ICT: Insured Cryptocurrency Transactions

Aydin Abadi[⋆1], Amirreza Sarencheh[⋆⋆2], Henry Skeoch[⋆⋆⋆3], Thomas Zacharias[†4]

[1] Newcastle University, United Kingdom
[2] The University of Edinburgh, United Kingdom
[3] University College London, United Kingdom
[4] University of Glasgow, United Kingdom

Abstract. Cryptocurrencies have emerged as a critical medium for digital financial transactions, driving widespread adoption while simultaneously exposing users to escalating fraud risks. The irreversible nature of cryptocurrency transactions, combined with the absence of consumer protection mechanisms, leaves users vulnerable to substantial financial losses and emotional distress. To address these vulnerabilities, we introduce Insured Cryptocurrency Transactions (ICT), a novel *decentralized insurance* framework designed to ensure financial recovery for honest users affected by fraudulent cryptocurrency transactions. We rigorously formalize the ICT framework, establishing strong security guarantees to protect against malicious adversaries. Furthermore, we present Insured Cryptocurrency Exchange (ICE), a concrete instantiation of ICT tailored for centralized cryptocurrency exchanges. ICE relies primarily on a standard smart contract and provides a robust mechanism to compensate users in cases of security breaches, insolvency, or fraudulent activities affecting the exchange. We have implemented ICE's smart contract and evaluated its on-chain costs. The evaluation results demonstrate ICE's low operational overhead. To our knowledge, ICT and ICE represent the first *formal* approaches to decentralized insurance frameworks in the cryptocurrency domain.

Keywords: Cryptocurrency, Financial Fraud, Smart Contracts, Decentralized Finance, Decentralized Insurance, Consumer Protection, Risk Management, Secure Transactions.

⋆ aydin.abadi@ncl.ac.uk
⋆⋆ amirreza.sarencheh@ed.ac.uk
⋆⋆⋆ henry.skeoch.19@alumni.ucl.ac.uk
† thomas.zacharias@glasgow.ac.uk

Table of Contents

1 Introduction

Cryptocurrencies have become a prominent financial and technological innovation, attracting attention from investors, entrepreneurs, and the general public. Their growing prominence is evidenced by substantial daily trading volumes. As of early 2025, the daily trading volume across all cryptocurrencies exceeds $162 billion [23]. With the increasing popularity of cryptocurrencies, malicious actors have become more active in seeking to exploit this market. *Cryptocurrency fraud* has surged in recent years, resulting in substantial financial losses documented by government agencies and regulatory bodies. According to the *Financial Times*, losses due to cryptocurrency fraud in the UK rose by 41% year-on-year, reaching £306 million in the 12 months leading up to March 2023, compared to £216.5 million in 2022. Notably, over a third of these losses occurred in November 2022, coinciding with the collapse of the cryptocurrency exchange FTX [88]. The FBI reported that losses from cryptocurrency-related investment fraud in the USA surged to $3.94 billion in 2023, reflecting a 53% increase from the previous year [37]. Unlike traditional banking transactions, cryptocurrency transactions lack consumer protections and are generally irreversible. If users fall victim to fraud, there is no established process for fund recovery, unlike credit card dispute resolution mechanisms [38]. Moreover, cryptocurrency transactions operate without intermediaries, such as banks, to assist with disputes or refunds.

The true cost of cryptocurrency fraud extends beyond immediate financial loss, often imposing additional burdens such as investigative costs, legal fees, and an emotional toll. Victims may incur significant expenses by hiring specialized cybersecurity firms or private investigators to trace stolen funds and gather evidence. In some cases, the victims pursue legal action through civil litigation or by involving law enforcement. Retaining legal representation and dealing with the complexities of the legal process can lead to substantial legal fees, further worsening the financial impact. Beyond financial losses, the emotional toll of cryptocurrency fraud can be profound. Victims may experience a sense of violation, betrayal, and loss of trust, which can lead to considerable psychological distress, including anxiety, depression, and, in extreme cases, post-traumatic stress disorder [31,34,66].

Currently, there is no scientific, formal, and technical mechanism to assist victims of cryptocurrency fraud in receiving reimbursement for their financial losses. To fill this void, we propose a *robust formal* insurance mechanism. Implementing this solution offers a practical and effective approach for several key reasons:

– *Financial Recovery*: Insurance can provide victims with a means to recover financial losses incurred due to fraud. This can help alleviate the immediate financial burden and offer a safety net, ensuring that individuals and businesses are not completely devastated by such incidents [95].
– *Market Stability*: Insurance helps protect users from the uncertainties and risks of the cryptocurrency market, encouraging greater participation with confidence [84]. By offering a safety net, it enhances the market's stability and credibility.
– *Addressing Regulatory Concerns*: Regulatory bodies often express concerns about consumer protection in the cryptocurrency ecosystem [49]. Implementing a dependable insurance mechanism demonstrates a commitment to safeguarding users' interests, potentially alleviating future regulatory pressures [80].

1.1 Our Contributions

In this work, we present "Insured Cryptocurrency Transactions" (ICT), a generic decentralized insurance framework that *insures* clients' financial cryptocurrency transactions with online service providers, i.e., servers. It guarantees that an honest client, who has made a payment but fails to receive the promised services, will receive financial compensation. Figure 1 provides an abstract view of ICT.

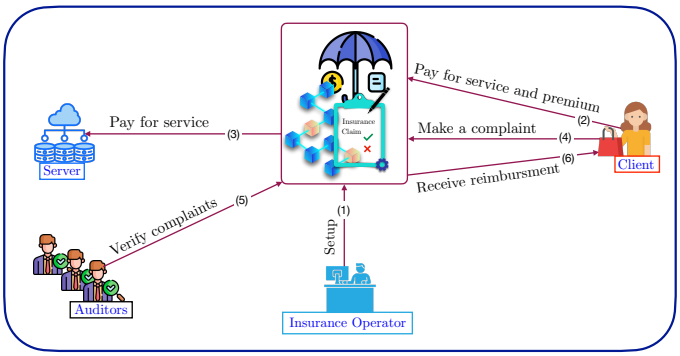

Fig. 1: A high-level illustration of the ICT platform.

We rigorously define ICT and establish its fundamental security guarantees. These assurances are vital for maintaining the framework's security, particularly in scenarios where either clients or servers are corrupted by malicious adversaries seeking to exploit vulnerabilities. Developing precise definitions has demanded far more than merely formalizing the intuitive, informal requirements of decentralized insurance. Our formalisms expand on standard foundational concepts (e.g., signatures [13,14,18] and encryption [45,54,28]), which account for strong adversaries with access to multiple oracles and queries both before and during the execution of the scheme instances. This framework enables adversaries to observe the outputs of various algorithms within the scheme prior to corrupting a party, ensuring a robust and comprehensive security model.

We also introduce Insured Cryptocurrency Exchange (ICE), a concrete instantiation of ICT, tailored for scenarios where the service provider is a *centralized cryptocurrency exchange*. We focus on centralized exchanges due to their broader adoption. For example, Binance, the largest centralized exchange, reported over $10 billion in 24-hour trading volume in mid-2024, compared to Uniswap, the largest decentralized exchange, with $640 million—over 15 times less [81,82]. Along the way, we introduce an abstract formal definition of a centralized cryptocurrency exchange. ICE ensures that users are compensated if the exchange encounters various issues such as security breaches, insolvency, or fraudulent activities. ICT and ICE offer several advantages over traditional insurance companies, including (a) decentralized dispute resolution, ensuring fairness and impartiality through external auditors; (b) real-time access and transparency, allowing immediate visibility into transactions and claims; (c) fixed policy and terms, guaranteeing unchangeable

agreements through smart contracts; and (d) minimized human intervention, reducing errors and biases with automated processes. See Section 8.2 for a detailed discussion. We have implemented ICE's smart contract and made the source code publicly accessible [1]. Our cost analysis indicates that the on-chain process, including smart contract deployment, incurs a total cost of approximately $10 in a single client-server scenario. Of this, around $7.9 is a one-time cost for deploying the smart contract, which can subsequently serve other clients as well.

Summary Of Our Contributions. In this work, we make the following key contributions:

- Introduce, for the first time, a formal concept of Insured Cryptocurrency Transactions (ICT).
- Propose Insured Cryptocurrency Exchange (ICE), a concrete instantiation of ICT, and rigorously prove its security.
- Implement and deploy ICE's smart contract, followed by an analysis of its overhead.

Further Applications. The concept of ICT can be extended to a variety of applications beyond cryptocurrency exchanges. Potential applications include:

– *E-commerce Transactions*: ICT can be used to insure purchases made on e-commerce platforms. In this case, if a customer pays for a product but does not receive it due to fraud, delivery issues, or receives a counterfeit item [92], they are compensated.
– *Online Lending*: When a loan is issued through a blockchain-based peer-to-peer lending platform, the funds are transferred from the lender to the borrower [63]. Using ICT, these transactions can be insured to ensure that the lender's capital is protected. If a borrower defaults on a repayment, the ICT mechanism ensures that the lender receives compensation.
– *Rental Services*: For platforms offering blockchain-based rental services (e.g., Bee Token [83], Dtraverl [30], or Propy [72]), ICT can insure both renters and owners against non-fulfillment of rental agreements.
– *Event Ticketing*: ICT can be used to insure transactions for blockchain-based event tickets (e.g., Aventus [11]), ensuring buyers receive valid tickets or are compensated if there are issues such as cancellations or fraud.
– *Central Bank Digital Currencies (CBDCs) and Stablecoins*: ICT can contribute to the CBDC [16,15,55] and stablecoin [62,86,76] ecosystems by mitigating risks such as fraud and operational disruptions through loss coverage.
– *Proofs of Storage*: Our platform can be used to insure users against data loss within the context of blockchain-based proof of storage [57]. If a storage provider fails to store the data correctly or loses it, the user can claim compensation through the ICT framework.
– *Verifiable Computation*: ICT can also be utilized in verifiable computation [12] to provide insurance against incorrect results. It guarantees that if a server delivers an incorrect computation result, the client is compensated accordingly.
– *Supply Chain*: ICT can insure transactions within blockchain-based supply chains (e.g., IBM Food Trust [51]), ensuring that payments are made for goods that are actually delivered, reducing the risk of fraud, and enhancing transparency.

2 Related Works

This section provides an overview of the solutions utilizing blockchain and smart contracts to improve insurance systems. Appendix A presents a survey of related works. Briefly, despite their potential, these approaches share a significant limitation: the absence of formal methodologies for cryptocurrency insurance, including well-defined security models, provable constructions, and formal proofs. Further limitations of these solutions are outlined below.

2.1 Blockchain-Based Insurance

Zhou et al. [94] propose MIStore, a blockchain-based medical insurance storage system designed for storing data on a blockchain. MIStore employs multiple servers to enhance system security but relies on a centralized insurance company for claim processing and is limited to health insurance. Several high-level blockchain-based cyber-insurance frameworks have been explored. Lepoint et al. [58] propose a system to assist centralized insurance companies by facilitating insurer-customer interactions through Hyperledger, a permissioned blockchain. Farao et al. [35] present a framework designed to support cybersecurity investment and insurance pricing decisions, using blockchain to record data, enhance transparency, and automate claim processing.

A scheme has been proposed to construct a smart contract encapsulating agreements between a customer and an insurance company [40], leveraging Ethereum to record information and transfer funds. However, it lacks rigorous technical details and depends on a centralized company. Loukil et al. [61] introduce a decentralized insurance system to automate policy management and claim processing. While implemented using Ethereum smart contracts, it is restricted to policies fully executable by smart contracts. Bhawana et al. [56] propose a framework that integrates fire detection and insurance systems into a unified platform for managing fire brigade service requests and insurance claims efficiently. Conceptually, it is an intriguing system. Nevertheless, it remains dependent on a centralized insurance company and lacks broad applicability. A recent cyber-insurance framework incorporates "Know Your Customer" (KYC) procedures [36]. It utilizes a decentralized identity management system to verify participant identities and facilitate KYC but still heavily relies on a centralized insurance company. Deb et al. [26] introduces an economic model and mechanism for insurance in Proof-of-Stake (PoS) blockchains. It aims to provide cryptoeconomic safety, to ensure that no honest users of the blockchain suffer financial losses even in the event of attacks on the blockchaian safety, e.g., chain reorganizations. Nevertheless, it lacks generality, due to its focus on only PoS and specific attack vectors.

2.2 Commercial Cryptocurrency Insurance

Several commercial cryptocurrency insurance platforms [24,43,70,59,8,64,17] have been developed, providing coverage against cyberattacks and technical failures. For example, "OneInfinity" [70] provides cryptocurrency wallet insurance, offering coverage against the risk of losing wallet private keys due to natural disasters, physical damage, or cyberattacks. However, these platforms suffer from a lack of detailed documentation and transparent methodologies, making it challenging to evaluate their security effectively.

2.3 Reversible Cryptocurrency Transactions

To mitigate theft-related losses in the blockchain ecosystem, Eigenmann [32] introduced the concept of reversible transactions, enabling payees to withdraw payments within a set time frame using standard ERC-20 contract templates. This approach was later enhanced by incorporating a panel of judges who review user queries and determine whether a transaction should be reversed [89,90]. Reversible transactions can be useful in certain situations, particularly when transactions involve indecent content [10]. However, their effectiveness in financial transactions is questionable, as a committee's authority to decide payment outcomes could undermine trust in the cryptocurrency system, leaving users vulnerable to losing funds based on the committee's decisions. This is not the case in ICT, as funds obtained by recipients are never subject to reversal.

3 Preliminaries

3.1 Notations and Assumptions

A scheme for Insured Cryptocurrency Transactions (ICT) involves five types of entities. Below, we informally explain each of them. We will provide a formal definition of the scheme in Section 4.

- **Insurance operator** ($\mathcal{O}$): A trusted third party that registers servers and auditors into a smart contract after local verification.
- **Servers** ($\mathcal{S}_1, \ldots, \mathcal{S}_k$): Service providers that accept digital currency in exchange for their services.
- **Clients** ($\mathcal{C}_1, \ldots, \mathcal{C}_n$): Customers of servers, referred to as victims if subjected to cryptocurrency fraud.
- **Smart contract** ($\mathcal{SC}$): A standard blockchain-based smart contract, e.g., in Ethereum.
- **Committee of auditors** ($\mathcal{D}_1, \ldots, \mathcal{D}_m$): Third-party authorities that handle complaints and issue verdicts.

We assume that parties perform their off-chain interactions over a secure communication channel and that the operator remains honest. Other parties may be corrupted by a malicious (i.e., active) adversary [44], (however, in the security games, the adversary should specify a challenge pair of a client and a server such that not both parties are corrupted). Additionally, a minority of auditors may be corrupted. The default value of any variable is $\perp$.

3.2 Distributed Ledger

We adopt the definitions of *persistence* and *liveness* for public transaction ledgers as described in [41]. These properties are critical for ensuring the consistency and progress of a distributed ledger. Here, we provide brief and informal definitions. Definitions 5 and 6 formally state them.

- *Persistence.* It ensures that once a transaction is incorporated into the blockchain of an honest participant at a depth of more than k blocks from the chain's end, it will, with overwhelming probability, be included in the blockchain of every other honest participant. Furthermore, the transaction will occupy a fixed and permanent position in the ledger.
- *Liveness.* It guarantees that all transactions initiated by honest account holders will eventually reach a depth greater than k blocks in the blockchain of at least one honest participant. Consequently, the adversary is unable to execute a selective denial-of-service attack against transactions originating from honest account holders.

Smart Contracts. Cryptocurrencies like Bitcoin [67] and Ethereum [91] offer more than just decentralized currency functionality; they also enable computational operations on transactions. These platforms allow users to encode specific computational logic into programs known as *smart contracts*. Currently, Ethereum is the leading cryptocurrency framework that supports the definition and deployment of arbitrary smart contracts. In this framework, contract code is stored on the blockchain and executed by all participants maintaining the cryptocurrency. The correctness of execution is guaranteed by the security of the underlying blockchain infrastructure. In this work, we utilize standard public (Ethereum) smart contracts.

4 Formal Framework

4.1 Predicates and Functions

An ICT scheme is parameterized by a set of predicates and functions, defined below.

- $\mathsf{Validate}^{(\mathcal{C})}(st_{\mathcal{C}_i}, adr_{\mathcal{S}_j}, adr_{\mathcal{SC}}) \to \hat{a}$: A predicate run by a client $\mathcal{C}_i$. It is used to check whether a server $\mathcal{S}_j$ has been registered in a specific smart contract and is reliable. It takes as input $\mathcal{C}_i$'s state $st_{\mathcal{C}_i}$ (which may contain information about $\mathcal{S}_j$), $\mathcal{S}_j$'s address $adr_{\mathcal{S}_j}$, and the smart contract's address $adr_{\mathcal{SC}}$. It returns 1 if the verification passes, and 0 otherwise.
- $\mathsf{Validate}^{(\mathcal{O})}(plc_\lambda, st_{\mathcal{O}}, msg_P) \to \hat{b}$: A predicate run by operator $\mathcal{O}$. It is used to check whether a party P is reliable according to the insurance's policy. It takes as input, $\mathcal{O}$'s policy: plc_λ, $\mathcal{O}$'s state: $st_{\mathcal{O}}$, and P's message: msg_P (that may contain P's address, adr_P). It returns 1 if the verification passes, and 0 otherwise.

- $\mathsf{CalPremium}(state_{\mathcal{SC}}, adr_{\mathcal{S}_j}, req_{\mathcal{C}_i}) \rightarrow \tau$: a function that determines the amount of premium that a $\mathcal{C}_i$ should pay. It can be run by any party. It takes as input smart contract state $state_{\mathcal{SC}}$, $\mathcal{S}$'s address $adr_{\mathcal{S}_j}$, and $\mathcal{C}_i$'s request $req_{\mathcal{C}_i}$ that includes the amount α a client $\mathcal{C}_i$ wants to pay to a server $\mathcal{S}_j$ for its service, and maybe a description of the service for which it wants to send α amount.[5] It returns the amount of premium τ.
- $\mathsf{CompRemAmount}(plc_\lambda, \gamma, req_{\mathcal{C}_i}, pp) \rightarrow amount$: a function that determines the amount of reimbursement a victim of fraud should receive. It takes insurance policy plc_λ, the total amount of coins γ (including premium) that $\mathcal{C}_i$ sent, $\mathcal{C}_i$'s request $req_{\mathcal{C}_i}$ for a service (that includes the amount α of the service payment that $\mathcal{C}_i$ wants to cover), and public parameters pp. It returns the reimbursement amount $amount$ that $\mathcal{C}_i$ should receive.
- $\mathsf{ExtractVerdict}(plc_\lambda, e_1, \ldots, e_m, adr_{\mathcal{C}_i}) \rightarrow \hat{r}$: a predicate that determines whether a client $\mathcal{C}_i$ must be reimbursed. It takes as input policy plc_λ (which may contain an integer t defining a threshold), all auditors' votes $e_1, \ldots, e_m$, and $\mathcal{C}_i$'s address $adr_{\mathcal{C}_i}$. It returns 1 if $\mathcal{C}_i$ should be reimbursed. It returns 0 otherwise.
- $\mathsf{UpdateList}(list, T, \theta) \rightarrow list$: an algorithm that takes a list of triples $list = \{(.,.,T_1), \ldots, (.,.,T_m)\}$ and value θ, where T is current time, T_i is a specific time, and θ is a positive value. It deletes each $(.,.,T_i)$ in the list if $T - T_i > \theta$. It returns the updated $list$.
- $\mathsf{CheckBudget}(req_{\mathcal{C}_i}, adr_{\mathcal{S}_j}, adr_{\mathcal{SC}}, \mathsf{balance}, state_{\mathcal{SC}}, plc_\lambda, \gamma, pp, L_{\mathsf{pnd}}) \rightarrow \hat{w}$: a predicate that determines if a smart contract $\mathcal{SC}$ has enough budget to serve a new client $\mathcal{C}_i$. It takes (1) $req_{\mathcal{C}_i}$: $\mathcal{C}_i$'s request that includes the amount α the client wants to send, (2) $adr_{\mathcal{S}}$: address of $\mathcal{S}_j$, (3) $\mathsf{balance}$: the amount of usable funds held by $\mathcal{SC}$, (4) plc_λ: policy of $\mathcal{S}_j$, (5) γ: the total amount (that includes service payment plus premium) that $\mathcal{C}_i$ wants to pay, (6) pp: public parameters, and (7) L_{pnd}: a list of pending transactions. If $\mathcal{SC}$ has enough budget, it returns 1. Otherwise, it returns 0.
- $\mathsf{Update.State}(st_P, pp, sk_P, data_{ledger}) \rightarrow st_P$: an algorithm executed by a party P. It takes P's state st_P, public parameters pp, P's secret key sk_P, and the blockchain's state $data_{ledger}$. The algorithm updates st_P and returns an updated state st_P.

