SaCI: a Blockchain-based Cyber Insurance Approach for the Deployment and Management of a Contract Coverage

Muriel Franco, Noah Berni, Eder Scheid,
Christian Killer, Bruno Rodrigues, Burkhard Stiller

Communication Systems Group CSG
Department of Informatics IfI, University of Zürich UZH
Binzmühlestrasse 14, CH—8050 Zürich, Switzerland
E-mail: [franco, scheid, killer, rodrigues, stiller]@ifi.uzh.ch, noah.berni@uzh.ch

Abstract. The cyber insurance market is still in its infancy but growing fast. Novel models and standards for this particular insurance market are essential due to the use of modern IT (Information Technology) and since insurance providers need to create suitable models for customers.
In this work, a refreshing approach SaCI for the deployment and management of contract coverage is introduced. SaCI translates relevant information of a cyber insurance contract to Smart Contracts (SC) running on the Blockchain (BC). Thus, SaCI *(i)* allows for recording agreements in an immutable way, *(ii)* simplifies interactions between stakeholders (e.g., customers and insurers), and *(iii)* ensures a trustworthy and transparent process during the life-cycle of the contract. A case study is provided to show evidence of the feasibility of the approach, which is backed by a cost analysis and discussion regarding especially the application of BCs.

Keywords: Cyber Insurance · Cybersecurity Economics · Blockchain · Smart Contract (SC).

1 Introduction

Cybersecurity stands as one of the key investment pillars for companies applying IT (Information Technology) to gain competitiveness in the market due to the continuously increase in the number of cyberattacks on IT systems 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 most significant transfer of economic wealth in history [9]. In this sense, to reduce the impact of successful attacks and to enable companies to recover faster and with less costs, different cybersecurity investment strategies have been investigated [14], in which one of the most prominent strategies includes cyber insurance coverage models [12]. Although the cyber insurance market is fast-paced and is under strong development [6], [7], cyber insurance approaches still have room to advance from a rarely used risk transfer tool to a critical requirement for companies risk management.

Currently, different cyber insurance approaches are explored by companies, effectively expanding the market, either *(a)* introducing new business models and mechanisms to gain advantages or *(b)* improving their insurance services by using new technologies. However, critical open challenges for a cyber insurance adoption exist, *e.g.*, the information asymmetry that has to be considered during the contract's design and the customer's eligibility for coverage [1]. Thus, different cyber insurance approaches have been proposed and new paradigms have been applied in such a context [16]. One such a new paradigm that is a relevant catalyst in the insurance market is the Blockchain (BC). BCs allow for the implementation of Smart Contracts (SC) to remove intermediaries, automate the deployment and management of insurance contracts, and support novel insurance models [4]. Due to the automation of SCs and the immutability of the BC, BC-based cyber insurance models can provide a trustworthy and immutable agreement between cyber insurers and customers; thus, both stakeholders can profit from the benefits introduced by the BC.

This paper introduces a BC-based approach for the creation, deployment, and management of a cyber insurance contract. SaCI correlates relevant customers' aspects and cyber insurance companies' (*i.e.*, insurers) requirements, such as business information, contract constraints, and security aspects, to create an SC that describes and manages the agreement between customers and insurers. Based on this, both stakeholders can interact with the SC to proceed with coverage requests, contract updates, and premium payments. SaCI ensures a trustworthy record of the contract coverage and all changes along time; thus, not only *(i)* providing automation of the process, but also *(ii)* acting as a referee or proof in case of disputes (*e.g.*, customers requesting payment for a loss due to a cyberattack that the insurer has denied payment for). Further, if funds are available and contractual requirements are satisfied, SaCI automatically transfer funds between stakeholder to execute payments, such as those related to premiums paid and loss coverage due to a cyberattack.

The remainder of this paper is organized as follows: Background and related work are reviewed in Section 2. While Section 3 introduces SaCI and details of the implementation, Section 4 discusses the feasibility of SaCI and presents a suitable case study subject to a cost evaluation. Finally, Section 5 summarizes the paper and outlines future work.

