using the obtained data, he concludes that he should report a violation of severity *medium* of the *Time to Mitigate* metric. Note that to make a successful request to `contracts/:address/violation`, a monitor has to include a message and signature to verify his identity (see Section 5.4). A monitor would thus generate a unique message and signature using Ethereum's elliptic curve digital signature algorithm

(ECDSA)[21] using a script such as that of Listing 6.1.

```
const ethUtils = require("ethereumjs-util")
const { v4: uuidv4 } = require("uuid")
const secret = require("./secret")

const message = uuidv4()
const privateKey = secret.privateKey

const signMessage = async (message, privateKey) => {
  const messageBuffer: Buffer = ethUtils.hashPersonalMessage(
    ethUtils.toBuffer(message)
  )
  const signature: {
    v: string
    r: string
    s: string
  } = ethUtils.ecsign(messageBuffer, Buffer.from(privateKey, "hex"))

  return ethUtils.toRpcSig(signature.v, signature.r, signature.s)
}
```

Listing 6.2: signing.ts, script to generate message and signature using ethereumjs-util

Note that `uuidv4()`[1] returns a universally unique identifier, which ensures that generated signatures are unique. The monitor now sends an HTTP POST request to `/contracts/:address/violation` including the obtained values for *message* and *signature*. Note that the values of 0 for *violationType* and 1 for *violationSeverity* correspond to *Time to Mitigate* and *medium* respectively. A monitor is informed about the encoding scheme regarding both *violationType* and *violationSeverity* in the response body of a call to `/orbitdb/monitors/add`.

```
{
    "message": "b9f9e1ee-55e9-4df7-a83a-d5156813e192",
    "signature": "0x4184f...24fef00",
    "violationType": 0,
    "violationSeverity": 1
}
```

Listing 6.3: Examplary truncated request body to `/contracts/:address/violation`, reporting a *medium* severity violation of metric *Time to Mitigate*.

Meanwhile the customer sees in the front end, that a violation has occured and the current compensation has been updated to an amount of 2.45 ETH (35% compensation at a price of 7 ETH). After a period of 7 days the service and thus its associated SLA SC expires, the contract terminates by refunding 2.45 ETH to the customer and releases the remaining funds to the provider.

[1] https://www.npmjs.com/package/uuidv4

6.2 Costs

To interact with the SCs, they first have to be deployed to one of the Ethereum networks. Executing operations on the Ethereum VM is associated with costs measured in *gas* which carry an associated *gas* price in *ether*. Deploying and interacting with Ethereum SCs is thus associated with real-world costs which shall be analyzed in this section for the setting of `Kirti`.

Of the two SCs, *Kirti.sol* has to be deployed only once while a new instance of *SLA.sol* has to be deployed for each purchased service. The deployment of *Kirti.sol* is associated with a cost of *3'383'264 gas*, while the deployment of *SLA.sol* amounts to a cost of *2'397'165 gas*. The *gas* costs of the publicly callable functions of *SLA.sol* are further displayed in Table 6.2. The specification of equivalent fiat currency values has been purposefully omitted due to the extreme fluctuations of *gas* prices. Note that at the start of writing this thesis (March 15th, 2020), *ether* was priced at *111*$[2] and the average *gas* price was around *17 GWei*[3]. At present day (September 1st, 2020), *ether* is valued at *435*$[4], while average *gas* prices have spiked to an astronomical *236 Gwei*[5]. Thus, executing any operation on the Ethereum network such as a contract deployment has become more expensive by a factor of over *50* over the period of less than half a year. It should be pointed out that from the outset, both SCs were not implemented with the intention of being maximally deployment cost efficient. As such, SCs providing similar functionality could be significantly more cost efficient.

