University of Zurich[UZH]

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

SC4CyberInsurance: Automated Cyber-Insurance Contracts

Berni, Noah
Zurich, Switzerland
Student ID: 14-728-166

Supervisor: Muriel Franco, Eder Scheid
Date of Submission: January 6, 2021

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

ifi

Master 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

Durch die ständig wachsende Automatisierung von Arbeitsprozessen und eine immer grösser werdende Interkonnektivität zwischen verschiedenen Unternehmungen, erhöht sich die Abhängigkeit von funktionierenden Systemen und die Auswirkungen, wenn ein solches ausfällt, verschlimmern sich. Cyberkriminelle nutzen Schwachpunkte der Systeme aus und greifen vermehrt Firmen an, die sich auf ihre Technologien verlassen. Da die Unternehmungen ein solches Risiko nicht tragen wollen, steigt die Nachfrage nach Cyberversicherungen. Weil der Cyberversicherungsmarkt erst am Anfang steht, sind noch keine standardisierten Versicherungsprodukte verfügbar. Es wird versucht, dem Kunden ein optimales Angebot bereit zu stellen, indem verschiedene Technologien verwendet werden. Eine der benutzten Technologien ist Blockchain. In dieser Arbeit wird untersucht, inwiefern sich Blockchain als Technologie eignet, um einen Cyberversicherungsvertrag als Smart Contract bereit zu stellen, welcher dann selbstständig die verschiedenen Anfragen des Kunden und des Versicherers entsprechend der vereinbarten Konditionen verarbeitet. Dadurch soll ein vertrauenswürdiges und transparentes Verhältnis zwischen den Parteien entstehen. Im Rahmen dieser Arbeit wird ein entsprechendes Design herausgearbeitet und ein Prototyp implementiert. Durch die Erstellung des Prototypen sollen die Stärken und Schwächen des Designs evaluiert und aufgezeigt werden.

Caused by the permanent growing automation of operating processes and the increasing interconnectivity between different organizations, the dependency to have working systems is rising, and the impact when such a system falls out is exacerbated. Cybercriminals take advantage of the vulnerabilities of the systems and attack organizations that rely on their technologies to an increasing degree. As the companies do not want to bear such a risk, the demand for cyber-insurance increases. Since the cyber-insurance market is still in its infancy, no standardized insurance products are available. Through the usage of different technologies, insurance providers try to create an optimal product for the customers. One of these technologies is blockchain. This work investigates the possibility of deploying a cyber-insurance contract as a smart contract in a blockchain, which can autonomously process the customer's and the insurer's requests according to the agreed conditions. Thereby a trustworthy and transparent relationship between the two parties should arise. Within the scope of the work, an appropriate design will be constructed and a prototype implemented. By creating the prototype, the strengths and weaknesses of the design should be evaluated and illustrated.

Acknowledgments

First of all, I would like to thank my supervisor Muriel Franco for always taking time and providing valuable feedback and support in the process of the thesis. It was always a pleasure to discuss the progress with you. I also want to thank Prof. Dr. Burkhard Stiller for the opportunity to write my master thesis at the Communication Systems Group of the University of Zurich. I am very grateful that I could tackle a topic that interested me so much. Last but not least, I want to thank my family and friends who are supporting me all the time.

Contents

Chapter 1

Introduction

The increasing reliance on advanced technologies and the rising networking under each other improves most organizations and governments' efficiency but creates a big surface for possible cyberattacks [21]. Because of that, cybersecurity becomes more and more important to mitigate the risks of such an attack. However, since both the technologies and the proceeding of cyberattacks continuously change, it is nearly impossible to achieve perfect or even near-perfect cybersecurity protection [6]. Due to the lack of perfect security, some cyberattacks are still successfully executed, leading to different suffering for the concerned organizations, such as direct financial losses, business disruption, or stolen data, which can cause great damage to the organizations. As the number of cyberattacks has been increasing continuously over the past years, predictions state that cybercrime will cost the world 10.5 trillion US$ annually by 2025, up from 3 trillion US$ in 2015, which represents the greatest transfer of economic wealth in history [22]. To reduce the impact of these successful attacks and to let the concerned organizations recover faster and with less loss of money, different cyber-insurance coverage models have been introduced [25]. Around the world, the cyber-insurance market is just at its beginning, and it is still under heavy development. However, cyber-insurance is advancing from a rarely used risk transfer tool to a critical requirement for enterprise risk management [25]. Initial predictions expect a heavy growth in the worldwide spending on cyber-insurance that it will achieve 9 billion US$ by 2020 [18].

1.1 Motivation

As the cyber-insurance market is still in its infancy, different approaches to boost the market are explored by the institutions. While some introduce new business models and mechanisms to gain advantages [5], others try to improve their insurance services by using new technologies [4]. A promising technology in insurance is the usage of blockchain [3] and accompanying to that, the usage of smart contracts. The usage of blockchain alone has been discussed and researched quite many times. For example, a blockchain-based continuous monitoring and processing system for cyber-insurance has been proposed by the authors in [7]. However, the combination of blockchain along with smart contracts is yet

not very well elaborated and the advantages introduced by smart contracts still have to be explored in the context of insurance in general. This work introduces SC4CyberInsurance, a system that creates and deploys smart contracts for cyber-insurance. The smart contracts describe the contract agreements between the insurer and its customers. Thanks to the transparency of the smart contracts and the immutability of the blockchain they are executed on, blockchains and smart contracts can provide a trustworthy and immutable agreement between the cyber-insurers and their customers so that both parties can profit from the advantages of the blockchain [3]. To explore the potential of a smart contract in the cyber-insurance use case is the project's main goal.

1.2 Description of Work

This work focus on the investigation and development of blockchain-based models for the cyber-insurance market. Thus, in an initial step, an extensive literature review is done to define a data model for cyber-insurance. Next, the system SC4CyberInsurance is designed and developed to allow the creation and deployment of smart contracts that describe the contract agreements between the insurer and the customer. Both insurer and customer are allowed to interact with SC4CyberInsurance and update the contract conditions on demand. Also, the payment execution is handled automatically by the smart contract, whether it is possible.

The work is divided into three steps. In the first step, the focus lies in research. The available literature about cyber-insurance, blockchain-based cybersecurity approaches, and the connection between cyber-insurance and blockchain is studied to gain the necessary background. The attributes and characteristics from the customer that are necessary for a cyber-insurance contract are collected and carried together from the related work. It is then defined which of the collected attributes and characteristics are stored in the contract and which are relevant to calculate the premium to pay afterwards.
In a second step, an initial architecture is provided for the proposed system SC4CyberInsurance, which includes data flows, involved actors, and principal components. Besides that, a data structure is defined to store all the previously described attributes and characteristics in a well-defined and extensible way. Then, a smart contract is designed to store and execute the cyber-insurance agreements. The smart contract considers different interactions between the actors involved and it is dynamically created and deployed using the data structure previously created.
In the third step, the prototype SC4CyberInsurance is developed. To prove that SC4CyberInsurance covers all steps of a cyber-insurance contract, different use cases are covered and evaluated. As cyber-insurance is in its infancy and its environment is changing permanently, SC4CyberInsurance is aimed to be extensible to include new information regarding different cyber-insurance models, agreements, and configurations. Finally, SC4CyberInsurance is evaluated considering different use cases to shows evidence of its performance and feasibility.

1.3 Thesis Outline

The rest of this thesis is structured as follows. First, in Chapter 2, the project's theoretical basis is given by providing the necessary background information, including an analysis of the current state of the art. In Chapter 3, different related work, which did research on the same topic as this work, are presented, and lacks in the literature are identified. Next, the solution design of SC4CyberInsurance is described in Chapter 4, while the implementation of the prototype is explained in Chapter 5. Different evaluations on the usefulness and the costs of the implementation are shown in Chapter 6. Finally, in Chapter 7, the thesis's most important aspects are summarized and some conclusions are drawn.

Chapter 2

Background

This Chapter is intended to give the reader a basic understanding of the technologies and concepts used. In Section 2.1 a general introduction to the topic of cyber-insurance is given and important specialities are reviewed. Section 2.2 gives a short overview of blockchains and smart contracts. Finally, the current economic state of cyberattacks and cyber-insurance is discussed and the economic impact of cyberattacks is explored in Section 2.3.

2.1 Cyber-Insurance

The term insurance is generally well known. Simply explained, insurance is a contract in which an insurant, which can be an individual or an entity, receives financial protection or reimbursement against losses from an insurance company. In return, the insurance company periodically receives a premium. By pooling the clients' risks, the premium payment can be made more affordable for the insured [17]. Cyber-insurance is a specific product of such an insurance contract and it is intended to cover damage caused by cyber-criminality. It is mainly aimed for companies, but also for governments or individuals, which want to protect their businesses from cyber risks. The focus of this paper will lay on the cyber-insurance products aimed for companies. A generalized process how such a cyber-insurance contract could be created is shown in the following [2]:

1. Risk Identification
    (a) Asset Identification
    (b) Threat Identification
    (c) Security/Vulnerability Identification
2. Risk Analysis
    (a) Likelihood Determination
    (b) Impact Determination
    (c) Risk Estimation
3. Establish Contract
    (a) Coverage Specification

(b) Premium Estimation
(c) Write and Sign Contract
(d) Claim Handling (optional)

First the main risks of the organization to insure are identified, which includes asset identification, threat identification and the identification of existing vulnerabilities in the security system. For each detected risk the probability that it is exploited and the impact when it is exploited are quantified. Both values are aggregated to the general risk estimation. The step of the risk analysis is quite complex in the cyberspace and will be described more detailed in Section 2.1.1. For now it is assumed, that the risk can be analyzed and assessed in an acceptable manner. After the risk analysis, an organization could also decide to increase its cybersecurity to decrease the probability and the impact of a specific risk exploitation. However, as this paper focuses on insurance it is assumed that the organization will pick the risk transfer option. After the risks are analyzed the customer specifies which risks should be covered. Dependent on the coverage the insurer estimates a premium and makes an offer to the customer. If both parties agree the contract is created and signed by both. During the validity of the contract the customer can make claims in the case an incident occurred. The insurer then has to cover the losses if the specific risk was specified in the contract [2].

As cyber-criminality is a huge term and the variety of cyberattacks is enormous and each attack can cause different effects it has to be described very detailed in the insurance contract what attacks and what costs are covered. As different customers have different requirements, there is no one fits all cyber-insurance product. One customer may want to cover the risks of a distributed denial-of-service (DDoS) attack, while another wants to cover the risk of a data breach and a third one wants to cover the risk of having any ransomware in the software. Since the list of possible attack is quite long and there are very likely yet unknown attack types, it can be very hard for the customer to define an appropriate list of attacks to cover. Because of that, most of the insurers offer the possibility to not only describe what attacks are covered, but mainly to describe which effects and hence which costs are covered. For example, a possible effect of an attack is that a company can not continue to sell its products, because its software is not working anymore, hence its business is interrupted. If the insurance contract covers such costs, the insurer has to pay the not earned money to the customer.
Section 2.1.2 gives an overview of the most typical costs caused by cybercrime and distributes the cost types in different groups. Considering the different attack types and the different cost types, it is also possible to combine the two into a new coverage type. For example, an insurance contract can cover the business interruption in the case of a DDoS attack, but not when it was caused by a ransomware. Each coverage type covers a specific risk and as more risks are covered, as higher the premium should be. Theoretically, each risk has a specific price to be covered and these prices are summed up, maybe with discount, when more than one risk is covered.

The customer and the insurance company therefore try to create a best fitting packet of coverage so that the remaining not covered risks are acceptable and the premium to pay is feasible for the customer. Since the surface of a software is huge and the software is always evolving, it is not possible to have a perfect or near perfect cybersecurity and there will

be always at least a small possibility that a risk is exploited [6]. Hence a possible benefit for risk minimization by using cyber-insurance is always given [2]. Due to that and the increasing number of cyberattacks, cyber-insurance is becoming more and more important over the years. Its volume is estimated to more than double from 3.4 - 4 billion US$ in 2017 to 8 - 9 billion US$ in 2020 [18]. However, the premium spent on cyber-insurance is still only a very small part of the total insurance volume, which exceeds 5 trillion US$ only in OECD economies in 2016 [1].

2.1.1 Risk Analysis in the Cyberspace

As mentioned in the introduction of this section, the insurance premium an organization has to pay is dependent of the amount of risks, which are covered and the assessments of these risks. Before, it was assumed that the cyber risks of an organisation can be assessed in an appropriate level. However, in the practical world it is very hard to rate a risk. The estimation of a risk is dependent on the likeability that such a risk is exploited and on the impact when it is exploited as illustrated in Figure 2.1. Figure 2.1 shows a 5x5 risk matrix, which is a classic tool to assess risks. There are some limitations in the risk matrices, for example that a linear risk distribution is used, which is mostly not true in the practical world [35], but in this case it is enough to understand the relationships between impact, likelihood and risk assessment. Considering this dependency the impact and the likelihood have to be calculated first, before the risk can finally be assessed. Compared to other insurance types it is very hard to quantify the impact and the likelihood of a risk in cyber-insurance. Considering the impact, it is quite hard to tell in advance what costs a possible cyberattack would cause, which will get more clear when the possible costs are explained in Section 2.1.2. Some key questions, which help to roughly quantify the impact of a successful cyberattack are listed in the following. However, they just help to get an approximated estimation.

- How high are the average daily sales?
- What monitoring is available, that a problem could be detected as fast as possible?
- Do countermeasures already exist, when a problem is detected?
- What is the size of the critical data and what is contained in the critical data?

To quantify the likelihood of a risk exploitation the insurer has to consider two variables, the probability that the customer is attacked and the chance that this attack is successful. The probability that a customer is attacked is dependent of different variables. In some attacks the attacker picks the victim randomly, mainly when he just wants to win some money from the victim, for example by extortion and he does not care who is at the other side. On the other hand, there are attacks where the attacker only wants to damage the target because of personal, political, or any other reasons. For such an attack, a non liked customer is more likely to be attacked, for example, when it is a political party.

While the customer can not do that much to reduce the probability of a cyberattack, he is mainly responsible for the chance that an attack is successful. This is because this value is

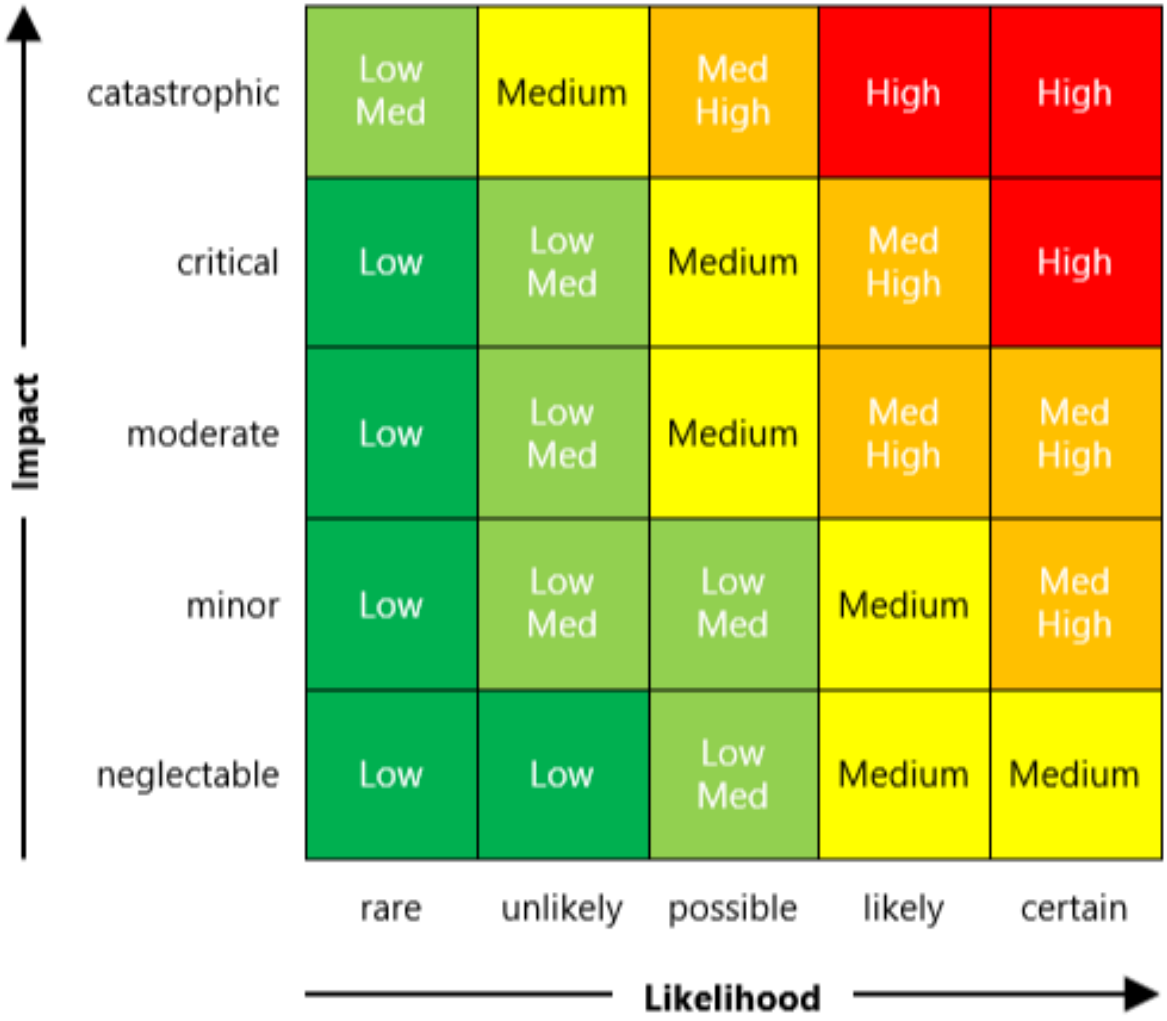

Figure 2.1: Classic 5x5 Risk Matrix by [35]

mainly dependent on the security of the customer's software, for which they are responsible themselves. As better the cybersecurity of the customer as less probable a cyberattack on the system is successful. In general, as more a customer invests in cybersecurity as smaller is the probability that its system is successfully attacked and hence the premium to pay for cyber-insurance should be smaller as well. However, this statement only applies if the insurer is able to correctly assess the cybersecurity of the customer, which is not as easy as it may sounds. There exists some techniques and methods, which can try to assess the security of a software, but they are not extremely exact and they need high involvement of the customer, which is not always given [8]. Beside that, the risk assessment, which is done today, is possibly not valid anymore tomorrow, because the insured software and the world around it is changing permanently and hence do the risks contained in it. These two problems will be tackled again in Section 2.1.3, where the specialities and the challenges in a cyber-insurance contract are described. Assuming that the insurer can assess the security of a software quite exact, the risks could be assessed then and a price tag could be assigned to cover a specific risk.

2.1.2 Costs Caused by Cyberattacks

A cyber-insurance contract protects the insured from specific cyber risks defined in the contract. To define which risks are actually present, the potential costs caused by cybercrime have to be analyzed. As software is mostly very complex and comprises different components, the negative effects of a cyberattack are mostly not so easy to track, because there are plenty of direct costs like direct loss caused by a service downtime, but also possible indirect costs due to reputation loos. Considering that, there are many costs, which seems to be clear, but also some that maybe often get forgotten. To gain a better understanding of costs in general, it helps to know the distinction between first- and third-party costs [24].
First-party costs are all costs which arise directly to the main party, while third-party costs are claims from another party, which were caused by a problem of the first-party. Considering an example of a car accident between two cars, where the driver in car A is the guilty party, these costs can be described more understandable. The driver in car A is considered as the first party. The first-party costs are all costs, which arise to the driver in car A directly. They include possible health costs for driver A and the costs to repair car A. The third-party costs are all costs, which arise through claims from other parties involved in the accident, including possible health costs from any other person involved in the accident, as well as the repair costs of the other involved car. In the following, these two terms will be defined more deeply in the context of cyberattacks and the most typical costs per category will be listed and explained. Some costs can not perfectly be assigned to one of the categories as they have characteristics of both. However, they are still assigned to only one category, where the specific cost seems to fit the best. It is important to understand that a cyber-insurance contract covers a specified list of the costs, which will be listed in the following. As more costs are covered as less risks the customer has to carry, but as higher the premium to pay to the insurer is.