2 Background and Related Work

A cyber insurance is a specific product of an insurance company, which is commercially offered to cover damage caused by cyber-incidents, direct or indirect impacts caused by cyberattacks. A cyber insurance is offered for companies, governments, or individuals, who want to reduce or share financial risks of an attack and which shall cover costs for recovering from an incident [7]. Typically, the process of cyber insurance contract creation involves three main steps: *(i)* Risk identification, which is based on the identification of assets that can be affected by different threats [14], *(ii)* Risk analysis, which determines the likelihood of a threat and also its impact, and *(iii)* Contract establishment with a focus on coverage specifications and premium definition. With the increase of cyber-attacks and their actual impacts, the cyber insurance market also has to evolve to handle different aspects, such as incomplete, asymmetric, or even insufficient data for pricing premiums and coverage, lack of regulations and standards, and the gap between cybersecurity and risk transfer [5].

According to a study conducted in South Korea [10], companies with high incomes, high education, and insurance contracts are more likely to "pay extra" for insurance policies using BCs and SCs. Thus, a strategic development of insurance products using BCs targeting these customers can increase the number of policyholders, which can, in turn, increase premium revenues. Thus, the application of BCs can provide efficiency and trust in the entire process, while insurers become innovators in their relation to customers.

In this context, [2] introduces a conceptual framework for cybersecurity investments and cyber insurance decisions. The framework advocates the use of SCs for cyber insurance coverage and premium management as one of its key pillars. A case study focuses on the maritime sector and shows evidence of the framework's applicability. However, no implementation details are provided at all. [17] provides a model for determining insurance premiums based on the Stackelberg Game to improve the time efficiency of BC applications. A BC-based crowdsourcing system was developed as a proof-of-concept to show how the cyber insurance model can protect blocks containing task information. Although this approach improves the time to perform each crowdsourcing task, focus is neither laid on information about contract coverage nor on interactions between customers and insurers.

Furthemore, BlockCIS [8] proposes a BC-based cyber insurance tool, which offers the insurer and the customer the possibility to reach an automated, real-time, and immutable feedback cycle for a dynamic risk assessment. For that, the system interconnects the insurer and the customer over a BC. However, BlockCIS is presented as a supporting tool and cannot be used as an individual tool to provide a cyber insurance service. For example, the paying of the premium and the payment of claims are not integrated into the system, and hence that has to be managed by external applications. However, such frameworks can well be used to assess cybersecurity correctly and, based on that, can calculate a fair premium for a cyber insurance contract.

Thus, although the demands in related work clearly indicate benefits of using BC-based approaches for cyber insurance, open issues remain, especially with regards to achieving an efficient model that considers different nuances of the market. In order to address this gap, SaCI focuses on the mapping of information and interactions, required to establish a trustworthy and automated interaction between customers and cyber insurers. Therefore, this work does contribute to the development of simplified, trustful, and efficient cyber insurance models.

3 The SaCI Approach

SaCI is proposed to handle different demands of cyber insurance in order to create a simplified, trustworthy, and automated process for cyber insurance contracts. For that, SaCI describes a JSON (JavaScript Object Notation) file structure to store relevant information about the contract and to translate it to SC code within well-defined functions allowing for interactions between customers and insurers. Therefore, the SaCI allows for the *(i)* payment of premiums and contract updates, *(ii)* request of damage coverage and dispute resolutions, and *(iii)* check of contract information and its integrity, whenever it is required (*e.g.*, in case one of the parties involved are not following the agreement defined).