Table 6.1: Publicly invocable functions of *SLA.sol* sorted by *gas* cost

Identifier	**Called ...**	***Gas* Cost**
`init()`	once	473717
`initOraclizeCallback()`	once	123659
`reportViolationFromAPI()`	0...n times	116283
`payForService()`	once	86852
`getState()`	1...n times	29599
`getValidityPeriod()`	1...n times	25815
`updateCompensation()`	1...n times	24471
`getViolations()`	1...n times	23693
`terminate()`	once	16352

[2]https://etherscan.io/chart/etherprice
[3]https://etherscan.io/chart/gasprice
[4]https://etherscan.io/chart/etherprice
[5]https://etherscan.io/chart/gasprice

6.3 Decentralization and Limitations

From the outset, `Kirti` has been designed in a decentralized manner and while it combines distributed technologies such as the Ethereum blockchain and IPFS, certain limitations concerning true decentralization arise. While `Kirti`'s architecture can be deemed decentralized, in its current form it can only be operated in a politically centralized manner, i.e by single operator acting as the system's owner. This limitation arises due to various technicalities of its implementation. As such, functions of *Kirti.sol*, which records major events of the system in the blockchain and handle the creation of new SLA SCs, are only callable by the contract owner. Along the same lines, some of the *API Layer*'s endpoints are required to be only accessible to the entity which owns the system. In the current state of `Kirti`, the system's owner pays for the deployment of SLA contracts, but deployment fees could trivially be allocated to customers and/or providers.

Besides these two types of actors, monitors form an integral part of `Kirti`, as they are in charge of reporting violations to SLA terms. It is essential that these monitors can be trusted to correctly report violations, implying that they must be capable of adequately surveilling protection services. This presents a challenge to the status quo of the cybersecurity market and raises the question of availability of solution providers who would act as monitoring solution in a setting as that of `Kirti`. Currently there exists no compensation scheme for monitoring solutions, nevertheless it seems reasonable to remunerate monitors according to a proportionate amount of the total service cost.

6.4 A Word on Rating Fraud

It should be mentioned that any reputation system which is based on subjective measures, such as user ratings, cannot fully mitigate the attacks of *Bad Mouthing* and *Ballot stuffing* - after all, there is by definition no way to quantify the correctness of a subjective sentiment in an objective manner. However, by ensuring that only verified customers are entitled to leave ratings, a reputation system can increase its resilience against rating fraud. Rater verification protects against *Bad Mouthing*, since the benefit incurred by rating a competitor's service negatively is likely outweighed by the cost of having to purchase said competitor's service in the first place. Additionally, `Kirti` ensures that a customer may only leave a single rating for each time he purchased a service. Otherwise the reputation system would be prone to rating attacks, as a customer could purchase a service only once and later on leave multiple malicious ratings.

It is worth noting that rater verification is not as effective against *Ballot stuffing* attacks however. In `Kirti`, a service provider could potentially purchase his own service and afterwards rate it most favorably. In the spirit of comprehensive decentralisation, `Kirti` does not enforce the creation of user accounts, as such an Ethereum address is sufficient to interact with the system. This implies that the cost of creating a new identity is trivial. Thus, a malicious service provider interested in boosting his own reputation can generate a number of different identities at a low cost.

Chapter 7

Conclusions and Future Work

The intent of this thesis was the design and prototypical implementation of a blockchain based reputation system for the cybersecurity market including automated SLA enforcement using SCs. While reputation systems in the cybersecurity market are not unheard of, they are generally implemented using a centralized architecture, while the proposed solution `Kirti` is implemented in a decentralized way, relying on distributed open source technologies such as the Ethereum blockchain and IPFS. SLA enforcement using SCs however is indeed a novelty which has only been mentioned sparsely in the academia and is still waiting to be applied to a real-world commercial setting. Besides offering a market place which allows cybersecurity providers to offer their protection services and customers to buy and rate these services, `Kirti` also provides an API which allows external parties to obtain reputation data of services as rated by customers of the system.

While industry players may still be hesitant to rely on public blockchain technology for mission critical systems, the cybersecurity market might shun the high costs of automated SLA enforcement using SCs. While this thesis has not conducted a cost-benefit analysis for the involved parties of an SLA agreement in a cybersecurity setting, its results have shown that the deployment of a SC which enforces the SLA agreement of a cybersecurity protection service can prove to be associated with high costs given a bullish Ethereum market.

Future work should investigate how the deployment and operation costs of the SCs can be reduced. Additionally, a business model for platform operators as well as incentives for monitoring solutions shall be identified. Further the technicalities of how a customer obtains a purchased protection service as well as the details regarding its deployment shall be investigated. As both service providers and customers need to have implicit trust in monitoring solutions and the correctness of their reported SLA violations, future work should investigate how this trust requirement can be mitigated. As such the feasibility of introducing a council of monitoring solutions as proposed by Zhou et al. [43] should be examined.