First Party Costs

In the context of cyberattacks, the first-party costs are the costs directly associated with the measures against an attack (a privacy breach or a security failure) on behalf of the attacked organization [23]. These costs include everything what the customer did not earn because of the attack and everything what the customer had to pay to mitigate the attack and to recover from it. In the following list typical first-party costs are shown along with a short explanation what actions and what direct costs they comprise [2] [24] [25] [26]. It has to be considered that mostly not all of these costs arise in one single attack. In a data breach for example, there will probably be no business interruption as the system is not directly affected, while in a DDoS attack no data will be stolen in the normal case.

- **Incident Response / Crisis Management Costs:** Measures to handle and mitigate the attack.
- **Data Recovery / Systems Restoration Costs:** Recovering damaged or lost data and restore the system.

- **Forensics Costs:** Forensic expenses to determine the extent of an attack and its perpetrators.
- **Notification / Call-center Costs:** Notification to the clients and to the employees about the attack and maybe setting up a call-center for urgent questions.
- **Monitoring Costs:** Monitoring of the illegal usage of the stolen data.
- **Public Relations / Reputation Management Costs:** Expenses related to public relations and to keep the reputation on an acceptable level.
- **Business Interruption / Contingent Business Interruption Costs:** Loss during the time a service was not available.
- **Cyber Extortion Costs:** Price to pay to the attacker that the attack is stopped or that some private data is not published.
- **Lost of Digital Assets Costs:** Loss, damage or theft of digital assets.

Third Party Costs

The third-party costs comprises the costs, which accumulate through claims made against the attacked party from outside of the organization itself, the so called third party [23]. These claims arise because the third party did suffer from the attack on the company as well, for example because a service of the attacked company was not available or because some data, which concerns them, was lost. Normally, the third-party costs coverage does not pay all the looses of the third parties, it covers the costs to find a settlement between the attacked company and its affected partners. A list with typical third-party costs is shown in the following along with the costs they comprise [2] [24] [25] [26]. Compared to the first-party, the costs do not comprise any action from the first-party here, as they all mainly fall into the first-party costs. Here the first party has only to pay the claims. As in the first-party costs not all of these costs arise in one single attack most of the times.

- **Privacy Law Violation:** Legal fees and costs to find settlements with the sufferer, as well as judgements resulting from a lawsuit. Mainly caused by a data breach.
- **Third-Party Business Interruption Costs:** Costs to find a settlement for third-party loss during the time a service was not available.
- **Regulatory fines and penalties:** Costs assessed by regulators.
- **Errors and Omissions Costs:** Costs to find a settlement caused by a failure of delivering a contractual obligation.
- **Multi-Media Liability Costs:** Legal expenses to cover costs caused by intellectual property infringements.
- **Administrative Reissuing Costs:** Costs when customer cards have to be reissued because of loss of data.

Some insurers focus to offer coverage mainly for the first-party costs, which is of course a very important part. However, in many cases the third-party costs are even more critical, because legal claims from affected third-parties can result in expenses of thousands or even millions of dollars, which negatively impacting a company's assets and its ability to continue operation. Hence cyber-insurance only forms an effective umbrella of protection when both first- and third-party costs are part of the policy [24].

2.1.3 Specialities and Challenges in Cyber-Insurance

Considering the big amount of different costs caused by cybercrime introduced in Subsection 2.1.2 and the need to somehow create insurance products, which offer coverage for all these costs, it seems clear that cyber-insurance is a very complex insurance product. However, beside the high complexity there are other important specialities and challenges that have to be considered in cyber-insurance compared to other insurance products. In the following, some key characteristics of the cyber-insurance products are listed and described. Along that, challenges, which appear through the key characteristics, are explained.

High Complexity of the Cyber-Insurance Products

As already indicated in the introduction of this subsection cyber-insurance products have a very high complexity compared to other insurance products. The high complexity of the products lies mainly on the big set of different risks and derived from that the big set of different costs, which are present in the cyberspace introduced in Section 2.1.2. Through that, it can get quite hard to define which coverage a specific customer actually needs and which not. Of course, there are other insurance products, which have a high complexity, maybe even higher than the one in the cyber-insurance products. However, there are two additional problems, which currently exist not in the cyber-insurance market in general, but in the current developing state of the cyber-insurance market.
First of all, the cyber-insurance market is currently at the beginning of its development and there are no standardized product yet and hence the offered coverage differs a lot between the different insurers [1]. Therefore the potential customers should have a clear understanding of their cyber risk exposures by themselves, to be able to determine what coverage and which coverage amount they need in their current situation. Exactly this understanding is missing to a lot of the potential customers [1]. According to a survey executed by Wyman [9], 49 percents of the respondents have "insufficent knowledge" about their cyber risk exposures to assess the type and coverage of insurances they need [1].

Beside the missing knowledge of the potential customers also the intermediate players between insurer and customer, namely the insurance brokers and insurance agents, having a non satisfying knowledge and understanding of cyber risks and cyber-insurance. They consider cyber-insurance as a key and growing market, but still missing the needed understanding. Through that the difficulty of selling cyber-insurance product is increased [1]. These two problems are some of the main reasons that the cyber-insurance market struggles in the complete penetration of the market and they intensify and underline

the cyber-insurance characteristic of the high complexity and make it to a mentionable speciality of the current cyber-insurance market.

Digital Dynamic Product to Insure

At the end of Section 2.1.1 it was mentioned that it is quite hard to assess the cybersecurity of a specific software or even of a whole organization, which is needed to quantify the risks and calculate a fair premium. Mainly due to newness and the scarcity of data, there do not exist standardized frameworks to rate cybersecurity yet [7] [1]. Besides that, it is possible that the cybersecurity changes from one day to the other. The main reason for these problems is that the insured product is a digital product, which mostly is dynamic and changes a lot. Software is very big and complex and bugs can be implemented everywhere. It takes a huge amount of time to go through the whole code to look for coding errors and even if this time is taken, someone possibly already implemented new bugs in the already checked code. Additionally, new yet unknown attack types evolve through new technologies, which can not be checked against, because they are not known yet. Hence probably no one can find the time to control the whole code for bugs and really assess its cybersecurity, which makes it hard to determine a fair premium [2].

An organization itself knows how much it has invested in cybersecurity during development and by using specific security tools. Hence they can estimate the security of their software. However, the insurer does not have this knowledge and can just try to guess the effort of the organization by asking for specific information. This information is mostly gained by questionnaires on the customer's security posture, which just do not deliver enough information for an adequate security assessment [7]. Therefore the insurer can hardly differentiate between an organization, which invested a lot in cybersecurity and one which did not, which is due to information asymmetry [6].

To really understand this speciality of insuring a digital product it makes sense to make a comparison with a car insurance. There the problem of information asymmetry exists as well and the insurer can not directly distinguish between a high-risk driver and a low-risk driver and this leads to the problem of a lemon market, that only the high-risk drivers buy an insurance, because for the low-risk drivers it is too expensive. However, the insurance providers found a way to steer against the lemon market by giving premium reductions when a customer had few accidents in the past and a normal driving style. It can be assumed that a person, which had few accidents and a normal driving style is a low-risk driver and will also have few accidents in the future, because in normal cases the driver style does not change. Therefore for low-risk drivers it makes sense again to buy a car insurance, because of the premium reduction. One can think that considering the history of successful attacks on a software could cause the same effect in cyber-insurance. Generally, it can be assumed that a customer, which suffered a lot of successful cyberattacks, will experience a lot of successful cyberattacks in the future, because the customer has probably a weak cybersecurity. However, the other way around is not really given, mainly because of two reasons. First, the targets of cyberattacks can be quite random and it is possible that an organization with no attacking history was just never attacked in the past, but will be attacked a lot in the future. Second, the security of a software is permanently changing and maybe a newly added component

includes a lot of cyber risks. In general a software piece changes a lot faster than the behaving of a human, which makes it a lot harder to predict the future. Considering that, cyber-insurance theoretically requires a continuous feedback loop between customers and insurers to dynamically update the risk assessment and the resulting premium [7]. In general there is still a high uncertainty in cyber risk assessment and leading out of that in pricing cyber risk coverage. Therefore insurer tend to overcharge, because they do not want to loose money [7] [1]. This is another reason, why cyber-insurance is not that far widespread yet.

Free Riding Problem

Considering that the challenges mentioned before are mainly caused because cyber-insurance is a quite new insurance product and by the time, when more knowledge is available and when there exist some standardized appropriate methods to dynamically process the risk assessment, these challenges will not be that crucial anymore. However, there exists another problem in cyber-insurance, which differentiates cyber-insurance from traditional insurance products, the so called free riding problem. Cyber risks have an interdependent and correlated nature [6], which means that when one user in a network does investments in cybersecurity it generates positive externalities for other connected users in the same network by increasing their individual security [6].

In a large distributed system such as the Internet the amount of connected users can be very high. Considering that a potential cyber-insurance customer is required to have a minimum level of cybersecurity and hence the customer has to improve the own cybersecurity, it is likely that the security of all Internet users is increased. This positive external effect may discourage some companies to buy cyber-insurance, because it would create a free riding problem for other companies, which could profit as well. This may reduces incentives to increase cybersecurity and to get cyber-insurance by oneself [1] [2]. Somehow an incentive has to be created for the users to invest in cybersecurity, although they experience positive externalities from other users investing in the network [6].

2.2 Blockchain / Smart Contracts

There is a big hype about blockchains and smart contracts. These technologies are praised as "The next big thing" [27] or as "The new Graal" [28], because they could bring huge benefits to many different sectors and even to everyday activities [3]. Blockchains mainly remove the need of trust, which makes expensive middlemen become unnecessary. However, the blockchain technology has some drawbacks and is not fully mature yet. Voices increase that blockchains are just overhyped and that the benefits could also be achieved using already emerged technologies [29] [30]. In Subsection 2.2.1 the blockchain technology is explained and its main characteristics are introduced. Smart contracts and its relation to the blockchain technology are introduced in Subsection 2.2.2.

2.2.1 Blockchain

A clear initial definition of a blockchain is given by Gatteschi et al. [3]:

> "A blockchain is a distributed ledger maintained by network nodes, recording transactions executed between nodes (i.e., messages sent from one node to another). Information inserted in the blockchain is public, and cannot be modified or erased."

This definition will be further explained in the following. In general a blockchain replaces the need to build a trustworthy environment between two non-trusted parties by using a cryptographic proof. Hence blockchain often makes a trusted third-party unnecessary [10]. As the name indicates a blockchain is a chain of blocks, where each block is linked to the previous block with a cryptographic hash. Inside a block a list of transactions, which change the state of the blockchain, are stored. Such a transaction is created and exchanged by the nodes inside the blockchain network [11]. Each transaction has to be signed with the private key of the transaction sender so that the correct sender later can be verified using its public key. Like that the chain of ownership of a monetary amount or simply a coin can be verified [10]. Figure 2.2 illustrates the idea behind this concept of transactions. Transactions can not only exchange coins between the nodes, but also

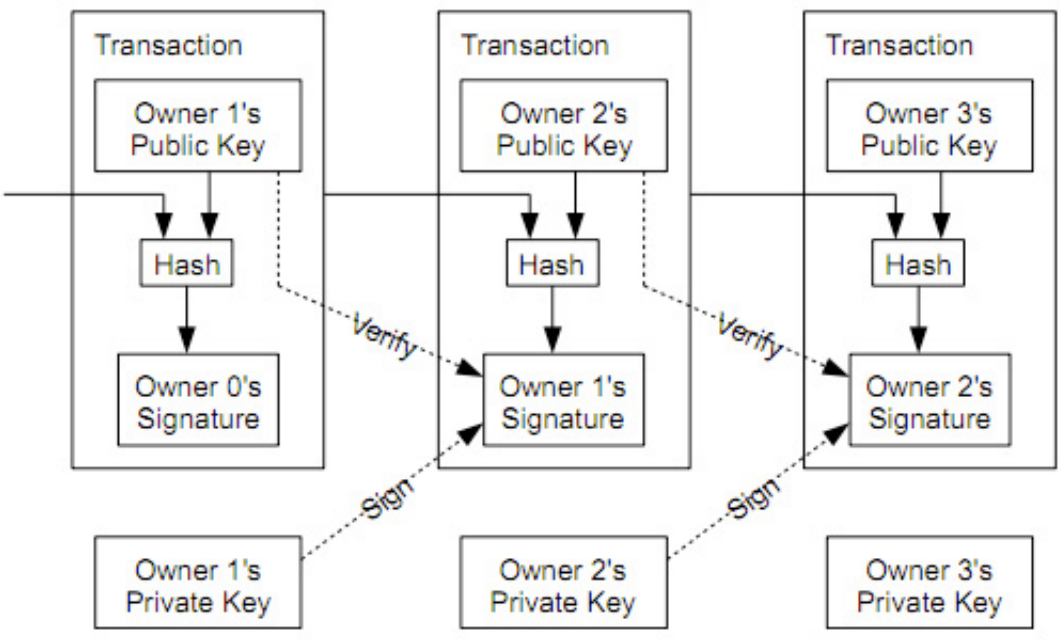

Figure 2.2: Transactions in a Blockchain by [10]

execute predefined code within a smart contract [11]. A smart contract is a self-executing contract, where the conditions are written in code [3]. More about smart contracts will be discussed in the next part of this section. The blockchain is shared among all nodes in the network and when new transactions are added, new blocks have to been created,

which will extend the current blockchain. What exactly happens when new transactions are added is shown in the following steps [10]:

1) New transactions are broadcast to all nodes.
2) Each node collects new transactions into a block.
3) Each node works on finding a difficult proof-of-work for its block.
4) When a node finds a proof-of-work, it broadcasts the block to all nodes.
5) Nodes accept the block only if all transactions in it are valid and not already spent.
6) Nodes express their acceptance of the block by working on creating the next block in the chain, using the hash of the accepted block as the previous hash.

First the new transactions are broadcast to all nodes in the network in step 1. The nodes then collect the new transactions into a block in step 2. It is possible that a fraudulent user tries to double spend his coins by creating a new block with transactions in it, where the now transferred coins were already used before. This problem is called the double spending problem. To avoid that a so called consensus algorithm is used in step 3. The consensus algorithm tries to make sure that the current valid blockchain is verified by the majority of the nodes and it hinders that each node creates its own individual blockchain by adding a new block to the chain and broadcasts it. The consensus algorithm used here is the proof of work algorithm, which is used in the popular blockchain Bitcoin [10]. This algorithm only allows to add a new block, when it can be proven that a specific amount of work was carried out by solving a puzzle. Through that the longest chain, which is the valid one, is built through the biggest computation power, which is in the most cases the whole set of non-fraudulent users. This will get more clear later. There exists other consensus algorithm as for example proof of stake, which ensures that the longest chain is created by the nodes with the majority of coins in the blockchain. The proof of stake consensus algorithm is used in Ethereum [12].

When the puzzle is solved by a node, the new block is broadcast to all other nodes in step 4. In step 5 the newly broadcast blockchain can be accepted by the other nodes. In this case, they stop working on their current block and continue with creating a new block, which should be an extension on the newly broadcast blockchain in step 6. If a node did not accept the new block, the node just continues to work on its current block and ignores the new block. Considering the second case, where a block was not accepted by some nodes, there will be more than one blockchain in progress. However, only the longest chain is considered to be the correct one, which is normally the one, where the most computation power is behind in the long-term [10].

As a lot of computation power has to be invested to solve the puzzle and then to be allowed to broadcast the new block, there need to be an incentive to try to solve the puzzle. The computation winner, which is the one who solved the puzzle first, will receive some coins on the blockchain as a reward, which should compensate his effort and motivate the nodes to continue solving the puzzles [10]. A node, which participate in this competition is called a miner. To better understand blockchain, its main properties along with a short explanation are shown in the following list [10][11].

- **Full Decentralization:** There is no central entity and everything is decentralized.
- **Immutability:** Data, which is stored on a blockchain once is nearly immutable. To remove or change stored data, a longer chain than the current longest chain that does not contain this data has to be created, which is only possible if one has the highest computation power. As older a block is, as more immutable it gets, because newly added blocks confirm older blocks by extending the chain after it.
- **Transparency**: All data and every update in a blockchain is transparent to its user and everyone can see all the information stored. Having a fully transparent system leads to a trade off that no privacy can be provided to the users.
- **Public Verifiability:** In a public blockchain anyone can verify the correctness of its state as it is transparent and everyone can see all the information. Miners accept a blocks if they start working on extending the chain with the new block. This replaces the need of third parties.
- **Integrity of Information:** Information is protected from unauthorized modifications. Anyone can verify the integrity of the data, which is linked to the public verifiability.
- **Traceability:** Every detail of a transaction on a blockchain can be traced back.
- **Redundancy of Data:** Through the replications of the blockchain data on every participating node, the redundancy of the data is inherently provided in a blockchain.

2.2.2 Smart Contracts

As indicated in the previous section a transaction in a blockchain can execute predefined code within a smart contract, which is a self-executing contract. In a smart contract the terms of agreement between two parties are directly written in code, which controls the execution. Therefore transactions are trackable and irreversible [15]. The term smart contract was initially defined by Nick Szabo as following [13]:

> "A smart contract is a computerized transaction protocol that executes the terms of a contract. The general objectives of smart contract design are to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimize exceptions both malicious and accidental, and minimize the need for trusted intermediaries. Related economic goals include lowering fraud loss, arbitration and enforcement costs, and other transaction costs."

Smart contracts are powerful, but they need an infrastructure around them to bring out their full potential [14]. The infrastructure should be able to run, execute and verify the contract's transaction data. A blockchain is perfectly appropriate to take in this part and provides a fully decentralized infrastructure. The blockchain Ethereum is currently the most famous blockchain, which supports the execution of smart contracts. In Ethereum the smart contracts are run in an Ethereum Virtual Machine(EVM) and the coin used

is called ether and its subunit wei. One ether contains 10^{18} wei. To reward the miners every transaction executed in the chain has to be paid by an amount of ether [16]. The paid coins are given to the miner that first mines a block, which includes the transaction. How many ether has to be paid for a specific transaction is dependent on the complexity of the transaction and the so called gas costs.
Gas is a special unit inside the EVM, which is used to calculate the complexity of a transaction. Each operation that can be performed by the EVM has a hardcoded gas cost. Hence for all operations in a transaction the gas costs are summed up and a total gas cost of the transaction is determined. Gas is not directly a subunit of ether, however it is converted into an amount of ether by using the gas costs. The gas cost determine how many wei one gas costs. The gas costs can be determined by the caller of the function and as higher the gas costs as higher the probability that a transaction will be included in one of the next blocks, because the miners earn more ether. If the gas costs are too low it is possible that a transaction will never be integrated in a block. A gas price of 20 gwei (20'000'000'000 wei) is the default value in Ethereum and according to [16] a transaction with such a gas price will usually be inserted in the next few blocks. While a transaction with a higher gas price of 40 gwei will nearly always be included in the next block, a transaction with a lower gas price of 2 gwei will probably takes some minutes until it is included in a block [16]. Beside the reward for the miner, the gas costs also help to prevent from potential denial-of-service attacks, since otherwise the operations of a smart contract could just be repeatedly called in a loop by an attacker and overload the EVM. Through the benefits of the blockchain smart contracts are ensured to be permanently stored and it is near to impossible to manipulate its content, because their code is stored on the blockchain, which is nearly immutable. Using smart contracts on a blockchain fully distributed digital organization can be created [14].

2.3 Economics of Cybersecurity

The costs caused by cybercrime and following out of that the need for cyber-insurance is increasing over the past few years and is predicted to increase a lot more in the near future. According to predictions, costs caused by cybercrime will increase from 3 trillion US$ in 2015 to 6 trillion US$ in 2021 to 10.5 trillion US$ in 2025, which is a doubling in only around 6 years. This would be the world's third-largest economy after the U.S. and China [22]. The volume of cyber-insurance is predicted to grow even faster and to more than double in around 3 years from 3.4 - 4 billion US$ in 2017 to 8 - 9 billion US$ in 2020 [1], which would be a massive increase. Considering this, the economics of cyberattacks and of cyber-insurance will be analyzed more deeply in this section.

2.3.1 Economics of Cyberattacks

As mentioned in the introduction to this section, the costs caused by cyberattacks are increasing extremely. In the first place, this is because the number of cyberattacks is increasing, but also because the economic impact caused by an individual cyberattack is growing as well. The rising number of cyberattacks is mainly caused by new technologies,

which offer new attack types and create new attacking surface [2][21][18]. For example a DDoS attack can nowadays just be bought over the internet for quite low prices, which was not possible several years ago, because the technology was not so far developed.