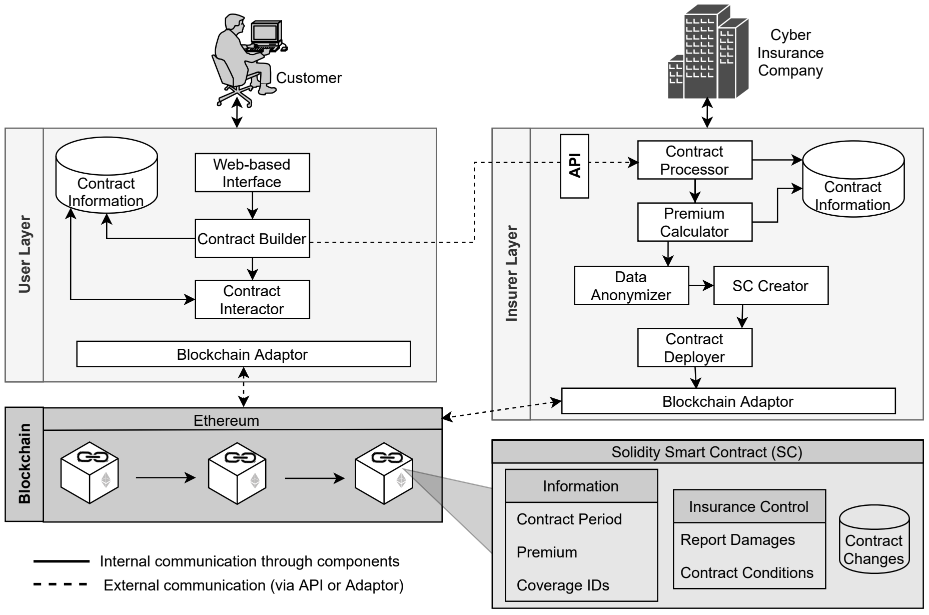


Fig. 1: SaCI Architecture.

The architecture of the SaCI (*cf.* Figure 1) determines the two different stakeholders (*i.e.*, customer and cyber insurer) at the top and enables the interaction with the system using those components running on their respective layers (*i.e.*, on their own infrastructures). The *User Layer* is composed out of a Web-based interface, with which the customer can access and add all information related to business and demands (*cf.* Table 1). This information is forwarded to the *Contract Builder* in charge of mapping these information into the JSON format. The respective JSON file is sent to the *Insurer Layer* using the SaCI's API.

Within the *Insurer Layer* the *Contract Processor* reads information from this JSON file and stores a copy of all contract information. The *Premium Calculator* estimates the premium for this contract's coverage.While this paper does not focus on an optimal premium calculation, it provides relevant information in a standardized format, *e.g.*, as input for a base rate pricing in which modifications for the calculation can be accommodated according to insurer preferences.

Table 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 companys 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 at 50%; data breach for third-person damage: coverage: at 100%

After the premium calculation, the *Data Anonymizer* component is in charge of removing from the contract all information that can be critical to identify the company and its risks. This is essential before deploying the contract within a public BC (*e.g.*, Ethereum or Cardano). The *SC Creator* uses all other information to transform the JSON file into an SC based on previously defined one (*i.e.*, Solidity code) and fills in missing information in those fields mapped. Finally, the contract is deployed on the BC and available for interactions between all stakeholders (Actors) involved (*cf.* Table 2)

In order to define the relevant information for the creation of the cyber insurance contract, and consequently, the SC, necessary information was defined based on the related cyber insurance market. Table 1 provides an overview of these main categories considered by SaCI. Every characteristic demanded for by a customer is assigned to one of these categories. Note that this type of information has to be provided by customers, which might result in "inaccurate" information and can be impacted by companies' biases, such as metrics related to risk assessment and threats impacts.

The business information contains standard information about the company, which are most likely to be known publicly. This information is needed to identify the company, but not relevant for a premium calculation. Basic conditions (*e.g.*, contract duration) are stored in contract constraints. Company conditions comprise all non-technical characteristics and mainly include information about business numbers. The following two categories are significantly related to each other and they encompass all technical characteristics. With the information of these two categories, the probability and partially the impact of a successful attack can be estimated to better understand all risks by both actors.

Table 2: Examples of SaCI Functions Implemented in the SC.