Bibliography

[1] Richard Dennis and Gareth Owen. Rep on the block: A next generation reputation system based on the blockchain. In *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 131–138. IEEE, 2015.

[2] Sulin Ba and Paul A Pavlou. Evidence of the effect of trust building technology in electronic markets: Price premiums and buyer behavior. *MIS quarterly*, pages 243–268, 2002.

[3] Gary E Bolton, Elena Katok, and Axel Ockenfels. How effective are electronic reputation mechanisms? an experimental investigation. *Management science*, 50(11):1587–1602, 2004.

[4] Mike Sharples and John Domingue. The blockchain and kudos: A distributed system for educational record, reputation and reward. In *European conference on technology enhanced learning*, pages 490–496. Springer, 2016.

[5] Matthew Buechler, Manosai Eerabathini, Christopher Hockenbrocht, and Defu Wan. Decentralized reputation system for transaction networks. *Technical report, University of Pennsylvania, Tech. Rep.*, 2015.

[6] Yuanfeng Cai and Dan Zhu. Fraud detections for online businesses: a perspective from blockchain technology. *Financial Innovation*, 2(1):20, 2016.

[7] M. F. Franco, B. Rodrigues, and B. Stiller. MENTOR: The Design and Evaluation of a Protection Services Recommender System. In *15th International Conference on Network and Service Management (CNSM 2019)*, pages 1–7, Halifax, Canada, October 2019.

[8] Martín Varela, Patrick Zwickl, Peter Reichl, Min Xie, and Henning Schulzrinne. From service level agreements (sla) to experience level agreements (ela): The challenges of selling qoe to the user. In *2015 IEEE International Conference on Communication Workshop (ICCW)*, pages 1741–1746. IEEE, 2015.

[9] SLA Management Handbook (2004) TMF G045. The Open Group.

[10] S Morgan. Official annual cybercrime report. Herjavec Group, 2019.

[11] Cybertango - The Cybersecurity Directory. https://www.cybertango.io/cybersecurity-vendors/. Last visit July 1, 2020.

[12] Alexander Schaub, Rémi Bazin, Omar Hasan, and Lionel Brunie. A trustless privacy-preserving reputation system. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 398–411. Springer, 2016.

[13] Emmanuelle Anceaume, Gilles Guette, Paul Lajoie-Mazenc, Nicolas Prigent, and V Viet Triem Tong. A privacy preserving distributed reputation mechanism. In *2013 IEEE International Conference on Communications (ICC)*, pages 1951–1956. IEEE, 2013.

[14] Richard Dennis and Gareth Owenson. Rep on the roll: a peer to peer reputation system based on a rolling blockchain. *International Journal for Digital Society*, 7(1):1123–1134, 2016.

[15] Chrysanthos Dellarocas. Immunizing online reputation reporting systems against unfair ratings and discriminatory behavior. In *Proceedings of the 2nd ACM conference on Electronic commerce*, pages 150–157, 2000.

[16] Linlin Wu and Rajkumar Buyya. Service level agreement (sla) in utility computing systems. In *Performance and dependability in service computing: Concepts, techniques and research directions*, pages 1–25. IGI Global, 2012.

[17] Michael Crosby, Pradan Pattanayak, Sanjeev Verma, Vignesh Kalyanaraman, et al. Blockchain technology: Beyond bitcoin. *Applied Innovation*, 2(6-10):71, 2016.

[18] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[19] Maher Alharby and Aad Van Moorsel. Blockchain-based smart contracts: A systematic mapping study. *arXiv preprint arXiv:1710.06372*, 2017.

[20] Zibin Zheng, Shaoan Xie, Hongning Dai, Xiangping Chen, and Huaimin Wang. An overview of blockchain technology: Architecture, consensus, and future trends. In *2017 IEEE international congress on big data (BigData congress)*, pages 557–564. IEEE, 2017.

[21] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[22] Andreas M Antonopoulos and Gavin Wood. *Mastering ethereum: building smart contracts and dapps*. O'reilly Media, 2018.

[23] Solidity - 0.6.10 Documentation. https://solidity.readthedocs.io/en/v0.6.10/. Last visit July 6, 2020.

[24] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 1997.

[25] Konstantinos Christidis and Michael Devetsikiotis. Blockchains and smart contracts for the internet of things. *Ieee Access*, 4:2292–2303, 2016.