4.2 Syntax

An ICT scheme comprises a set of algorithms and protocols, which we formally define below. The complexity of the presented syntax arises from our deliberate effort to capture a broad range of real-world scenarios in the context of insurance while preserving generality. These scenarios include: (i) enabling clients to withdraw funds, (ii) allowing the operator to vet auditors and servers, (iii) allowing clients or servers to decide whether to transact with their counterparts based on their responses, and (iv) permitting the decentralized insurance system to determine whether to serve a client based on its available budget.

For each party P, we define a state st_P that incorporates its background knowledge about other parties. During the execution of certain algorithms or protocols, the party updates its state.[6]

- $\underline{\mathtt{Operator.Setup}}(1^\lambda, plc_\lambda) \rightarrow (st_{\mathcal{O}}, pp, adr_{\mathcal{SC}}, \langle\mathcal{SC}\rangle, sk_{\mathcal{O}})$: an algorithm run by $\mathcal{O}$. It takes as input the security parameter 1^λ and the insurance policy plc_λ, where plc_λ specifies (i) an *insurance period* Θ: the duration that a client is insured, and (ii) a *transaction period* Δ: a fixed period during which certain processes must be executed with respect to this time frame. It generates a public-secret key pair for the operator $(sk_{\mathcal{O}}, pk_{\mathcal{O}})$; it also generates a smart contract $\mathcal{SC}$ (which encodes plc_λ), an initial state $st_{\mathcal{O}}$ for $\mathcal{O}$, and public parameters pp that include $pk_{\mathcal{O}}, \Theta$, and Δ. Let $\langle\mathcal{SC}\rangle$ denote the code of $\mathcal{SC}$. The algorithm deploys $\mathcal{SC}$ on the blockchain, initialized with a deposit of Γ coins, where Γ is specified in plc_λ. $\mathcal{SC}$ maintains a variable $\mathsf{balance} \leftarrow \Gamma$ and a (pending) list L_{pnd} initialized as empty. Let $adr_{\mathcal{SC}}$ be the address of the deployed $\mathcal{SC}$. It returns $st_{\mathcal{O}}, pp, adr_{\mathcal{SC}}, \langle\mathcal{SC}\rangle$, and $sk_{\mathcal{O}}$. The operator $\mathcal{O}$ publishes $adr_{\mathcal{SC}}$ and sends pp to $\mathcal{SC}$.

[5] In traditional insurance, this would be known as the policy limit or cover.

[6] Defining a dynamic state offers two distinct advantages: (i) it enables us to capture the real-world context in which each party assesses its interactions with other parties based on the background knowledge it possesses about them, and (ii) it allows us to account for the dynamic nature of a party's background knowledge, which evolves over time in response to its counterparts' past known behavior.

- $\texttt{Party.GenParams}(1^\lambda, (P, adr_P)) \rightarrow (sk_P, pk_P)$: an algorithm run by each client, server, and auditor, denoted as P. It takes as input the security parameter 1^λ and the address adr_P associated with P, and returns the secret and public parameters (sk_P, pk_P).
- $\texttt{Register}\Big(\langle P(msg_P, sk_P), \mathcal{O}(st_\mathcal{O}, adr_{\mathcal{SC}}, sk_\mathcal{O})\rangle\Big) \rightarrow \hat{c}$: a protocol between $\mathcal{O}$ and a party P, where P is either an auditor or a server. The party P initiates registration by sending a message msg_P to $\mathcal{O}$. The message msg_P includes adr_P and, if P is a server, also includes the description of the service function $\mathtt{F}_P$ and a set $\mathbf{V}_P$ of clients' inputs that are valid for the service provided by P. $\mathcal{O}$ takes as input the state $st_\mathcal{O}$, the address $adr_{\mathcal{SC}}$, and the secret key $sk_\mathcal{O}$. It reads plc_λ from $\mathcal{SC}$ and runs $\mathsf{Validate}^{(\mathcal{O})}(plc_\lambda, st_\mathcal{O}, msg_P) \rightarrow \hat{b}$. If $\hat{b} = 1$, $\mathcal{O}$ sets $\hat{c} \leftarrow 1$ and registers P by sending $(adr_P, \hat{c}, \mathtt{F}_P, \mathbf{V}_P, \alpha_P)$ or $(adr_P, \hat{c})$ to $\mathcal{SC}$, depending on whether P is a server or an auditor, respectively. If $\hat{b} = 0$, $\mathcal{O}$ sets $\hat{c} \leftarrow 0$ and sends the message $(\mathsf{Registration}, \hat{c})$ to $\mathcal{SC}$.
- $\texttt{C.Init}(inpt_{\mathcal{C}_i}, sk_{\mathcal{C}_i}, pk_{\mathcal{C}_i}, adr_{\mathcal{S}_j}, adr_{\mathcal{SC}}, st_{\mathcal{C}_i}) \rightarrow (inpt^*_{\mathcal{C}_i}, \hat{a}, pp_{\mathcal{C}_i})$: an algorithm run by a client $\mathcal{C}_i$. It takes as input $\mathcal{C}_i$'s input $inpt_{\mathcal{C}_i}$ (which may depend on a specific service of interest), $\mathcal{C}_i$'s secret and public keys $(sk_{\mathcal{C}_i}, pk_{\mathcal{C}_i})$, the address of a server $adr_{\mathcal{S}_j}$, as well as $adr_{\mathcal{SC}}$ and the state $st_{\mathcal{C}_i}$. The algorithm runs $\mathsf{Validate}^{(\mathcal{C})}(st_{\mathcal{C}_i}, adr_{\mathcal{S}_j}, adr_{\mathcal{SC}}) \rightarrow \hat{a}$ to check whether $\mathcal{S}_j$ has been registered and is considered reliable. If:
  - $\hat{a} = 1$, it generates public parameters $pp_{\mathcal{C}_i}$ (including an identifier $id\_inpt_{\mathcal{C}_i}$ linked to $inpt_{\mathcal{C}_i}$), and the input's representation $inpt^*_{\mathcal{C}_i}$ that includes $id\_inpt_{\mathcal{C}_i}$. It returns $(inpt^*_{\mathcal{C}_i}, \hat{a}, pp_{\mathcal{C}_i})$. $\mathcal{C}_i$ sends $(inpt^*_{\mathcal{C}_i}, adr_{\mathcal{C}_i})$ to $\mathcal{S}_j$ and $pp_{\mathcal{C}_i}$ to $\mathcal{SC}$.
  - $\hat{a} = 0$, it returns $(\perp, \hat{a}, \perp)$, and $\mathcal{C}_i$ takes no further action.
- $\texttt{S.Init}(inpt^*_{\mathcal{C}_i}, adr_{\mathcal{C}_i}, sk_{\mathcal{S}_j}, pk_{\mathcal{S}_j}, st_{\mathcal{S}_j}, adr_{\mathcal{SC}}) \rightarrow (\hat{e}, resp_{\mathcal{S}_j})$: an algorithm run by $\mathcal{S}_j$ upon receiving $(inpt^*_{\mathcal{C}_i}, adr_{\mathcal{C}_i})$ from $\mathcal{C}_i$. It takes as input $inpt^*_{\mathcal{C}_i}$, $adr_{\mathcal{C}_i}$, $sk_{\mathcal{S}_j}$, $pk_{\mathcal{S}_j}$, $st_{\mathcal{S}_j}$, and $adr_{\mathcal{SC}}$. The algorithm reads $pp_{\mathcal{C}_i}$ and pp from $\mathcal{SC}$ and checks the validity of $inpt^*_{\mathcal{C}_i}$. If the check passes, it sets $\hat{e} \leftarrow 1$. Otherwise, it sets $\hat{e} \leftarrow 0$. It returns $\hat{e}$ and a server's response $resp_{\mathcal{S}_j}$. $\mathcal{S}_j$ sends $(id\_inpt_{\mathcal{C}_i}, \hat{e}, resp_{\mathcal{S}_j}, adr_{\mathcal{C}_i})$ to $\mathcal{SC}$.
- $\texttt{SendTransaction}(\mathcal{C}_i(id\_inpt_{\mathcal{C}_i}, sk_{\mathcal{C}_i}, inpt_{\mathcal{C}_i}, st_{\mathcal{C}_i}, \alpha, adr_{\mathcal{S}_j}, adr_{\mathcal{SC}}), \mathcal{SC}) \rightarrow (\gamma, tx_i, req_{\mathcal{C}_i})$: a protocol between $\mathcal{C}_i$ and $\mathcal{SC}$. $\mathcal{C}_i$ takes as input $id\_inpt_{\mathcal{C}_i}$, $sk_{\mathcal{C}_i}$, $inpt_{\mathcal{C}_i}$, $st_{\mathcal{C}_i}$, the amount α that $\mathcal{C}_i$ wants to pay for a service that will be offered by $\mathcal{S}_j$, $adr_{\mathcal{S}_j}$, and $adr_{\mathcal{SC}}$. It reads $\mathcal{SC}$, including $\hat{e}, resp_{\mathcal{S}_j}$, and pp. It decides whether it wants to proceed (based on $\mathcal{S}_j$'s response) by running the predicate $\mathsf{Decide}(st_{\mathcal{C}_i}, \alpha, inpt_{\mathcal{C}_i}, resp_{\mathcal{S}_j})$ that outputs $\hat{q}$. It proceeds only if $\hat{e} = 1$ and $\hat{q} = 1$ (else, $\mathcal{C}_i$ outputs $\perp$). It generates a service request $req_{\mathcal{C}_i}$ (that includes $id\_inpt_{\mathcal{C}_i}$ and α) and sets $\gamma \leftarrow \mathsf{CalPremium}(state_{\mathcal{SC}}, adr_{\mathcal{S}_j}, req_{\mathcal{C}_i}) + \alpha$. Client $\mathcal{C}_i$ sends $req_{\mathcal{C}_i}$, $adr_{\mathcal{S}_j}$, and γ coins to $\mathcal{SC}$ which checks if it holds sufficient budget to serve $\mathcal{C}_i$, by calling $\mathsf{CheckBudget}(req_{\mathcal{C}_i}, adr_{\mathcal{S}_j}, adr_{\mathcal{SC}}, \mathsf{balance}, state_{\mathcal{SC}}, plc_\lambda, \gamma, pp, L_{\mathsf{pnd}}) \rightarrow \hat{w}$. If the check does not pass, it returns $\perp$ and transfers γ coins to $\mathcal{C}_i$ and takes no further action. Otherwise, it generates a unique transaction identifier tx_i (that includes $adr_{\mathcal{C}_i}$ and $adr_{\mathcal{S}_j}$), and sets two flags $\hat{f}$ and $\hat{g}$ to 0. Let T_1 be the time when tx_i was generated. $\mathcal{SC}$ appends $(tx_i, \mathsf{CompRemAmount}(plc_\lambda, \gamma, req_{\mathcal{C}_i}, pp), T_1)$ to L_{pnd}. When $\mathcal{SC}$ does not return $\perp$, $\mathcal{C}_i$ sends $(tx_i, inpt_{\mathcal{C}_i})$ to $\mathcal{S}_j$.
- $\texttt{VerRequest}(adr_{\mathcal{SC}}, tx_i, inpt_{\mathcal{C}_i}) \rightarrow \hat{f}$: an algorithm run by $\mathcal{S}_j$ upon receiving $(tx_i, inpt_{\mathcal{C}_i})$ from $\mathcal{C}_i$. It takes as input $adr_{\mathcal{SC}}$, tx_i, and $inpt_{\mathcal{C}_i}$. It reads $\mathcal{SC}$ (including $adr_{\mathcal{C}_i}, \hat{e}, resp_{\mathcal{S}_j}, req_{\mathcal{C}_i}$, and $pp_{\mathcal{C}_i}$ associated with tx_i and pp). It proceeds if $\hat{e} = 1$. It checks the validity of $req_{\mathcal{C}_i}$. If the check passes, it sets $\hat{f} \leftarrow 1$. It returns $\hat{f}$. Subsequently, $\mathcal{S}$ sends $\hat{f}$ to $\mathcal{SC}$.
- $\texttt{Withdraw}(adr_{\mathcal{C}_i}, pp, tx_i, T, T_1) \rightarrow \hat{g}$: it is run by $\mathcal{SC}$ (invoked by $\mathcal{C}_i$). It takes as input $adr_{\mathcal{C}_i}$, pp, tx_i, current time T, and the time T_1 when $\mathcal{C}_i$ registered a transaction tx_i. It checks whether $\mathcal{C}_i$ is entitled to withdrawal. If the check passes, it returns γ coins (that $\mathcal{C}_i$ paid via transaction tx_i) to address $adr_{\mathcal{C}_i}$, updates L_{pnd}, and sets $\hat{g} \leftarrow 1$. Otherwise, it sets $\hat{g} \leftarrow 0$. It returns $\hat{g}$.
- $\texttt{Transfer}(\hat{e}, \hat{f}, \hat{g}, tx_i, \alpha, \gamma, \Delta, T, T_1) \rightarrow \hat{h}$: an algorithm run by $\mathcal{SC}$ (invoked by $\mathcal{S}_j$). It takes as input $\hat{e}, \hat{f}, \hat{g}, tx_i$, the amount α (specified in $req_{\mathcal{C}_i}$) to be sent to $\mathcal{S}_j$ (specified in tx_i), the amount γ sent to $\mathcal{SC}$ via tx_i, Δ, current time T, and time T_1 when $\mathcal{C}_i$ registered tx_i. It proceeds if $\hat{e} = 1, T - T_1 > \Delta$, and $\hat{g} = 0$ (else, it outputs $\perp$). Let $state_{\mathcal{SC}}[T_1]$ be the data added to $\mathcal{SC}$ before time T_1. If $\hat{f} = 0$, it returns $\hat{h} = 0$ and $\mathcal{SC}$ sends γ coins back to $\mathcal{C}_i$, updates L_{pnd}, and does not take any action regarding tx_i. If $\hat{f} = 1$, it proceeds as follows. It reads request $req_{\mathcal{C}_i}$ associated with tx_i. If $\gamma \geq \mathsf{CalPremium}(state_{\mathcal{SC}}[T_1], adr_{\mathcal{S}_j}, req_{\mathcal{C}_i}) + \alpha$, it

(i) transfers α coins to $\mathcal{S}_j$, (ii) keeps $\mathsf{CalPremium}(state_{\mathcal{SC}}[T_1], adr_{\mathcal{S}_j}, req_{\mathcal{C}_i})$ coins, accordingly updates its balance, (iii) returns the remaining of γ to $\mathcal{C}_i$, and (iv) sets $\hat{h} \leftarrow 1$. Otherwise, it (i) sends γ coins back to the sender $\mathcal{C}_i$ of the coins, (ii) updates L_{pnd}, and (iii) sets $\hat{h} \leftarrow 0$. Let T_2 be the time that `Transfer` algorithm is completed.

- $\underline{\texttt{Serve}(sk_{\mathcal{S}_j}, adr_{\mathcal{SC}}, tx_i) \rightarrow (service_j, \pi_j)}$: an algorithm run by $\mathcal{S}_j$ upon receiving α coins from $\mathcal{SC}$ for transaction tx_i. It takes as input $sk_{\mathcal{S}}, adr_{\mathcal{SC}}$, and tx_i. It reads $\mathcal{SC}$. It proceeds only if the bit $\hat{h}$ associated with tx_i is 1 (else, it outputs $\perp$). It sets $service_j$ to the output of $\mathtt{F}_{\mathcal{S}_j}(inpt^*_{\mathcal{C}_i}, req_{\mathcal{C}_i}, pp)$ and sets π_j to the detail of the service delivery (that may include proof of the service delivery). When the service is not delivered, it sets $service_j$ and π_j to $\perp$. It returns $(service_j, \pi_j)$. $\mathcal{S}_j$ sends $service_j$ to $\mathcal{C}_i$ and π_j to $\mathcal{SC}$.
- $\underline{\texttt{GenComplaint}(sk_{\mathcal{C}_i}, service_j, tx_i, adr_{\mathcal{SC}}) \rightarrow complaint_i}$: an algorithm run by $\mathcal{C}_i$ upon receiving $service_j$ from $\mathcal{S}_j$ or if the current time is greater than $T_2 + \Delta$ (where $\Delta \in pp$) and $\mathcal{C}_i$ has received no service. It takes as input $sk_{\mathcal{C}_i}, service_j$, tx_i, and $adr_{\mathcal{SC}}$. It reads $\mathcal{SC}$ (including π_j and $req_{\mathcal{C}_i}$). It verifies if the service has been delivered. If verification does not pass, it generates a complaint $complaint_i$ (that contains evidence, $adr_{\mathcal{C}_i}$, and tx_i). Otherwise, it sets $complaint_i \leftarrow \perp$. In either case, it returns $complaint_i$. If $complaint_i \neq \perp$, $\mathcal{C}_i$ sends $complaint_i$ to $\mathcal{SC}$. Let T_3 be the time $complaint_i$ is registered in $\mathcal{SC}$.
- $\underline{\texttt{Reimburse}\Big(\langle \mathcal{D}_1(adr_{\mathcal{SC}}, sk_{\mathcal{D}_1}), \ldots, \mathcal{D}_m(adr_{\mathcal{SC}}, sk_{\mathcal{D}_m}), \mathcal{SC}(plc_\lambda, \alpha, \gamma, complaint_i, pp)\rangle\Big) \rightarrow amount_i}$: a protocol run among registered auditors $\mathcal{D}_1, \ldots, \mathcal{D}_m$ and $\mathcal{SC}$. Each $\mathcal{D}_\ell$ takes the following steps: (i) reads $\mathcal{SC}$ (including $T_2, complaint_i$, and plc_λ); (ii) makes a voting decision $\hat{d}_\ell \in \{0, 1\}$; and (iii) sends $(complaint_i, \hat{d}_\ell)$ to $\mathcal{SC}$ before time $T = T_3 + \Delta$. Upon receiving the decision bits $\hat{d}_1, \ldots, \hat{d}_m$ from auditors (if auditor $\mathcal{D}_\ell$ is not registered or it did not send its decision bit until time $T = T_3 + \Delta$, then $\hat{d}_\ell$ is set to $\perp$) and on input $plc_\lambda, \alpha, \gamma, complaint_i$, and pp, $\mathcal{SC}$ takes the following steps: (i) calls $\mathsf{ExtractVerdict}(plc_\lambda, \hat{d}_1, \ldots, \hat{d}_m, adr_{\mathcal{C}_i}) \rightarrow \hat{r}$ to determine the final verdict; (ii) if $\hat{r} = 0$, sets $amount_i = 0$; (iii) if $\hat{r} = 1$, determines the amount of reimbursement that $\mathcal{C}_i$ must receive by calling $\mathsf{CompRemAmount}(plc_\lambda, \alpha, \gamma, req_{\mathcal{C}_i}, pp) \rightarrow amount_i$ (where $req_{\mathcal{C}_i}$ is the request linked to tx_i contained in $complaint_i$); (iv) if $amount_i \neq 0$, updates as $\mathsf{balance} \leftarrow \mathsf{balance} - amount_i$, deletes $(tx_i, \cdot, \cdot)$ from L_{pnd}, and sends $amount_i$ coins to $adr_{\mathcal{C}_i}$.