Function	**Actor**	**Parameters**	**Description**
payPremium	Customer	-	Pays the premium converted in Ethereum's Wei, increases time of validity.
reportDamage	Customer	uint date, uint amoumt, string type_of_attack string logfileHash, uint damage_id	Creates a damage struct on the contract.
acceptDamage	Insurer	uint damage_id	Accepts damage with ID and pays out reported damage.
acceptCounterOffer	Customer	uint damage_id	Accepts counter offer, which is paid out automatically.
resolveDispute	Customer	uint damage_id	Resolves a dispute about a damage reported, when a solution is found off-chain.
proposeTo-UpdateContract	Both	uint new_premium, string new_file_hash	Makes a proposal to update the contract.

While the company security category describes different metrics about security deployed and measures taken to improve the security, the company's infrastructure includes all information of hardware, software, and technology as well as about critical parts of those. Finally, within the contract coverage category, details about every contract's coverage are stored in an unlimited list. For every attack, the costs covered and possibly other constraints of the specific coverage (*e.g.*, maximum indemnification of insurer) are defined. The contract coverage is the most important part besides the risk assessment to calculate the premium. Listing 1.1 shows an example of a contract coverage against four different threats (*e.g.*, business interruption due to a DDoS attack and third-person damage due to a data breach) defined in the JSON file's descriptor. Finally, upon entering information of all categories, the content can be forwarded to the *Premium Calculator*, which will calculate the premium and inputs the SC generation.

At this point, the contract is deployed on the BC and can be accessed by the insurer and the customer utilizes functions available in the contract (*cf.* Table 2).

This list is not exhaustive and other functions are available in the proposed SC, too, all details are available within the implementation [11].

After the premium is paid and the contract is enacted, the actors can interact. For instance, in case an attack happened, the customer can call the *reportDamage()* function (*cf.* Listing 1.2) to ask for refunding or help. The insurer can accept or deny the coverage requested. If accepted (*i.e.*, *acceptDamage(id)*), the payment is made automatically via the SC according to what was defined previously in the contract. Note that the customer can also provide a hash of a log file as proof of the attack. This hash is also stored in the BC to further enable an integrity check. At the same time, the file itself has to be stored off-chain, especially inside the contract information datasets maintained by both actors.

```
"contract_coverage": [
  { "name": "DDoS",
   "coverage": [{
     "name": "Business Interruption",
     "coverage_ratio": 100,
     "deductible": 1000,
     "max_indemnification": 300000 }]},
  { "name": "Data Breach",
   "coverage": [
    { "name": "Third-party damage",
     "coverage_ratio": 100,
     "deductible": 1000,
     "max_indemnification": 300000 }]}]
```

Listing 1.1: Contract Coverage in a JSON Format.

If the parties cannot reach a conclusion, counteroffers can be made by the insurer (*i.e.*, payment for a specific loss but not for all financial losses). Figure 2 shows the state diagram of possible interactions after a *reportDamage()* is called by the customer. The report damage process has one of the following states: *New*, *Paid*, *UnderInvestigation*, *Dispute*, *Resolved*, or *Canceled*. This diagram exmplifies the different functions's use (*e.g.*, *reportDamage()*, *acceptDamage()*, and *acceptCounterOffer()*) to claim a settlement.

The *Canceled* status is an ending state, reached only if the customer cancels the request. *Paid* status defines that the insurer accepted to cover the damage, and it was automatically paid. If the contract has a lower balance than the value to pay out, the insurer has to transfer funds to the contract, when accepting the coverage. If the insurer declines the coverage payment, a reason is provided and a counteroffer is issues. If a counteroffer is not possible to be offered at that time, the status is defined as *UnderInvestigation*, which means that further manual investigations have to be placed off-chain before a counteroffer can be placed.