The increasing interconnectivity between different systems and even between different organization and the automating of more and more components in most of the businesses are the main reason for the growing impact, when a cyberattack is successful. Considering that one system has a downtime, everyone who relies on it has to suffer from an attack as well. In the past when every application was running by itself and had no dependencies, this was not the case and hence only the attacked application was harmed.
Thanks to internet of things and 5G, which may will come soon, this growth is not expected to slow down in the future as more new technologies will be invented, which offer new attacking possibilities [21][18]. Because of the above reasons cybercrime is seen as one of the biggest challenges that humanity will face in the next two decades [18]. Waren Buffet, a billionaire businessman, even calls cybercrime the number one problem with mankind, and cyberattacks a bigger threat to humanity than nuclear weapons [19]. To better understand the extreme costs of cyberattacks Table 2.1 shows an overview of the biggest attacks and data breaches in the past along with their estimated economic impact if available [31].

Table 2.1: Biggest Cyberattacks and their Economic Impacts

Cyberattack	**Type**	**Impact**
WannaCry	Ransomware attack that infected up to 400 thousand computers. Encrypted files and blackmailed the users.	$4 billion
Petya	Ransomware attack that infected up to 300 thousand computers. Encrypted whole hard drive and blackmailed the users.	$3 billion
Epsilon Data Breach	Data breach, where the addresses and names of 60 million users were exposed.	$4 billion
Yahoo Data Breach	Data breach, where personal data of 3 billion accounts was stolen.	?
FriendFinder Networks Data Breach	Data breach, where sensitive information of 412 million user accounts was disclosed.	?
Marriott Hotels Data Breach	Data breach, where where personal information including credit card numbers of 500 million customers were stolen.	?

As seen in the table the economic impacts are extremely high by only some single attacks. The attacks with the biggest global economic impact are mainly data breaches and ransomware since they mostly do concern a huge amount of users and victims and not only a single organization as other types like denial-of-service attacks mostly do. However, in the view of a single organization these attacks can be as critical as the others.
It can be thought that it is only very bad luck, if a specific company is attacked by a cyberattack. However, already in a survey in the year 2013 54% of the participating organizations mentioned that they have been subject of a cyberattack in the last 3 years, while 17% did not answer or were not able to answer the question if they were attacked.

Research from the same year revealed that in average the financial impact to companies due to cyber incidents was $9.4 million [2]. Considering these evaluations it is more probable that an organization is attacked in the next 3 years than it is not. As these numbers are assumed to have been increased a lot over the past years and will be increasing more in the future, the probability is now even higher and will continue to increase. To look at this more concretely the current numbers of DDoS attacks will be considered as a specific example. In the first half of 2020 the amount of DDoS attacks has been increased by 151% compared to the same period in 2019, which is a huge increase for only a single year. Beside the number of attacks the DDoS attacks are also increasing in its size, since the cyber-criminals can compromise more endpoints with commercialized DDoS services. Organizations also improve their measures to defend against and mitigate DDoS attacks, hence the attackers have to create bigger volumes to overwhelm the defenses [20]. Considering that, it seems obvious that the era of cybercrime has just begun.

2.3.2 Economics of Cyber-Insurance

To analyze the even more increasing volume of cyber-insurance, this subsection investigates the cyber-insurance market considering some different dimensions as place, company size and company sector. Table 2.2 shows the cyber-insurance volume in some countries and compares it to the total volume of insurance in general [1][32].

Table 2.2: Cyber-Insurance Volume in some key Economies by [1]

Economy	**Cyber-Insurance Premiums (US$, million)**	**Total Insurance Premiums (US$, billion)**	**Cyber-Insurance Premiums as a Proportion of Total Insurance Premiums**
Brazil	0.6458 (2016)	58.9 (2016)	0.001%
Germany	105 - 117 (2016)	327.3 (2016)	0.03%
India	27.9 (2017)	69.8 (2016)	0.04%
Japan	134.2 (2017)	407.4 (2016)	0.03%
South Korea	26.4 (2016)	186.6 (2016)	0.01%
The U.S.	2500 (2016) 6200 (2020)	2703.8 (2016)	0.09%
Total	3500 (2016)	5205.8 (2016)	0.067%

In the table it can be seen that in every shown country the cyber-insurance volume is just a very small part of the total insurance volume. Cyber-insurance did not totally penetrate the market yet and it is still at its very beginning. More than the half of the cyber-insurance premiums are paid in the United States, which is not only because the United States have the biggest total insurance volume. There the cyber-insurance volume also has the highest proportion of the total volume, close to 1%. The main reason for this is because in the United States the cyber-insurance market is further developed and there are already a lot more cyber-insurance providers. However, it is expected that it will still increase a lot in the future. In Brazil, where organizations are not as automatized as in

the other shown economies, the proportion of cyber-insurance is a lot smaller. This makes sense as in Brazil the thread of cybercrime is not as big as in countries, where everything is dependent on technologies.

Besides the location, also the size and the type of an organization is important regarding the probability if a company has a cyber-insurance contract or not. In OECD countries the penetration for stand-alone cyber-insurance was above 50% for the large companies, while it was under 10% for small and medium sized enterprises [1]. Data intensive companies are more likely buying a cyber-insurance, which seems obvious, because they carry more risks being attacked. These include mainly banking, financial service and insurance providers, IT and Telecom organizations and the Healthcare sector, as they all have to work with personal data. Retail and manufacturing companies are not that threatened and hence the cyber-insurance volume from them is smaller in proposition to the previously mentioned sectors [33]. As mentioned before, mainly because of the increasing thread through cybercrime and the growing awareness that cyber-insurance could minimize these risks, the volume is predicted to increase a lot more in the future. According to predictions from March 2020 the premiums paid for cyber-insurance will increase to 28.6 billion US$ by 2026 [33].

Chapter 3

Related Work

There are several effort in the field of cyber-insurance and to set it up using a blockchain. In this chapter, relevant work related to this thesis is highlighted. In the first section, several efforts to solve the problem of the cybersecurity assessment is introduced. In a second part, different uses cases are presented, in which a blockchain can be applied in the insurance business. Also, a set of the benefits and the drawbacks of using a blockchain is discussed. Finally, the most important external effects of cybersecurity are presented.

3.1 Cybersecurity Assessment

In this section two works will be analyzed that try to solve the difficulty of correctly assess the cybersecurity of a complex ecosystem. In a first work, [8] introduced the SEConomy framework, which is a strictly-step based framework. The mapping and the modeling of risks is proceeded in 5 sequential stages. This concept allows the creation of economic models with fine-grained estimates. The 5 stages are shown in Figure 3.1. In the first stage the actors contained in the ecosystem are defined, along with their functions and the interactions between them. For each actor identified in stage 1 the system/components and processes that are performed are determined in stage 2. For an initial attribution of the capital and operational expenditures (CAPEX/OPEX) the actors legal implications have to be taken into account as well in the second stage. In stage 3 possible risks and threats are analyzed based on the mapping of actors, systems and processes done in stage 2. Risk models are generated and for each risk the impact is estimated when the risk is exploited. Possible prevention measures and incident response actions are listed as well. In stage 4 for each risk identified before potential costs are mapped and it is determined if proactive or reactive approaches are the better economic alternative for each specific risk. Finally in stage 5 the outputs of the previous stages are collected and a general feedback is given that includes the overall economic impacts, the determination of improvement actions and best practices.

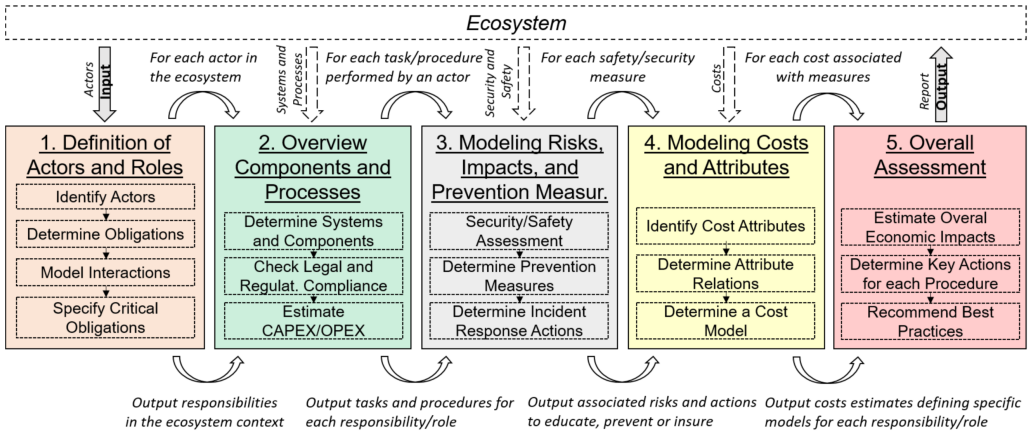

Figure 3.1: SEConomy Framework

SEConomy was generally designed to support the decisions if a security investment should be applied or not by considering the cyberrisks. Since cyber-insurance is a potential proactive security measure an outcome for a company using the SEConomy framework could be that some risks are not mitigated but transferred using cyber-insurance. It was not the primal intention of SEConomy to be used by an insurer to improve the premium calculation. However, as it gives a good overall assessment of the cybersecurity of an ecosystem, it is well suitable to support an insurer by the risk classification of a potential customer.

In another work, [7] introduced a blockchain-based continuous monitoring and processing system for cyber-insurance, which is called BlockCIS. Similarly to the SEConomy framework the usage of BlockCIS should improve the risk assessment of an ecosystem. However, while SEConomy was not directly aimed to be used in a cyber-insurance use case BlockCIS was explicitly designed for cyber-insurance. The system mainly tackles the inherent challenge of cyber-insurance mentioned in Subsection 2.1.3 that software can change permanently and the risk assessment done one day is maybe not valid anymore the other day.
Instead of doing a static risk assessment once at the beginning of a contract, the BlockCIS system offers the insurer and the customer the possibility to have an automated, real-time and immutable feedback cycle and hence to apply a dynamic risk assessment. To do so BlockCIS interconnects the insurer and the customer over a blockchain. When a contract is created a BlockCIS node is deployed in the network of both sides. The node deployed at the customer network monitors the enterprise services and try to gain information about the security of the customer. It collects analytics and statistics like firewall logs or database logs and forwards it to the blockchain. The insurer then uses the forwarded information to better assess the security and to calculate a dynamic premium. Beside the customer and the insurer two additional parties are connected to the blockchain system: Third party services and auditors. Third party services support the insurer to assess the likelihood of a cyberattack and its impact by analyzing possible threats in the internet in general. The collected information is also forwarded to the blockchain and used by the insurer. Auditors solve disputes between the insurer and the customer and possibly regulate the actions of the insurer. Figure 3.2 shows an overview of the BlockCIS system.

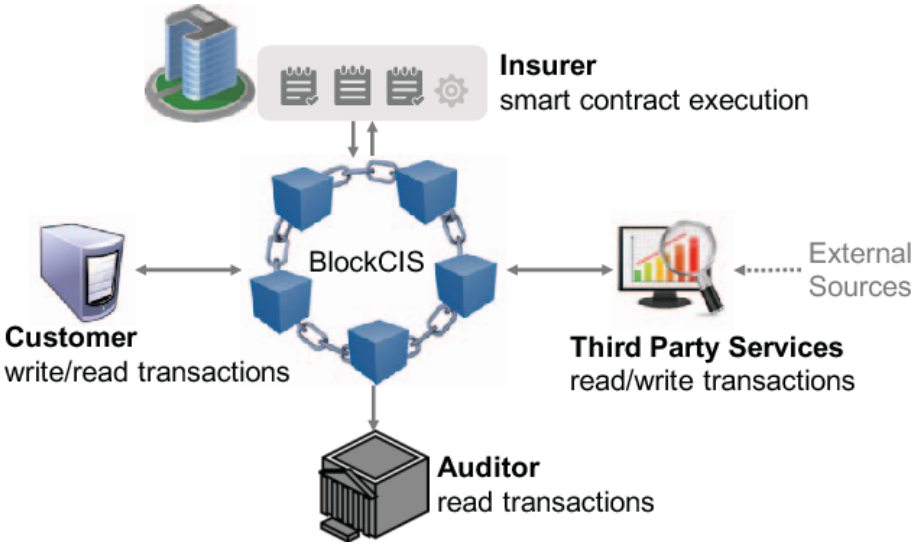

Figure 3.2: BlockCIS Overview by [7]

In general BlockCIS is a tool that supports dynamic and fair premium calculation. However, BlockCIS is just a supporting tool and can not be used as an individual tool to provide a cyber-insurance service. For example the paying of the premium and the paying of claims is not included in the system and hence that has to be managed by other applications. However, frameworks like these could be well used to correctly assess the cybersecurity and based on that to calculate a fair premium in a cyber-insurance contract.

3.2 Blockchain and Smart Contracts in Insurance

The previous introduction of the BlockCIS system, already introduced a possible approach to use a blockchain to build up cyber-insurance contracts. This section focuses on the analysis of the work presented in [3], that considers the usage of blockchain and smart contract in connection with insurance in general and analyses if these technologies are appropriate in the insurance sector. The work mainly investigated the debate if blockchain and smart contracts are just adopted in insurance because of a current hype of the technology or if it indeed brings some major advantages for different use cases. Different use cases of blockchain and smart contracts in insurance analyzed by [3] are shown in the following list.

- **Improvement of Customer Experience and Reduction of Operation Costs:** Blockchain and smart contracts can be used to automatize or at least simplify the premium payment and the claim processing by integrating these processes in smart contract functions. While the insurer can save some resources to manage these processes, the customer experience is increased as well, because the customers claims can be handled faster. A total automatization is mostly difficult in claim processing since the majority of claims need to be evaluated by an expert before being settled. However, the process can still be simplified.

- **Data Entry/Identity Verification:** Through the cryptographic mechanism in a blockchain each participant is identified and verified by its account. This could reduce the insurers effort to manual data entry and to verify customers. However, as the customers could loose its credential and could not access the account anymore, there are also some drawbacks here.

- **Premium Computation/Risk Assessment/Frauds Prevention:** As done in the BlockCIS system [7], the blockchain can be used to gather different information about the customer and its environment from the customer itself or by using third parties. This information can be used to calculate a fair premium or also to prevent fraud by analyzing the collected data. The privacy of the customer is a relevant issue.

- **Pay-Per-Use/Micro-Insurance:** Through smart contracts micro-insurance gets profitable for an insurer since the administrative costs can be reduced. Through that the possibility to create quick and cheap policies is increased. Though the automatic and permanent collection of data over the blockchain, also pay-per-use insurance gets possible. By knowing when a service is used by a customer the insurer can create pay-per-use products, where the customer has to pay premium only when the insured service was used. This makes the premiums more exact.

- **Peer-to-Peer Insurance:** As everyone can participate in a blockchain and call its functions peer to peer insurance can be applied, where everyone can insure everyone. To do that just an according smart contract has to be deployed on the blockchain.

The shown uses cases give an insight of potential applications of the blockchain technology in the insurance sector and there seem to be numerous possible advantages. However, until now only a few prototypes were created and it is not yet clear if they give a clear benefit to the insurance sector or if the tackled problems could be solved in other easier ways than using a blockchain. The previously introduced BlockCIS system [7] is a perfect example for a system that applies the third mentioned use cases of improving the risk assessment and premium calculation by monitoring the customer over a blockchain. In this work mainly the first and the second use cases mentioned above are handled, while the premium calculation is also tried to be improved.

The authors of [3] also proceeded a SWOT analysis and tried to summarize the advantages and disadvantages of applying blockchain in the insurance sector. The SWOT analysis is shown in Figure 3.3. While most of the shown strenghts and opportunities were already mentioned at some point before, there are several weakness and threats, which were not tackled yet. For example one of the most relevant weakness is the current lack of scalability. Through the required computational power and consequently the high energy consumption during mining new blocks, the number of transactions that can be handled per second is extremely low compared to other systems. As in insurance the number of transactions is not as high as in other organizations like a banking service, this drawback is not so grave in this case. However, still considering all the disadvantages that blockchain brings it is hard to tell if it will be adopted more in the next years.

	Strengths	Weaknesses
Internal	- Fast and low-cost money transfers - No need for intermediaries - Automation (by means of smart contracts) - Accessible worldwide - Transparency - Platform for data analytics - No data loss/modification/falsification - Non-repudiation	- Scalability - Low performance - Energy consumption - Reduced users' privacy - Autonomous code is "candy for hackers" - Need to rely to external oracles - No intermediary to contact in case of loss of users' credentials - Volatility of cryptocurrencies - Still in an early stage (no "winning" blockchain, need of programming skills to read code, blockchain concepts difficult to be mastered) - Same results achieved with well-mastered technologies
	Opportunities	**Threats**
External	- Competitive advantage (if efforts to reduce/hide the complexity behind blockchain are successful, or in case of diffusion of IoT) - Possibility to address new markets (e.g., supporting car and house sharing, disk storage rental, etc.) - Availability of a huge amount of heterogeneous data, pushed in the blockchain by different actors	- Could be perceived as unsecure/unreliable - Low adoption from external actors means lack of information - Governments could consider blockchain and smart contracts "dangerous" - Medium-long term investment - Not suitable for all existing processes - Customers would still consider personal interaction important

Figure 3.3: SWOT Analysis of the Adoption of Blockchain by [3]

3.3 External Effects in Cybersecurity

If contract discrimination exists amongst customers, a better cybersecurity should lead to a lower premium for cyber-insurance. Considering that, one could assume that the customers have an incentive to increase their cybersecurity, because they want to pay less premium. Taking this assumption, cyber-insurance theoretically should increase the cybersecurity of the customers. [6] investigated this statement and analyzed the free riding problem in cybersecurity mentioned in Subsection 2.1.3, which is caused by the external effects of cybersecurity investments. They analyzed different market setups like a monopolistic market and derived the satisfaction of the stakeholders. In most of the market setups the insurer had no expected profit and hence no incentive to participate in the market. To remove this issue alliances between insurers and security software vendors can be created. A customer that buys security software from a partner gets a premium reduction. In return the insurer gets a part of the profit of the security vendor. In special cases the insurer can even sell security software by himself and generate benefit. Like this both parties can profit from the collaboration and the cybersecurity is increased because of the presence of cyber-insurance.

Chapter 4

SC4CyberInsurance Approach

Not least due to the usage of blockchain and smart contracts the proposed solution should bring the following benefits:

- Simplify the way of creating a cyber-insurance contract
- Reducing the complexity of the cyber insurance contract
- Less Effort through automatization
- Trustworthy and immutable agreement
- Secure and transparent, but still anonymized

In this chapter it is explained how these advantages are tried to be reached by introducing the design of the SC4CyberInsurance approach and giving a first overview of the solution model. In Section 4.1 the general architecture is initially explained. The information that the customer has to fill out is discussed more in Section 4.2. Afterwards each component of the architecture is described individually in Section 4.3. Finally, in Section 4.4 the design ideas of the smart contract are investigated more deeply.

4.1 Architecture

The general architecture has to facilitate the thesis goals and should be as simple as possible. Figure 4.1 shows a high level architecture of the SC4CyberInsurance system. The components of the system that are running at the insurer respectively at the customer side are represented in the dashed rectangles. Both actors mainly access their part of the system over an API. Theoretically, they can also access the smart contract directly using its function. However, using the API brings several advantages and sometimes it is required to use the API to experience the whole benefit of the system because some components of the architecture are activated over the API. Out of this reason the direct interaction to the smart contract is not shown in the architecture.