4.3 Properties

An ICT must meet *correctness* and *security* properties. Informally, correctness entails that when both the client and server are honest:

- when a client does not want to withdraw its payment from $\mathcal{SC}$ within a certain period, then at the conclusion of the protocol's execution: (i) the operator registers the servers and auditors, (ii) the server accepts the client's input, (iii) the server accepts the client's request (iv) $\mathcal{SC}$ pays the server, and (v) the client accepts the service, its proof, and does not invoke the auditors.
- when a client wants to withdraw its payment from $\mathcal{SC}$ within a predefined period, then: (i) the operator registers the servers and auditors, (ii) the server accepts the client's input, (iii) the server accepts the client's request, and (iv) the client fully recovers its payment from $\mathcal{SC}$.

Security states that: (a) an honest client does not accept an invalid service or it will receive a predefined compensation for an invalid service delivered by a server, (b) an honest server will not accept an invalid request from a client, and (c) a client will not receive reimbursement if the server delivered a valid service.

5 Security Model

5.1 Special Commands

Each property of an ICT scheme is formalized through a game between a challenger Ch and the adversary $\mathcal{A}$. The challenger Ch initially runs $\texttt{Operator.Setup}(1^\lambda, plc_\lambda)$ and plays the role of the honest parties, e.g., it maintains the variable $\mathsf{balance}$ and the list L_{pnd} of $\mathcal{SC}$. Also, $\mathcal{A}$ controls the corrupted parties and orchestrates the execution by issuing the following types of special commands:

- (PARAMS_GENERATION, adr_P): if this is the first time that it has received this command and party P is honest, Ch executes $\texttt{Party.GenParams}(1^\lambda, adr_P)$, which produces a pair (sk_P, pk_P); otherwise, it returns $\perp$ to $\mathcal{A}$. It sends pk_P to $\mathcal{A}$.
- (REGISTRATION, msg_P): if this is the first time that it has received this command and P is an auditor or a server, Ch executes $\mathcal{O}$'s steps in the $\texttt{Register}$ protocol (else it returns $\perp$ to $\mathcal{A}$). The format of msg_P should be such that it includes adr_P and, if P is a server, the description of (i) the service function $\texttt{F}_P$, (ii) the set $\mathbf{V}_P$ of clients' inputs that are valid for the service that P provides, and (iii) the expected amount α to be paid for delivering its service (else it returns $\perp$ to $\mathcal{A}$). Then, it updates the data added to $\mathcal{SC}$ according to the output $\hat{b}$ of $\mathsf{Validate}^{(\mathcal{O})}(plc_\lambda, st_\mathcal{O}, msg_P)$.
- (CLIENT_INIT, $inpt_{\mathcal{C}_i}, adr_{\mathcal{C}_i}, adr_{\mathcal{S}_j}$): if it has previously received the command (PARAMS_GENERATION, $adr_{\mathcal{C}_i}$) and $\mathcal{C}_i$ is honest, Ch runs the algorithm $\texttt{C.Init}(inpt_{\mathcal{C}_i}, sk_{\mathcal{C}_i}, pk_{\mathcal{C}_i}, adr_{\mathcal{S}_j}, adr_{\mathcal{SC}}, st_{\mathcal{C}_i})$ (else it returns $\perp$ to $\mathcal{A}$). If the algorithm outputs $(\perp, 0, \perp)$, it takes no further action. If the algorithm outputs $(inpt^*_{\mathcal{C}_i}, 1, pp_{\mathcal{C}_i})$, Ch adds $pp_{\mathcal{C}_i}$ to $\mathcal{SC}$ and sends $(inpt^*_{\mathcal{C}_i}, adr_{\mathcal{C}_i})$ to $\mathcal{S}_j$.
- (SEND_TRANSACTION, $id\_inpt_{\mathcal{C}_i}, \alpha, adr_{\mathcal{C}_i}, adr_{\mathcal{S}_j}$): if $\mathcal{C}_i$ is honest and there exists a tuple $(id\_inpt_{\mathcal{C}_i}, \hat{e}, resp_{\mathcal{S}_j}, adr_{\mathcal{C}_i})$ in $data_{\mathcal{SC}}$, Ch runs $\texttt{SendTransaction}(\mathcal{C}_i(id\_inpt_{\mathcal{C}_i}, sk_{\mathcal{C}_i}, inpt_{\mathcal{C}_i}, st_{\mathcal{C}_i}, \alpha, adr_{\mathcal{S}_j}, adr_{\mathcal{SC}}), \mathcal{SC}) \rightarrow (\gamma, tx_i, req_{\mathcal{C}_i})$ between $\mathcal{C}_i$ and $\mathcal{SC}$ (else it returns $\perp$ to $\mathcal{A}$). Specifically, Ch runs $\mathsf{Decide}(st_{\mathcal{C}_i}, inpt_{\mathcal{C}_i}, resp_{\mathcal{S}_j})$ that outputs $\hat{q}$. It proceeds if $\hat{e} = 1$ and $\hat{q} = 1$ (else, it returns $\perp$ to $\mathcal{A}$). It generates a request $req_{\mathcal{C}_i}$ for the service, including $id\_inpt_{\mathcal{C}_i}$ and α, and computes an amount $\gamma \leftarrow \mathsf{CalPremium}(data_{\mathcal{SC}}, adr_{\mathcal{S}_j}, req_{\mathcal{C}_i}) + \alpha$. It adds $(id\_inpt_{\mathcal{C}_i}, \gamma, adr_{\mathcal{S}_j})$ to $\mathcal{SC}$. It checks if there is sufficient budget to serve $\mathcal{C}_i$, by calling $\mathsf{CheckBudget}(req_{\mathcal{C}_i}, adr_{\mathcal{S}_j}, adr_{\mathcal{SC}}, \mathsf{balance}, data_{\mathcal{SC}}, plc_\lambda, \gamma, pp, L_{\mathsf{pnd}}) \rightarrow \hat{w}$. If the check passes, it generates a transaction identifier tx_i and sets two associated flags $\hat{f}$, $\hat{g}$ to 0 (if there is no sufficient balance, it returns γ coins to $\mathcal{C}_i$ and $\perp$ to $\mathcal{A}$). It sends $(tx_i, inpt_{\mathcal{C}_i})$ to $\mathcal{S}_j$ on behalf of $\mathcal{C}_i$.
- (WITHDRAW, $adr_{\mathcal{C}_i}, tx_i$): Ch reads the current time T, the delay parameter Δ, and the time T_1 when $\mathcal{C}_i$ registered transaction tx_i (if tx_i is not registered, it returns $\perp$ to $\mathcal{A}$). It runs $\texttt{Withdraw}(adr_{\mathcal{C}_i}, pp, tx_i, T, T_1)$ that updates a flag $\hat{g}$ associated with tx_i. It sends $\hat{g}$ to $\mathcal{A}$.
- ADVANCE_CLOCK: Ch increments the time counter by 1 and returns $\top$ to $\mathcal{A}$.
- (CORRUPT, adr_P): Ch marks party P (client, server, or auditor) as corrupted and returns its state st_P to $\mathcal{A}$ along with the party's secret parameters sk_P (if already generated).

5.2 Correctness

Broadly speaking, an ICT protocol satisfies correctness if, for any valid inputs provided by the participating parties, the following condition holds. When the servers and clients honestly follow the protocol and provide valid inputs, the protocol ensures that each party correctly receives its designated output. In this scenario, (i) clients always receive the services, they never generate complaints, and they will never receive compensation, and (ii) the servers are always paid for the services they provide.

Overview of Correctness Game $\mathcal{G}^{\Sigma_{\mathrm{ICT}},\mathcal{A}}_{\mathsf{correct}}(1^\lambda, \mathbf{S}, \mathbf{C}, \mathbf{D}, plc_\lambda)$. The game is parameterized by the security parameter λ, sets of servers $\mathbf{S} = \{\mathcal{S}_1, \ldots, \mathcal{S}_k\}$, clients $\mathbf{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$, and auditors $\mathbf{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_m\}$, and a family of policy descriptions $plc = \{plc_\lambda\}_{\lambda \in \mathbb{N}}$. The size of parameters k, n, m, and plc are polynomial in λ. The game comprises four phases: Initialization, Execution, Challenge, and Termination.

- In the **Initialization** phase, the challenger Ch initializes the insurance operator and the smart contract. It provides the adversary $\mathcal{A}$ with: the public parameters and all parties' descriptions and addresses.
- In the **Execution** phase, Ch and $\mathcal{A}$ engage in an interaction where the latter is allowed to send all types of commands described in Section 5.1. Thus, besides adaptively corrupting the parties of its choice, $\mathcal{A}$ orchestrates the execution, including the engagement of the honest parties.
- In the **Challenge** phase, $\mathcal{A}$ provides Ch with the addresses of a client $\tilde{\mathcal{C}}$ and a server $\tilde{\mathcal{S}}$, $\tilde{\mathcal{C}}$'s input $inpt_{\tilde{\mathcal{C}}}$, and an amount $\tilde{\alpha}$.
- In the **Termination** phase, the winning conditions for $\mathcal{A}$ are specified. Informally, $\mathcal{A}$ wins the game if:
  1. $\tilde{\mathcal{C}}$ and $\tilde{\mathcal{S}}$ are honest, their secret and public parameters have been generated, and $\tilde{\mathcal{S}}$ is successfully registered, and

2. Either of the following holds:
   (a) $\tilde{\mathcal{C}}$ does not validate $\tilde{\mathcal{S}}$'s reliability or registration status.
   (b) $\tilde{\mathcal{S}}$ rejects a valid $inpt_{\tilde{c}}$.
   (c) $\tilde{\mathcal{C}}$ makes a late withdrawal request, but the smart contract $\mathcal{SC}$ carries out the request execution.
   (d) $\mathcal{SC}$ does not complete a withdrawal request from $\tilde{\mathcal{C}}$ that was submitted on time.
   (e) $\tilde{\mathcal{S}}$ does not deliver the correct service as requested by $\tilde{\mathcal{C}}$.
   (f) $\tilde{\mathcal{C}}$ does not validate a service honestly executed by $\tilde{\mathcal{S}}$.

Detailed Correctness Game. We proceed to formally present the correctness game $\mathcal{G}_{\mathsf{correct}}^{\Sigma_{\mathrm{ICT}},\mathcal{A}}(1^\lambda, \mathbf{S}, \mathbf{C}, \mathbf{D}, plc_\lambda)$, in Figure 2.

The Correctness game $\mathcal{G}_{\mathsf{correct}}^{\Sigma_{ICT},\mathcal{A}}(1^\lambda, \mathbf{S}, \mathbf{C}, \mathbf{D}, plc_\lambda)$.

- *Initialization*: On behalf of $\mathcal{O}$, Ch runs $\texttt{Operator.Setup}(1^\lambda, plc_\lambda)$, which outputs $(st_{\mathcal{O}}, pp, adr_{\mathcal{SC}})$. It sets a time counter to 0 and generates an address adr_P for every party $P \in \mathbf{S} \cup \mathbf{C} \cup \mathbf{D}$. It provides $\mathcal{A}$ with $pp, adr_{\mathcal{SC}}$, and $\{(P, adr_P)\}_{P \in \mathbf{S} \cup \mathbf{C} \cup \mathbf{D}}$.
- *Execution*: Ch and $\mathcal{A}$ engage in an interaction where Ch plays the role of $\mathcal{O}$, $\mathcal{SC}$, and the honest parties, while $\mathcal{A}$ controls the corrupted parties and can send the commands described in Section 5.1.
- *Challenge*: $\mathcal{A}$ provides Ch with challenge addresses $adr_{\tilde{c}}$ and $adr_{\tilde{s}}$, an input $inpt_{\tilde{c}}$, and an amount $\tilde{\alpha}$.
- *Termination*: The game returns a bit as follows:
  1. If there is no registered auditor, then the game returns 0.
  2. If $adr_{\tilde{c}}$ is not the address of an honest $\tilde{\mathcal{C}}$ or $adr_{\tilde{s}}$ is not the address of an honest $\tilde{\mathcal{S}}$, the game returns 0.
  3. If $\mathcal{A}$ has not sent commands $(\textsc{Params\_Generation}, adr_{\tilde{c}})$, $(\textsc{Params\_Generation}, adr_{\tilde{s}})$, the game returns 0.
  4. If $\mathcal{A}$ has not sent $(\textsc{Registration}, msg_{\tilde{s}})$ such that Ch has returned 1, the game returns 0. If $\mathcal{A}$ has sent such a command (i.e., $\tilde{\mathcal{S}}$ is registered), let $\mathtt{F}_{\tilde{s}}$ be the service function and $\mathbf{V}_{\tilde{s}}$ be the set of valid clients' inputs for the service that $\tilde{\mathcal{S}}$ provides.
  5. Ch runs $\texttt{C.Init}(inpt_{\tilde{c}}, sk_{\tilde{c}}, pk_{\tilde{c}}, adr_{\tilde{s}}, adr_{\mathcal{SC}}, st_{\tilde{c}})$.
  6. If the algorithm C.Init outputs $(\perp, 0, \perp)$, the game returns 1 ($\tilde{\mathcal{C}}$ considered $\tilde{\mathcal{S}}$ non-registered or unreliable).
  7. If C.Init outputs $(inpt^*_{\tilde{c}}, 1, pp_{\tilde{c}})$, Ch runs $\texttt{S.Init}(inpt^*_{\tilde{c}}, adr_{\tilde{c}}, sk_{\tilde{s}}, pk_{\tilde{s}}, st_{\tilde{s}}, adr_{\mathcal{SC}})$.
  8. If S.Init outputs $(\hat{e}, resp_{\tilde{s}})$ such that $\hat{e} = 0$ and $inpt_{\tilde{c}} \notin \mathbf{V}_{\tilde{s}}$, the game returns 0.
  9. If S.Init outputs $(\hat{e}, resp_{\tilde{s}})$ such that $\hat{e} = 0$ and $inpt_{\tilde{c}} \in \mathbf{V}_{\tilde{s}}$, the game returns 1 ($\tilde{\mathcal{S}}$ rejected a valid client's input).
  10. If S.Init outputs $(\hat{e}, resp_{\tilde{s}})$ such that $\hat{e} = 1$, then Ch runs protocol $\texttt{SendTransaction}(\tilde{\mathcal{C}}(id\_inpt_{\tilde{c}}, sk_{\tilde{c}}, inpt_{\tilde{c}}, st_{\tilde{c}}, \tilde{\alpha}, adr_{\tilde{s}}, adr_{\mathcal{SC}}), \mathcal{SC})$. If $\mathsf{Decide}(st_{\tilde{c}}, \tilde{\alpha}, inpt_{\tilde{c}}, resp_{\tilde{s}})$ outputs $\hat{q} = 0$, the game returns 0. If $\mathcal{SC}$ sends $\perp$ to $\tilde{\mathcal{C}}$ (no sufficient balance), the game returns 0. Otherwise, the protocol outputs an amount $\tilde{\gamma}$, a unique transaction identifier $\tilde{tx}$ and a client's request $req_{\tilde{c}}$. Let T_1 be the time that $\tilde{tx}$ was registered.
  11. Ch runs $\texttt{VerRequest}(adr_{\mathcal{SC}}, \tilde{tx}, inpt_{\tilde{c}})$. If VerRequest outputs $\hat{f} = 0$, the game returns 1 ($\tilde{\mathcal{S}}$ rejected a valid client's request).
  12. If the VerRequest outputs $\hat{f} = 1$, Ch provides $\mathcal{A}$ with $(\tilde{tx}, req_{\tilde{c}})$.
  13. $\mathcal{A}$ can send a number of $\textsc{Advance\_Clock}$ commands of its choice. It provides Ch with a bit $\tilde{g}$.
  14. If $\tilde{g} = 0$, then Ch runs $\texttt{Transfer}(1, 1, 0, \tilde{tx}, \tilde{\alpha}, \tilde{\gamma}, \Delta, T, T_1)$, where T is the current time, and goes to Step 20.
  15. If $\tilde{g} = 1$, Ch runs $\texttt{Withdraw}(adr_{\tilde{c}}, pp, \tilde{tx}, T, T_1)$—recall $\Delta \in pp$.
  16. If Withdraw returns 1 and $T - T_1 \leq \Delta$, the game returns 0.
  17. If Withdraw returns 1 and $T - T_1 > \Delta$, the game returns 1 ($\mathcal{SC}$ completed a withdraw request that was submitted too late).
  18. If Withdraw returns 0 and $T - T_1 \leq \Delta$, the game returns 1 ($\mathcal{SC}$ did not perform a withdrawal that was submitted on time).

19. If `Withdraw` returns 0 and $T - T_1 > \Delta$, Ch runs $\texttt{Transfer}(1, 1, 0, \tilde{tx}, \tilde{\alpha}, \tilde{\gamma}, \Delta, T, T_1)$.
20. If `Transfer` outputs $\hat{h} \neq 1$, the game returns 0.
21. If `Transfer` outputs $\hat{h} = 1$, Ch runs $\texttt{Serve}(sk_{\tilde{\mathcal{S}}}, adr_{\mathcal{SC}}, \tilde{tx})$ that outputs $(service_{\tilde{\mathcal{S}}}, \tilde{\pi})$.
22. If $service_{\tilde{\mathcal{S}}} \neq \mathsf{F}_{\tilde{\mathcal{S}}}(inpt^*_{\tilde{\mathcal{C}}}, req_{\tilde{\mathcal{C}}}, pp)$, the game returns 1 ($\tilde{\mathcal{S}}$ did not deliver the correct service as requested by $\tilde{\mathcal{C}}$).
23. Ch runs $\texttt{GenComplaint}(sk_{\tilde{\mathcal{C}}}, service_{\tilde{\mathcal{S}}}, \tilde{tx}, adr_{\mathcal{SC}})$.
24. If `GenComplaint` outputs $\perp$, the game returns 0.; else, it returns 1 ($\tilde{\mathcal{C}}$ did not validate an honestly executed service by $\tilde{\mathcal{S}}$).

Fig. 2: The Correctness game for Σ_{ICT} w.r.t. $plc = \{plc_\lambda\}_{\lambda\in\mathbb{N}}$ between the challenger Ch and the adversary $\mathcal{A}$ with the sets of servers $\mathbf{S} = \{\mathcal{S}_1, \ldots, \mathcal{S}_k\}$, clients $\mathbf{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$, and auditors $\mathbf{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_m\}$.

Definition 1 (Correctness). *Let λ be the security parameter and k, n, m be integers polynomial in λ. Let Σ_{ICT} be a ICT scheme with sets of servers $\mathbf{S} = \{\mathcal{S}_1, \ldots, \mathcal{S}_k\}$, clients $\mathbf{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$, and auditors $\mathbf{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_m\}$. Let $plc = \{plc_\lambda\}_{\lambda\in\mathbb{N}}$ be a family of policy descriptions of size polynomial in λ. Then, Σ_{ICT} satisfies* Correctness w.r.t. *plc, if for every PPT adversary $\mathcal{A}$, it holds that:*

$$\Pr\left[\mathcal{G}^{\Sigma_{ICT},\mathcal{A}}_{\mathsf{correct}}(1^\lambda, \mathbf{S}, \mathbf{C}, \mathbf{D}, plc_\lambda) = 1\right] = \mathsf{negl}(\lambda).$$

5.3 Client-Side Security

Informally, client-side security ensures that an honest client does not accept an invalid service or receives a predefined compensation for an invalid service delivered by a server.