If the insurer provides a counteroffer (*e.g.*, a lower amount than the initially requested compensation for that incident) and the customer does not accept it, the state changes to *Dispute*. This refers to the fact that no agreement has been found yet. Either the insurer creates a better counteroffer or the two actors have to solve the dispute off-chain for which a third party may be considered. If the dispute can be solved, the final status of *Resolved* will be achieved. Using the SC function *getAllReportedDamagesWithStatus* all reported damages with a

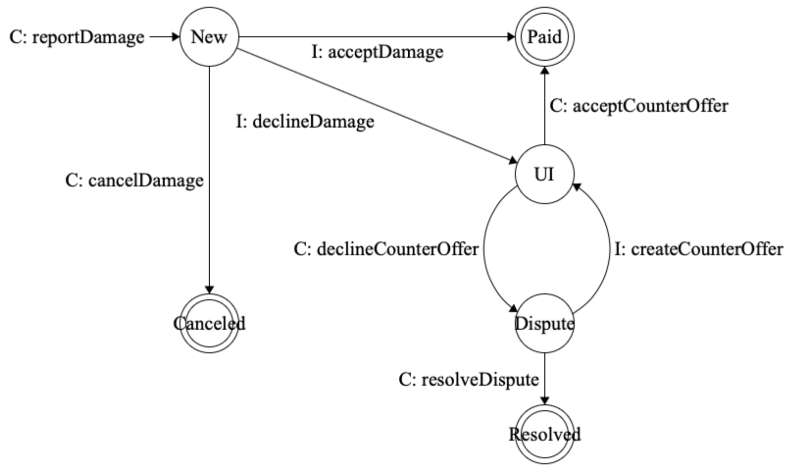


Fig. 2: Claims Settlement State Diagram.

specific status can be returned, which also allows verifying the history of past interactions, *e.g.*, accepted, declined, and under investigation coverage requests.

A prototype of the SaCI was implemented using Python as backend language and Solidity for the SC development. The Ethereum blockchain running on the Ganache testbed has been used for the deployment and tests of SC functionality. For SaCI's Application Programming Interface (API) Flask was used in its latest version. Finally, for the off-chain storage, the prototype uses SQLite. The source-code and all documentation is publicly available at [11].

The code of the function to report a damage is shown, as an example, in Listing 1.2. 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 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 contract covers the date the damage occurred. To do so, the date of the damage is compared to the contract attributes `start_date` and `valid_until` in line 11. Since damage should not be overwritten, it must be ensured that there is 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 contract's 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 lines 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 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 afterward also should be able to challenge paid-out damage automatically in case of fraudulent behavior. Hence, 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 1.3.

```
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 payment
31        if(amount_of_damage < premium && count_of_damages < 4){
32            automaticPayOut(damage_id, false);
33        }
34    }
```

Listing 1.2: Example of the SC Function for Damage Report.

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 counteroffer should be paid 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 lines 3 - 6 checks if the damage was already paid out, canceled, or otherwise resolved to protect the insurer of unintended double payout. If the damage status is not in an ending state, the amount to pay is calculated in lines 7 - 12. Considering the parameter `is_counter_offer`, either the initial value of the reported damage or the value of the counteroffer is converted into Wei using the exchange rate returned from the oracle again. Afterward, 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`.

```
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 1.3: SC for the Automatic Payment.

4 Evaluation

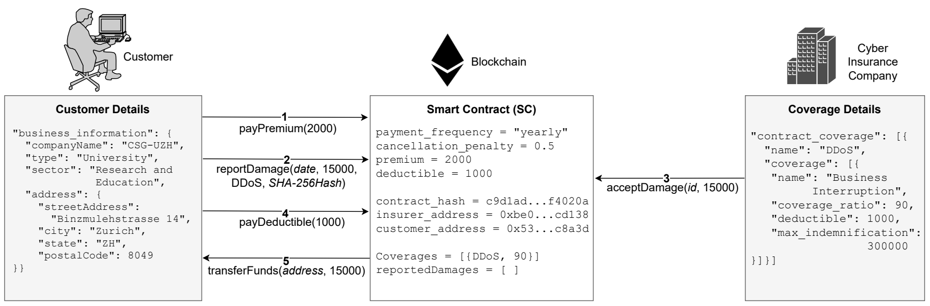
While evaluations of cyber insurance models as such will cover the precision of risk models and their prediction granularity, SaCI's evaluation here focuses on the systems' operations, which are based on a real-world case. Furthermore, cost analysis and discussion concerning its BC-based implementation are provided.

4.1 Case Study

Suppose that a customer wants to protect her business from financial loss possibly caused by Distributed Denial-of-Service (DDoS) attacks. The customer will access SaCI's Web-based interface and fills all information related to her business and respective requirements, such as the company's conditions (*e.g.*, sector, revenue, and number of employees), security aspects (*e.g.*, attacks history, risk assessment, available protections), and coverage demands. The insurer uses this information to propose a contract offering coverage of 90% of all financial loss, if a business interruption happens due to a DDoS attack until a maximum amount of 300,000 €. For that, the deductible amount of 1,000 € is considered besides a yearly premium of 2,000 €. Figure 3 provides an overview of all interactions and actors considered for this case study.

After the customer and insurer decided about the contract off-chain, this generates a JSON file with all information and SC is created with the anonymization of private information (*cf.* Section 3). Finally, the contract is deployed on the

BC and the hash of the JSON file with all contact information is stored together with the SC. Both actors also store a copy of the JSON file (*i.e.*, all contract information without anonymization) in private databases for further reference, while the hash stored in the BC allows for an integrity validation. The customer will finally call the function *payPremium(amount)* to initiate the coverage.