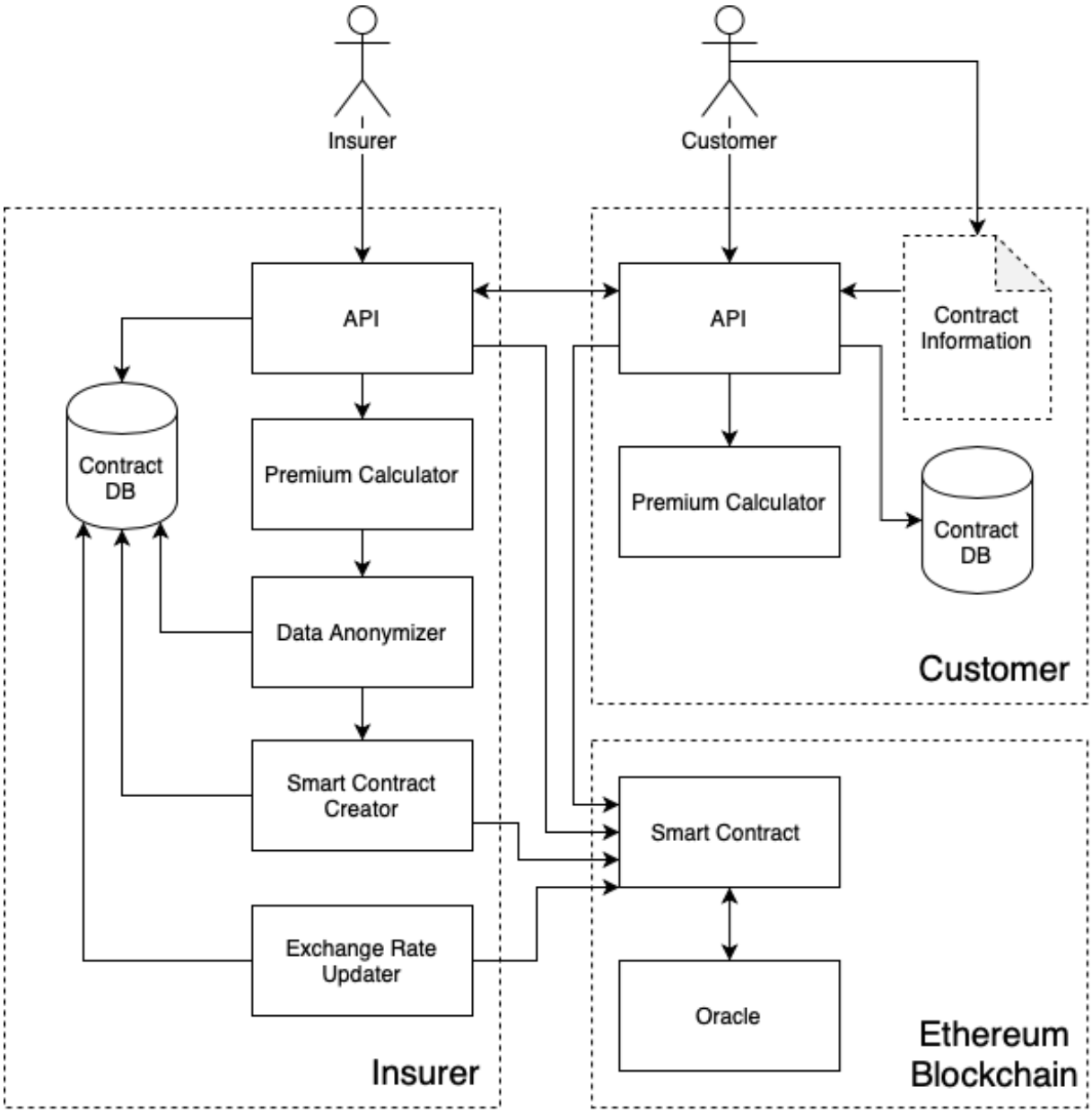

Figure 4.1: System Architecture of SC4CyberInsurance

At the beginning of the lifetime of an insurance contract the customer has to fill in the information, that is needed to set up a contract. The content is entered in a prepared file and should contain information about the company in general and about the coverage. What information is contained in the file will be explained in more detail in Section 4.2. When the file is completely filled out the customer can calculate the offered premium using the `Premium Calculator` component. When the premium is accepted by the customer, the insurers system part is notified that a contract file is ready to be processed. The insurer system then automatically reads the file over the customers API. The `Premium Calculator` on the insurer side then uses the read information and calculates the premium. Since the `Premium Calculator` is equivalent on both sides the calculated premium should be equal to the one calculated on the customer side before.

As private information of the customer, which should not get publicly published on a blockchain, was necessary to calculate the premium, the `Data Anonymizer` only forwards anonymized information to the `Smart Contract Creator` and stores the whole content of the file in a local database. The `Data Anonymizer` forwards the hash of the file, the calculated premium and some contract constraints from the file. Since both actors have to be in possession of the contract agreements the file content have to be stored on both databases at the insurer and the customer side. At the customer side the data is directly stored over the API when a premium is accepted.
With the forwarded information the `Smart Contract Creator` dynamically creates a smart contract and deploys it on the Ethereum blockchain. The address of the deployed contract is stored in the databases of both sides that the contract can be accessed later again using its address. Some of the functions of the smart contract are restricted to be called only by the insurer and some only by the customer. Until the end of the validity of the deployed contract the customer has to pay the premium to the insurer and is allowed to report suffered damage. If the damage is covered in the contract agreements the insurer has to pay the damage. Both actors can also update the contract conditions, for example if the customer has a new operating system. In such a case a new premium is calculated and the updated agreements are stored in the database again.
The components between the API and the deployed smart contract, namely the `Premium Calculator`, the `Data Anonymizer` and the `Smart Contract Creator` are only used when a new contract is deployed or partly when a currently existing one is updated by a party. In every other cases the API directly access the smart contract functions and the data stored in the database.
Since the value of a cryptocurrency on a blockchain is very volatile it is very risky for both parties to calculate the premium in the cryptocurrency. If the cryptocurrency gets more valuable the premium increases and if it gets less valuable the premium decreases. As an insurance should transfer risk and not create new risks, the premium is calculated in the currency of the specific country, which is generally less volatile. Because of that, the values to pay have to be converted from the country currency into the cryptocurrency each time when some money is transferred. Hence the exchange rate between the two currencies have to be known on the blockchain. Therefore an oracle is used that access a third-party website, where the exchange rate is stored. Using that exchange rate, the contract can convert the value in the country currency into the same value in the cryptocurrency. In the following, euro is considered as country currency. As the update of the exchange rate on the smart contract calls an external oracle, it takes some time and also costs something. That the normal contract functions are not disturbed and slowed down by the updating of the exchange rate, the updating is managed by a separate component, the `Exchange Rate Updater`. This component updates the exchange rate on the contract in specific time intervals, but only when the exchange rate increased by a given percentage to avoid unnecessary calls. When the customer wants to update the exchange rate, because it decreased, he can call the update function by himself.

4.2 Contract Information

This section investigates what information has to be entered by the insurer at the beginning of a contract lifetime. The necessary information was carried together from research of related work. Beside defining the constraints of the contract, the entered information should also support the insurer to calculate a fair premium. Table 4.1 gives an overview of the main categories. Every needed characteristic of the customer is assigned to one of these categories. The six main categories are business information, contract constraints, company conditions, company security, company infrastructure and contract coverage.

The business information contains various standard information about the company itself, which are most likely already publicly known. This information is mainly needed to identify the company and it is not relevant for the premium calculation. A possible example for this category is the name of the company.
The basic conditions like the duration of the contract are stored in the contract constraints. It comprises every non-technical constraints of the contract. Parts of the information of the first category as well as all the information of the second category are not anonymized by the `Data Anonymizer` when a contract is created, because it is either needed to identify the customer company in the blockchain or simply to define the contract conditions in the smart contract. The following four categories are all completely anonymized and they are mainly relevant for the premium to pay. It is possible that a customer does not fill out some of these characteristics, for example if they are not known or not available at the moment. However, it is recommended to be as honest as possible, since it has a negative impact on the premium if too much information is missing.
The company conditions comprises all non-technical characteristics of the company and it mainly includes information about the business numbers of the company. Possible characteristics of this category are the yearly revenue or the number of employees in the company.

The next two categories are very related to each other and they encompass all technical characteristics of the company. With the information of these two categories the probability and partially the impact of a successful attack can be estimated. Hence the information of these categories is mainly intended to support the risk assessment. While the company security category describes different metrics about the security of the company and realized measures to improve the security, the company infrastructure includes all information of the company's hardware and software technologies as well as everything about critical parts of the software. For example the amount of critical data is also part of the company infrastructure.
Finally, in the contract coverage category, all the details about the coverage of the contract are stored. In an unlimited list every defined coverage is written down. For every attack the costs, which are covered, are defined as well as the coverage percent and possibly some other constraints of the specific coverage like a maximal indemnification of the insurer. Obviously the contract coverage is the most important part beside the risk assessment to calculate the premium. When the information of all categories is entered, the file content can be forwarded to the `Premium Calculator`, which then can calculate the premium.

Table 4.1: Contract Information

Category	Description	Example
Business Information	Standard Information about the company, which is not relevant for the premium, but which is needed to identify the company.	Company name, Company address
Contract Constraints	Information about the non-technical constraints of the contract, which have to be completely defined in each contract.	Duration of the contract, Payment frequency
Company Conditions	Non-technical information about the company's business number, which affect the premium.	Yearly revenue, Number of employees
Company Security	Information about the measures of the company to increase its cyber security as well as different metrics to measure it.	Risk assessment metrics, Attack history, Security software, Security training
Company Infrastructure	Information about the hardware and software used by the company.	Used technologies, Critical data amount
Contract Coverage	Information about what attacks and impacts are covered by the contract and by which conditions.	DDoS attack: business interruption, coverage: 50% Data breach: Third person damage, coverage: 100%

4.3 Intermediate Components

A fair calculation of the premium is very hard in cyber-insurance, mainly because the risks are hard to assess as mentioned in Subsection 2.1.3 a few times. However, through the previously collected characteristics and metrics from the customer, it should be possible to at least estimate the risk level and dependent on that to approximate a fair premium. This calculation is part of the `Premium Calculator`. In general, the `Premium Calculator` has to use the entered information and apply a predefined algorithm to calculate the premium. Figure 4.2 shows how the premium calculation can be proceeded. First, the risk level is estimated using the characteristics of the company's security and its infrastructure. Using the risk level along with the company conditions and the defined coverage the basic premium is calculated. This premium is adjusted regarding the contract constraints. For example when the contract lifetime is very high, the premium is lower in general.

As indicated in the architecture the `Premium Calculator` is contained at the insurer and at the customers system part. The logic of the `Premium Calculator` is the same in both part. In the system part at the customer side the `Premium Calculator` serves as a deciding tool and helps the customer to find a good configuration of the contract. At the insurer side, the `Premium Calculator` is more a supporting tool. The component calculates and forwards the premium to the `Data Anonymizer` using the previously entered file content.

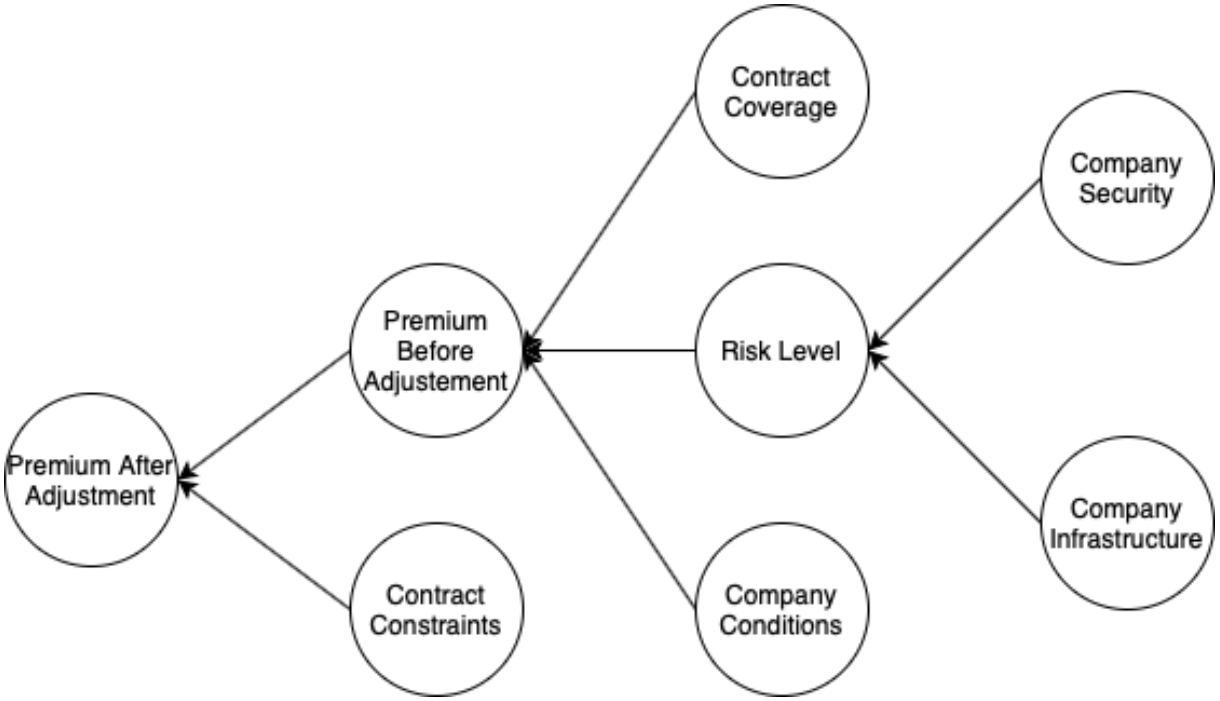

Figure 4.2: Premium Calculation

The `Data Anonymizer` then extracts and stores the information that should be hold private on a database and forwards the rest to the `Smart Contract Creator`. Only the hash of the file content, parts of the business information and the contract constraints are forwarded, while the rest is stored on the database. While it seems clear that the business information is needed to identify the customer and the contract constraints are needed to define the restrictions of the contract, the necessity of the hash is not so obvious. Since the data, which is not stored on the blockchain, is needed to check the contract agreements when a damage is reported, the file hash is stored to be able to track the effective file. For example considering that a damage occurs during the contract lifetime, the customer or the insurer can prove that the damage is covered respectively not covered by showing it in the file. Since the whole file is not publicly available both parties have to prove somehow, that they have the correct file. By storing the hash of the file on the blockchain the correctness of the shown file can be validated by comparing its hash with the hash stored on the blockchain. If the hashes match, the file is the correct one. Hence by storing the file hash, all information remains trackable even though it is not effectively stored. This is also the reason, why both parties have to store the file content in the database, because both have to be able to prove something with the actual file. This data anonymization is the main task of the `Data Anonymizer`.

After the private data is extracted and stored in the database by the `Data Anonymizer` the remaining information is forwarded to the `Smart Contract Creator`. The `Smart Contract Creator` then uses the information and dynamically creates the code of the smart contract. To do so, the forwarded information is entered in a prepared string, where the contract functions are already present. What functions there are available will be explained in the next section. After the code of the smart contract is created the `Smart Contract Creator` compiles the code and deploys it on the Ethereum blockchain. The

address of the deployed contract is stored in the database afterwards with the file hash as primary key, because the address has to be known later on, when the contracts functions are called. Therefore the contract address is also forwarded to the customer. Since the deploying address is assigned as the insurer address in the contract, it is necessary that the insurer deploys the smart contract. The customers address is passed in the constructor of the contract by the `Smart Contract Creator`. Like this, the identification of the different parties is inherently given by the system.

While the above explained components are essential the API mainly aims to simplify the usage of the system. Since the direct invocation of smart contract functions is not very intuitive, the API offers API functions that take on the access to the smart contracts. Beside this simplification, the API also interconnects the insurers and the customers system part by interchange some information. Mainly during the creation phase of the smart contract, the APIs send information to the API of the other system part, like for example the file content or the smart contract address. Finally, the API also supports the customer with the storage of the content in the database and offers additional possibilities to access the information of the database. However, without all these functions of the API, the SC4CyberInsurance system could theoretically still be used, but a lot more tasks would have to be done manually. Hence the API is also a very important component to achieve the goals of the project.

4.4 Smart Contract

At this point the contract is deployed on the blockchain and can be accessed by the insurer and the customer. Hence it has to be explained what the contract is actually able to do. In a very simplified and optimistic view the contract allows the customer to report damages, which the insurer has to pay, in the case the damage was covered in the agreements. In return, the customer has to pay a premium to the insurer to keep the contract valid. However, it is also possible that there is disagreement during the claims settlement and there are other use cases expected like the updating of the contract agreements. Hence there are plenty more functions that the smart contract has to offer. This section investigates what function are available and who can call them. In the final solution these functions will all be called over the API. Table 4.2 gives an overview of the most important functions. The calling actor C stands for customer and I for insurer.

The first shown function is already an exceptional case as it represents the constructor of the smart contract and it is only called once at the beginning when the contract is created. As previously mentioned, the constructor has to be called by the insurer, because the address of the caller is assigned to the insurer address in the contract. The constructor takes the address of the customer as parameter, which will be assigned to the customer address in the contract. All the following functions can only be called by the according addresses, which were assigned to the contract in the constructor.
The next two functions are interconnected to each other and they are both responsible if a deployed contract is still valid. The `paySecurity` function is a sort of protection for the insurer. As a contract is binding to both insurer and customer, the customer should not be able to cancel the contract by just not paying the premium anymore. Because of

that, the customer has to pay a security at the beginning of the contract lifetime, which will be refunded at the end of the validity. If a customer decides to still quit to pay the premium, the payed security is lost and the insurer can keep it. Before the security is not payed, the contract is not valid and none of its function can be called.

Table 4.2: Smart Contract Functions

Method	**Actor**	**Parameters**	**Description**
constructor	I	address customer_address	Assigns the correct addresses.
paySecurity	C	-	Pays the security converted in wei.
payPremium	C	-	Pays the premium converted in wei and increases the time of validity.
reportDamage	C	uint date, uint amoumt, string type_of_attack string logfileHash, uint damage_id	Creates a damage struct on the contract.
cancelDamage	C	uint damage_id	Cancels the damage with the ID.
acceptDamage	I	uint damage_id	Accepts the damage with the ID and pays out the reported damage.
declineDamage	I	uint damage_id, string reason, uint counter_offer	Declines a reported damage and creates an optional counter offer.
acceptCounterOffer	C	uint damage_id	Accepts the counter offer, which then automatically gets paid out.
declineCounterOffer	C	uint damage_id	Declines the counter offer.
resolveDispute	C	uint damage_id	Resolves a dispute about a reported damage, when a solution was found of the chain.
getAllReported-DamagesWithStatus	C & I	uint status	Returns all reported damages with the given state.
proposeToUpdate-Contract	C & I	uint new_premium, string new_file_hash	Makes a proposal to update the contract.
agreeToUpdate-Contract	C & I	-	Accepts the current proposal to update the contract.
declineToUpdate-Contract	C & I	-	Declines the current proposal to update the contract.
isNewProposal-Available	C & I	-	Returns true when a new proposal is available.
updateExchange-Rate	C & I	-	Updates the exchange rate of the contract.
annulTheContract	I	-	Annuls the contract if the premium was not payed for a while.

The `payPremium` function is responsible for the payment of the premium. Every time when the premium is payed, the contracts validity date is increased by the payment period. In

both functions the customer transfers ether from his account to the smart contract, which then can be earned by the insurer.

The functions `reportDamage`, `cancelDamage`, `acceptDamage`, `declineDamage`, `acceptCounterOffer`, `declineCoutnerOffer` and `resolveDispute` all serve to support the claims settlement. A reported damage has one of the following states: `New`, `Paid`, `UnderInvestigation`, `Dispute`, `Resolved` or `Canceled`.
Figure 4.3 shows the state of a damage regarding to the functions, which were called. First of all a damage is created with the function `reportDamage`. It takes as parameter the amount of damage that should be covered adjusted regarding the percent of coverage, the date and the type of the attack to check if the contract covers the incident and a hash of a logfile, where some details about the attack can be checked. The concept of the data anonymization of the logfile is the same as with the file content of the contract agreements. Beside that, a damage id is given to each damage, which will be used in all of the following functions to identify which damage is considered.