Overview of Client-side *t*-Security Game $\mathcal{G}^{\Sigma_{\text{ICT}},\mathcal{A}}_{t-\mathsf{cln.sec}}(1^\lambda, \mathbf{S}, \mathbf{C}, \mathbf{D}, plc_\lambda)$. The game is parameterized by λ, a real number $t \in [0, 1)$, $\mathbf{S} = \{\mathcal{S}_1, \ldots, \mathcal{S}_k\}$, $\mathbf{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$, $\mathbf{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_m\}$, and $plc = \{plc_\lambda\}_{\lambda\in\mathbb{N}}$. The parameters k, n, m, plc are polynomial in λ. The game comprises four phases, where phases **Initialization**, **Execution**, and **Challenge** are the same as the ones discussed in the Correctness game (Section 5.2). In the **Termination**, the winning conditions for $\mathcal{A}$ are specified. Informally, $\mathcal{A}$ wins the game (i.e., the game outputs 1) if the following conditions are met:

1. Client $\tilde{\mathcal{C}}$ is honest and its secret and public parameters have been generated and the server $\tilde{\mathcal{S}}$ is corrupted.
2. $\mathcal{A}$ has corrupted less than t fraction of all registered auditors.
3. At least one of the following conditions holds: (i) $\tilde{\mathcal{C}}$ validates $\tilde{\mathcal{S}}$'s reliability and registration, but $\tilde{\mathcal{S}}$ is not registered, (ii) $\tilde{\mathcal{C}}$ accepts an invalid service provided by $\tilde{\mathcal{S}}$, or (iii) $\tilde{\mathcal{C}}$ generated a complaint for an invalid service of $\tilde{\mathcal{S}}$, but `Reimburse` concludes that $\tilde{\mathcal{C}}$ should not be reimbursed.

Detailed Description of Client-Side Security Game. Next, we present a detailed description of the game $\mathcal{G}^{\Sigma_{\text{ICT}},\mathcal{A}}_{t-\mathsf{cln.sec}}(1^\lambda, \mathbf{S}, \mathbf{C}, \mathbf{D}, plc_\lambda)$ for client-side security, in Figure 3.

The Client-side t-Security game $\mathcal{G}^{\Sigma_{ICT},\mathcal{A}}_{t-\mathsf{cln.sec}}(1^\lambda, \mathbf{S}, \mathbf{C}, \mathbf{D}, plc_\lambda)$.

- *Initialization*: On behalf of the insurance operator $\mathcal{O}$, Ch runs $\texttt{Operator.Setup}(1^\lambda, plc_\lambda)$ that outputs $(st_{\mathcal{O}}, pp, adr_{\mathcal{SC}})$. Then, it sets a time counter to 0 and generates an address adr_P for every party $P \in \mathbf{S} \cup \mathbf{C} \cup \mathbf{D}$. It provides $\mathcal{A}$ with $pp, adr_{\mathcal{SC}}$ and $\{(P, adr_P)\}_{P\in\mathbf{S}\cup\mathbf{C}\cup\mathbf{D}}$.
- *Execution*: Ch and $\mathcal{A}$ engage in an interaction where Ch plays the role of $\mathcal{O}$, $\mathcal{SC}$ and the honest parties, while $\mathcal{A}$ controls the corrupted parties and can send commands Params_Generation, Registration, Client_Init, Send_Transaction, Withdraw, Advance_Clock, Corrupt, as described in Section 5.1.
- *Challenge*: $\mathcal{A}$ provides Ch with challenge addresses $adr_{\tilde{\mathcal{C}}}$ and $adr_{\tilde{\mathcal{S}}}$, an input $inpt_{\tilde{\mathcal{C}}}$, and an amount $\tilde{\alpha}$.
- *Termination*: The game returns a bit as follows:
  1. If $\mathcal{A}$ has corrupted at least t fraction of all successfully registered auditors or if there is no successfully registered auditor, then the game returns 0.

2. If $adr_{\tilde{\mathcal{C}}}$ is not the address of an honest client $\tilde{\mathcal{C}}$ or $adr_{\tilde{\mathcal{S}}}$ is not the address of a corrupted server $\tilde{\mathcal{S}}$, then the game returns 0.
3. If $\mathcal{A}$ has not sent a command (Params_Generation, $adr_{\tilde{\mathcal{C}}}$), then the game returns 0.
4. Ch runs the algorithm $\mathtt{C.Init}(inpt_{\tilde{\mathcal{C}}}, sk_{\tilde{\mathcal{C}}}, pk_{\tilde{\mathcal{C}}}, adr_{\tilde{\mathcal{S}}}, adr_{\mathcal{SC}}, st_{\tilde{\mathcal{C}}})$.
5. If the algorithm $\mathtt{C.Init}$ outputs $(\perp, 0, \perp)$, the game returns 0.
6. If the algorithm $\mathtt{C.Init}$ outputs $(inpt^*_{\tilde{\mathcal{C}}}, 1, pp_{\tilde{\mathcal{C}}})$, but $\mathcal{A}$ has not sent a command (Registration, $msg_{\tilde{\mathcal{S}}}$) such that Ch has returned 1, then the game returns 1 ($\tilde{\mathcal{C}}$ validated a non-registered server).
7. If the algorithm $\mathtt{C.Init}$ outputs $(inpt^*_{\tilde{\mathcal{C}}}, 1, pp_{\tilde{\mathcal{C}}})$ and $\tilde{\mathcal{S}}$ is successfully registered, let $\mathtt{F}_{\tilde{\mathcal{S}}}$ be the service function and $\mathbf{V}_{\tilde{\mathcal{S}}}$ be the set of valid clients' inputs for the service that $\tilde{\mathcal{S}}$ provides. If $inpt_{\tilde{\mathcal{C}}} \notin \mathbf{V}_{\tilde{\mathcal{S}}}$, then the game returns 1. If $inpt_{\tilde{\mathcal{C}}} \in \mathbf{V}_{\tilde{\mathcal{S}}}$, Ch provides $\mathcal{A}$ with $(inpt^*_{\tilde{\mathcal{C}}}, pp_{\tilde{\mathcal{C}}})$.
8. $\mathcal{A}$ provides Ch with a bit $\tilde{e}$ and a response $resp_{\tilde{\mathcal{S}}}$.
9. If $\tilde{e} = 0$, the game returns 0. If $\tilde{e} = 1$, Ch adds $(id\_inpt_{\tilde{\mathcal{C}}}, \tilde{e}, resp_{\tilde{\mathcal{S}}}, adr_{\tilde{\mathcal{C}}})$ to $\mathcal{SC}$ and runs the protocol $\mathtt{SendTransaction}(\tilde{\mathcal{C}}(id\_inpt_{\tilde{\mathcal{C}}}, sk_{\tilde{\mathcal{C}}}, inpt_{\tilde{\mathcal{C}}}, st_{\tilde{\mathcal{C}}}, \tilde{\alpha}, adr_{\tilde{\mathcal{S}}}, adr_{\mathcal{SC}}), \mathcal{SC})$. If $\mathsf{Decide}(st_{\tilde{\mathcal{C}}}, \tilde{\alpha}, inpt_{\tilde{\mathcal{C}}}, resp_{\tilde{\mathcal{S}}})$ outputs $\hat{q} = 0$, then the game returns 0. If $\mathcal{SC}$ sends $\perp$ to $\tilde{\mathcal{C}}$ (no sufficient balance), then the game returns 0. Otherwise, the protocol outputs an amount $\tilde{\gamma}$, a unique transaction identifier $\tilde{tx}$ and a client's request $req_{\tilde{\mathcal{C}}}$. Let T_1 be the time that $\tilde{tx}$ was registered. Ch provides $\mathcal{A}$ with $(\tilde{tx}, req_{\tilde{\mathcal{C}}})$.
10. $\mathcal{A}$ can send a number of Advance_Clock commands of its choice. Then, it provides Ch with bits $\tilde{f}$ and $\tilde{g}$.
11. If $\tilde{f} = 0$, then the game returns 0.
12. If $\tilde{g} = 0$, then Ch runs the algorithm $\mathtt{Transfer}(1, 1, 0, \tilde{tx}, \tilde{\alpha}, \tilde{\gamma}, \Delta, T, T_1)$, where T is the current time, and goes to Step 16.
13. If $\tilde{g} = 1$, Ch executes the algorithm $\mathtt{Withdraw}(adr_{\tilde{\mathcal{C}}}, pp, \tilde{tx}, T, T_1)$.
14. If the algorithm $\mathtt{Withdraw}$ returns 1 or $T - T_1 \leq \Delta$, then the game returns 0.
15. If the algorithm $\mathtt{Withdraw}$ returns 0 and $T - T_1 > \Delta$, Ch runs $\mathtt{Transfer}(1, 1, 0, \tilde{tx}, \tilde{\alpha}, \tilde{\gamma}, \Delta, T, T_1)$.
16. If the algorithm $\mathtt{Transfer}$ outputs $\hat{h} \neq 1$, the game returns 0.
17. If the algorithm $\mathtt{Transfer}$ outputs $\hat{h} = 1$, then Ch provides $\mathcal{A}$ with $\hat{h}$. Let T_2 be the time that $\mathtt{Transfer}$ algorithm is completed.
18. $\mathcal{A}$ can send a number of Advance_Clock commands of its choice. Upon receiving any such command, Ch also checks if $T - T_2 > \Delta$ (and no service has been received). If so, then it runs the algorithm $\mathtt{GenComplaint}(sk_{\mathcal{C}_i}, \perp, \tilde{tx}, adr_{\mathcal{SC}})$ and goes to Step 22.
19. $\mathcal{A}$ provides Ch with a pair $(service_{\tilde{\mathcal{S}}}, \tilde{\pi})$.
20. If $service_{\tilde{\mathcal{S}}} = \mathtt{F}_{\tilde{\mathcal{S}}}(inpt^*_{\tilde{\mathcal{C}}}, req_{\tilde{\mathcal{C}}}, pp)$, then the game returns 0.
21. If $service_{\tilde{\mathcal{S}}} \neq \mathtt{F}_{\tilde{\mathcal{S}}}(inpt^*_{\tilde{\mathcal{C}}}, req_{\tilde{\mathcal{C}}}, pp)$, Ch runs $\mathtt{GenComplaint}(sk_{\mathcal{C}_i}, service_{\tilde{\mathcal{S}}}, \tilde{tx}, adr_{\mathcal{SC}})$.
22. If the algorithm $\mathtt{GenComplaint}$ outputs $\perp$, then the game returns 1 ($\tilde{\mathcal{C}}$ validated an incorrect service provided by $\tilde{\mathcal{S}}$).
23. If algorithm $\mathtt{GenComplaint}$ outputs $complaint_{\tilde{\mathcal{C}}}$, Ch engages with $\mathcal{A}$ in an execution of the protocol $\mathtt{Reimburse}\Big(\langle \mathcal{D}_1(adr_{\mathcal{SC}}, sk_{\mathcal{D}_1}), \ldots, \mathcal{D}_m(adr_{\mathcal{SC}}, sk_{\mathcal{D}_m}), \mathcal{SC}(plc, \tilde{\alpha}, \tilde{\gamma}, complaint_{\tilde{\mathcal{C}}}, pp)\rangle\Big)$ as follows:
    (a) Let $\mathbf{I}$ be the subset of $[m]$ that specifies the subset of corrupted auditors. Playing the role of every honest auditor $\mathcal{D}_{\ell'}$, Ch decides a bit $\hat{d}_{\ell'}$, where $\ell' \in [m] \setminus \mathbf{I}$.
    (b) It provides $\mathcal{A}$ with all honest decision bits $\{\hat{d}_{\ell'}\}_{\ell' \in [m] \setminus \mathbf{I}}$.
    (c) $\mathcal{A}$ replies with the corrupted decision bits $\{\hat{d}_{\ell}\}_{\ell \in \mathbf{I}}$ of its choice. If the decision bit of some non-registered corrupted auditor is not $\perp$, then the game returns 0.
    (d) Ch runs $\mathsf{ExtractVerdict}(plc, \hat{d}_1, \ldots, \hat{d}_m, adr_{\tilde{\mathcal{C}}})$ that outputs the final verdict bit $\hat{r}$.
    (e) If $\hat{r} = 0$, then the game returns 1 ($\tilde{\mathcal{C}}$ will not be reimbursed although $\tilde{\mathcal{S}}$ did not provide the correct service). Otherwise, the game returns 0.

Fig. 3: The Client-side t-Security game for Σ_{ICT} w.r.t. $plc = \{plc_\lambda\}_{\lambda \in \mathbb{N}}$ between the challenger Ch and the adversary $\mathcal{A}$ with the sets of servers $\mathbf{S} = \{\mathcal{S}_1, \ldots, \mathcal{S}_k\}$, clients $\mathbf{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$, and auditors $\mathbf{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_m\}$.

Definition 2 (Client Security). *Let λ be the security parameter, k, n, m be integers polynomial in λ, and $t \in [0, 1)$. Let Σ_{ICT} be a ICT scheme with sets of servers $\mathbf{S} = \{\mathcal{S}_1, \ldots, \mathcal{S}_k\}$, clients $\mathbf{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$, and auditors $\mathbf{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_m\}$. Let $plc = \{plc_\lambda\}_{\lambda \in \mathbb{N}}$ be a family of policy descriptions of size polynomial in λ. Then, Σ_{ICT}* satisfies Client-side t-Security w.r.t. *plc* and $\mathbf{D}$*, if for every PPT adversary $\mathcal{A}$, it holds that:*

$$\Pr\left[\mathcal{G}^{\Sigma_{ICT},\mathcal{A}}_{t-\mathsf{cln.sec}}(1^\lambda, \mathbf{S}, \mathbf{C}, \mathbf{D}, plc_\lambda) = 1\right] = \mathsf{negl}(\lambda).$$

5.4 Server-Side Security

Server-side security ensures that an honest server will not accept an invalid input or request from a client, and that the client will not receive reimbursement if the server has delivered a valid service.

Overview of Server-side t-Security Game $\mathcal{G}^{\Sigma_{\text{ICT}},\mathcal{A}}_{t-\mathsf{srv.sec}}(1^\lambda, \mathbf{S}, \mathbf{C}, \mathbf{D}, plc_\lambda)$. The game comprises four phases, where **Initialization**, **Execution** are similar to the ones in the Correctness game. In the **Challenge** phase, the adversary provides the challenger with the addresses of a client $\tilde{\mathcal{C}}$ and a server $\tilde{\mathcal{S}}$. In the **Termination**, the winning conditions for $\mathcal{A}$ are specified; $\mathcal{A}$ wins the game, if the following conditions are satisfied:

1. The server $\tilde{\mathcal{S}}$ is honest, its secret and public parameters have been generated and it is registered, and $\tilde{\mathcal{C}}$ is corrupted.
2. $\mathcal{A}$ corrupted less than t fraction of registered auditors.
3. At least one of the following conditions holds: (i) $\tilde{\mathcal{S}}$ accepts an invalid (encoded) input from $\tilde{\mathcal{C}}$, (ii) $\tilde{\mathcal{S}}$ accepts an invalid request from $\tilde{\mathcal{C}}$, or (iii) $\tilde{\mathcal{C}}$ received reimbursement although $\tilde{\mathcal{S}}$ provided a valid service.

Detailed Description of Server-Side Security Game. Now we provide a detailed description of the game $\mathcal{G}^{\Sigma_{\text{ICT}},\mathcal{A}}_{t-\mathsf{srv.sec}}(1^\lambda, \mathbf{S}, \mathbf{C}, \mathbf{D}, plc_\lambda)$, which models server-side security, in Figure 4.

The Server-side t-Security game $\mathcal{G}^{\Sigma_{ICT},\mathcal{A}}_{t-\mathsf{srv.sec}}(1^\lambda, \mathbf{S}, \mathbf{C}, \mathbf{D}, plc_\lambda)$.

- *Initialization*: On behalf of the insurance operator $\mathcal{O}$, Ch runs $\texttt{Operator.Setup}(1^\lambda, plc_\lambda)$ that outputs $(st_{\mathcal{O}}, pp, adr_{\mathcal{SC}})$. Then, it sets a time counter to 0 and generates an address adr_P for every party $P \in \mathbf{S} \cup \mathbf{C} \cup \mathbf{D}$. It provides $\mathcal{A}$ with $pp, adr_{\mathcal{SC}}$ and $\{(P, adr_P)\}_{P \in \mathbf{S} \cup \mathbf{C} \cup \mathbf{D}}$.
- *Execution*: Ch and $\mathcal{A}$ engage in an interaction where Ch plays the role of $\mathcal{O}$, $\mathcal{SC}$ and the honest parties, while $\mathcal{A}$ controls the corrupted parties and can send commands of type Params_Generation, Registration, Client_Init, Send_Transaction, Withdraw, Advance_Clock, Corrupt, as described in Section 5.1.
- *Challenge*: $\mathcal{A}$ provides Ch with two challenge addresses $adr_{\tilde{\mathcal{C}}}$ and $adr_{\tilde{\mathcal{S}}}$.
- *Termination*: The game returns a bit according to the following steps:
  1. If $\mathcal{A}$ has corrupted at least t fraction of all successfully registered auditors or if there is no successfully registered auditor, then the game returns 0.
  2. If $adr_{\tilde{\mathcal{C}}}$ is not the address of a corrupted client $\tilde{\mathcal{C}}$ or $adr_{\tilde{\mathcal{S}}}$ is not the address of an honest server $\tilde{\mathcal{S}}$, then the game returns 0.
  3. If $\mathcal{A}$ has not sent a command $(\text{Params\_Generation}, adr_{\tilde{\mathcal{S}}})$, then the game returns 0.
  4. If $\mathcal{A}$ has not sent a command $(\text{Registration}, msg_{\tilde{\mathcal{S}}})$ such that Ch has returned 1, then the game returns 0. If $\mathcal{A}$ has sent such a command (i.e., $\tilde{\mathcal{S}}$ is successfully registered), let $\mathtt{F}_{\tilde{\mathcal{S}}}$ be the service function and $\mathbf{V}_{\tilde{\mathcal{S}}}$ be the set of valid clients' inputs for the service that $\tilde{\mathcal{S}}$ provides.
  5. $\mathcal{A}$ provides Ch with a triple $(inpt^*_{\tilde{\mathcal{C}}}, \hat{a}, pp_{\tilde{\mathcal{C}}})$.
  6. If $\hat{a} \neq 1$, the game outputs 0. Otherwise, Ch runs $\texttt{S.Init}(inpt^*_{\tilde{\mathcal{C}}}, adr_{\tilde{\mathcal{C}}}, sk_{\tilde{\mathcal{S}}}, pk_{\tilde{\mathcal{S}}}, st_{\tilde{\mathcal{S}}}, adr_{\mathcal{SC}})$.
  7. If the algorithm $\texttt{S.Init}$ outputs $(\hat{e}, resp_{\tilde{\mathcal{S}}})$ such that $\hat{e} = 0$, then the game returns 0.
  8. If the algorithm $\texttt{S.Init}$ outputs $(\hat{e}, resp_{\tilde{\mathcal{S}}})$ such that $\hat{e} = 1$, then if $inpt_{\tilde{\mathcal{C}}} \notin \mathbf{V}_{\tilde{\mathcal{S}}}$, then the game returns 1 ($\tilde{\mathcal{S}}$ accepted an invalid client's input). Otherwise, Ch sends $(id\_inpt_{\tilde{\mathcal{C}}}, \hat{e}, resp_{\tilde{\mathcal{S}}}, adr_{\tilde{\mathcal{C}}})$ to $\mathcal{SC}$ and provides $\mathcal{A}$ with $(id\_inpt_{\tilde{\mathcal{C}}}, 1)$.