```
"business_information": {
  "companyName": "CSG-UZH",
  "type": "University",
  "sector": "Research and
            Education",
  "address": {
    "streetAddress":
      "Binzmulehstrasse 14",
    "city": "Zurich",
    "state": "ZH",
    "postalCode": 8049
}}
```

```
payment_frequency = "yearly"
cancellation_penalty = 0.5
premium = 2000
deductible = 1000

contract_hash = c9d1ad...f4020a
insurer_address = 0xbe0...cd138
customer_address = 0x53...c8a3d

Coverages = [{DDoS, 90}]
reportedDamages = [ ]
```

```
"contract_coverage": [{
  "name": "DDoS",
  "coverage": [{
   "name": "Business
          Interruption",
   "coverage_ratio": 90,
   "deductible": 1000,
   "max_indemnification":
                  300000
}]}]
```

Fig. 3: Case Study's Information and Flows.

If an attack happened at the customer's IT resulting in 15,000 € of loss, a request for coverage is placed by calling the function *reportDamage(date, amount, type_of_attack, logFile_hash)*. Based on this, the insurer automatically checks, if the request complies to the contract and calls the function *acceptDamage(amount)*, ensuring that the amount is available in the SC for the payment. The amount is automatically sent to the customer in order to pay for her losses. If the damage was not accepted, a counteroffer will be placed or further investigations are required, as discussed above. The logFile_hash allows for the verification of the attack and losses if required. Thus, the insurer can ask the customer to send log files via a secure channel, *e.g.*, containing network traces, reports, or internal analysis data explaining the incident. The hash stored in the BC provides a trustworthy record in case a dispute is required.

4.2 SC Cost Evaluation

Of key relevance for the economic efficiency of such an approach are costs related to the BC-based solution. Thus, Table 3 summarizes all costs for calling functions available in the SC, including the deployment (*i.e.*, Constructor) of the contract. These Gas costs in Ethereum were estimated using the function *estimateGas* provided by the Web3 library. Gas defines the internal pricing to run a transaction or a contract in the Ethereum BC. Gas does "measure" the computational usage in terms of monetary costs (*e.g.*, Gas per Swiss Franc or €) [3]. These functions as of today within the proof-of-concept were not yet optimized in terms of Gas costs; they can be reduced for a production deployment by *(a)* using different implementations of BC, which support SCs, and also *(b)* by optimizing the overall process, such as by increasing the time to process transactions to reduce the amount of Gas that have to be spent. Furthermore, as many BC projects (*e.g.*, Cardano and Polkadot) are promising efficient features, the can enable a cheapest and most efficient way to implement cyber insurance models that rely on SCs.

Gas costs were converted into Wei (*i.e.*, smallest denomination of Ether) using a Gas cost of 20 GWei per Gas, which is the default value of Ganache. The Ether value was converted into € using an exchange rate of 600 € per Ether, which is approximately the current exchange rate as of January 2021; in general the exchange rate from Ether to € changes permanently. The most expensive function is the one that deploys the contract (*i.e.*, Constructor), followed by *reportDamage*.

Table 3: Cost Estimations of SaCI's Functions.

Function	**Estimation in Ether (20 GWei/Gas)**	**Converted in € (600 €/Ether)**
Constructor	0.10893	65.36
paySecurity	0.00080	0.48
payPremium	0.00084	0.50
reportDamage	0.00435	2.61
acceptDamage	0.00109	0.65
declineDamage	0.00174	1.04
acceptCounterOffer	0.00082	0.49
proposeToUpdateContract	0.00264	1.58