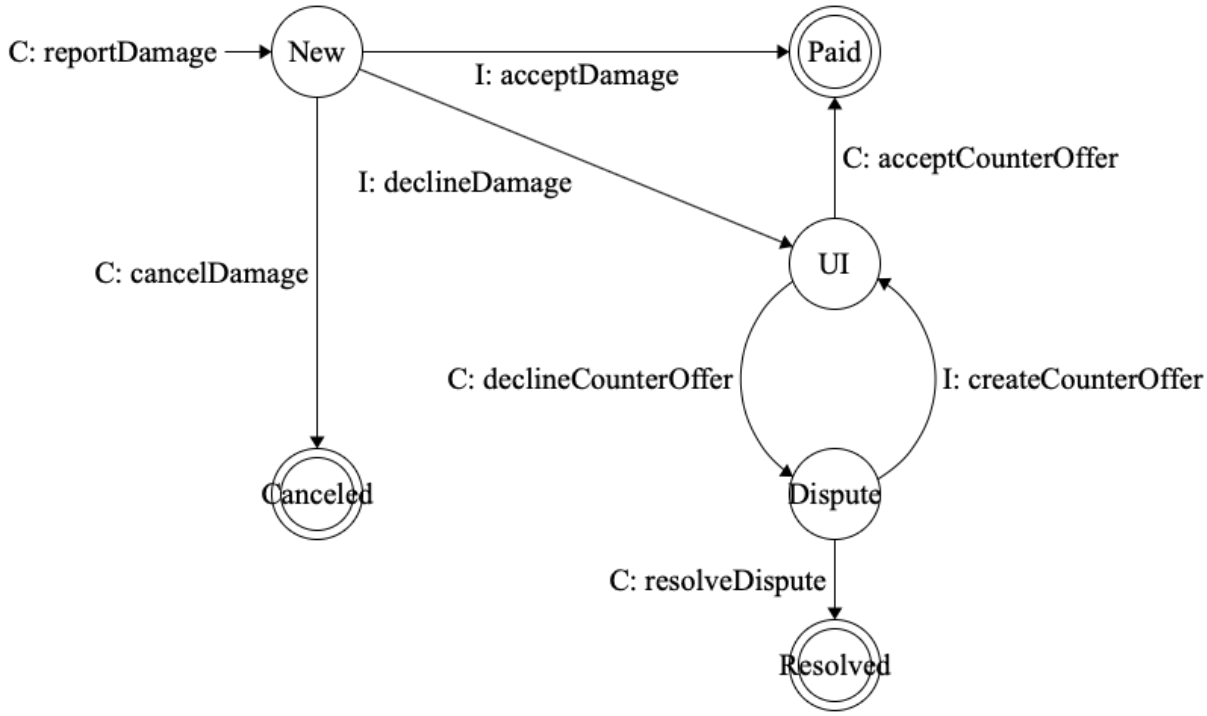

Figure 4.3: Claims Settlement State Diagram

If a damage was reported by accident, the customer can cancel the damage, which then will change its status to `Canceled`, which is an ending state and nothing has to be done anymore. If the damage is not canceled the insurer can either accept the damage or decline it and offer a counter offer. When the insurer accepts the damage the reported amount is paid out automatically to the insurer and the state changes to `Paid`, which is also an ending state. If the contract has a lower balance then the value to pay out, the insurer has to transfer coins to the contract when accepting a damage. In the other case, the insurer declines the reported damage and creates a counter offer. When the damage

is declined a reason has to be provided along with the amount of the counter offer. If the insurer does not want to make a counter offer, the parameter is set to 0. The state then changes to `UnderInvestigation`.
At this stage, the customer can either accept the counter offer or decline it. When he accepts it, the amount of the counter offer is automatically paid out and the state change to `Paid`. If it is declined the state changes to `Dispute`. This state means that yet no agreement could be found regarding the damage. Either the insurer creates a better counter offer or the two parties have to solve the dispute off-chain. In the second case a third-party, which can solve the dispute, may be taken into consideration. When the two parties could find an agreement of-chain the customer can resolve the dispute, which makes the state change to `Resolved`. This state is an ending state as well. When there are a lot of damages in the `Resolved` state, this means that either the insurer is not willing to pay damages he actually should or the customer just reports a lot of damages, which are not covered by the contract. However, the states of damages from different customers can indicate if a specific insurer is trustworthy. Using the function `getAllReportedDamagesWithStatus` all reported damages with a specific status can be returned.

The function `proposeToUpdateContract`, `agreeToUpdateContract` and `declineToUpdateContract` handle the use case that it should be possible to update the contract agreements. Both the insurer and the customer can propose new contract agreements by updating the input file. They can enter the updated file to the system and over the `Premium Calculator` a premium is calculated and proposed. The hash of the new file content is proposed as well. The other party then can accept or decline the proposed updated contract agreements by calling the function `agreeToUpdateContract` respectively `declineToUpdateContract`. While an updated contract is proposed it can not be proposed other agreements and the current proposal has to be accepted or declined first. Like this no party can try to trick the other party by overriding the new contracts agreement many time and try to make the other party accept new conditions, which they did not want to accept. To find out if a proposal is available on a contract, the function `isNewProposalAvailable` returns if a proposal is available.

The function `updateExchangeRate` updates the exchange rate stored on the contract by calling an external oracle. The `Exchange Rate Updater` on the insurer side calls this function at a specific time interval but only when the rate changed by a specific percentage, since otherwise a lot of expensive calls would be unnecessarily executed. Besides the `Exchange Rate Updater`, the function can be called by both the insurer and the customer directly whenever they want. Finally, the function `annulTheContract` gives the insurer the possibility to annul a contract, when the customer did not pay the premium for a while.

Chapter 5

Implementation

In Chapter 4 it was described how the design of the SC4CyberInsurance system looks like. In this chapter, a prototype implementation of the system is shown and explained. First, an overview of the used technologies is given in Section 5.1. In Section 5.2 the used data format and the structure of the input file is shown. Afterwards the concrete implementations of the components in the system architecture shown in Figure 4.1 are explained in Section 5.3. The implementation of the smart contract is investigated in Section 5.4. Last but not least, the implemented API is described in Section 5.5.

5.1 Implementation Overview

To have a general overview of the implemented solution it is important to consider the architecture and think of the technologies, that could be used for the specific components. The following list shows for each component, what technology is used in the prototype.

- Contract Information: Mapped to a JSON file
- API: Python API using Flask
- Premium Calculator: Python
- Data Anonymizer: Python
- Smart Contract Creator: Python
- Exchange Rate Updater: Python
- Contract DB: SQLite
- Smart Contract: Solidity
- Blockchain: Ethereum with Ganache

To store the contract information the JSON format was picked, since it is a very common standard data format and it uses human-readable text to store and transmit data objects, which is adequate for the use case as it can easily be processed afterwards. The customer has to enter the contract information in a predefined JSON-file. When he completely filled out the file, it is stored on the system and the path of the file has to be entered in the API, that its content can be processed afterwards. The API is a simple Python API using flask, which is a python web framework. Flask is used because it does not require particular tools or libraries and hence it is very applicable to be used in a prototype. To improve the customer experience here, the process of entering the information could be integrated into a nice looking interface instead of simply filling out a file. A web based interface that access the API would be nice to have for a final solution of the system to improve the usability and hence the customer experience. However, for this prototype it was decided to only implement an API and to keep the focus on the actual functionality. The `Premium Calculator`, the `Data Anonymizer`, the `Smart Contract Creator` and the `Exchange Rate Updater` are implemented in python, because Python offers different libraries to easily interact with a smart contract and it is best to keep as less different languages as possible to reduce the complexity. The database that is used to store the private data is a SQLite database. SQLite is suitable for the prototype, because the database engine is self-contained and can be directly integrated into the application without the usage of any other server software. Considering that, SLQlite is still full-featured and has a high-reliability. Hence it is perfectly appropriate for the prototype. The smart contract is written in Solidity source code, which is the most common language for smart contracts and especially used in combination with the Ethereum blockchain, which is also the blockchain used in the project.

5.2 Structure and Content of Input JSON File

As indicated in the previous section, the needed customer information, which was described in Section 4.2, is mapped into a JSON file to be transmitted and processed later on. Using the JSON format the described categories of information can be separated having an individual key per category with a JSON object as value. All the according information then is stored in the JSON object of the specific key. Hence each category has an own paragraph in the file. For the categories contract contraints, company security and contract coverage the paragraphs of the JSON file will be shown in a shortened way in the following. The complete JSON file structure is available in Appendix C of this thesis. The values entered in the examples are randomly and do not belong to a specific company.

Listing 5.1 shows the JSON part of the contract constraints category. The information, which is stored there, is very important, as it defines the non functional restrictions of the contract. The smart contract creator mainly uses this information to dynamically create the smart contract. Hence all the characteristics will be available on the blockchain. The contract constraints contain information about the time of validity of the contract in the `startDate`, `endDate` and `duration` keys. One of these keys is redundant and hence they should fit to each other. How many times per year the customer has to pay the premium

is stored in the `paymentFrequencyPerYear` key. In the `cancellationPenaltyInPercent` key it is stored how many percent of the premium the customer has to pay as security at the beginning of a contract lifetime. The security is lost when the customer stops to pay the premium and hence it is like a cancellation penalty. Of course, as lower the penalty is, as higher the premium is because the insurer takes greater risks. The information stored here is not explicitly special for a cyber-insurance contract and similar characteristics has to be defined in each insurance contract.

```
1  "contract_constraints": {
2    "startDate": "01.01.2021",
3    "duration": 3,
4    "endDate": "31.12.2023",
5    "paymentFrequencyPerYear": "half-yearly",
6    "cancellationPenaltyInPercent": 50
7  }
```

Listing 5.1: Contract Constraints in JSON Format

The JSON section of the company security category is shown in Listing 5.2. As this information is the most important factor to calculate the risk level of the company, which is very important for the final premium, it is important to understand what characteristics are entered there. The section consists of different lists of security metrics and security measures of the company. Each list can contain zero or many entries. However, to have a better overview always one entry is shown per list in Listing 5.2. First of all, a list of risk assessment metrics can be entered, that are assessments of the company's security executed from an external party outside of the company. Such an assessment generally contains the name or type of the assessment and its result. The example shown in the listing is a metric that assess the known vulnerabilities. This metric is classified to be medium, which is around the average of all the assessed companies. Such an external assessment of the company can be assumed to be trustworthy, mainly when it is an acknowledged assessment from a reliable third-party.
Besides that, the history of successful attacks also indicates how probable it is that the company will be successfully attacked in the future. As more attacks happened in the past, as more attacks will probably happen in the future. For each attack different characteristics are described, that also help to estimate the impact of the attack. In a third list, the current available security software is entered. This list contains every security software available in the company, that supports to block an attack, to detect and monitor them and also to mitigate them. Finally, all security trainings and workshops are listed that were undertaken in the company. This can comprise security training of the customers as shown in Listing 5.2, but also other security workshops, as a penetration test to detect current problems.
Considering all these characteristics, it is very hard to automatically calculate a security level because some of them are just plain text and it is hard to process them without manual support. For example in one risk assessment metric a high value is bad, while in another one a high value is good. How should that be processed automatically then. Generally, that problem can be solved by having predefined metrics with predefined values that can be selected by the customer or just left empty. Like this all metrics can be known by the algorithm and hence be considered correctly. Using a interface this can be

supported very well by using a drop down list or something similar. However, as it will be explained afterwards in Section 5.3, the focus of the project does not lie in a perfect risk level assessment and premium calculation and hence it is enough here to just summarize all the necessary information needed and move the improvement to possible future work.

```
"company_security": {
"risk_assessment_metrics": [
  {
    "name": "Known vulnerabilities",
    "result": "medium"
  }
],
"attacks_history": [
  {
    "type": "DDoS",
    "date": "09/12/2019",
    "time_to_recovery": "15 hours",
    "details": "Mirai Botnet",
    "mitigated": false
  }
],
"security_software": [
  {
    "name": "Dynatrace",
    "type": "monitoring"
  }
],
"security_training": [
  {
    "name": "ZYX security certificate",
    "type": "education of employees",
    "date": "26/04/2020",
    "provider": "International CyberSecurity Institute"
  }
]
}
```

Listing 5.2: Company Security in JSON Format

Besides the risk level, the contract coverage is the most important factor for the premium calculation. Therefore also this part is investigated and shown in Listing 5.3. In this passage all the coverage are listed up individually as a combination of what costs are covered in which attack type. It was decided to separate the attack types first, and then to list up all costs that are covered for this specific attack type. In the shown example the business interruption and the cyber extortion is covered in the case of a DDoS attack, while third person damage is covered when a data breach occurs. Each coverage is described by the name, the coverage ratio, the deductible and the maximal indemnification. The deductible and the maximal indemnification are optional to be set. While the name

describes what cost is covered, the coverage ratio determines the percentage of the cost that is covered by the insurer. When it is equal to 100, the whole damage is covered, expect when one or both of the other two optional values is set as well. The deductible determines how many the customer has to pay in each individual attack before the insurer pays any expenses. Small attacks with caused costs under the deductible value are not covered at all. At the other side, the maximal indemnification defines the maximum amount of money that is paid to the customer per attack. If the coverage ratio is small and the deductible and the maximal indemnification are set, it can be assumed that the premium is quite low.
Compared to the company security this information is easier to automatically process and to work with it in the premium calculation as most of the values are numbers with a strict ordering. However, a predefined list of attacks and costs is also advantageous or even necessary, since otherwise it can be entered anything as the name of a cost and that makes it impossible to process the coverage by the premium calculation algorithm.

```
    "contract_coverage": [{
      "name": "DDoS",
      "coverage": [{
          "name": "business Interruption",
          "coverage_ratio": 100,
          "deductible": 1000,
          "max_indemnification": 300000
        },
        {
          "name": "cyber extortion",
          "coverage_ratio": 100,
          "deductible": 1000,
          "max_indemnification": 300000
        }
      ]
    },
    {
      "name": "data breach",
      "coverage": [
        {
          "name": "third person damage",
          "coverage_ratio": 100,
          "deductible": 1000,
          "max_indemnification": 300000
        }
      ]
    }
  ]
```

Listing 5.3: Contract Coverage in JSON Format

5.3 Intermediate Components Implementation

Now that it is defined, how the contract information is forwarded the implementation of the intermediate components can be analyzed. As mentioned before, the `Premium Calculator` is written in python. Using the JSON format the `Premium Calculator` can access all the entered information very easily and apply its algorithm. Since the focus of this work did not lie in the process of the premium calculation only a very simple algorithm is defined in the SC4CyberInsurance system yet and it is not applicable in practice. The algorithm simply models the dependencies on some of the most important characteristics. For example as more coverage are defined as higher the premium is or as better the level of security as lower the premium is. Hence the implemented `Premium Calculator` is only a very simple proof of concept of a `Premium Calculator` that could be used in production code. However, the concept behind the implemented `Premium Calculator` is practically applicable and hence it is enough to show that the concept of the SC4CyberInsurance system works.

After the premium is calculated the entered information is forwarded to the `Data Anonymizer`. The implementation of the `Data Anonymizer` is very short and quite simple. First, the SHA256 hash of the JSON content is calculated and then the JSON content is stored in the database with the hash as primary key. The information, which is needed later in the `Smart Contract Creator`, is stored in an class instance called `ContractInformation`. When initializing the class instance the needed fields are read out of the JSON content, which is given as input parameter. This class instance then is only used temporarily to store the necessary fields and to forward the information from the `Data Anonymizer` to the `Smart Contract Creator`. It contains the attributes `company_name`, `start_date`, `end_date`, `payment_frequency`, `cancellation_penalty`, `premium` and `json_hash`.
Hence the `Smart Contract Creator` can create and deploy a smart contract using the `ContractInformation` instance and the addresses of the insurer and the customer. The rest of the JSON content is not needed at this point anymore. The `Smart Contract Creator` dynamically creates the contract code by inserting the attributes of the `ContractInformation` instance in a predefined contract code and then compiles the resulting code. Since the contract calls an oracle to get the exchange rate, the code of the oracle has to be compiled previously as well. As oracle the Provable oracle is used, which is a widely used oracle service for smart contracts and blockchain applications [34]. The oracle can access the exchange rates provided by third-parties. After the contract code is created it is deployed on the Ethereum blockchain. The contract address then is stored in the database along with the contract ABI, that the contract can be accessed later again.

5.4 Smart Contract

The smart contract is the key component in the SC4CyberInsurance system and to fulfill its tasks it has to offer all the functionalities explained in Section 4.4. This section shows how the smart contract is set up and gives an introduction into the implementation. As indicated before the smart contract is implemented in the Solidity language.

5.4.1 Attributes of the Smart Contract

Table 5.1 gives an overview of the attributes that are used in each contract. They will be explained more deeply in the following. By understanding the idea of all the attributes, it is much easier to understand the final implementation.

Table 5.1: Smart Contract Attributes

Name	**Type**	**Purpose**
customer_name	string	Stores the name of the customer. Used to identify the customer.
start_date	uint	Defines from when the contract is valid.
end_date	uint	Defines until when the contract is valid at most.
valid_until	uint	Defines until when the contract is valid currently. It is increased when a premium is paid by the customer.
payment_frequency	uint	Defines the time interval, in which the customer should pay the premium.
valid	bool	Defines if the contract is currently valid.
cancellation_penalty	uint	Defines the percentage of the premium that the customer has to pay as security.
initial_premium	uint	Stores the initial premium of the contract. This value is never updated.
premium	uint	Stores the premium of the contract. This value may be updated.
json_hash	string	Stores the hash of the json content.
proposed_new_ premium	uint	If there is a proposal, it stores the proposed new premium.
proposed_new_ json_hash	string	If there is a proposal, it stores the hash of the proposed json content.
new_proposal_ available	bool	Defines if a proposal is currently available.
address_of_ accepting_party	address	Stores the address of the account who is allowed to accept or decline the proposed updated conditions of the contract.
reported_damages	mapping (uint => Reported_ Damage)	Stores all reported damages.
count_of_damages	uint	Stores the number of reported damages.
list_of_damage_ids	uint[50]	Stores all available damage IDs.
exchange_rate	uint	Defines the exchange rate to be used, when money is transferred.
customer_address	address payable	Stores the address of the customer.
insurer_address	address	Stores the address of the insurer.

The restrictions of the validity of the contract is defined in the attributes `start_date`, `end_date`, `valid_until` and `valid`. Since the Solidity language does not support a date type, the dates are stored as `integers` that represent a time in seconds. While the attributes `start_date`, `end_date` and `valid_until` represent specific dates, the attribute `payment_frequency` stores the time interval in seconds that the customer should pay the premium. At a specific time a contract is valid from the value in `start_date` until the value in `valid_until`. When a damage is reported, the time of the attack has to be inside the time of validity, otherwise it is not covered. At the beginning the `valid_until` attribute is set equal to `start_date`. Each time when a premium is paid, the date in `valid_until` is increased by the time period in `payment_frequency` until it has reached the date in `end_date`. Then no more premiums can be paid anymore as the contract conditions are met. Before any premium can be paid, the contract has to be valid and the `valid` attribute has to be `True`, which is reached by paying the security. The amount, which has to be paid to the insurer as security, is the product of the value in `initial_premium` and the percentage in `cancellation_penalty`. Since the security is payed back when the contract conditions are met, the initial premium has to be known at the end, in order to pay the correct amount back. Hence this value is kept constant, while the actual premium can vary.

During the lifetime of the contract both the insurer and the customer can propose new contract conditions by updating the JSON content and offer the new conditions to the counterparty. When doing that, the new values are entered in the attributes `proposed_new_premium` and `proposed_new_json_hash`.
The attribute `new_proposal_available` will be set to true, which means that a proposal is available and no other proposal can be entered currently. If the counterparty, whose address is checked with the value in the attribute `address_of_accepting_party`, accepts the new conditions, the new values are assigned to the attributes `premium` and `json_hash`, which define the current contract conditions.

```
1     struct Reported_Damage {
2         uint date_of_damage;
3         uint amount_of_damage;
4         StatusDamage status;
5         uint damage_id;
6         string type_of_attack;
7         string logfile_hash;
8         string decline_reason;
9         uint counter_offer;
10    }
```

Listing 5.4: Reported_Damage Struct

Each reported damage is stored in a `Reported_Damage` struct. The setup of such a struct is shown in Listing 5.4. The attributes `date_of_damage`, `amount_of_damage`, `damage_id`, `type_of_attack` and `logfile_hash` are given by the customer when a damage is reported and they were described previously in Section 4.4. In the case a damage is declined and a counter offer is given by the insurer, the attributes `decline_reason` and `counter_offer` are assigned with the respective values. The status of a damage is tracked by the attribute `status`, which can have the values `New`, `Paid`, `UnderInvestigation`, `Dispute`, `Resolved` or `Canceled` as described previously in Section 4.4 as well. All reported damages are mapped by their `damage_id` into the attribute `reported_damages`. The attribute

`list_of_damage_ids` stores all ids and the attribute `count_of_damages` counts the number of damages. The attribute `exchange_rate` defines the exchange rate between the currency defined in the contract, in this case euro, and the cryptocurrency. The value is updated on demand either by the customer or by the insurer. The `Exchange Rate Updater` component on the insurer side regularly updates the exchange rate, when it has changed by a defined percentage. The new exchange rate is assigned using an oracle and hence using the service of a third party.