9. $\mathcal{A}$ provides Ch with a request $req_{\tilde{c}}$ that specifies an amount $\tilde{\alpha}$ and an amount of $\tilde{\gamma}$ coins.
10. Ch (playing the role of $\mathcal{SC}$) runs $\mathsf{CheckBudget}(req_{\tilde{c}}, adr_{\tilde{c}}, adr_{\mathcal{SC}}, \mathsf{balance}, data_{\mathcal{SC}}, plc_\lambda, \tilde{\gamma}, pp, L_{\mathsf{pnd}}) \rightarrow \hat{w}$. If $\hat{w} = 0$, then the game returns 0. Otherwise, it generates a unique transaction identifier $\tilde{tx}$ and provides $\mathcal{A}$ with $\tilde{tx}$. Let T_1 be the time that $\tilde{tx}$ was registered.
11. $\mathcal{A}$ provides Ch with an input $inpt_{\tilde{c}}$.
12. Ch runs the algorithm $\mathtt{VerRequest}(adr_{\mathcal{SC}}, \tilde{tx}, inpt_{\tilde{c}})$.
13. If the algorithm $\mathtt{VerRequest}$ outputs $\hat{f} = 0$, then the game returns 0.
14. If $\mathtt{VerRequest}$ outputs $\hat{f} = 1$, if $(inpt_{\tilde{c}}, req_{\tilde{c}}, pp) \notin Domain(\mathtt{F}_{\tilde{s}})$, the game returns 1 ($\tilde{\mathcal{S}}$ accepted an invalid input or an invalid request from $\tilde{\mathcal{C}}$). Otherwise, Ch sends $\hat{f}$ to $\mathcal{SC}$.
15. $\mathcal{A}$ can send a number of ADVANCE_CLOCK commands of its choice. It provides Ch with a bit $\tilde{g}$.
16. If $\tilde{g} = 0$, then Ch runs $\mathtt{Transfer}(1, 1, 0, \tilde{tx}, \tilde{\alpha}, \tilde{\gamma}, \Delta, T, T_1)$, where T is the current time, and goes to Step 20.
17. If $\tilde{g} = 1$, Ch executes the algorithm $\mathtt{Withdraw}(adr_{\tilde{c}}, pp, \tilde{tx}, T, T_1)$.
18. If the algorithm $\mathtt{Withdraw}$ returns 1, then the game returns 0.
19. If $\mathtt{Withdraw}$ returns 0, Ch runs $\mathtt{Transfer}(1, 1, 0, \tilde{tx}, \tilde{\alpha}, \tilde{\gamma}, \Delta, T, T_1)$.
20. If the algorithm $\mathtt{Transfer}$ outputs $\hat{h} \neq 1$, the game returns 0.
21. If the algorithm $\mathtt{Transfer}$ outputs $\hat{h} = 1$, then Ch runs the algorithm $\mathtt{Serve}(sk_{\tilde{s}}, adr_{\mathcal{SC}}, \tilde{tx})$ that outputs $(service_{\tilde{s}}, \tilde{\pi})$. Then, it provides $\mathcal{A}$ with $service_{\tilde{s}}$.
22. $\mathcal{A}$ can send a number of ADVANCE_CLOCK commands of its choice. Then, it provides Ch with a complaint $complaint_{\tilde{c}}$.
23. If $complaint_{\tilde{c}} = \perp$, the game returns 0. Otherwise, Ch engages with $\mathcal{A}$ in the execution of protocol $\mathtt{Reimburse}\Big(\langle \mathcal{D}_1(adr_{\mathcal{SC}}, sk_{\mathcal{D}_1}), \ldots, \mathcal{D}_m(adr_{\mathcal{SC}}, sk_{\mathcal{D}_m}), \mathcal{SC}(plc, \tilde{\alpha}, \tilde{\gamma}, complaint_{\tilde{c}}, pp)\rangle\Big)$ as follows:
    (a) Let $\mathbf{I}$ be the subset of $[m]$ that specifies the subset of corrupted auditors. Playing the role of every honest auditor $\mathcal{D}_{\ell'}$, Ch decides a bit $\hat{d}_{\ell'}$, where $\ell' \in [m] \setminus \mathbf{I}$.
    (b) It provides $\mathcal{A}$ with all honest decision bits $\{\hat{d}_{\ell'}\}_{\ell' \in [m] \setminus \mathbf{I}}$.
    (c) $\mathcal{A}$ replies with the corrupted decision bits $\{\hat{d}_\ell\}_{\ell \in \mathbf{I}}$ of its choice. If the decision bit of some non-registered corrupted auditor is not $\perp$, then the game returns 0.
    (d) Ch runs $\mathsf{ExtractVerdict}(plc, \hat{d}_1, \ldots, \hat{d}_m, adr_{\tilde{c}})$ that outputs the final verdict bit $\hat{r}$.
    (e) If $\hat{r} = 1$, then the game returns 1 ($\tilde{\mathcal{C}}$ will be reimbursed although $\tilde{\mathcal{S}}$ provided the correct service). Otherwise, the game returns 0.

Fig. 4: The Server-side t-Security game for Σ_{ICT} w.r.t. $plc = \{plc_\lambda\}_{\lambda \in \mathbb{N}}$ between the challenger Ch and the adversary $\mathcal{A}$ with the sets of servers $\mathbf{S} = \{\mathcal{S}_1, \ldots, \mathcal{S}_k\}$, clients $\mathbf{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$, and auditors $\mathbf{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_m\}$.

Definition 3 (Server Security). *Let λ be the security parameter, k, n, m be integers polynomial in λ, and $t \in [0, 1)$. Let Σ_{ICT} be a ICT scheme with sets of servers $\mathbf{S} = \{\mathcal{S}_1, \ldots, \mathcal{S}_k\}$, clients $\mathbf{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$, and auditors $\mathbf{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_m\}$. Let $plc = \{plc_\lambda\}_{\lambda \in \mathbb{N}}$ be a family of policy descriptions of size polynomial in λ. Then, Σ_{ICT} meets Server-side t-Security w.r.t. plc and $\mathbf{D}$, if for every PPT adversary $\mathcal{A}$, it holds that:*

$$\Pr\left[\mathcal{G}^{\Sigma_{ICT}, \mathcal{A}}_{t-\mathsf{srv.sec}}(1^\lambda, \mathbf{S}, \mathbf{C}, \mathbf{D}, plc_\lambda) = 1\right] = \mathsf{negl}(\lambda).$$

5.5 ICT Security

An ICT scheme is secure, if it satisfies client-side and server-side security, regarding Definitions 2 and 3, respectively. Formally,

Definition 4. *Let λ be the security parameter, k, n, m be integers polynomial in λ, and $t \in [0, 1)$. Let Σ_{ICT} be a ICT scheme with sets of servers $\mathbf{S} = \{\mathcal{S}_1, \ldots, \mathcal{S}_k\}$, clients $\mathbf{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$, and auditors $\mathbf{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_m\}$. Let $plc = \{plc_\lambda\}_{\lambda \in \mathbb{N}}$ be a family of policy descriptions of size polynomial in λ. We say that Σ_{ICT} is t-secure w.r.t. plc and $\mathbf{D}$ if it satisfies (i) Client-side t-Security and (ii) Server-side t-Security according to Definitions 2 and 3, respectively.*

6 Insured Cryptocurrency Exchange

This section presents a concrete instantiation of ICT for centralized cryptocurrency exchange scenarios, using a modular approach. We first define the syntax of the Centralized Cryptocurrency Exchange (CCE) in Section 6.1, followed by key subroutines in Section 6.2. We present an abstract overview of the ICE protocol in Sections 6.3, followed by a high level description of its various phases in Section 6.4. Subsequently, we construct the Insured Cryptocurrency Exchange (ICE) protocol in Section 6.5, building on CCE and the subroutines.

6.1 Centralized Cryptocurrency Exchange

At an abstract level, a CCE scheme consists of (i) a client $\mathcal{C}_i$, who seeks to exchange a specified amount of one cryptocurrency for another, and (ii) a server $\mathcal{S}$, which facilitates the exchange and is commonly known as the exchange. Typically, $\mathcal{S}$ informs the public about the currency types it can accommodate. It publishes a list $pp_{\mathcal{S}}$ containing (i) the identifiers of currencies $[C_1, \ldots, C_z]$ valid for exchange and (ii) the corresponding maximum allowed amount per transaction $[M_1, \ldots, M_z]$; specifically, no more than an amount M_i in currency C_i can be exchanged by the server per transaction. The service function $\mathsf{F}_{\mathcal{S}}$ that $\mathcal{S}$ supports is defined as: $\mathsf{F}_{\mathcal{S}}(C_\ell, C_{\ell'}, \alpha, \rho) \rightarrow (\alpha \cdot \rho, C_{\ell'})$ which expresses the transaction where α amount of coins in C_ℓ are exchanged to $\alpha \cdot \rho$ amount of coins $C_{\ell'}$ with rate ρ.
The server's state $st_{\mathcal{S}}$ is initialized as (i) a vector $\mathbf{AF} = \langle (\beta_1, C_1), \ldots, (\beta_z, C_z) \rangle$, where β_ℓ denotes the initial available funds in C_ℓ coins, (ii) a $z \times z$ matrix $\mathbf{RT} = [\rho_{\ell,\ell'}]$, where $\rho_{\ell,\ell'}$ is the rate for exchanging C_ℓ to $C_{\ell'}$, and (iii) an *adjustment factor* η, a parameter that adjusts the exchange rate based on changes in available funds. A client $\mathcal{C}_i$'s valid input is defined as a tuple $inpt_{\mathcal{C}_i} = (\alpha, X, Y)$, that has identifier id and encodes the statement "I want to exchange an amount of α in currency X to currency Y".

Underlying Functions. A typical CCE scheme may make use of the following two procedures:

- $\mathsf{Decide}(st_{\mathcal{C}}, \alpha, inpt_{\mathcal{C}}, inpt_{\mathcal{S}}) \rightarrow \hat{q}$: a predicate run by a client $\mathcal{C}$. It parses $inpt_{\mathcal{C}}$ as (α^*, X, Y) and $inpt_{\mathcal{S}}$ as (ρ, μ), where ρ is the exchange's proposed rate and μ is the amount for exchange in currency Y for this transaction. It determines if $\mathcal{C}$ wants to transact with $\mathcal{S}$ based on $\mathcal{C}$'s state $st_{\mathcal{C}}$, $\mathcal{C}$'s input $inpt_{\mathcal{C}}$, and $\mathcal{S}$'s input $inpt_{\mathcal{S}}$. It returns 1 if it concludes that $\mathcal{C}$ will transact with $\mathcal{S}$ and returns 0 otherwise. In Section 6.5, we will use this predicate to help a client decide whether to interact with the exchange. The implementation is straightforward, with the predicate containing an internal constant value $diff$. It verifies the following conditions: (i) $\alpha = \alpha^*$, (ii) ρ is at least the difference between $\mathcal{C}$'s ideal rate (specified in $st_{\mathcal{C}}$) and $diff$, and (iii) $\mu \geq \alpha \cdot \rho$. If the conditions are met, it returns 1. Else, it returns 0.
- $\mathsf{UpdateRate}(\mathbf{RT}, \mathbf{AF}, req, \eta) \rightarrow \mathbf{RT}$: an algorithm used by the exchange. It takes as input (1) a $z \times z$ matrix $\mathbf{RT} = [\rho_{\ell,\ell'}]$, where $\rho_{\ell,\ell'}$ is the rate for exchanging C_ℓ to $C_{\ell'}$, (2) a vector $\mathbf{AF} = \langle (\beta_1, C_1), \ldots, (\beta_z, C_z) \rangle$, where β_ℓ denotes the available funds in C_ℓ coins, (3) a current customer's request $req = (\alpha, C_x, C_y, \rho_{x,y})$, where α is the amount of money/coins in the source currency C_x, C_y is the destination currency, and $\rho_{x,y}$ is the current exchange rate, and (4) an *adjustment factor* η, a parameter that adjusts the exchange rate according to changes in available funds. The algorithm updates its input matrix and returns an updated matrix $\mathbf{RT}$. There are various methods available for implementing this algorithm. In our case, we can use a relatively straightforward approach to implement it. It initially computes a new rate $\rho'_{x,y}$ for exchanging C_x to C_y as:

$$\rho'_{x,y} = \rho_{x,y} \cdot \left(1 + \eta \cdot \left(\frac{\beta_y + \alpha \cdot \rho_{x,y}}{\beta_x - \alpha} - \frac{\beta_y}{\beta_x}\right)\right)$$

  It updates its current related rates as: $\rho_{x,y} \leftarrow \rho'_{x,y}$ and $\rho_{y,x} \leftarrow \frac{1}{\rho'_{x,y}}$. Note that η is a theoretical parameter used to model how an exchange rate might dynamically adjust in response to changes in supply and demand. In real-world financial markets and exchanges, several parameters influence exchange rates, including liquidity and all current (buy and sell) demands.

Description. We define CCE via the following six algorithms.

- $\underline{\texttt{CCE.C.Init}(inpt_{c_i}, ID_{\mathcal{S}}, pp_{\mathcal{S}}) \rightarrow inpt^*_{c_i}}$: it is run by a client $\mathcal{C}_i$. It parses $inpt_{c_i}$ as (α, X, Y). It returns $inpt^*_{c_i} = (X, Y)$. $\mathcal{C}_i$ sends $inpt^*_{c_i}$ to $\mathcal{S}$.
- $\underline{\texttt{CCE.S.Init}(inpt^*_{c_i}, ID_{c_i}, st_{\mathcal{S}}) \rightarrow (\hat{e}, \rho)}$: it is executed by a server $\mathcal{S}$ upon receiving $inpt^*_{c_i}$ from $\mathcal{C}_i$. It reads $inpt^*_{c_i}$ as (X_{c_i}, Y_{c_i}). If $X_{c_i}, Y_{c_i} \in pp_{\mathcal{S}}$, then it sets a bit $\hat{e}$ to 1, reads the matrix $\mathbf{RT} \in st_{\mathcal{S}}$ and fetches (i) the rate ρ^* in $\mathbf{RT}$ that corresponds to the pair (X_{c_i}, Y_{c_i}); (ii) the available funds β_{c_i} in $\mathbf{AF}$ for Y_{c_i}; and (iii) the maximum amount per transaction $M_{c_i} \in pp_{\mathcal{S}}$ for Y_{c_i}. It sets $\rho \leftarrow \rho^*$ and $\mu \leftarrow \min\{\beta_{c_i}, M_{c_i}\}$. Otherwise, it sets $\hat{e}$ to 0 and ρ, μ to $\perp$. $\mathcal{S}$ sends $(\hat{e}, (\rho, \mu))$ to $\mathcal{C}_i$.
- $\underline{\texttt{CCE.C.Request}(inpt_{c_i}, \alpha, ID_{\mathcal{S}}, (\rho, \mu), st_{c_i}) \rightarrow req_i}$: it is run by $\mathcal{C}_i$ upon receiving $(\hat{e}, (\rho, \mu))$ from $\mathcal{S}$. It runs $\mathsf{Decide}(st_{c_i}, \alpha, inpt_{c_i}, (\rho, \mu))$ that outputs $\hat{q}$. If $\hat{q} = 1$, it generates a request $req_{c_i} = (\alpha, X_{c_i}, Y_{c_i}, \rho)$; $\mathcal{C}_i$ sends req_{c_i} to $\mathcal{S}$. Else, it returns $\perp$.
- $\underline{\texttt{CCE.S.VerRequest}(req_{c_i}, ID_{c_i}, st_{\mathcal{S}}) \rightarrow \hat{f}}$: it is run by $\mathcal{S}$ upon receiving req_{c_i} from $\mathcal{C}_i$. It reads α, X_{c_i}, Y_{c_i}, and ρ from req_i and fetches ρ^*, β_{c_i}, and M_{c_i} as in $\texttt{CCE.S.Init}$. If $\rho^* = \rho$ and $\min\{\beta_{c_i}, M_{c_i}\} \geq \alpha \cdot \rho$, it sets a bit $\hat{f}$ to 1; otherwise, it sets $\hat{f}$ to 0. It adds $(\hat{f}, \alpha, X_{c_i}, Y_{c_i}, \rho)$ to $st_{\mathcal{S}}$. Server $\mathcal{S}$ returns $\hat{f}$ to $\mathcal{C}_i$.
- $\underline{\texttt{CCE.C.Transfer}(req_i, \hat{f}, ID_{\mathcal{S}}) \rightarrow (\alpha, X_{c_i})}$: it is run by $\mathcal{C}_i$. It reads α, X_{c_i} from req_i. If $\hat{f} = 1$, client $\mathcal{C}_i$ sends α amount of X_{c_i} to $\mathcal{S}$.
- $\underline{\texttt{CCE.S.Serve}(req_{c_i}, \alpha^*, X^*, ID_{c_i}, st_{\mathcal{S}}) \rightarrow (\phi, Y_{c_i})}$: it is run by $\mathcal{S}$ upon receiving α^* amount of X^* from $\mathcal{C}_i$. It parses req_{c_i} as $(\alpha, X_{c_i}, Y_{c_i}, \rho)$. If there is a $(1, \alpha, X_{c_i}, Y_{c_i}, \rho)$ in $st_{\mathcal{S}}$, $\alpha = \alpha^*$, and $X_{c_i} = X^*$; and if there is at least $\alpha \cdot \rho$ of Y_{c_i} then it computes $\mathsf{F}_{\mathcal{S}}(X_{c_i}, Y_{c_i}, \alpha, \rho) \rightarrow (\phi, Y_{c_i})$, where $\phi = \alpha \cdot \rho$. It updates $st_{\mathcal{S}}$ as follows: (i) in $\mathbf{AF}$, it increases the available funds in X_{c_i} by α and reduces the funds in Y_{c_i} by ϕ, (ii) in $\mathbf{RT}$, it updates the rates by calling $\mathsf{UpdateRate}(\mathbf{RT}, \mathbf{AF}, req_i, \eta) \rightarrow \mathbf{RT}$, and (iii) it deletes $(1, \alpha, X_{c_i}, Y_{c_i}, \rho)$ from $st_{\mathcal{S}}$. Server $\mathcal{S}$ sends ϕ amount of Y_{c_i} to $\mathcal{C}_i$.