[26] Daniel Macrinici, Cristian Cartofeanu, and Shang Gao. Smart contract applications within blockchain technology: A systematic mapping study. *Telematics and Informatics*, 35(8):2337–2354, 2018.

[27] Juan Benet. Ipfs-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561*, 2014.

[28] Kien Nguyen, Thinh Nguyen, and Yevgeniy Kovchegov. A p2p video delivery network (p2p-vdn). In *2009 Proceedings of 18th International Conference on Computer Communications and Networks*, pages 1–7. IEEE, 2009.

[29] IPFS Documentation. https://docs.ipfs.io/concepts/what-is-ipfs. Last visit July 6, 2020.

[30] Protoschool - Interactive tutorials on decentralized web protocols. https://proto.school/#/anatomy-of-a-cid/01. Last visit July 6, 2020.

[31] OrbitDB. https://orbitdb.org/. Last visit July 6, 2020.

[32] OrbitDB - GitHub Repository. https://github.com/orbitdb/orbit-db/. Last visit July 6, 2020.

[33] OrbitDB - GitHub Repository. https://github.com/orbitdb/orbit-db/blob/f2cfaa8d1a10bc1f4d589442308f88bb5e976cb2/GUIDE.md. Last visit July 6, 2020.

[34] IPFS Log - GitHub Repository. https://github.com/orbitdb/ipfs-log. Last visit July 6, 2020.

[35] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.

[36] Sepandar D Kamvar, Mario T Schlosser, and Hector Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. In *Proceedings of the 12th international conference on World Wide Web*, pages 640–651, 2003.

[37] Li Xiong and Ling Liu. Peertrust: Supporting reputation-based trust for peer-to-peer electronic communities. *IEEE transactions on Knowledge and Data Engineering*, 16(7):843–857, 2004.

[38] Runfang Zhou and Kai Hwang. Powertrust: A robust and scalable reputation system for trusted peer-to-peer computing. *IEEE Transactions on parallel and distributed systems*, 18(4):460–473, 2007.

[39] Kuan Lun Huang, Salil S Kanhere, and Wen Hu. A privacy-preserving reputation system for participatory sensing. In *37th Annual IEEE Conference on Local Computer Networks*, pages 10–18. IEEE, 2012.

[40] Davide Carboni. Feedback based reputation on top of the bitcoin blockchain. *arXiv preprint arXiv:1502.01504*, 2015.

[41] Sergi Delgado-Segura, Cristian Tanas, and Jordi Herrera-Joancomartí. Reputation and reward: Two sides of the same bitcoin. *Sensors*, 16(6):776, 2016.

[42] Rafael Brundo Uriarte, Rocco De Nicola, and Kyriakos Kritikos. Towards distributed

sla management with smart contracts and blockchain. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 266–271. IEEE, 2018.

[43] Huan Zhou, Cees de Laat, and Zhiming Zhao. Trustworthy cloud service level agreement enforcement with blockchain based smart contract. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 255–260. IEEE, 2018.

[44] E. J. Scheid B. Rodrigues, L. Z. Granville, and B. Stiller. Enabling dynamic sla compensation using blockchain-based smart contracts. In *IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 53–61, Washington DC, USA, 2019.

[45] Jéferson C Nobre, Lisandro Z Granville, Alexander Clemm, and Alberto Gonzalez Prieto. Decentralized detection of sla violations using p2p technology. In *8th International Conference on Network and Service Management (CNSM)*, pages 100–107, Las Vegas, USA, 2012. IEEE.

[46] Business Software and Services Reviews | G2. https://www.g2.com/search?utf8=%E2%9C%93&query=security. Last visit July 7, 2020.

[47] Microsoft Azure Marketplace. https://azuremarketplace.microsoft.com/en-us/marketplace/apps/category/security. Last visit July 7, 2020.

[48] AWS Marketplace. https://aws.amazon.com/marketplace/b/2649363011?nc2=h_ql_mp_sec. Last visit July 7, 2020.

[49] Eder J Scheid, Manuel Keller, Muriel F Franco, and Burkhard Stiller. Bunker: a blockchain-based trusted vnf package repository. In *International Conference on the Economics of Grids, Clouds, Systems, and Services*, pages 188–196. Springer, 2019.