The attributes `customer_name`, `start_date`, `end_date`, `payment_frequency`, `cancellation_penalty`, `premium`, `initial_premium` and `json_hash` are assigned dynamically by the `Smart Contract Creator` using the input information of the customer as described in Section 5.3.
The addresses of the insurer and the customer stored in the attributes `insurer_address` and `customer_address` are assigned in the constructor, which is shown in Listing 5.5. These addresses are very important as they are responsible for the identity management and access control of the contract latter on. For example, only the address that is stored as the customer can report damages and receive the insurance payment and only the insurer address can take money from the contract. If this is not given, the contract does not work at all. Additionally, it has to be assumed that the owner of the insurer address can be identified because otherwise it is not known who has to pay damage. Every customer could create a contract by himself, report some damage and then legally complain that the insurer did not pay the damage. This would make the system collapse. In the constructor also the initial exchange rate is set. For testing reasons a fixed value of 320 is used here to make the test results deterministic. However, in practice the initial exchange rate should be set using the external oracle.

```
1     constructor(address payable company_address) public {
2         insurer_address = msg.sender;
3         customer_address = company_address;
4         exchange_rate = 320;
5     }
```

Listing 5.5: constructor

In the `Smart Contract Creator` the smart contract code is dynamically created and afterwards the constructor is called, which assigns the addresses. Afterwards the contract is ready to be used. It is interesting to look at the concrete implementation of the most important functions now.

5.4.2 Functions in the Smart Contract

The functions, which have to be implemented in the contract, were shown in Table 4.2 and explained in Section 4.4. In this subsection the concrete implementation of some of the most important functions will be shown and explained. Since the concept of implementing the functions is similar for all the functions it is enough to investigate some of them to better understand the others. As described in Section 2.2.2, every function call that modifies the state of the contract is included in an own blockchain transaction.
In Listing 5.6 the code of the function to pay the premium is shown. Since the customer

wants to pay the premium in this function, it is necessary that money is transferred to the contract. In the function header in line 1 it can be seen that the function is `payable`. This means that if the caller sends money in the transaction of the function call, the sent money is transferred to the smart contract.
Considering the function body, most of it consists of restrictions that define if the caller is allowed to call this function. First of all, the restriction in lines 2 - 5 checks if the address of the message sender is equal to the address of the customer stored in `customer_address`. If it is not, the red printed error message in line 4 is returned. The verification of the correct message sender is always the first restriction in every function, since the following code should not be executed if the message sender is not allowed to call the respective function. When the message sender is indeed the stored customer the function proceeds and defines the premium to pay. As the premium is defined in euro it has to be converted to wei first, which is the smallest denomination of ether, the cryptocurrency coin used on the Ethereum blockchain. Since the exchange rate can change permanently, it has to be read from outside of the blockchain using an oracle. The converting to wei is done in the function `convertEuroToWei`, which finally returns the converted value of the input parameter.

```
1   function payPremium() public payable {
2       require(
3           customer_address == msg.sender,
4           "Only the registered customer can pay the premium."
5       );
6       uint premium_in_wei = convertEuroToWei(premium);
7       // if its the last premium, the already paid security is
            subtracted from the premium
8       if(valid_until + payment_frequency == end_date){
9           uint security_in_wei = convertEuroToWei(initial_premium) /
                100 * cancellation_penalty;
10          premium_in_wei = premium_in_wei - security_in_wei;
11      }
12      require(
13          msg.value >= premium_in_wei,
14          "Not enough Ether provided to pay the premium."
15      );
16      require(
17          valid,
18          "Have to pay the security first, because otherwise the
                contract is not valid."
19      );
20      require(
21          valid_until < end_date,
22          "End date of the contract is reached. No need to pay
                additional premiums."
23      );
24      increaseTimeOfValidity();
25      //refund the extra paid
26      msg.sender.transfer(msg.value - premium_in_wei);
27  }
```

Listing 5.6: Smart Contract Function payPremium

In lines 8 - 11 it is checked if the premium to pay is the last premium to pay in the contract. If it is, the customer fulfilled its duty of paying the premiums until the end

of the contract and the security will be refunded. Hence in this case the value of the security is subtracted from the premium to pay. It is possible, that the resulting value will be negative, which means that the customer actually gets some money from the contract. This can be the case when the premium and hence the security was very high at the beginning and during the lifetime of the contract the premium sank due to updated conditions. After the premium to pay was calculated it is checked if the transaction sent by the customer contains enough ether to pay the previously calculated premium. If not, another error message is returned.
When enough ether is provided, in lines 16 - 23 it is investigated if the contract is already valid, which means that the security was payed before, and if the contract did not reach its ending date. If one of the conditions does not meet, a suitable error message is returned. When all checks were passed, the `valid_until` attribute of the contract is increased in the function `increaseTimeOfValidity` in line 24. Finally, remaining coins that overlapped the premium to pay are sent back to the customer in line 26. When the premium to pay is negative, more money is transferred back than sent by the customer.

```
1  function reportDamage(  uint date_of_damage,
2                          uint amount_of_damage,
3                          uint damage_id,
4                          string memory type_of_attack,
5                          string memory logfile_hash) public{{
6      require(
7          customer_address == msg.sender,
8          "Only the registered customer can report a damage."
9      );
10     require(
11     date_of_damage > start_date && date_of_damage <= valid_until,
12             "The contract was not valid at the date of damage."
13     );
14     //check if the id is already given away
15     require(
16         reported_damages[damage_id].amount_of_damage == 0,
17         "Already exists a damage with the selected id."
18     );
19     reported_damages[damage_id]
20         = Reported_Damage(  date_of_damage,
21                             amount_of_damage,
22                             StatusDamage.New,
23                             damage_id,
24                             type_of_attack,
25                             logfile_hash,
26                             "",
27                             0);
28     list_of_damage_ids[count_of_damages] = damage_id;
29     count_of_damages = count_of_damages + 1;
30     //Possibly allow an automatic payout
31     if(amount_of_damage < premium && count_of_damages < 4){
32         automaticPayOut(damage_id, false);
33     }
34 }
```

Listing 5.7: Smart Contract Function reportDamage

The code of the function to report a damage is shown in Listing 5.7. It takes the date

the damage happened, the amount of damage, the damage id, the type of attack and the logfile hash as input parameters as described in Table 4.2.
As in the `payPremium` function first some restrictions are checked. In lines 6 - 9 it is verified again if the sender of the message is the customer. After that, it is checked if the date the damage occurred is actually covered by the contract. To do so, the date of the damage is compared to the contract attributes `start_date` and `valid_until` in line 11. Since a damage should not be overwritten, it has to be ensured that there exists no damage yet, with the same damage id as the new reported damage. This check is done in lines 15 - 18.
When the restrictions are met a `Reported_Damage` struct is created and mapped by the id into the contract attribute `reported_damages`. The struct is created with the values passed by the function and default values for the counter offer. The current status of the damage is set to `New`. The new damage id is added to the contracts list of ids in line 28 and the count of reported damages is increased by 1 in line 29. Theoretically, it is possible to automatically pay out some damages without a check from the insurer as shown in line 31 - 33. For example when the damage amount is quite small and the last reported damages were all covered. This would reduce the administrative effort of the insurer and increase the customers satisfaction. However, it offers an additional possibility for fraud and the conditions when automatic payment is possible should be chosen very well. The insurer afterwards also should be able to challenge an automatically paid out damage when it was fraudulent. Hence the lines 31 - 33 are not mandatory to be included in the contract, but they offer an additional possibility to the insurer. The code of the `automaticPayOut` function that is called in line 32 is shown in Listing 5.8.

```
1   function automaticPayOut (uint damage_id, bool is_counter_offer)
        private{
2       StatusDamage current_status = reported_damages[damage_id].status
            ;
3       require(
4           current_status != StatusDamage.Paid && current_status !=
                StatusDamage.Canceled && current_status != StatusDamage.
                Resolved,
5           "This damage is already paid, deleted or resolved otherwise.
                "
6       );
7       uint payOutInWei = 0;
8       if(is_counter_offer){
9           payOutInWei = convertEuroToWei(reported_damages[damage_id].
                counter_offer);
10      }else{
11          payOutInWei = convertEuroToWei(reported_damages[damage_id].
                amount_of_damage);
12      }
13      require(
14          address(this).balance >= payOutInWei,
15          "Not enough Ether available in the contract."
16      );
17      customer_address.transfer(payOutInWei);
18      reported_damages[damage_id].status = StatusDamage.Paid;
19  }
```

Listing 5.8: Smart Contract Function automaticPayOut

The function takes as parameter the id of the damage and a `boolean` named `is_counter_offer`. The `boolean` defines if the value of the counter offer should be payed out or the value of the initially reported damage. As this function should not be called from outside of the contract it is assigned to be private. The restriction in line 3 - 6 checks if the damage was already payed out, cancelled or otherwise resolved to protect the insurer of unintended double pay-out. If the damage status is not in an ending state the amount to pay out is calculated in line 7 - 12. Considering the parameter `is_counter_offer`, either the initially value of the reported damage or the value of the counter offer is converted into wei using the exchange rate returned from the oracle again. Afterwards it is checked if the contract currently has enough balance to pay out the damage. In the case, that there is not enough balance the insurer is notified by the error message in line 15. Otherwise the calculated amount is transferred to the customer address stored in the contract and the status of the damage changes to `Paid`.

Listing 5.9 shows the code of the acceptDamage function. It takes as parameter the id of the reported damage that should be accepted. The function body is very simple and it consists only of a verification that the message sender is the insurer and the function call of the `automaticPayOut` function. The verification is necessary as otherwise the customer could accept its damage by himself. Since the value of the initial reported damage should be payed out the parameter `is_counter_offer` is set to `false`. The code of this function is mainly shown because it illustrates how simple the smart contract functions can be built. Most of the functions consists of some restrictions at the beginning and then mostly only some lines that change the state of a reported damage and maybe transfer some money. This simple setup is used in every other key function of the contract.

```
1     function acceptDamage(uint damage_id) public payable{
2         require(
3             insurer_address == msg.sender,
4             "Only the insurer can accept a reported damage."
5         );
6         automaticPayOut(damage_id, false);
7     }
```

Listing 5.9: Smart Contract Function acceptDamage

5.5 API

The API mainly serves to facilitate the interconnection with the blockchain for both the insurer and the customer. Theoretically, this part could be left out in a prototype and the blockchain could be accessed directly. However, the API takes also some other parts in the system, which are mandatory and should not be left out. The API handles all the connections with the database and provides different database requests to show summarized information of specific contracts. Besides that, it is responsible for the information exchange between the insurer and the customer outside of the blockchain.
To understand better the benefit of the API Table 5.2 gives an overview of the functions the API provides. Since the API on the insurer side is different from the API on the customer side, there are 3 groups where an API function can be assigned to. While the

`Common` functions are available on both sides, the `Insurer` and the `Customer` functions are available only at the insurer respectively at the customer side. As indicated before, there are many API functions that simplify the connection with the blockchain. These functions just call the fitting function on the blockchain with the given parameters. The functions, which just set up this connection, are marked with `true` in the SCFunction column in Table 5.2. Since these functions just take access to the blockchain, they are not explained any deeper here. The other functions, which are marked as `false` there, can also access the blockchain, but they do additional tasks in the API.

Table 5.2: API Functions

Group	**Function Name**	**Parameter**	**SCFunction**
Common	identify	String address	false
Common	calculatePremium	String jsonFile	false
Common	getCoverage	String jsonHash	false
Common	getAvailableContracts	-	false
Common	getDamagesOfCurrentContract	string status	false
Common	getAllDamages	status	false
Common	useContract	string jsonHash	false
Common	proposeToUpdateContract	String newJsonFile	true
Common	agreeToUpdateContract	-	true
Common	declineToUpdateContract	-	true
Common	isNewProposalAvailable	-	false
Common	getContentOfProposal	string hash	false
Common	checkProposal	string hash, string fileName	false
Insurer	deployContract	String jsonFile	false
Insurer	getContractAddress	String jsonHash	false
Insurer	getContractABI	String jsonHash	false
Insurer	acceptDamage	int id	true
Insurer	declineDamage	int id, string reason, int counterOffer	true
Insurer	getLogFileContent	string logFileHash	false
Customer	createContract	String jsonFile	false
Customer	getJsonContent	String jsonFile	false
Customer	getCustomerAddress	-	false
Customer	paySecurity	float ether	true
Customer	payPremium	float ether	true
Customer	reportDamage	String date, int id, String typeOfAttack, String hashOfLogFile	true
Customer	acceptCounterOffer	int id	true
Customer	declineCounterOffer	int id	true
Customer	resolveDispute	int id	true
Customer	getLogContent	string logFileHash	false

These functions will be explained in the following. The remaining functions of the `Common` group will be investigated first.
The API function `identify` is responsible for the identification of the users. It includes an approach to solve the identification problem that it is not known if a current user is an insurer or a customer. By giving the own address as input it is checked if the given address is stored in a predefined list of insurer addresses. If it is, the user is identified as an insurer otherwise as a customer. Only a insurer should then be able to deploy contracts. Since the address is given as a parameter and everything could be entered the concept of course does not really work like this. However, considering that the function is extended and that it also checks the credentials of the entered address, the problem of fake insurers could be solved. Hence the functions is currently only integrated to show the idea behind the identification but does not really solve the issue.
As both parties have to be able to calculate the premium the `calculatePremium` function is available in the `Common` group. Thanks to this function, the customer can repeatedly adapt the conditions and calculate the premium until he has found the best fitting contract conditions. The functions `getCoverage` and `getAvailableContracts` provide additional functionalities to read specific contract information out of the database. The `getCoverage` function lists up all coverage of a specific contract. It simplifies and accelerate the use case when a party want to check if a specific case is covered in a contract without having to access and read the whole JSON file. The second function gives an overview of all currently stored contracts along with their JSON hash. Both these functions help to have a quick summary of the contracts and their content. The functions `getDamagesOfCurrentContract` and `getAllDamages` allow both parties to list up all damages with a given status stored in a specific contract or in any contract they participate. This can especially help the insurer to detect new damages reported in any contract.
When either the insurer or the customer wants to call one or many functions of a specific contract, he has to call the `useContract` function first, which assigns the contract with the given hash as the currently used contract. All functions that call smart contract functions will then access this contract until a new one is assigned.

The remaining functions `deployContract`, `getContractAddress` and `getContractABI` from the insurer API and the `createContract`, `getJsonContent` and `getCustomerAddress` from the customer API are all responsible for the creation and deployment of a smart contract on the blockchain. Since it would be cumbersome to call all these functions individually to create a contract, they are interconnected and call each other. When a customer calls the `createContract` API function the creation is triggered and all the other functions are called at some point. Figure 5.1 shows the flow of the process when a contract is created.
First, when the customer is happy with the contract conditions, he calls the function `createContract` on his API with the path to the JSON file as input. The function directly forwards the path to the insurer API and triggers it to deploy the contract by calling the `deployContract` function. Since the specific content of the file and the address of the customer is needed to deploy the contract, the `deployContract` function call the functions `getJsonContent` and `getCustomerAddress` on the customer API. Using that information the contract can be deployed by the components explained before in Section 5.3. When the deployment at the insurer side is finished, the customer API has to get the contract address and the ABI of the contract by calling the functions `getContractAd-`

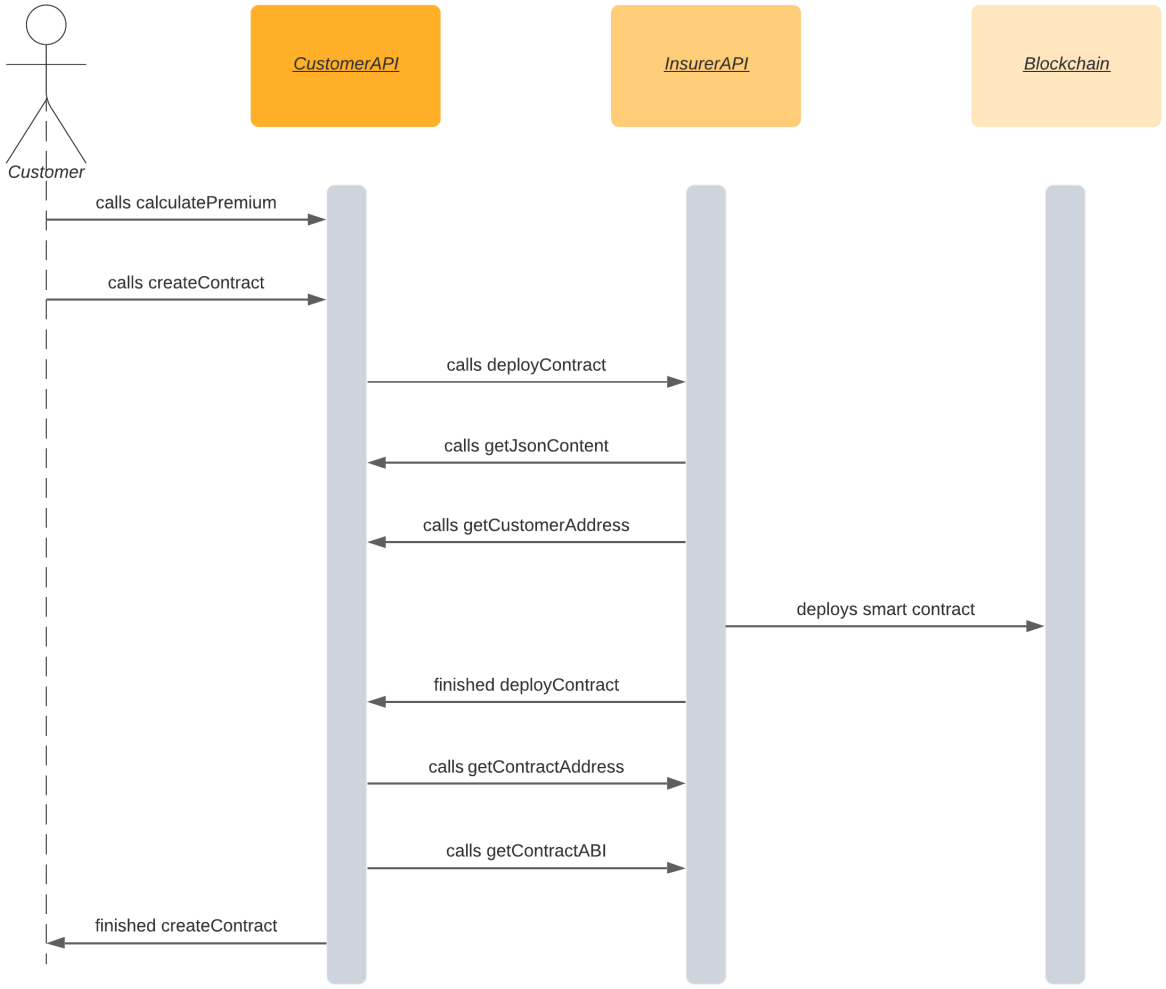

Figure 5.1: Creation of a Contract

`dress` and `getContractABI` on the insurers API. The returned information is then stored in the database to be reused later. After that, the creation is finished and the contract is ready to be used. A similar interconnectivity between the APIs is given when a new proposal is available. First using the `isNewProposalAvailable` it is found out, if on any available contract a proposal is available. If there is, the content of the proposal can be checked by calling the `checkProposal` function, which automatically calls the `getContentOfProposal` function at the proposer side. The response is then written to a file that can be checked by the approving party. If the proposal is accepted the database will be updated accordingly with the new valid JSON file. Finally, another interconnectivity of the API is needed to read the content of a log file that describes a damage. The function `getLogFileContent` at the insurer API calls the `getLogContent` at the customer API.

Considering the interconnectivity of the APIs, it has to be ensured that only the mentioned functions can be called from outside of the own network. The other functions have to be restricted. Also only the other party should be allowed to call them. Using an interface that can send files between the different parties could help to replace the interconnectivity between the APIs and hence also reduce the risk of publishing private data.

Chapter 6

Evaluation

In Chapter 5 a prototype implementation of the SC4CyberInsurance system was shown and explained. This chapter evaluates the implementation in different dimensions. Since the performance of the application is mainly dependent of the average block time, which is the time it takes to mine a block, it can be relinquished to measure the performance as the block time can not be influenced. Considering that in the most use cases the other party has to process a following task, it is anyway not really important if an operation is processed in 1 second or in 1 hour. Hence the performance is not really crucial in most of the operations. However, all the API functions that are not connected to the blockchain are responding relatively fast. The most important parts to evaluate are the functionality of the system and the cost respectively the gas that has to be payed to call specific functions. Hence Section 6.1 investigates four different case studies and analyzes if the SC4CyberInsurance system provides the needed functionality in each case study. In Section 6.2 the gas cost of the most important functions are measured and the result is examined.