6.2 Subroutines

Subroutine for Updating State. During the execution of certain algorithms, such as `Register` and `C.Init`, it is crucial for each party to maintain an updated state. An up-to-date state enables a party to decide whether to engage with its counterpart. To facilitate state updates, we introduce the algorithm $\mathsf{Update.State}$, which incorporates records of disqualified service providers and qualified auditors into the party's state. Figure 5 provides a detailed explanation of $\mathsf{Update.State}$.
The algorithm makes a black-box call to another algorithm, $\mathsf{Find.SusAccount}(data_{ledger}, param) \rightarrow \boldsymbol{v}$, which can be executed by a party P to identify suspicious blockchain accounts. It takes as input the latest blockchain state $data_{ledger}$ and a set of parameters $param$, returning suspicious account addresses. Various machine-learning-based approaches [3,52,93] can implement $\mathsf{Find.SusAccount}$.[7]

Subroutines to Check Parties' Status. To enable both the operator $\mathcal{O}$ and the client $\mathcal{C}$ to verify the reliability of their counterparts, we introduce two algorithms $\mathsf{Validate}^{(\mathcal{O})}$ and $\mathsf{Validate}^{(\mathcal{C})}$. The algorithm $\mathsf{Validate}^{(\mathcal{O})}$ enables $\mathcal{O}$ to ensure the integrity of a party P by checking if it has a clean record and is qualified as an auditor. This step is important for maintaining the trustworthiness of the system, as it safeguards against potential malicious actors or unqualified entities. The algorithm $\mathsf{Validate}^{(\mathcal{C})}$ allows $\mathcal{C}$ to verify the legitimacy of a server. By employing this algorithm, $\mathcal{C}$ can confirm that the server is not flagged as a known bad actor and has previously undergone the necessary registration process facilitated by $\mathcal{O}$. This verification mechanism ensures that clients interact only with trustworthy servers that have been vetted by $\mathcal{O}$. Figures 6 and 7 present detailed descriptions of $\mathsf{Validate}^{(\mathcal{O})}$ and $\mathsf{Validate}^{(\mathcal{C})}$, respectively.

[7] Throughout this paper, we assume that $\mathsf{Find.SusAccount}$ produces no false positives, i.e., it never flags honest parties, as correctness would otherwise be compromised.

$\mathsf{Update.State}(st_P, pp, sk_P, data_{ledger}) \rightarrow st_P$

- *Input.* $st_P = (st_P^{\mathcal{S}}, st_P^{\mathcal{D}})$: current state; pp: public parameters including pointers to two databases, (i) $db_{\mathcal{S}}$: disqualified service providers' database and (ii) $db_{\mathcal{D}}$: approved auditors' database; $data_{ledger}$: the blockchain's state.
- *Output.* st_P: updated state.

1. Call $\mathsf{Find.SusAccount}(data_{ledger}, param) \rightarrow \boldsymbol{v}$.
2. For every element e of $db_{\mathcal{S}}$ that is not in $st_P^{\mathcal{S}}$, insert e to $st_P^{\mathcal{S}}$. Similarly, for every e' of $db_{\mathcal{D}}$ not in $st_P^{\mathcal{D}}$, insert e' to $st_P^{\mathcal{D}}$.
3. For every $e \in \boldsymbol{v}$ if $e \notin st_P^{\mathcal{S}}$, then insert e to $st_P^{\mathcal{S}}$.
4. Return $st_P := (st_P^{\mathcal{S}}, st_P^{\mathcal{D}})$.

Fig. 5: Algorithm to update party P's state.

$\mathsf{Validate}^{(\mathcal{O})}(plc_\lambda, st_{\mathcal{O}}, msg_P) \rightarrow \hat{b} \in \{0, 1\}$

- *Input.* plc_λ: $\mathcal{O}$'s policy stating P must not be a bad actor server or unqualified auditor, $st_{\mathcal{O}}$: the state of $\mathcal{O}$, and msg_P: party P's message that contains P's address adr_P.
- *Output.* $\hat{b} = 1$: if P is considered valid; $a = 0$: otherwise.

1. Set $\hat{b}$ as follows.
   - if P is a server (i.e., $server \in msg_P$), check if P is not a known bad actor, i.e., $adr_P \notin st_{\mathcal{O}}^{\mathcal{S}}$. If the check passes, set $\hat{b} = 1$. Otherwise, set $\hat{b} = 0$.
   - if P is an auditor (i.e., $auditor \in msg_P$), check if P is a qualified auditor , i.e., $adr_P \in st_{\mathcal{O}}^{\mathcal{D}}$. If the check passes, set $\hat{b} = 1$. Otherwise, set $\hat{b} = 0$.
2. Return $\hat{b}$.

Fig. 6: Operator's algorithm to check the status of a server or auditor.

$\mathsf{Validate}^{(\mathcal{C})}(st_{\mathcal{C}_i}, adr_P, adr_{\mathcal{SC}}) \rightarrow \hat{a} \in \{0, 1\}$

- *Input.* $st_{\mathcal{C}_i}$: state of the client and adr_P: P's address.
- *Output.* $\hat{a} = 1$: if P is considered valid; $\hat{a} = 0$: otherwise.

1. Set $\hat{a}$ as follows.
   - check if P is not a known bad actor, i.e., $adr_P \notin st_{\mathcal{C}_i}^{\mathcal{S}}$, where $st_{\mathcal{C}_i}^{\mathcal{S}} \in st_{\mathcal{C}_i}$. If the check passes, set $\hat{a} = 1$. Else, set $\hat{a} = 0$.
   - check if adr_P has been registered in $\mathcal{SC}$.
   - if the two checks pass, set $\hat{a} = 1$. Otherwise, set $\hat{a} = 0$.
2. Return $\hat{a}$.

Fig. 7: Client's algorithm to check the status of a server.

Subroutine to Check Budget. Ensuring that a smart contract $\mathcal{SC}$ can evaluate its financial capacity to accommodate new clients is important for maintaining its functionality, particularly when managing client premiums. To facilitate this capability, we define the CheckBudget algorithm. It serves as a vital mechanism

for $\mathcal{SC}$ to evaluate its available budget against the coverage requested by new clients and its commitment to its existing clients. A detailed description of this algorithm is provided in Figure 8.

CheckBudget$(req_{\mathcal{C}}, adr_{\mathcal{S}}, adr_{\mathcal{SC}}, \mathsf{balance}, data_{\mathcal{SC}}, plc_{\lambda}, \gamma, pp, L_{\mathsf{pnd}}) \rightarrow \hat{w}$

- *Input.* (1) $req_{\mathcal{C}}$: $\mathcal{C}$'s request, of the form $(id\_inpt_{\mathcal{C}}, \alpha, C_{\ell}, C_{\ell'}, \rho)$, where α amount of coins in C_{ℓ} are exchanged to $\alpha \cdot \rho$ amount of coins $C_{\ell'}$ with rate ρ, (2) $adr_{\mathcal{S}}$: address of $\mathcal{S}$, (3) balance: the amount of usable funds held by $\mathcal{SC}$, (4) plc_{λ}: policy of $\mathcal{S}$, (5) γ: the total amount (that includes service payment plus premium) that $\mathcal{C}$ wants to pay, (6) pp: public parameter, and (7) L_{pnd}: a list of pending transactions. This list consists of triples of the form $(\tilde{tx}, \tilde{v}, \tilde{T})$, where $\tilde{tx}$ is a transaction identifier, $\tilde{v}$ is a compensation amount associated with $\tilde{tx}$, and $\tilde{T}$ is the time when $\tilde{tx}$ was registered.
- *Output.* $\hat{w} = 1$: if $\mathcal{SC}$ has enough budget to serve $\mathcal{C}$; $\hat{w} = 0$: otherwise.

1. Calculate the amount of premium that $\mathcal{C}$ must pay, by calling: $a \leftarrow \mathsf{CalPremium}(data_{\mathcal{SC}}, adr_{\mathcal{S}}, req_{\mathcal{C}})$.
2. Determine the amount of compensation $\mathcal{C}$ will receive, by calling: $b \leftarrow \mathsf{CompRemAmount}(plc_{\lambda}, \gamma, req_{\mathcal{C}}, pp)$.
3. Set $\hat{w}$ as follows. If $\mathsf{balance} + a < b + \sum_{(\tilde{tx}, \tilde{v}, \tilde{T}) \in L_{\mathsf{pnd}}} \tilde{v}$, then set $\hat{w} = 0$; otherwise, set $\hat{w} = 1$.
4. Return $\hat{w}$.

Fig. 8: $\mathcal{SC}$'s algorithm to check its budget.

Subroutine to Check Exchange. To enable a client to check if it has received the exchanged coins from the exchange $\mathcal{S}_j$, we define an algorithm VerifyExchage, presented in Figure 9. It allows the client to access its account and check if it has the amount that $\mathcal{S}_j$ was supposed to transfer as a result of a coin exchange.

Subroutine for Final Verdict Extraction. To let $\mathcal{SC}$ determine the final verdict based on the individual decisions of all auditors regarding whether a client should be reimbursed, we define a specific algorithm $\mathsf{ExtractVerdict}(plc_{\lambda}, e_1, \ldots, e_m, adr_P) \rightarrow \hat{r}$. This algorithm, presented in Figure 10, consolidates the auditors' verdicts into a final decision. The algorithm returns 1, in favor of the client, if at least a predefined number of honest auditors have decided so.

6.3 An Overview of ICE

Initially, the insurance operator $\mathcal{O}$ deploys a smart contract $\mathcal{SC}$ and publishes its details. $\mathcal{O}$ deposits a certain amount of coins into $\mathcal{SC}$ to cover future legitimate claims. $\mathcal{O}$ registers the details of a set of servers (i.e., cryptocurrency exchanges) and auditors in $\mathcal{SC}$. When a client $\mathcal{C}$ wishes to exchange a certain amount of coins, they send a message to $\mathcal{SC}$ and one of the registered servers, $\mathcal{S}$, who processes $\mathcal{C}$'s request and provides the exchange rate to $\mathcal{SC}$. If $\mathcal{C}$ decides to proceed, they transfer the desired amount of coins to $\mathcal{SC}$ for the server to exchange. Additionally, $\mathcal{C}$ transfers a specified premium to $\mathcal{SC}$ to receive coverage. Given this amount, $\mathcal{SC}$ determines whether it can serve $\mathcal{C}$. $\mathcal{SC}$ refunds $\mathcal{C}$ when it concludes that it cannot serve $\mathcal{C}$. Otherwise, it withholds $\mathcal{C}$'s coins for a certain period. In this case, $\mathcal{C}$ notifies $\mathcal{S}$ by sending a request detailing the amount of exchange. $\mathcal{S}$ verifies $\mathcal{C}$'s request.
During the predefined period within which $\mathcal{SC}$ holds $\mathcal{C}$'s coins, $\mathcal{C}$ can request a withdrawal. This feature helps prevent fraud if $\mathcal{C}$ quickly concludes that $\mathcal{S}$ may not be trustworthy. If the coins have not been withdrawn,

VerifyExchage($sk_{\mathcal{C}}, req_{\mathcal{C}}, \pi_j$) $\rightarrow (\hat{z}, \zeta_{\mathcal{C}})$

- *Input.* $sk_{\mathcal{C}}$: secret key $sk_{\mathcal{C}}$ of a client $\mathcal{C}$, $req_{\mathcal{C}}$: the client's request which includes the amount of transfer and the currency to which the amount is transferred to, and π_j: a proof/statement provided by $\mathcal{S}_j$ and asserts that the amount has been transferred.
- *Output.* $\hat{z} = 1$: if the amount specified in $req_{\mathcal{C}}$ has been transferred; in this case $\zeta_{\mathcal{C}} = \perp$. $\hat{z} = 0$: if the specified amount has not been transferred; in this case, $\zeta_{\mathcal{C}}$ is set to $\mathcal{C}$'s account statement, summarizing the financial transactions in the account over a certain period of time.

1. Access the account to which the exchanged coins are transferred (the account type is specified in $req_{\mathcal{C}}$), as follows.
   - if it is a private account (e.g., a bank account), then log in to the account using $sk_{\mathcal{C}}$.
   - otherwise (if it is a public account, e.g., Ethereum), access the public account.
2. Verify π_j by checking if the transfer took place.
   - if the check passes, then set $\hat{z} = 1$ and $\zeta_{\mathcal{C}} = \perp$.
   - otherwise, set $\hat{z} = 0$ and $\zeta_{\mathcal{C}} = AccountStatement$.
3. Return $(\hat{z}, \zeta_{\mathcal{C}})$.

Fig. 9: Client's algorithm to check if the requested exchange occurred.

ExtractVerdict($plc_{\lambda}, e_1, \ldots, e_m, adr_P$) $\rightarrow \hat{r}$

- *Input.* plc_{λ}: operator $\mathcal{O}$'s policy including a threshold t specifying the fraction of potential corrupt registered auditors; auditors' verdicts: $e_1, \ldots, e_m$; the address adr_P of client P for which the verdict is decided.
- *Output.* $\hat{r} = 1$: if P must be reimbursed; $\hat{r} = 0$: otherwise.

1. Set final verdict $\hat{r}$ as follows.
   - initiate an empty vector $\vec{vec}$.
   - append every binary verdict to $\vec{vec}$ as follows.

$$\forall j, 1 \leq j \leq m: \text{ if } e_j \in \{0,1\}, \text{ then } \vec{vec} \leftarrow e_j$$

   - count the total number of votes in favor of a client: $counter \leftarrow \sum_{i=1}^{|\vec{vec}|} e_i$, where $e_i \in \vec{vec}$.
   - if $|\vec{vec}| = 0$, then set $counter \leftarrow 0$.
   - if $counter \geq |\vec{vec}| \cdot (1-t)$, set $\hat{r} = 1$. Else, set $\hat{r} = 0$.
2. Return $\hat{r}$.

Fig. 10: $\mathcal{SC}$'s algorithm to retrieve final verdict.

$\mathcal{SC}$ retains $\mathcal{C}$'s premium and transfers the rest of $\mathcal{C}$'s payment to $\mathcal{S}$ after a certain period. Upon receiving the payment, $\mathcal{S}$ transfers the required amount in the destination currency to $\mathcal{C}$. Then, $\mathcal{S}$ sends the transaction's proof to $\mathcal{SC}$, which may not necessarily be a cryptographic proof. If $\mathcal{C}$ later concludes that the service has not been provided in accordance with their agreement, they submit a complaint to $\mathcal{SC}$. Each registered auditor reviews the complaint and provides their verdict to $\mathcal{SC}$. If the auditors' verdicts support the complaint, $\mathcal{SC}$ reimburses $\mathcal{C}$ from its budget.

6.4 Overview of Each Phase of ICE

To enhance conceptual understanding of the various phases of ICE, this section provides a high-level overview of each phase, along with Figures 11 and 12 depicting parties' interactions within these phases.

1. Insurance Operator-side Setup: The insurance operator develops and deploys a smart contract into a blockchain, e.g., Ethereum. It publishes the address of this smart contract, so that everyone can access it. This smart contract will primarily function as a conventional insurer, while also offering additional features. The operator stores a comprehensive insurance policy within the smart contract and deposits in it an initial capital amount, which serves as the starting funding for the insurance. The operator updates its state, by adding to it the details of known misbehaving servers and qualified auditors.
2. Key Generation: Each party that wants to use the platform independently generates their pair of private and public keys to send transactions to the smart contract and receive payments. They publish their public key.
3. Operator-side Registration: When the operator receives a request from a party seeking registration in the smart contract (as either a server which will provide currency exchange or an auditor), it initiates a verification process. This process ensures that the requesting party is not a known bad actor, in the case of server registration, or that they meet the qualifications required to be an auditor. Upon successful verification, the operator registers the party's details in the smart contract, along with the specifics of the service they will provide, if the party is being registered as a server. The operator may register different parties at different times.
4. Client-side Initiation: When a client wishes to exchange a certain amount of cryptocurrency for another currency, it first retrieves a list of registered servers from the smart contract. The client then filters this list to identify servers it considers valid. After selecting one of the valid servers, the client sends a request to both the server and the smart contract, including the details of the desired exchange.

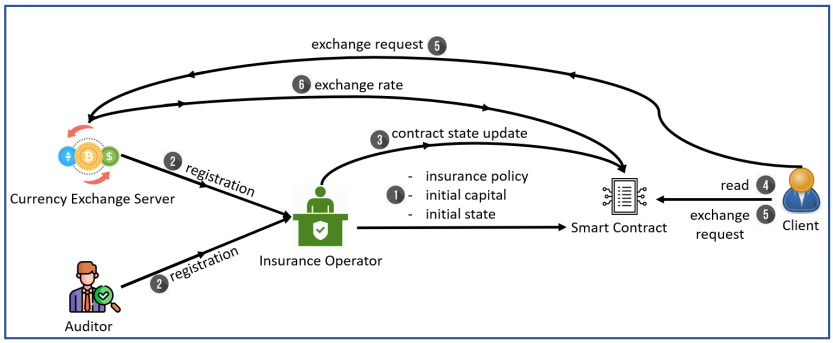

Fig. 11: Algorithms 1–6.

5. Server-side Initiation: Upon receiving a request from a client, the server first checks the validity of the request. If the request is deemed valid, the server then determines the exchange rate for the specific transaction. It subsequently sends this rate to the smart contract.
6. Transaction Transmission: Given the rate provided by the server, the client decides whether to accept the offer. If the client chooses to proceed, it generates a new request and calculates the required premium. The client then submits this request, along with the necessary funds to cover both the exchange amount and the premium, to the smart contract. Upon receiving the request, the smart contract verifies whether it has sufficient funds to complete the transaction. If the smart contract confirms it cannot fulfill the request, it will refund the client.
7. Request Verification: Upon receiving the client's final request and payment, the server verifies two main conditions: (i) whether it accepts the client's requested rate, and (ii) whether it has sufficient budget to fulfill the client's request. If both conditions are met, the server sends a flag, set to 1, to the smart contract.

8. Withdrawal: Upon receiving a message from the client, the smart contract determines if the client is eligible for withdrawal by verifying that the request was made within a predefined time window. If this condition is met, the smart contract processes the refund to the client.
9. Fund Transfer: After the withdrawal period has expired, the smart contract initiates several checks to ensure the following: (1) the server has approved the client's request, (2) the client has not already withdrawn the payment, and (3) the client has transferred a sufficient amount of coins. If conditions (1) and (3) are not met, the smart contract automatically refunds the client. If all conditions are satisfied, the contract deducts the premium amount from the client's payment, adds it to its balance, and transfers the server's share. If the client has overpaid, the contract refunds the excess amount to the client.
10. Serving a Customer: After receiving the client's payment via the smart contract, the server fulfills the client's request by transferring the specified amount of money or cryptocurrency in the quoted currency, based on the agreed-upon rate. As proof, the server sends the confirmation of this transaction to the smart contract. The server then updates its local ledger to reflect the transaction.

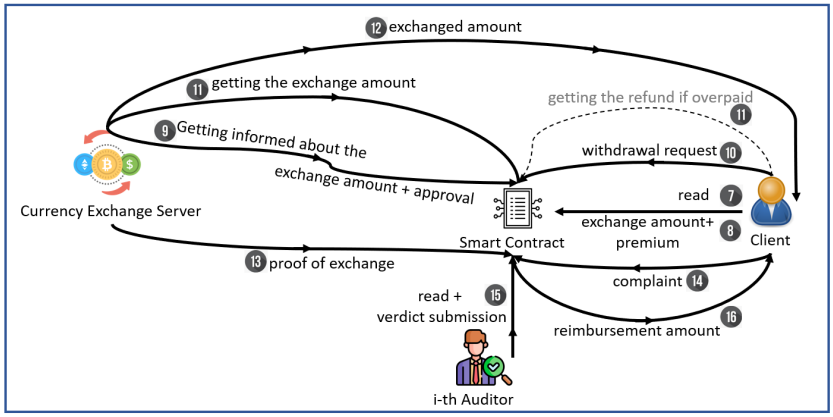

Fig. 12: Algorithms 7–12

11. Generating a Complaint: The client verifies whether the service has been completed and the agreed-upon amount has been transferred. If the verification fails, the client generates a complaint and sends it to the smart contract, including the transaction details through which the amount was transferred.
12. Reimbursing a Customer: Each auditor retrieves the client's complaint and processes it locally. After evaluation, each auditor generates a verdict and submits it to the smart contract. The smart contract then determines whether the client should be reimbursed, based on the collective verdicts and a predefined function. If the final decision favors the client, the smart contract calculates the reimbursement amount according to predefined rules and transfers the funds to the client. The smart contract also updates its state to reflect this transaction.