agreeToUpdateContract	0.00098	0.59

Although this amount has to be paid by the actors involved, this value does not represent a high values, since it is paid only when the function is called. Therefore, 65 € are paid for the deployment of the contract and 4.5 € have to be paid, when a coverage request is done. Note that all of these values already represent the most expensive case, in which the blocks are mined as fast as possible. Taking a Gas cost of 2 GWei, which is considered a price that usually persists a transaction in a block within the next minutes in the Ethereum network [13], the final cost to deploy a contract can be divided by ten, thus, resulting in a cost of 6.5 €.

These costs can also be affected due to the choice of the BC technology to be used. For this prototype, Ethereum was used for convenience (*i.e.*, support to SC, extensive documentation, and frameworks for development). However, the approach proposed by SaCI can be implemented using any permissioned or permissionless BCs that support SCs implementation, such as Cardano, Polkadot, and Hyperledger Fabric [15]. The decision might depend upon the insurer's demands in terms of performance, privacy, and scalability.

5 Summary, Conclusions, and Future Work

This work presented SaCI, a blockchain-based approach for the creation, deployment, and life-cycle management of cyber insurance contracts. SaCI handles the translation of human-readable demands (*e.g.*, JSON file) to SC contracts executed on the BC. The approach proposed allows users' information input, provides the SC code with all functions for interactions, and deploys the contract coverage information as an SC running on the public Ethereum BC for any interactions required between customers and insurers.

Concluding, the proof-of-concept implementation of SaCI is fully operational and was developed taking into consideration real-life actors and their interactions. Moreover, the system is fully decentralized, with no intermediaries due to the usage of a BC. However, off-chain disputes are still possible to resolve open issues that require interactions, since they cannot be automated at this step (*e.g.*, analysis of log files, agreement between the premium price, and decision about the coverage payment). SaCI's feasibility was investigated by conducting a case study and cost analysis that shows basic interactions of the approach as well as concerns regarding the costs while using public BCs. Besides the advantages introduced by this approach (*e.g.*, automation and trust), it is important to conduct further investigations to verify the role of BC in the future of cyber insurance, such as introducing trust and simplifying the process while reducing its costs.

Future work includes: *(i)* the development of a Web-based interface for the interaction with SaCI and the contract running on the BC, *(ii)* the investigation of premium calculation models that can provide a fair way to define the value of the premium and the coverage amount, and *(iii)* an analysis of different types of BCs (private and hybrid) and distributed systems (*e.g.*, Inter-Planetary File Systems) to increase the efficiency of this solution (in terms of costs, privacy, and time to process transactions), while reducing its overall complexity. Furthermore, additional studies are still required in the field of cyber insurances to map and improve all different tasks required from the creation (*e.g.*, contract underwriting and premium definition) until the termination of a contract.

Acknowledgements. This paper was supported partially by *(a)* the University of Zürich UZH, Switzerland and *(b)* the European Union's Horizon 2020 Research and Innovation Program under Grant Agreement No. 830927, the CONCORDIA Project.