6.1 Functionality Evaluation

In this section the functionality of the system is analyzed. As mentioned in Chapter 4 and in Chapter 5 many times an interface could increase the customer experience and hence the usability a lot. However, it was decided to focus on the improvement of the implementation of the functionality. Hence this section will mainly evaluate if the system provides all the necessary functionality and if it could theoretically be used to manage a cyber-insurance contract. To do so, four case studies are analyzed, each in a separate subsection. For each case study a figure is provided, that shows the API functions that are called by the insurer respectively the customer. To improve the readability of the figures, the names used in the figures and also the parameters given are not fitting perfectly to the API functions. However, the names can be mapped to each other easily and the missing parameters are specific parameters, which are not relevant in the case studies. Each case study was executed manually by calling the respective API functions. Beside that, tests were also implemented that automatically execute the flow of the case studies. These tests

could be used to repeatedly check if the core functions are still working without having to do it manually. At the end of each automatic test, the contract that was inserted in the database has to be removed again. Otherwise the test can not be executed another time, since there would be already a contract with the same hash stored. In the following, the manual tests are considered to evaluate the functionality. As the tests have to be deterministic to check the resulting values for correctness, the exchange rate is fixed here. Like this the premiums can be defined in fixed ether values as otherwise the concrete premium would be dependent on non manipulable factors. Because of that, all money values will be described directly in ether in the following. The exchange rate can be fixed by not activating the `Exchange Rate Updater` and initializing the exchange rate with a given value instead of calling the oracle.

In the first case study the creation of a contract and the interconnection between the customer and the insurer is investigated. It is checked if a customer can update the conditions several time without having to pay something. In the second case study a standard case is investigated, where a reported damage is reviewed and afterwards accepted by the insurer. The opposite case that the insurer does not accept the damage and propose a counter offer is investigated in the third case study. The customer then does accept the counter offer. The case that the counter offer is not accepted is not evaluated here, as it needs some legal action outside of the SC4CyberInsurance system. Finally, a case study is evaluated, where a customer has installed new security software and wants to update the contract conditions. In case study two, three and four the creation of a smart contract is just shown as a single call of the `createContract` function from the customer. The other steps are left out to improve the readability as they are anyway called automatically.

6.1.1 Case Study 1

In the first case study the creation of a contract is analyzed. The creation does include all the steps which have to be undertaken until the smart contract can be accessed and its functions can be called. It also includes the process of finding the best fitting contract conditions and the system should allow to update the conditions easily until the wanted ones are found. The process of the creation of a contract is already explained in Section 5.5 and shown in Figure 5.1. In this section the process is evaluated and it is checked if the SC4CyberInsurance system allows to easily update the conditions before the contract is created. First, the customer has to enter his data in the JSON file. It was already indicated, that this process has to be simplified using an interface and also the answer possibilities should be fixed by a drop down list or something similar, since otherwise everything could be entered and the information could not be processed automatically. When the information is completely entered the customer can calculate the according premium. Considering that the premium is too high, he may remove a coverage or adapt the contract conditions and calculates the premium again. It is important here that this process is very easy to execute and does not cost a lot, because it may has to be executed several time.
Since in the SC4CyberInsurance system the `Premium Calculator` is available on both APIs, all the calculations can be executed directly on the customer API and no costly operations or network calls have to be executed. Hence these requirements of easily updating

the conditions is satisfied in the SC4CyberInsurance system. After the wanted conditions are found, the contract can be created. It only has to be called the `createContract` function at the customer API. However, inside this API call a lot of outgoing API calls are contained. It has to be ensured here, that only the needed API functions can be called from outside of the network and only by the allowed persons. Otherwise it would be a huge security issue. In general if the security problem can be solved the creation of the contract is very simple and the conditions can be adapted very flexible before the contract is deployed. For the following case studies it is assumed that a contract with a premium of 2 ether was created. The cancellation penalty is 50%, which makes a security of 1 ether to be paid by the customer.

6.1.2 Case Study 2

The most important part for the customer is that he can report a damage and that the damage gets insured afterwards. As it needs first the reporting of a damage at the customer side and then a review of the damage at the insurer side, both parties are investigated here. In this case study, the damage is accepted and the customers damage is covered. The customer first creates a contract similarly as described in case study 1 with a premium of 2 ether and a security of 1 ether and the contracts lifetime starts. The customer fulfills his duty and pays the security and two premiums. Hence the contracts validity is extended twice and 5 ether are transferred to the contract. During the time of validity, the customer experience a damage of 10 ether through a business interruption caused by a DDoS attack, which is covered completely by the contract. Hence the customer reports the damage to the insurer. The insurer checks the data of the attack and decides to accept the damage. As there are currently only 5 ether on the contract he has to send 5 additional ether to the contract. The full damage of 10 ether is then paid out to the customer. In the following the described case study will be investigated considering the SC4CyberInsurance system. Figure 5.1 visualizes the executed API calls and shows the interactions between the customer, the insurer and the smart contract. As previously mentioned, the names in the figure and the given parameters are not fitting perfectly to the names and parameters of the API functions.

The creation of the contract was covered in the first case study and will not be evaluated here again. In each of the following steps it is important that the right contract is accessed, since otherwise a function is called at the wrong contract. At the customer side it is very likely that only one contract is available and hence it should not be a problem. If there are more than one contract the correct contract has to be accessed by calling the `useContract` function. The selected contract is then assigned to be the current contract and each afterwards called function will access this contract until a new one is selected. When a contract is created the new contract is selected as the current contract automatically. In this example it is assumed that only one contract exists on the customer side and no other contract is accessed at any time. At the insurer side there are most likely more than one contract from different customers and the accessed contract has to switch many times. It will be mentioned at the specific positions when the accessed contract changes.

The payment of the security and the premium is handled similar. An API function is called and it has to be defined, how many ether is forwarded to pay the security respectively

the premium. Since the exchange rate is fixed and hence the premium in ether is fixed as well, it is easy in this case to know how many ether have to be sent to pay the security and the ether. Considering that the exchange rate changes, the exact ether amount to pay is probably not known by the customer. Hence the customer may send too much or too little ether. In the case he send too little ether, he is notified that the sent ether does not suffice to execute the called operation and he can just try it again with more ether sent. When too much ether is sent, the remaining ether is just sent back to the customer. Hence there is no threat here that may something gets lost. Generally, the payment of the security and the premium is very easy and nothing unexpected can happen to a party.

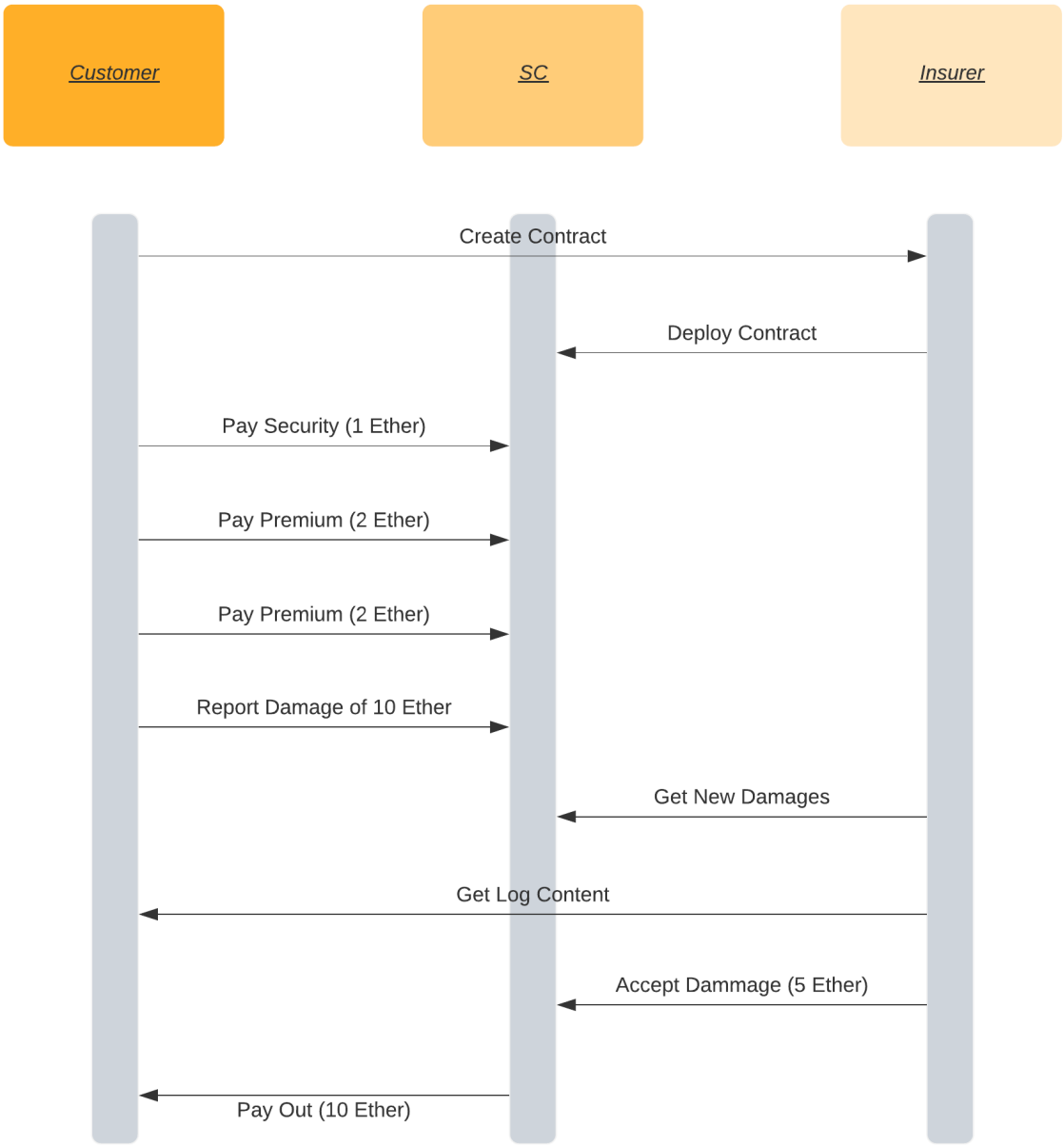

Figure 6.1: Case Study 2

When a damage appears the customer can report it by calling the API function `reportDamage`. The date when the damage appeared, the amount of damage that should be covered, the id of the damage, the type of attack and a suitable log file, which should prove that the attack happened, are given as parameter. Each parameter is considered shorty. The date of the damage will be converted into a timestamp and it has to be between the contract values in `start_date` and `valid_until`. Otherwise the damage will be rejected automatically. In the second parameter it is important to consider, that not the total amount of damage that appeared is contained, only the expected covered damage should be listed. This is because the blockchain does not know the concrete coverage defined in the contract and hence can not calculate it. This is a point that may get forgotten quite fast. Theoretically, the API could calculate the effective coverage out of the reported total damage. However, this is hard because any coverage can be created in the constraints since no limitations are given currently. The usage of a interface could help there. In the id parameter it has only to be considered, that no damage with the same id is reported yet in the contract. If there is one, another id has to be picked. The last two parameters are relevant for the insurer to decide if the damage is covered. The customer has to chose a log file that provides as much information about the damage as possible and that can prove that indeed some damage occurred. The log file should also contain information about the effective damage, which is quite hard to quantify. Hence it is questionable if the insurer can decide if the damage is covered by only considering the log file. This problem walks hand in hand with the high complexity of cyber-insurance.

When a new damage is reported, the insurer is not actively notified. He has to pull the new damages outside of all contracts. To do so all the damages with the status new are pulled out of the contracts frequently by a claims processor. When one or more new damages are available one damage is selected and the log content of the selected damage is read from the customers API. When the log content is read, the according contract is directly set as the current active contract and all the following actions will access this contract. With the content of the log and the other attributes of the reported damage the insurer has to decide if the damage is accepted or declined and maybe a counter offer is created. As mentioned before it is questionable if a reported damage can be assessed correctly with the available information and maybe more information has to be collected in disputable cases in the claim processing. In this case the damage is accepted by the insurer by calling the API function `acceptDamage`. Since currently only 5 ether are available in the contract, the insurer has to send at least 5 additional ether when accepting the damage. Otherwise he gets notified, that the damage can not be accepted as there are too little ether available. After the damage is accepted the customer gets paid out automatically.

In general, the case study can be handled very well by the SC4CyberInsurance system and the processes to execute are intuitive and no unnecessary operations have to be called. The only thing, which is not really clear here, is if the reported damages can really be assessed by the insurer with the provided information. However, if important information is missing, the function to report a damage can easily be extended that it contains more information. The general idea of reporting and processing a damage is tackled and solved by the system.

6.1.3 Case Study 3

In the second case study the case that a damage is accepted by the insurer was investigated. In this case study the other way is considered, that a damage is declined and a counter offer is proposed. Until the reporting of the damage all the steps are equal to the previous case study. However, this time only four ether of the total damage amount were caused by the business interruption and 6 ether were caused by third-party damage, which is not covered in the contract. The customer still reported a damage of 10 ether. It is important to mention here again, that the damages are reported in euro and not directly in ether. However, for simplicity and to avoid complex conversion everything is explained in ether here. As before, the insurer checks the data of the attack, but this time decides to decline the damage, because the third-party damage is not covered. Since the damage caused by the business interruption is covered, he proposes a counter offer of 4 ether. The customer then considers why his damage was declined. He agrees with the reason and accepts the counter offer. The amount of the counter offer is then paid out automatically to the customer.

The according API calls that are executed in the SC4CyberInsurance system are shown in Figure 6.2. Only the steps from the denial of the damage are explained here.After the log content is read from the customer he notices that the reported damage is too high. By calling the API function `declineDamage` the insurer declines the damage and adds the reason why it is declined as parameter. The counter offer is added as parameter as well. The customer is not notified directly that a damage was accepted or declined and he has to pull the information similarly as done at the customer side to get the new damages. When he notices that a damage was declined, he checks the given decline reason. In this case he agrees with the reason and accepts the counter offer. It is also possible that he declines the counter offer, which would change the damage status to `Dispute`. Since such a state needs legal support and it can not be handled by a program this case is not considered more deeply.

The case study of a declined damage can be handled by the SC4CyberInsurance system as well. No additional complexity is added compared to the second case study. It is a bit cumbersome for the customer, that he has to pull the current information of its reported damage actively and does not get notified. However, it is also possible to create additional components that take the notification part and frequently check the status of the blockchain. For a prototype the functionality provided now is sufficient.

6.1.4 Case Study 4

In a forth case study the case is examined that a customer has installed new security software and wants to update the contract conditions. After the customer payed the first premium he updates the conditions and proposes them to the insurer. The insurer checks the updated conditions and agrees with them. Since the customer has now a better risk level, the premium decreased from 2 ether to 1.85 ether per period. Figure 6.3 shows the executed API calls in the SC4CyberInsurance system. Until the payment of the first premium everything is still the same as in the previous case studies. To update

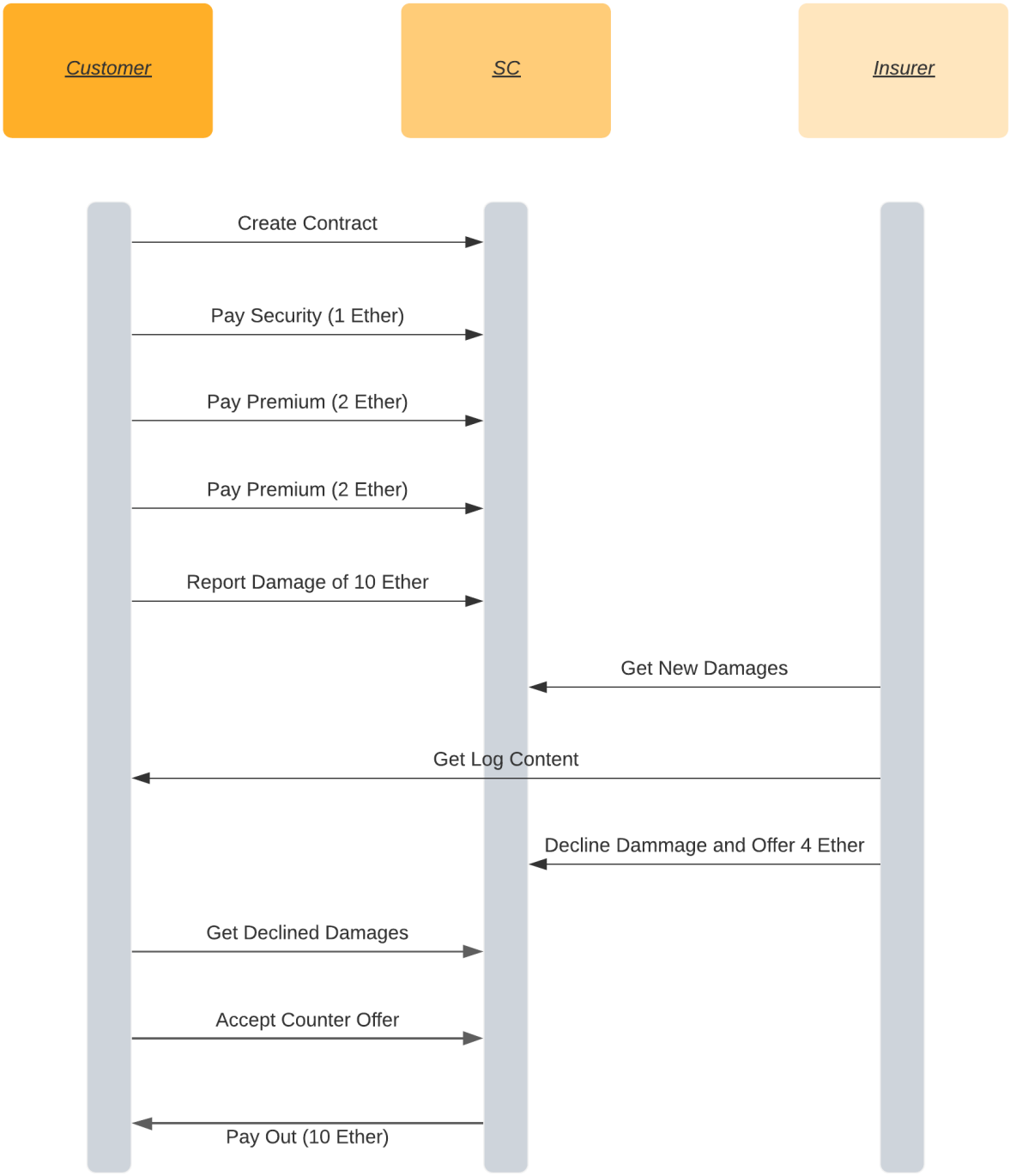

Figure 6.2: Case Study 3

the contract conditions the customer can just use the initial JSON file and adapt it considering his new changes. He then calls the API function `proposeToUpdateContract` and proposes the new JSON file to become the valid one. The insurer is again not actively informed that a new proposal is available and similarly as he got the new damages, the insurer frequently checks if any new proposals are available in one of his contracts. If there is a new one, the insurer can get the content of the proposal directly from the customer. Of course this content should not get public as well, because it again contains

a lot of the customers private information as the initial JSON file. Considering the new JSON content, the insurer decides if the proposal is accepted or declined by simply calling the according API function. After the proposal is accepted the premium is accordingly adapted. The SC4CyberInsurance system can also handle the case study of updating the

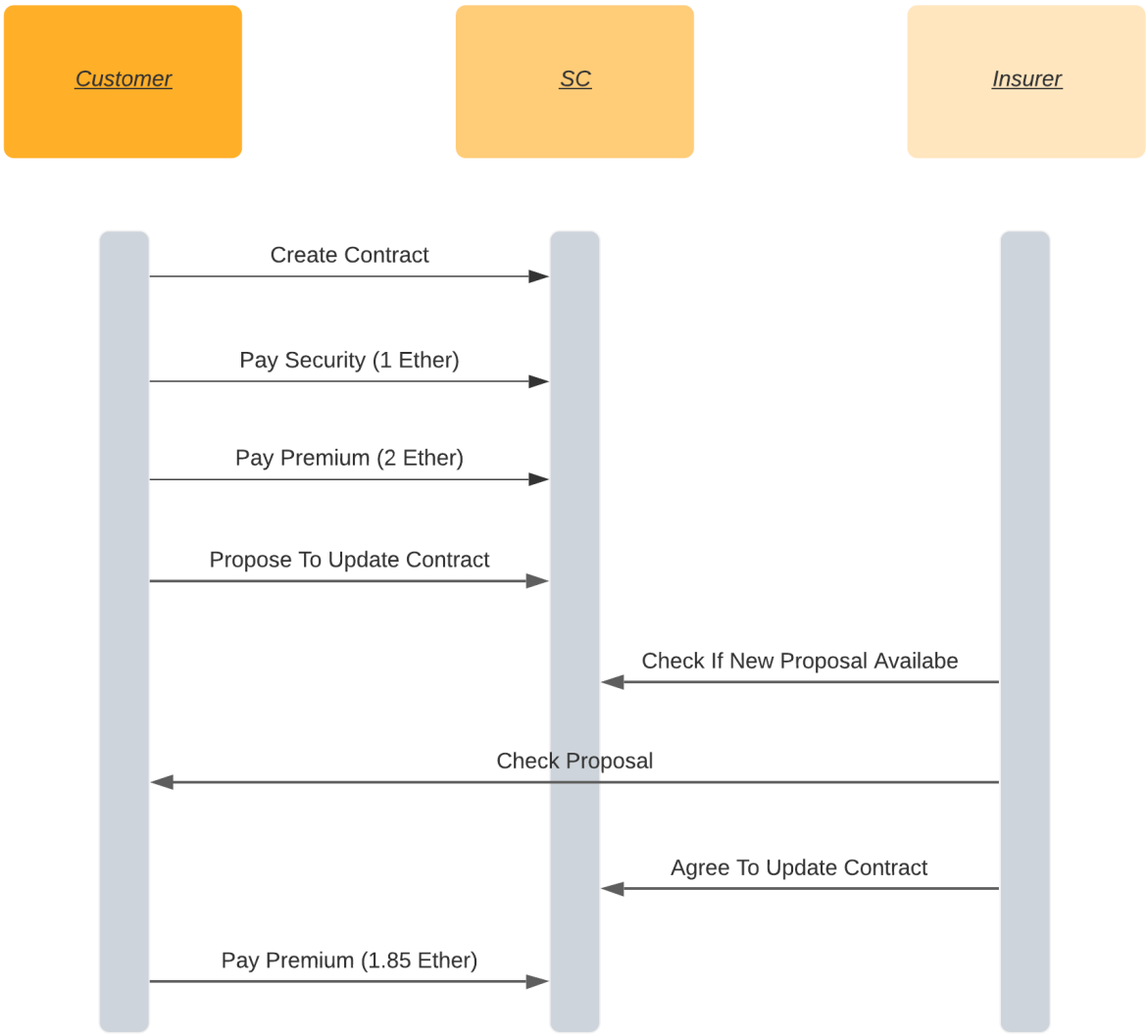

Figure 6.3: Case Study 4

contract conditions. Similar as before an additional component could help to frequently check for potential proposals and to ensure that no proposal gets forgotten. However, also with the current prototype the updating of the contract conditions is possible.