6.5 Detailed Description of ICE

We present the ICE protocol, assuming all parties sign outgoing messages and verify signatures before processing them locally.

1. Insurance Operator-side Setup: $\texttt{Operator.Setup}(1^\lambda, plc_\lambda) \to (st_{\mathcal{O}}, pp, adr_{\mathcal{SC}}, \langle \mathcal{SC} \rangle, sk_{\mathcal{O}})$
   $\mathcal{O}$ takes the following steps.
   (a) initiates an empty set pp. It generates a pair of a private key $sk_{\mathcal{O}}$ and a public key $pk_{\mathcal{O}}$. It appends $pk_{\mathcal{O}}$ to pp.
   (b) develops a smart contract $\mathcal{SC}$ and deploys it. Let $adr_{\mathcal{SC}}$ be the address of the deployed $\mathcal{SC}$. It publishes the address $adr_{\mathcal{SC}}$ of $\mathcal{SC}$, e.g., on its website.

(c) records its policy plc_λ in $\mathcal{SC}$, where plc_λ includes (i) insurance period: Θ, (2) transaction period Δ, (3) a list L_{pnd} initially empty, (4) a parameter t determining the fraction of potential corrupted registered auditors, and (5) a pointer to two databases: (i) $db_{\mathcal{S}}$, the database of disqualified service providers and (ii) $db_{\mathcal{D}}$, the database of qualified/approved auditors. It stores pp in $\mathcal{SC}$ which appends plc_λ to pp.
(d) deposits Γ amount in $\mathcal{SC}$, where Γ is specified in plc_λ.
(e) initiates its state by setting $st_{\mathcal{O}} = \emptyset$.
(f) updates its $st_{\mathcal{O}}$ by calling $\mathsf{Update.State}(st_{\mathcal{O}}, pp, sk_{\mathcal{O}}, adr_{\mathcal{SC}}) \rightarrow st_{\mathcal{O}}$.
(g) locally stores $st_{\mathcal{O}}$.

2. Key Generation: $\mathtt{Party.GenParams}(1^\lambda, adr_P) \rightarrow (sk_P, pk_P)$
Each party P generates a pair of a private key sk_P and a public key pk_P for the blockchain's signature scheme.

3. Operator-side Registration: $\mathtt{Register}\Big(\langle P(msg_P, sk_P), \mathcal{O}(st_{\mathcal{O}},\ adr_{\mathcal{SC}},\ sk_{\mathcal{O}})\rangle\Big) \rightarrow \hat{c}$
It involves $\mathcal{O}$ and a party P that can be an auditor or a server.
(a) P sends message msg_P to $\mathcal{O}$, where msg_P contains the address of the party and also when the party is the server it contains: (i) the service function $\mathsf{F}_{\mathcal{S}}$ which is the function's description: "cryptoExchange" and (ii) a pair $\mathbf{V}_{\mathcal{S}}$ defined as a set of triple $= (val, X, Y)$ containing a possessive integer val and the identifiers of currencies valid for exchange from type X to supported types Y specified in the list $pp_{\mathcal{S}}$.[8]
(b) upon receiving msg_P, $\mathcal{O}$ updates its state $st_{\mathcal{O}}$ by reading from the blockchain permanent state and then calling algorithm $\mathsf{Validate}^{(\mathcal{O})}(plc_\lambda, st_{\mathcal{O}}, msg_P) \rightarrow \hat{b}$, presented in Figure 6.
– If $\hat{b} = 1$: sets $\hat{c} \leftarrow 1$. If P is a server $\mathcal{S}$, it stores $(adr_{\mathcal{S}}, \hat{c}, \mathsf{F}_{\mathcal{S}}, pp_{\mathcal{S}},\ \mathbf{V}_{\mathcal{S}})$ in $\mathcal{SC}$. If P is an auditor $\mathcal{D}$, it stores $(adr_{\mathcal{D}}, \hat{c})$ in $\mathcal{SC}$.
– If $\hat{b} = 0$, it sets $\hat{c} \leftarrow 0$ and sends $(\mathsf{Registration}, \hat{c})$ to $\mathcal{SC}$.

4. Client-side Initiation: $\mathtt{C.Init}(inpt_{\mathcal{C}_i}, sk_{\mathcal{C}_i}, pk_{\mathcal{C}_i}, adr_{\mathcal{S}_j}, adr_{\mathcal{SC}},\ st_{\mathcal{C}_i}) \rightarrow (inpt^*_{\mathcal{C}_i}, \hat{a}, pp_{\mathcal{C}_i})$
This phase involves a client $\mathcal{C}_i$.
(a) updates its state: $\mathsf{Update.State}(st_{\mathcal{C}_i}, pp, sk_{\mathcal{C}_i}, adr_{\mathcal{SC}}) \rightarrow st_{\mathcal{C}_i}$.
(b) validates server $\mathcal{S}_j$: $\mathsf{Validate}^{(\mathcal{C})}(st_{\mathcal{C}_i}, adr_{\mathcal{S}_j}, adr_{\mathcal{SC}}) \rightarrow \hat{a}$. If $\hat{a} = 0$, it returns $(\perp, \hat{a}, \perp)$ and does not proceed to the next steps. Otherwise (when $\hat{a} = 1$), it proceeds to the next step.
(c) if $inpt_{\mathcal{C}_i} \notin \mathbf{V}_{\mathcal{S}_j}$, it returns $(\perp, 0, \perp)$ and halts.
(d) expresses interest in being served by $\mathcal{S}_j$ with address $adr_{\mathcal{S}_j}$ by taking the following steps.
i. executes $\mathtt{CCE.C.Init}(inpt_{\mathcal{C}_i}, adr_{\mathcal{S}_j}, pp_{\mathcal{S}_j}) \rightarrow (X, Y)$, where $inpt_{\mathcal{C}_i} = (\alpha, X, Y)$ and $pp_{\mathcal{S}_j} \in \mathcal{SC}$.
ii. chooses a unique identifier $id\_inpt_{\mathcal{C}_i}$.
iii. sends $inpt^*_{\mathcal{C}_i} = (id\_inpt_{\mathcal{C}_i}, X, Y)$ and $adr_{\mathcal{C}_i}$ to $\mathcal{S}_j$.
iv. sets $pp_{\mathcal{C}_i} \leftarrow inpt^*_{\mathcal{C}_i}$ and sends $pp_{\mathcal{C}_i}$ to $\mathcal{SC}$.

5. Server-side Initiation: $\mathtt{S.Init}(inpt^*_{\mathcal{C}_i}, adr_{\mathcal{C}_i}, sk_{\mathcal{S}_j}, pk_{\mathcal{S}_j}, st_{\mathcal{S}_j},\ adr_{\mathcal{SC}}) \rightarrow (\hat{e}, resp_{\mathcal{S}_j})$
This phase involves $\mathcal{S}_j$.
(a) checks if there are registered auditors; if not, sets $\hat{e} \leftarrow 0$ and $resp_{\mathcal{S}_j} \leftarrow \perp$, sends $(inpt^*_{\mathcal{C}_i}, \hat{e}, resp_{\mathcal{S}_j},\ adr_{\mathcal{C}_i})$ to $\mathcal{SC}$ and halts.
(b) checks the validity of $inpt^*_{\mathcal{C}_i}$, by calling $\mathtt{CCE.S.Init}((X,\ Y), adr_{\mathcal{C}_i}, st_{\mathcal{S}}) \rightarrow (\hat{e}, (\rho, \mu))$.
(c) sets $resp_{\mathcal{S}_j} \leftarrow (\rho, \mu)$ and sends $(inpt^*_{\mathcal{C}_i}, \hat{e}, resp_{\mathcal{S}_j}, adr_{\mathcal{C}_i})$ to $\mathcal{SC}$ which stores them if the server has already been registered.

6. Transaction Transmission: $\mathtt{SendTransaction}(\mathcal{C}_i(id\_inpt_{\mathcal{C}_i},\ sk_{\mathcal{C}_i}, inpt_{\mathcal{C}_i}, st_{\mathcal{C}_i}, \alpha, adr_{\mathcal{S}_j}, adr_{\mathcal{SC}}), \mathcal{SC}) \rightarrow (\gamma, tx_i,\ req_{\mathcal{C}_i})$
It involves $\mathcal{C}_i$ and $\mathcal{SC}$.
(a) client $\mathcal{C}_i$ takes the following steps:
i. reads the content of $\mathcal{SC}$, including $\hat{e}, resp_{\mathcal{S}_j}$ and pp. It proceeds only if $\hat{e} = 1$ (otherwise, outputs $\perp$).

[8] X can be viewed as a native digital asset of the underlying blockchain, while Y is a digital asset supported by that blockchain and exists within its ecosystem.

ii. runs $\mathsf{Decide}(st_{\mathcal{C}_i}, \alpha, inpt_{\mathcal{C}_i}, inpt_{\mathcal{S}_j}) \rightarrow \hat{q}$. It proceeds if $\hat{q} = 1$.
iii. generates a request: $\texttt{CCE.C.Request}(inpt_{\mathcal{C}_i}, \alpha, adr_{\mathcal{S}_j}, resp_{\mathcal{S}_j}, st_{\mathcal{C}_i}) \rightarrow req^*_{\mathcal{C}_i} = (\alpha, X, Y, \rho)$. It sets $req_{\mathcal{C}_i} \leftarrow (id\_inpt_{\mathcal{C}_i}, req^*_{\mathcal{C}_i})$.
iv. calculates the total amount it must pay by setting: $\gamma \leftarrow \mathsf{CalPremium}(state_{\mathcal{SC}}, adr_{\mathcal{S}_j}, req_{\mathcal{C}_i}) + \alpha$.
v. sends γ amounts of coin, $adr_{\mathcal{S}_j}$, and $req_{\mathcal{C}_i}$ to $\mathcal{SC}$.

(b) $\mathcal{SC}$ takes the following steps:
i. parses L_{pnd} as a list of triples of the format $(\tilde{tx}, \tilde{v}, \tilde{T})$, where (i) $\tilde{tx}$ is a transaction identifier, (ii) $\tilde{v}$ is a compensation amount associated with $\tilde{tx}$, and (iii) $\tilde{T}$ is the time when $\tilde{tx}$ was registered. Let T be the current time.
ii. updates its pending list, $\mathsf{UpdateList}(L_{\mathsf{pnd}}, T, \theta) \rightarrow L_{\mathsf{pnd}}$, to ensure that the transactions that will not receive insurance coverage will be excluded from the list.
iii. checks if it has enough budget to serve $\mathcal{C}_i$ through calling: $\mathsf{CheckBudget}(req_{\mathcal{C}_i}, adr_{\mathcal{S}_j}, adr_{\mathcal{SC}}, \mathsf{balance}, state_{\mathcal{SC}}, plc_\lambda, \gamma, pp, L_{\mathsf{pnd}}) \rightarrow \hat{w}$.
iv. if $\hat{w} = 0$: the algorithm returns $\perp$. This indicates that there is no sufficient balance to guarantee compensation for this transaction. In this case, it takes no further action. Otherwise (when $\hat{w} = 1$): $\mathcal{SC}$ generates a unique transaction identifier tx_i and sets two associated flags $\hat{f}$ and $\hat{g}$ to 0. The identifier tx_i includes the addresses $adr_{\mathcal{C}_i}$ and $adr_{\mathcal{S}_j}$.
v. adds $(tx_i, \mathsf{CompRemAmount}(plc_\lambda, \gamma, req_{\mathcal{C}_i}, pp), T_1)$ to L_{pnd}.

(c) $\mathcal{C}_i$ sends $(tx_i, inpt_{\mathcal{C}_i})$ to $\mathcal{S}_j$.

7. Request Verification: $\texttt{VerRequest}(adr_{\mathcal{SC}}, tx_i, inpt_{\mathcal{C}_i}) \rightarrow \hat{f}$
This phase involves $\mathcal{S}_j$.
(a) given $adr_{\mathcal{SC}}$ and tx_i, reads $\hat{e}, req_i$, and $adr_{\mathcal{C}_i}$ from $\mathcal{SC}$. It proceeds to the next steps, only if $\hat{e} = 1$.
(b) verifies $req_{\mathcal{C}_i}$ by first checking if $req_{\mathcal{C}_i} = (id\_inpt_{\mathcal{C}_i}, \alpha, X, Y, \rho)$ includes $inpt_{\mathcal{C}_i}$ (i.e., if $inpt_{\mathcal{C}_i} = (\alpha, X, Y)$). If so, it runs $\texttt{CCE.S.VerRequest}((\alpha, X, Y, \rho), adr_{\mathcal{C}_i}, st_{\mathcal{S}_j}) \rightarrow \hat{f}$. Else, it sets $\hat{f} \leftarrow 0$.
(c) sends $\hat{f}$ to $\mathcal{SC}$, that stores it if $\mathcal{S}_j$ has already been registered.

8. Withdrawal: $\texttt{Withdraw}(adr_{\mathcal{C}_i}, pp, tx_i, T, T_1) \rightarrow \hat{g}$
This phase involves $\mathcal{SC}$, which itself is invoked by $\mathcal{C}_i$.
(a) checks if $T - T_1 \leq \Delta$, where Δ is a delay parameter in pp.
(b) if the checks pass, returns γ amount (that $\mathcal{C}_i$ paid via transaction tx_i) to address $adr_{\mathcal{C}_i}$, deletes $(tx_i, \cdot, \cdot)$ from L_{pnd}, and sets $\hat{g} \leftarrow 1$. Otherwise, it proceeds to the next step.
(c) keeps $\hat{g}$ as its public state.

9. Fund Transfer: $\texttt{Transfer}(\hat{e}, \hat{f}, \hat{g}, tx_i, \alpha, \gamma, \Delta, T, T_1) \rightarrow \hat{h}$
This phase involves $\mathcal{SC}$, which can be invoked by any party.
(a) checks the value of $\hat{e} = 1$ and proceeds only if $\hat{e} = 1$, $\mathcal{S}_j$ is registered, $T - T_1 > \Delta$, and $\hat{g} = 0$ (else, it outputs $\perp$).
(b) if $\hat{f} = 0$, returns $\hat{h} = 0$ and sends γ coins back to $\mathcal{C}_i$, deletes $(tx_i, \cdot, \cdot)$ from L_{pnd}, and does not take any action regarding tx_i. If $\hat{f} = 1$, it reads $req_{\mathcal{C}_i}$ associated with tx_i and checks γ.
– if $\gamma \geq \mathsf{CalPremium}(state_{\mathcal{SC}}[T_1], adr_{\mathcal{S}_j}, req_{\mathcal{C}_i}) + \alpha$, it:
i. calls $\texttt{CCE.C.Transfer}(req_{\mathcal{C}_i}, \hat{f}, adr_{\mathcal{S}_j}) \rightarrow (\alpha, X_{\mathcal{C}_i})$ which transfers α amount of coins in $X_{\mathcal{C}_i}$ to $\mathcal{S}_j$.
ii. keeps $\mathsf{CalPremium}(state_{\mathcal{SC}}[T_1], adr_{\mathcal{S}_j}, req_{\mathcal{C}_i})$ amount of coins.
iii. updates its $\mathsf{balance}$ by executing: $\mathsf{balance} \leftarrow \mathsf{balance} + \mathsf{CalPremium}(state_{\mathcal{SC}}[T_1], adr_{\mathcal{S}_j}, req_{\mathcal{C}_i})$.
iv. returns $\gamma - \mathsf{CalPremium}(state_{\mathcal{SC}}[T_1], adr_{\mathcal{S}_j}, req_{\mathcal{C}_i}) - \alpha$ amount to $\mathcal{C}_i$.
v. sets $\hat{h} \leftarrow 1$.
– if $\gamma < \mathsf{CalPremium}(state_{\mathcal{SC}}[T_1], adr_{\mathcal{S}_j}, req_{\mathcal{C}_i}) + \alpha$, it:
i. sends γ amount of coins back to $\mathcal{C}_i$.
ii. deletes $(tx_i, \cdot, \cdot)$ from L_{pnd}.
iii. sets $\hat{h} \leftarrow 0$.

(c) returns $\hat{h}$.

10. Serving a Customer: $\texttt{Serve}(sk_{\mathcal{S}_j}, adr_{\mathcal{SC}}, tx_i) \rightarrow (service_j, \pi_j)$
This phase involves $\mathcal{S}_j$.
   (a) reads $\mathcal{SC}$. It proceeds if bit $\hat{h}$ associated with tx_i equals 1 (else, it outputs $\perp$). It initially sets $service_j$ and π_j to a special symbol $\perp$. It reads from $\mathcal{SC}$ request $req_{\mathcal{C}_i}$ associated with tx_i.
   (b) calls $\texttt{CCE.S.Serve}(req_i^*, \alpha, X_{\mathcal{C}_j}, adr_{\mathcal{C}_i}, st_{\mathcal{S}}) \rightarrow (\phi, Y_{\mathcal{C}_i})$, where $\phi = \alpha \cdot \rho$.
   (c) sets $service_j$ to $(\phi, Y_{\mathcal{C}_i})$ and returns $service_j$. By invoking $\texttt{CCE.S.Serve}$, $\mathcal{S}_j$ sends ϕ amount of coins of type $Y_{\mathcal{C}_i}$ to $\mathcal{C}_i$. Let π_j be the transaction confirmation. $\mathcal{S}_j$ sends π_j to $\mathcal{SC}$.
11. Generating a Complaint: $\texttt{GenComplaint}(sk_{\mathcal{C}_i}, service_j, tx_i, adr_{\mathcal{SC}}) \rightarrow complaint_i$
This phase involves $\mathcal{C}_i$.
   (a) reads $\mathcal{SC}$, including π_j and $req_{\mathcal{C}_i}$ associated with tx_i.
   (b) checks whether the service has been delivered, by calling $\mathsf{VerifyExchage}(sk_{\mathcal{C}}, req_{\mathcal{C}}, \pi_j) \rightarrow (\hat{z}, \zeta_{\mathcal{C}})$.
   (c) if verification does not pass, i.e., $\hat{z} = 0$, generates $complaint_i$ (that contains $\zeta_{\mathcal{C}}$ and tx_i). Otherwise, it sets $complaint_i = \perp$.
   (d) if $complaint_i \neq \perp$, $\mathcal{C}_i$ sends $complaint_i$ to $\mathcal{SC}$. Let T_3 be the time $complaint_i$ is registered in $\mathcal{SC}$.
12. Reimbursing a Customer: $\texttt{Reimburse}\Big(\langle \mathcal{D}_1(adr_{\mathcal{SC}}, sk_{\mathcal{D}_1}), \ldots, \mathcal{D}_m(adr_{\mathcal{SC}}, sk_{\mathcal{D}_m}), \mathcal{SC}(plc_\lambda, \alpha, \gamma, complaint_i, pp)\rangle\Big) \rightarrow amount_i$
This phase involves the registered auditors $\mathcal{D}_1, \ldots, \mathcal{D}_m$ and $\mathcal{SC}$.
   (a) each $\mathcal{D}_\ell$ takes the following steps:
      i. reads $\mathcal{SC}$ (including T_3, $complaint_i$, and plc_λ).
      ii. decides its verdict $\hat{d}_\ell \in \{0, 1\}$.
      iii. sends $(complaint_i, \hat{d}_\ell)$ to $\mathcal{SC}$ before time $T = T_3 + \Delta$.
   (b) $\mathcal{SC}$ takes the below steps, upon receiving $\hat{d}_1, \ldots, \hat{d}_m$ from auditors and on input $plc_\lambda, \alpha_{\mathcal{S}_j}, \gamma, complaint_i$, and pp.
      i. if $\mathcal{D}_\ell$ has not been registered or it did not send its verdict before time $T = T_3 + \Delta$, then sets $\hat{d}_\ell$ to $\perp$.
      ii. calls $\mathsf{ExtractVerdict}(plc_\lambda, \hat{d}_1, \ldots, \hat{d}_m, adr_{\mathcal{C}_i}) \rightarrow \hat{r}$ to determine the final verdict.
         – if $\hat{r} = 0$, sets $amount_i = 0$ and takes no further action.
         – otherwise (when $\hat{r} = 1$), it:
            A. determines the amount of reimbursement that $\mathcal{C}_i$ must receive by calling the algorithm $\mathsf{CompRemAmount}(plc_\lambda, \gamma, req_{\mathcal{C}_i}, pp) \rightarrow amount_i$, where $req_{\mathcal{C}_i}$ is the request associated with tx_i contained in $complaint_i$.
            B. updates its balance as $\mathsf{balance} \leftarrow \mathsf{balance} - amount_i$.
            C. deletes $(tx_i, \cdot, \cdot)$ from L_{pnd}. It sends $amount_i$ coins to address $adr_{\mathcal{C}_i}$, where $adr_{\mathcal{C}_i} \in complaint_i$.