References

1. B. Aziz, Suhardi, Kurnia: A Systematic Literature Review of Cyber Insurance Challenges. In: International Conference on Information Technology Systems and Innovation (ICITSI 2020). Padang, Indonesia, 2020, pp. 357–363
2. A. Farao, S. Panda, S. A. Menesidou, E. Veliou, N. Episkopos, G. Kalatzantonakis, F. Mohammadi, N. Georgopoulos, M. Sirivianos, N. Salamanos, S. Loizou, M.

Pingos, J. Polley, A. Fielder, E. Panaousis, C. Xenakis: SECONDO: A Platform for Cybersecurity Investments and Cyber Insurance Decisions. In: S. Gritzalis, E. R. Weippl, G. Kotsis, A. M. Tjoa, I. Khalil (eds.) Trust, Privacy and Security in Digital Business. Springer International Publishing, Cham, Switzerland, 2020, pp. 65–74

3. M. F. Franco, E. J. Scheid, L. Z. Granville, B. Stiller: BRAIN: Blockchain-based Reverse Auction for Infrastructure Supply in Virtual Network Functions-as-a-Service. In: IFIP Networking 2019 (Networking 2019). Warsaw, Poland, May 2019, pp. 1–9
4. V. Gatteschi, F. Lamberti, C. Demartini, C. Pranteda, V. Santamara: Blockchain and Smart Contracts for Insurance: Is the Technology Mature Enough? Future Internet **10**(2), 2018
5. E. Kenneally: Ransomware: A Darwinian Opportunity for Cyber Insurance. Connecticut Insurance Law Journal Fall Symposium Edition **28.1**, 1–13, 2020
6. N. Kshetri: The Economics of Cyber-Insurance. IT Professional **20**(6), 9–14, 2018
7. N. Kshetri: The evolution of cyber-insurance industry and market: An institutional analysis. Telecommunications Policy **44**(8), 102007, 2020
8. T. Lepoint, G. Ciocarlie, K. Eldefrawy: Blockcisa blockchain-based cyber insurance system. In: IEEE International Conference on Cloud Engineering (IC2E 2018). Orlando, USA, 2018, pp. 378–384
9. S. Morgan: Cybercrime To Cost The World $10.5 Trillion Annually By 2025, November 2020, `https://cybersecurityventures.com/hackerpocalypse-cybercrime-report-2016/`, Last visit June, 2021.
10. S. O. Nam: How much are insurance consumers willing to pay for blockchain and smart contracts? a contingent valuation study. Sustainability **10**(4332), 1–11, 2018
11. M. F. Noah Berni: SaCI - Prototype and Source-Code, January 2021, `https://gitlab.ifi.uzh.ch/franco/saci`, Last visit June, 2021.
12. R. Pal, L. Golubchik, K. Psounis, P. Hui: Will Cyber-Insurance Improve Network Security? A Market Analysis. In: IEEE Conference on Computer Communications (INFOCOM 2014). Toronto, Canada, 2014, pp. 235–243
13. A. Rajeevan: Tokens, Gas and Gas limit in Ethereum, February 2019, `https://arunrajeevan.medium.com/tokens-gas-and-gas-limit-in-ethereum-f07790f56d8f`, Last visit June, 2021.
14. B. Rodrigues, M. F. Franco, G. Paranghi, B. Stiller: SEConomy: A Framework for the Economic Assessment of Cybersecurity . In: 16th International Conference on the Economics of Grids, Clouds, Systems, and Services (GECON 2019). Springer LNCS, Leeds, UK, September 2019, pp. 1–9
15. E. J. Scheid, B. Rodrigues, C. Killer, M. Franco, S. R. Niya, B. Stiller: Blockchains and Distributed Ledgers Uncovered: Clarifications, Achievements, and Open Issues, pp. 1–29. No. 1 in IFIP AICT Festschrifts, Springer, Cham, Switzerland, August 2021, `https://www.springer.com/gp/book/9783030817008`
16. J. Wargin: Insurance Company Technology Trends Transforming the Industry in 2021, January 2021, `https://www.duckcreek.com/blog/insurance-technology-trends/`, Last visit June, 2021.
17. J. Xu, Y. Wu, X. Luo, D. Yang: Improving the Efficiency of Blockchain Applications with Smart Contract based Cyber-insurance. In: IEEE International Conference on Communications (ICC 2020). Dublin, Ireland, 2020, pp. 1–7