[50] Truffle | Truffle Suite. https://www.trufflesuite.com/truffle. Last visit August 23, 2020.

[51] Ganache | Truffle Suite. https://www.trufflesuite.com/ganache. Last visit August 23, 2020.

[52] Provable - blockchain oracle service, enabling data-rich smart contracts. https://provable.xyz/. Last visit August 23, 2020.

Abbreviations

API	Application Programming Interface
EOA	Externally owned (Ethereum) Account
EVM	Ethereum Virtual Machine
HTTP	Hypertext Transfer Protocol
URL	Uniform Resource Locator
IPFS	InterPlanetary File System
SC	Smart Contract
SLA	Service Level Agreement
CRDT	Conflict-free replicated data type
VNF	Virtual Network Function

List of Figures

List of Tables

Appendix A

Installation Guidelines

A.1 Setup

1. **Initial Setup**
   (a) in case NVM (Node Version Manager) is not installed, install it by running
   `$ curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.35.3/install.sh | bash`

2. **Clone Repository**
   (a) Clone the GitHub repository by running
   `$ git clone https://github.com/acharl/kirti`
   (b) navigate to the root of the project with `$ cd kirti`

3. **Start Application**
   (a) Make sure that you're using node v10+,
   in case you're running into errors, run `$ nvm use 12`
   (b) Install dependencies with `$ npm install`
   (c) Start local blockchain with `$ npm run truffle`
   (d) In the same directory but a new tab run `$ npm run deployContracts`
   (e) Enable automated contract termination with
   `$ npm run oraclize`
   (f) In the same directory but in a new tab run `$ npm run server`
   (g) In the same directory but in a new tab run `$ npm run start`

4. **Setup MetaMask**
   (a) Install the latest MetaMask extension from
   `https://metamask.io/download.html`
   (b) Import an existing seed phrase or create a new wallet.

(c) In MetaMask's main menu, click on the badge with "Ethereum Main Net" and select "special RPC".

(d) Enter `http://127.0.0.1:9545` as the new RPC-URL.

(e) In the top right corner, click on the round button and next click on "import account".

(f) In the appropriate field enter a private key from one of the Truffle accounts. The tab in which `$ npm run truffle` was executed contains 10 test accounts and their private keys.

(g) Open a chrome browser window at `http://localhost:4200/` and verify that the imported Truffle account in MetaMask is marked as "connected". Optionally reload the page. You should now be seeing the *Buy* page with currently no services up for sale.

A.2 Testing

To test the functionality of the prototype, a reader may refer to the following section. To make calls to the API one may use a client such as the Postman API Client.[1]

1. **Register as Monitor**

   Send a POST request to `localhost:1008/orbitdb/monitors/add` including the contents of `add-monitor.json` as the request body, which can be found in the `/examples` folder at the root of the project. Make sure to replace the value for *address* with an address from Truffle or any valid Ethereum address for which you own the private key.

2. **Upload Service as a Service Provider**

   Send a POST request to `localhost:1008/orbitdb/services/add` including the contents of `add-service.json` as the request body, which can be found in the `/examples` folder at the root of the project.

3. **Buy Protection Service**

   (a) In the *Buy* tab of the application, click on the green button of a protection service to purchase it.

   (b) Confirm the transaction with MetaMask.

   (c) Navigate to the *My Services* to inspect the current state of your service.

4. **Report Violation**

   (a) Execute `$ npm run sign -- privateKey`, where *privateKey* is the private key associated with the monitor address specified in step 1. (Register as Monitor).

[1] https://www.postman.com/product/api-client/

(b) Send a POST request to `localhost:1008/contracts/`*`address`*`/violation` including the contents of `report-violation.json` as the request body, which can be found in the `/examples` folder at the root of the project. Make sure to replace the values for *message* and *signature* with the obtained values from (a). Note that *address* in the request URL has to be replaced with the contract address which can be seen in the *My Services* page.

(c) Navigate to *My Services* in the UI to see the updated state of the service. (Reload page if necessary)

5. **Leave Rating**

   (a) In the UI navigate to the service you purchased earlier in the *Buy* tab and click on *leave rating* button. After confirming the rating you may inspect the new rating via *show all ratings*.

Appendix B

Contents of the CD

1. **Thesis as PDF**
2. **complete source code as ZIP**
3. **slides of intermediate presentation as PDF**