6.2 Cost Evaluation

In Section 6.1 the functionality of the system was analyzed and all the investigated case studies could be handled by the SC4CyberInsurance system. However, it is not only important if the functionality is given, it is also important how much it costs to execute

the according functions. Since each executed transaction on the blockchain has to be paid in gas, every called function costs some ether.
Hence in this section the cost of the smart contract functions are analyzed. Table 6.1 gives an overview of the cost of the most important smart contract functions. The gas costs were estimated using the `estimateGas` function provided by the `web3` library. The estimations can differ when the state of the smart contract changes during time. However, the differences are relatively small and the highest found estimated values are used in Table 6.1 to consider the worst case. The functions were not intentionally optimized in gas costs and the costs may can be reduced a lot. Even so, it is assumed that the costs stay in approximately the same dimension.
The gas costs are converted into wei using a gas cost of 20 gwei per gas, which is the default value of Ganache. The ether value is converted into euro using an exchange rate of 600 euro per ether, which is approximately the current exchange rate. The exchange rate from ether to euro is changing permanently and it is likeable that the converted values are not nearly correct anymore after a short time. Currently the exchange rate is increasing quite fast, which makes each ether to pay because of gas consumption more expensive and hence the calculated values are possibly to low soon. In the following analysis only the calculated values in Table 6.1 are considered. However, it should be hold in mind that maybe the costs increase a lot in the near future. Each time when one of the functions is called, the caller has to pay the above costs.

Considering first the constructor of the contract, which is the most expensive function, a price of approximately 65 euro per call is extremely high. Since the insurer always has to deploy the contract, this price is always paid by the insurer. Thinking of Section 5.3, where it was described that the insurer automatically deploys a contract and the customer theoretically can deploy infinite contracts without paying anything, it seems clear that this concept does not work considering the cost of 65 euro to pay per contract deployed. Therefore a bit of automation should be removed in the system and the insurer should decide, which contract to deploy and which not, as otherwise an attacker could easily create thousands of contracts and force the insurer to pay the creation. Considering that only correct contracts are deployed in the blockchain, a price of 65 euro per contract is still quite high. As the time that it takes to deploy the contract is not really relevant, it makes sense here to decrease the gas cost and wait a bit longer until the contract is deployed. Taking a gas cost of 2 gwei, which is considered a price that usually still gets the transaction in a block within the next few minutes [16], the final cost can be divided by 10 as well. Hence the deployment would cost 6.5 euro. Assuming that only real active contracts are deployed, this is an acceptable price. To secure the insurer even more, it makes sense to let the customer pay the price of deploying the contract. Then the process could be automatized again as the insurer has no risks anymore and an attacker can not create a lot of requests, because it is too expensive.

Another important function to analyze is the `updateExchangeRate` function, because this function is probably the function called the most times. The cost of approximately 2.5 euro per call is very expensive as well. Assuming that it has to be called in average one time per day per contract it makes around 912.5 euro in 365 days just for updating the exchange rate. This cost is so high, because the oracle, which is called by the function, has to be paid inside the function as well. Again considering that the performance of this function is not extremely important, the gas cost could be reduced as well to 2 gwei,

Table 6.1: Cost Estimations Smart Contract Functions

Function	**Estimation in Ether (20 gwei/gas)**	**Converted in Euro (600 euro/ether)**
Constructor	0.1089376	65.36256
paySecurity	0.00080804	0.484824
payPremium	0.00084042	0.504252
reportDamage	0.00435256	2.611536
acceptDamage	0.00109892	0.659352
declineDamage	0.00174466	1.046796
acceptCounterOffer	0.00082004	0.492024
proposeToUpdateContract	0.00264966	1.589796
agreeToUpdateContract	0.00098624	0.591744
updateExchangeRate	0.004194334	2.5166004

which reduces the cost to around 91.25 euro. Taking a contract of a small customer with a yearly premium of 500 euros, nearly 20% of the premium would flow in the payment of the updating of the exchange rate. Hence it is clear that even with a gas cost of 2 gwei such a contract can not be updated once per day in average. Therefore the `Exchange Rate Updater` should be fine tuned, that the exchange rate is updated as few as possible and that it considers the premium of a contract. For example if a contract of a big customer has a yearly premium of 20'000 euro it makes sense to update the exchange rate more times than in the contract of the small customer, because the exchange rate impacts a lot more money.
Since the other functions are normally called quite rarely, the price of these functions is acceptable. If a customer does not want to pay 2.6 euros to report a damage, he can also reduce the gas price there and just wait a bit longer until his transaction is mined.

Chapter 7

Conclusions and Future Work

In this work a blockchain-based model for cyber-insurance was investigated and developed. Using different supporting components implemented in python, a system that can deploy cyber-insurance contracts on the Ethereum blockchain was built. In the system the contracts are deployed as smart contracts, which can automatically execute the contract agreements. A premium calculation usable in practice as well as the design of an interface was left at the side to focus on the functionality of the smart contract and could be designed and implemented in future work.
Considering the aimed benefits mentioned at the beginning of Chapter 4 the implementation of the SC4CyberInsurance system includes all of them, even though some of them only partially. The way to create a contract is simplified in the SC4CyberInsurance system by automating the processes as much as possible. A customer can create the contract by themselves without having to contact the insurer. Only a JSON file with the according information has to be filled out and it can directly be deployed on the blockchain as a valid contract. Having a functional interface could simplify the creation and support the customer even more. Also the complexity could be reduced a lot by having an interface, as the JSON fields could be filled out using predefined answers and it would get more clear what is important in cyber-insurance. However, the JSON file setup and the API functions are set up as intuitive as possible and they should still reduce the complexity of the cyber-insurance contract by helping to focus on the most important parts.
Thanks to the usage of smart contracts to execute the contract agreements the different functionalities were automatized where possible. When deploying a smart contract a trustworthy and immutable agreement is created as it is securely and transparently deployed on the blockchain. Thanks to the `Data Anonymizer` the private data of the customers is not published but still trackable. However, the data transfer between the insurer and the customer outside of the blockchain should be improved as through the interconnectivity of the APIs new risks are created.

All the case studies evaluated could successfully be handled by the SC4CyberInsurance system. Beside the missing interface and the problem of the interconnectivity between the APIs already mentioned above, there is another downside of the implementation, namely that a party is not actively informed if the state of a contract changes. For example if a damage is reported, the insurer has to actively monitor the contract to detect that a new damage is available. Even so, this problem can be handled by creating according

components that monitor the contracts and that inform the customer when something happened. Hence all these problems can be solved by improving the prototype in future work.
However, the biggest downside of the SC4CyberInsurance system is inherent in the blockchain itself. A blockchain user has to pay a specific amount of gas to the miners that the miners include the transaction in their next block. As higher the complexity of the transaction as higher the cost. Since the smart contract functions have a relatively high complexity the costs can get very high and at some point it is not lucrative anymore for the insurer and the customer to use the system. By reducing the gas costs the final costs can be reduced as well, in return the time until the transaction is mined is increased. Since not in all smart contract functions the gas costs are high and the performance time is not extremely crucial in the functions, where it is high, it should be possible to work with the worse performance there.
However, it is questionable if the advantages of trustworthiness and transparency could not be gained by another implementation outside of a blockchain that does not bring the mentioned disadvantages. It remains one of the key questions in the blockchain technology, if the technology is really needed or if it is just an additional complexity inserted in the system.

Bibliography

[1] Nir Kshetri: The Economics of Cyber-Insurance, University of North Carolina, November/December, 2018.

[2] A. Marotta, F. Martinelly, S. Nanni, A. Orlando, A. Yautsiukhin: Cyber-Insurance Survey, January 3, 2017.

[3] V. Gatteschi, F. Lamberti, C. Demartini, C. Pranteda, V. Santamaría: Blockchain and Smart Contracts for Insurance: Is the Technology Mature Enough?, future internet, February 20, 2018.

[4] Jeff Wargin: 8 Insurance Technology Trends Transforming the Industry in 2020, Duck Creek Technologies, December 30, 2020.

[5] Nidhi Agrawal: Four New Consumer-centric Business Models in Insurance, Mantra Labs, May 11, 2020.

[6] R. Pal, L. Golubchik, K. Psounis, P. Hui: Will Cyber-Insurance Improve Network Security? A Market Analysis, IEEE Conference on Computer Communications, 2014.

[7] G. Ciocarlie, K. Eldefrawy, T. Lepoint: BlockCIS—A Blockchain-based Cyber Insurance System, IEEE International Conference on Cloud Engineering, 2018.

[8] B. Rodrigues, M. Franco, G. Parangi, B. Stiller: SEConomy: a Framework for the Economic Assessment of Cybersecurity, Communication Systems Group, University of Zurich, 2020.

[9] Oliver Wyman: CYBER RISK IN ASIA-PACIFIC THE CASE FOR GREATER TRANSPARENCY, Asia Pacific Risk Center, 2017.

[10] Satoshi Nakamoto: Bitcoin: A Peer-to-Peer Electronic Cash System, 2009.

[11] Karl Wüst and Arhut Gervais: Do you need a Blockchain?, 2017.

[12] Imran Bashir: Mastering Blockchain, pp. 29, March 2017.

[13] Nick Szabo: Smart Contracts, `https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html`, 1994.

[14] Thomas Bocek and Burkhard Stiller: Smart Contracts - Blockchains in the Wings, UZH, 2017

[15] Jake Frankenfield: Smart Contracts, Investopedia, https://www.investopedia.com/terms/s/smart-contracts.asp, October 08, 2019.

[16] Arun Rajeevan: Tokens, Gas and Gas limit in Ethereum, https://arunrajeevan.medium.com/tokens-gas-and-gas-limit-in-ethereum-f07790f56d8f, February 11, 2019.

[17] Julia Kagan: Insurance, Investopedia, https://www.investopedia.com/terms/i/insurance.asp, July 29, 2020.

[18] Agence France-Presse: Cyber Insurance Market to Double by 2020, Says Munich Re, SECURITY WEEK, https://www.securityweek.com/cyber-insurance-market-double-2020-says-munich-re, September 09, 2018.

[19] Akin Oyedele: BUFFETT: This is 'the number one problem with mankind', https://www.businessinsider.com/warren-buffett-cybersecurity-berkshire-hathaway-meeting-2017-5?r=US&IR=T, May 6, 2017.

[20] Dan Raywood: DDoS Attacks Hit 1 Tbps in 2020, https://www.infosecurity-magazine.com/news/ddos-1tbps-2020/, September 17, 2020.

[21] Martin Kreuzer: Cyber Insurance: Risks and Trends 2020, https://bsabh.com/cyber-insurance-risks-and-trends-2020/, April, 2020.

[22] Steve Morgan: Cybercrime To Cost The World $10.5 Trillion Annually By 2025, Cybercrime Magazine, https://cybersecurityventures.com/hackerpocalypse-cybercrime-report-2016/, November 13, 2020.

[23] Business Insurance Center: WHAT ARE FIRST-PARTY AND THIRD-PARTY COSTS IN A CYBER ATTACK?, https://www.businessinsurancecenter.com/what-are-first-party-and-third-party-costs-in-a-cyber-attack/, July 29, 2019.

[24] Chris Murray: Breaking Down the First- and Third-Party Costs of a Cyber-Attack, Caitlin Morgan, https://www.caitlin-morgan.com/breaking-down-the-first-and-third-party-costs-of-a-cyber-attack/, August 31, 2018.

[25] Dan Burke: Cyber 101: Understand the Basics of Cyber Liability Insurance, WOODRUFF SAWYER, https://woodruffsawyer.com/cyber-liability/cyber-101-liability-insurance-2021/, November 2, 2020.

[26] K. Owens, S. Knight, S. Panensky and S. Groeber: What Does Cyber Insurance Really Bring Table and...Are You Covered?; RSA Conference, 2019, USA.

[27] Bernhard Niedermayer: Blockchain – The Next Big Thing, Catalysts, https://www.catalysts.cc/en/big-data/blockchain-the-next-big-thing/, February 22, 2017.

[28] Pierre Jean Duvivier: Is the blockchain the new graal of the financial sector ?, Linkedin, https://www.linkedin.com/pulse/blockchain-new-graal-financial-sector-pierre-jean-duvivier, June 1, 2016.

[29] Gideon Greenspan: Avoiding the pointless blockchain project, MultiChain, https://www.multichain.com/blog/2015/11/avoiding-pointless-blockchain-project/, November 22, 2015.

[30] Kasey Panetta: As blockchain evolves, CIOs can avoid unnecessarily failure by acknowledging common pitfalls., Gartner, https://www.gartner.com/smarterwithgartner/top-10-mistakes-in-enterprise-blockchain-projects/, July 1, 2019.

[31] Technology.org: Biggest Cyber Attacks and Their Cost for the Global Economy, https://www.technology.org/2019/07/17/biggest-cyber-attacks-and-their-cost-for-the-global-economy/, July 17, 2019.

[32] OECD Data: Gross insurance premiums, https://data.oecd.org/insurance/gross-insurance-premiums.htm, November 18, 2020.

[33] Aarti Goswami and Pramod Borasi: Cyber Insurance Market Outlook - 2026, https://www.alliedmarketresearch.com/cyber-insurance-market, March, 2020.

[34] Provable Dev Community: Provable, https://docs.provable.xyz, August 28, 2019.

[35] B Advisory: Risk Matrices – Why they don't work, https://b-advisory.ch/risk_matrices.html, November 12, 2020.

Abbreviations

ABI	Application Binary Interface
API	Application Programming Interface
DDoS	Distributed Denial-of-Service
EVM	Ethereum Virtual Machine
JSON	JavaScript Object Notation
OECD	Organisation for Economic Co-operation and Development

List of Figures

List of Tables

Appendix A

Installation Guidelines

Preconditions:

- Install Python3 and Ganache
- Start a Quickstart Ethereum Workspace in Ganache with at least 3 users

Start Insurer API:

- Open a command line interface
  - Move in the prototype folder of the project
  - Call command "python3 APIInsurer.py"
    - The first ganache account will now use port 5000 as an insurer
  - The url http://localhost:5000/ can now be used to access the insurer API
- When the Ganache workspace is restarted, the API has to be restarted as well

Start Customer API:

- Open a command line interface
  - Move in the prototype folder of the project
  - Call command "python3 APICustomer.py"
    - The second ganache account will now use port 5001 as a customer
  - The url http://localhost:5001/ can now be used to access the customer API
- When the Ganache workspace is restarted, the API has to be restarted as well

Manually use the system:
To manually use the system, the API functions can be called. In the folder named ”test”, the calls for the case studies are predefined in http files and the calls can be executed one by one using an IDEA. It is important to remember, that only one contract with the same JSON file can be created at once. Hence the contract have to be deleted in the database or the JSON file has to be adapted. To delete the contract from the database, the automatized tests can be called, which always clean up the contract they have built at the beginning and at the end.

Calling the automatized case studies:

- Open a command line interface
  - Move in the prototype folder of the project
  - Call command ”python3 CaseStudy1.py” (respectively the case study to be tested

Using the Exchange Rate Updater
To use the the oracle from the local test network a Etherum bride has to be set up.

- Set up an Ethereum bridge
  - Open a command line interface
    - Call command ”ethereum-bridge -H localhost:7545 -a 2”
    - This command will use the third Ganache account as bridge account for the oracle.
    - Adapt line 32 in the file SmartContractCode.py with the appropriate line of the response in the command line interface if necessary.
  - Open a command line interface
    - Move in the prototype folder of the project
    - Call command ”python3 ExchangeRateUpdater.py”

Appendix B

Contents of the CD

- Thesis as PDF
- Source code
- Presentation slides

Appendix C

JSON file

```
"business_information": {
    "companyName": "TestAG",
    "type": "AG",
    "sector": "Electronic Store",
    "address": {
      "streetAddress": "Examplestreet 1",
      "city": "Zurich",
      "state": "ZH",
      "postalCode": 8000
    },
    "contact": [
      {
        "type": "phone",
        "number": "123456789"
      },
      {
        "type": "mail",
        "number": "abc.de@testAG.ch"
      }
    ]
  },
  "contract_constraints": {
    "startDate": "01.01.2021",
    "duration": 3,
    "endDate": "31.12.2023",
    "paymentFrequencyPerYear": 2,
    "cancellation": {
      "allowed": true,
      "penaltyInPercent": 50
    }
  },
  "company_conditions": {
    "yearly_revenue": 2500000,
```

```
    "revenue": 0.2,
    "basedOnYear": 2019,
    "numberOfEmployees": 10
  },
  "company_security": {
    "risk_assessment_metrics": [
      {
        "name": "Known vulnerabilities",
        "result": "medium"
      }
    ],
    "attacks_history": [
      {
        "type": "DDoS",
        "date": "09/12/2019",
        "time_to_recovery": "15 hours",
        "details": "Mirai Botnet",
        "mitigated": false
      }
    ],
    "security_software": [
      {
        "name": "Dynatrace",
        "type": "monitoring"
      }
    ],
    "security_training": [
      {
        "name": "ZYX security certificate",
        "type": "education of employees",
        "date": "26/04/2020",
        "provider": "International CyberSecurity Institute"
      }
    ]
  },
  "company_infrastructure": {
    "number_connected_devices": 50,
    "number_systems": 2,
    "technologies": [{
        "type": "operating system",
        "name": "windows",
        "version": "XP",
        "updates": false
      },
      {
        "type": "core software",
        "name": "Docker",
```

```
        "version": "19.03.8",
        "updates": false
      }
    ],
    "critical_data": "3 TB",
    "critical_services": [
      "data privacy",
      "marketplace",
      "payment system"
    ]
  },
  "contract_coverage": [{
      "name": "DDoS",
      "coverage": [{
          "name": "business Interruption",
          "coverage_ratio": 100,
          "deductible": 1000,
          "max_indemnification": 300000
        },
        {
          "name": "cyber extortion",
          "coverage_ratio": 100,
          "deductible": 1000,
          "max_indemnification": 300000
        }
      ]
    },
    {
      "name": "data breach",
      "coverage": [
        {
          "name": "third person damage",
          "coverage_ratio": 100,
          "deductible": 1000,
          "max_indemnification": 300000
        }
      ]
    }
  ]
}
```

Listing C.1: Customer Information Mapped on JSON file