7 Correctness and Security Analysis

In this section, we state and prove our main ICE Correctness and Security Theorem 1, under the assumptions that the blockchain and the corresponding signature scheme are secure. First, we describe the algorithmic steps that an (honest) auditor should take so that set of auditors guarantees ICE security.

7.1 Auditor Description for ICE Security

Below, we present the algorithmic $\mathcal{D}_\ell(st_{\mathcal{D}_\ell}, complaint_{\mathcal{C}})$ steps that each of the (non-corrupted) auditors should follow to guarantee client-side and server-side security w.r.t. corruption threshold t.

– *Input.* $st_{\mathcal{D}_\ell}$: the auditor's state. $complaint_{\mathcal{C}}$: the complaint generated by client $\mathcal{C}$.
– *Output.* $d_\ell = 1$: if $\mathcal{C}$ must be reimbursed; $d_\ell = 0$: otherwise.

◇ Decide verdict d_ℓ as follows.
- parse $complaint_{\mathcal{C}}$ as pair (ζ, tx).
- if tx does not include the address $adr_{\mathcal{C}}$ of $\mathcal{C}$ or the address $adr_{\mathcal{S}}$ of some server $\mathcal{S}$, then return $d_\ell \leftarrow 0$.
- if tx is a transaction identifier such that $(completed, tx)$ is in $st_{\mathcal{D}_\ell}$, then return $d_\ell \leftarrow 0$.
- let T be the current time and T_1 be the time when transaction tx was registered. If the insurance period of $\mathcal{C}$ has expired, i.e., $T > T_1 + \Theta$, then return $d_\ell \leftarrow 0$.
- if tx is such that any of the following associated information is missing from the $\mathcal{SC}$:
  - $\mathcal{C}$'s public parameters $pp_{\mathcal{C}}$ including identifier $id\_inpt_{\mathcal{C}}$;
  - the response information from $\mathcal{S}$: $(id\_inpt_{\mathcal{C}}, 1, resp_{\mathcal{S}}, adr_{\mathcal{C}})$;
  - the triple $(req_{\mathcal{C}}, adr_{\mathcal{S}}, \gamma)$ submitted by $\mathcal{C}$, where $req_{\mathcal{C}} = (id\_inpt_{\mathcal{C}}, \alpha, X, Y, \rho)$;
  - $\mathcal{S}$'s acceptance of $\mathcal{C}$'s request (i.e., $\hat{f}$ associated with tx is set to 1);
  - the withdrawal status $\hat{g}$ is set to 0;
  - the transfer status $\hat{h}$ is set to 1;

  then add $(completed, tx)$ to $st_{\mathcal{D}_\ell}$ and return $d_\ell \leftarrow 0$.
- if $\zeta = \perp$, add $(completed, tx)$ to $st_{\mathcal{D}_\ell}$ and return $d_\ell \leftarrow 0$.
- if $\zeta \neq \perp$, then parse ζ as string *AccountStatement* and read $\mathcal{S}$'s associated proof/statement π.
- if (i) *AccountStatement* is a statement asserting $\mathcal{S}$ did not transfer $\alpha \cdot \rho$ coins to the account of $\mathcal{C}$ and (ii) π is an invalid statement of the transaction, then add $(completed, tx)$ to $st_{\mathcal{D}_\ell}$, then add $(completed, tx)$ to $st_{\mathcal{D}_\ell}$ and return $d_\ell \leftarrow 1$. Else, add $(completed, tx)$ to $st_{\mathcal{D}_\ell}$ and return $d_\ell \leftarrow 0$.[9]

7.2 ICE Correctness and Security Theorem

Below, we state our main Correctness and Security Theorem for the ICE scheme described in Section 6.5.

Theorem 1. *Let Σ_{ICE} be the ICE protocol described in Section 6.5 w.r.t. (i) the policy family $plc = \{plc_\lambda\}_{\lambda \in \mathbb{N}}$ and (ii) the set of auditors $\mathbf{D}$, where every (honest) auditors behaves according to the steps in Section 7.1. Assume that the underlying blockchain has persistence (Definition 5) and liveness (Definition 6), and the digital signature scheme (Definition 7) used in the smart contract is existentially unforgeable under a chosen message attack (Definition 8). Then, for every $t \in [0, \frac{1}{2}]$, Σ_{ICE} is correct (Definition 1) and t-secure (Definition 4) w.r.t. plc and $\mathbf{D}$.*

7.3 Proof of Correctness

Let $\mathcal{A}$ be an adversary that participates in the Correctness security game defined in Section 5.2. After completing the **Initialization** and **Execution** phases of the game, $\mathcal{A}$ provides Ch with two challenge addresses, $adr_{\tilde{\mathcal{C}}}$ and $adr_{\tilde{\mathcal{S}}}$, an input $inpt_{\tilde{\mathcal{C}}}$, and an amount $\tilde{\alpha}$.

Proof. Without loss of generality, we may assume that: (i) there is at least one successfully registered auditor, (ii) $\tilde{\mathcal{C}}$ is an honest client that has generated its key pair, and (iii) $\tilde{\mathcal{S}}$ is an honest server that has generated its key pair and is also registered. Indeed, if any of these conditions do not hold, the adversary would lose the game in one of Steps 1-4 (see Section 5.2). Hence, Step 5 in Section 5.2 is run by Ch, which invokes $\mathtt{C.Init}(inpt_{\tilde{\mathcal{C}}}, sk_{\tilde{\mathcal{C}}}, pk_{\tilde{\mathcal{C}}}, adr_{\tilde{\mathcal{S}}}, adr_{\mathcal{SC}}, st_{\tilde{\mathcal{C}}})$.
Given the correctness of $\mathsf{Find.SusAccount}$ (i.e., an honest server is not identified as suspected), and considering that: (i) the initial state of $\tilde{\mathcal{C}}$ contains no bad actors, and (ii) $adr_{\tilde{\mathcal{S}}}$ is registered in $\mathcal{SC}$, it follows that $\mathsf{Validate}^{(\mathcal{C})}(st_{\tilde{\mathcal{C}}}, adr_{\tilde{\mathcal{S}}}, adr_{\mathcal{SC}})$ will output 1 (and will not output $(\perp, 0, \perp)$, ensuring that $\mathcal{A}$ will not win due to the condition in Step 6. Thus, Ch will invoke $\mathtt{S.Init}(inpt^*_{\tilde{\mathcal{C}}}, adr_{\tilde{\mathcal{C}}}, sk_{\tilde{\mathcal{S}}}, pk_{\tilde{\mathcal{S}}}, st_{\tilde{\mathcal{S}}}, adr_{\mathcal{SC}})$, where $inpt^*_{\tilde{\mathcal{C}}}$ is the output of $\mathtt{CCE.C.Init}(inpt_{\tilde{\mathcal{C}}}, adr_{\tilde{\mathcal{S}}}, pp_{\tilde{\mathcal{S}}})$ (Step 7 will then be executed).
Assume that $\mathtt{S.Init}$ outputs $(\hat{e}, resp_{\tilde{\mathcal{S}}})$ such that $\hat{e} = 0$. Given that there is at least one successfully registered auditor, this can occur only if the underlying $\mathtt{CCE.S.Init}(inpt^*_{\mathcal{C}_i}, adr_{\mathcal{C}_i}, st_{\mathcal{S}})$ returns $\hat{e} = 0$. In turn, this can

[9] If both *AccountStatemnt* and π are valid, or if *AccountStatemnt* is invalid, then the auditor decides that the client should not be reimbursed. Note that concurrent validity of *AccountStatemnt* and π (i.e., concurrent evidence that the transaction took place and that it did not take place) is an inconsistency that will not happen in our setting.

happen only if $inpt^*_{\tilde{\mathcal{C}}}$ specifies a pair (X, Y) of cryptocurrencies that the server cannot support. This implies that $inpt_{\tilde{\mathcal{C}}} \notin \mathbf{V}_{\tilde{\mathcal{S}}}$, and, following the condition in Step 8, $\mathcal{A}$ will lose the game. Therefore, without loss of generality, we can assume that $inpt_{\tilde{\mathcal{C}}} \in \mathbf{V}_{\tilde{\mathcal{S}}}$, i.e., (X, Y) is a valid pair, which implies that `S.Init` outputs $\hat{e} = 1$ and a response $resp_{\tilde{\mathcal{S}}} = (\rho, \mu)$ (thus, $\mathcal{A}$ will not win due to the condition in Step 9). Thus, Ch will execute the protocol $\texttt{SendTransaction}(\tilde{\mathcal{C}}(id\_inpt_{\tilde{\mathcal{C}}}, sk_{\tilde{\mathcal{C}}}, inpt_{\tilde{\mathcal{C}}}, st_{\tilde{\mathcal{C}}}, \tilde{\alpha}, adr_{\tilde{\mathcal{S}}}, adr_{\mathcal{SC}}), \mathcal{SC})$ (Step 10 will then be executed).

Then, according to the ICE protocol description, Ch will execute $\texttt{CCE.C.Request}(inpt_{\tilde{\mathcal{C}}}, \tilde{\alpha}, adr_{\tilde{\mathcal{S}}}, (\rho, \mu), st_{\tilde{\mathcal{C}}})$. If $\mathsf{Decide}(st_{\tilde{\mathcal{C}}}, \tilde{\alpha}, inpt_{\tilde{\mathcal{C}}}, resp_{\tilde{\mathcal{S}}})$ outputs $\hat{q} = 0$, then $\mathcal{A}$ will lose the game. Hence, by the description of the $\mathsf{Decide}(\cdot)$ function and without loss of generality, we may assume that: (i) $\tilde{\alpha}$ matches the amount specified in $inpt_{\tilde{\mathcal{C}}}$ (i.e., $inpt_{\tilde{\mathcal{C}}} = (\tilde{\alpha}, X, Y)$), (ii) $\tilde{\mathcal{C}}$ accepts ρ, and (iii) $\mu \geq \tilde{\alpha} \cdot \rho$. The execution of `CCE.C.Request` will produce a request $req_{\tilde{\mathcal{C}}} = (id\_inpt_{\mathcal{C}_i}, \tilde{\alpha}, X, Y, \rho)$.

Ch computes the amount $\tilde{\gamma} \leftarrow \mathsf{CalPremium}(state_{\mathcal{SC}}, adr_{\tilde{\mathcal{S}}}, req_{\tilde{\mathcal{C}}}) + \tilde{\alpha}$ as described in the ICE protocol and checks if there is sufficient balance. Without loss of generality, we assume that $\mathsf{CheckBudget}(req_{\tilde{\mathcal{C}}}, adr_{\tilde{\mathcal{S}}}, adr_{\mathcal{SC}}, \mathsf{balance}, state_{\mathcal{SC}}, plc_\lambda, \tilde{\gamma}, pp, L_{\mathsf{pnd}}) \rightarrow 1$ (otherwise, $\mathcal{SC}$ outputs $\perp$ and $\mathcal{A}$ loses the game). At the end of this step, Ch has computed a unique transaction identifier $\tilde{tx}$ along with $\tilde{\gamma}$ and $req_{\tilde{\mathcal{C}}}$. It records T_1 at the time at which $\tilde{tx}$ is registered.

Ch runs $\texttt{VerRequest}(adr_{\mathcal{SC}}, \tilde{tx}, inpt_{\tilde{\mathcal{C}}})$ (Step 11 is executed). Given that $\hat{e} = 1$ and since $req_{\tilde{\mathcal{C}}}$ includes $inpt_{\tilde{\mathcal{C}}}$ where $inpt_{\tilde{\mathcal{C}}} = (\tilde{\alpha}, X, Y)$, Ch will run $\texttt{CCE.S.VerRequest}((\tilde{\alpha}, X, Y, \rho), adr_{\tilde{\mathcal{C}}}, st_{\tilde{\mathcal{S}}})$. In particular, it fetches the rate ρ^* in $\mathbf{RT}$ that corresponds to the pair (X, Y). Since the rate has not changed from the beginning of the challenge phase[10], and given that $req_{\tilde{\mathcal{C}}}$ has been honestly generated, it holds that $\rho^* = \rho$. Additionally, the available funds $\beta_{\tilde{\mathcal{C}}}$ have not changed from the beginning of the challenge phase[11], which implies that $\min\{\beta_{\tilde{\mathcal{C}}}, M_{\tilde{\mathcal{C}}}\} = \mu$. Since the output of $\mathsf{Decide}(st_{\tilde{\mathcal{C}}}, \tilde{\alpha}, inpt_{\tilde{\mathcal{C}}}, resp_{\tilde{\mathcal{S}}})$ is 1, we have that $\mu \geq \tilde{\alpha} \cdot \rho$. Hence, $\min\{\beta_{\tilde{\mathcal{C}}}, M_{\tilde{\mathcal{C}}}\} \geq \tilde{\alpha} \cdot \rho$. Therefore, both checks of `CCE.S.VerRequest` pass, and $\texttt{VerRequest}(adr_{\mathcal{SC}}, \tilde{tx}, inpt_{\tilde{\mathcal{C}}})$ outputs $\hat{f} = 1$ (so, $\mathcal{A}$ will not win due to the condition in Step 12).

After the execution of Step 13, Ch provides $\mathcal{A}$ with $(\tilde{tx}, req_{\tilde{\mathcal{C}}})$. In turn, in Step 14, $\mathcal{A}$ advances the clock and provides Ch with a bit $\tilde{g}$.

If $\tilde{g} = 1$, then Ch executes $\texttt{Withdraw}(adr_{\tilde{\mathcal{C}}}, pp, \tilde{tx}, T, T_1)$ (Step 16). By the description of `Withdraw`, the algorithm will output 1 if and only if $T - T_1 \leq \Delta$ (recall that the withdrawal bit $\hat{g}$ is by default set to 0). Therefore, $\mathcal{A}$ will not win due to the condition of either Step 17 or Step 18. In addition, if $\texttt{Withdraw}(adr_{\tilde{\mathcal{C}}}, pp, \tilde{tx}, T, T_1)$ outputs 1, then $\mathcal{A}$ will lose the game. So, without loss of generality, we may assume that either of the two following cases holds for $\tilde{g}$: either (i) $\tilde{g} = 0$ (Step 15), or (ii) $\tilde{g} = 1$ and $\texttt{Withdraw}(adr_{\tilde{\mathcal{C}}}, pp, \tilde{tx}, T, T_1)$ outputs 0 and $T - T_1 > \Delta$ (Step 20). We observe that in both above cases (i) and (ii), it holds that $\hat{g} = 0$ and Ch runs $\texttt{Transfer}(1, 1, 0, \tilde{tx}, \tilde{\alpha}, \tilde{\gamma}, \Delta, T, T_1)$ and then goes to Step 21.

By the description of `Transfer` and the facts that $\hat{e} = \hat{f} = 1$ and $\hat{g} = 0$, the algorithm will not output 1 unless one of the two conditions holds: (i) $T - T_1 \leq \Delta$; (ii) $\tilde{\gamma} < \mathsf{CalPremium}(state_{\mathcal{SC}}, adr_{\tilde{\mathcal{S}}}, req_{\tilde{\mathcal{C}}}) + \tilde{\alpha}$. The second condition does not hold since $\tilde{\gamma}$ has been set equal to $\mathsf{CalPremium}(state_{\mathcal{SC}}, adr_{\tilde{\mathcal{S}}}, req_{\tilde{\mathcal{C}}}) + \tilde{\alpha}$. Regarding the first condition, we may assume that $\mathcal{A}$ advances the clock such that $T - T_1 > \Delta$ (otherwise, $\mathcal{A}$ loses the game in Step 21).

By the above discussion, $\texttt{Transfer}(1, 1, 0, \tilde{tx}, \tilde{\alpha}, \tilde{\gamma}, \Delta, T, T_1)$ outputs $\hat{h} = 1$, and $\mathcal{SC}$ has transferred $\tilde{\alpha}$ amount of coins to $\tilde{\mathcal{S}}$. Then, on behalf of $\tilde{\mathcal{S}}$, Ch runs $\texttt{Serve}(sk_{\tilde{\mathcal{S}}}, adr_{\mathcal{SC}}, \tilde{tx})$, which outputs $(service_{\tilde{\mathcal{S}}}, \tilde{\pi})$ (Step 22 will be executed). By the description of `Serve`, the algorithm $\texttt{CCE.S.Serve}(req_{\tilde{\mathcal{C}}}, \tilde{\alpha}, X, adr_{\tilde{\mathcal{C}}}, st_{\tilde{\mathcal{S}}})$ will be invoked, producing the output $(\tilde{\alpha} \cdot \rho, Y)$. Thus, it holds that $service_{\tilde{\mathcal{S}}} = \mathtt{F}_{\tilde{\mathcal{S}}}(inpt^*_{\tilde{\mathcal{C}}}, req_{\tilde{\mathcal{C}}}, pp)$, hence, $\mathcal{A}$ will not win due to the condition of Step 23. Next, Ch will run $\texttt{GenComplaint}(sk_{\tilde{\mathcal{C}}}, service_{\tilde{\mathcal{S}}}, \tilde{tx}, adr_{\mathcal{SC}})$, which will verify the transfer via $\mathsf{VerifyExchage}(sk_{\tilde{\mathcal{C}}}, req_{\tilde{\mathcal{C}}}, \tilde{\pi}) \rightarrow (1, \perp)$. Therefore, `GenComplaint` will output $\perp$, and the adversary will lose the game. □

[10] Note that $\rho \in \mathbf{RT}$ and $\mathbf{RT}$ get updated by calling $\mathsf{UpdateRate}$ upon receiving a request. The assumption is that the game between the adversary and the challenger is based on a fixed request in this game. As a result, there is no change in the rate because no other requests are being processed concurrently.

[11] Recall, for the same reason discussed earlier, that since there are no other requests, the available funds remain unchanged, similar to the exchange rate.

7.4 Proof of Client-Side Security

Let $\mathcal{A}$ be an adversary that participates in the Client-side security game defined in Section 5.3. After completing the **Initialization** and **Execution** phases of the game, $\mathcal{A}$ provides Ch with two challenges addresses $adr_{\tilde{\mathcal{C}}}$ and $adr_{\tilde{\mathcal{S}}}$, an input $inpt_{\tilde{\mathcal{C}}}$, and an amount $\tilde{\alpha}$.

Proof. Without loss of generality, we may assume that:(i) there is at least one successfully registered auditor, (ii) $\mathcal{A}$ has corrupted less than t fraction of all successfully registered auditors, (iii) $\tilde{\mathcal{C}}$ is an honest client that has generated its key pair, and (iv) $\tilde{\mathcal{S}}$ is corrupted. Indeed, if any of these conditions do not hold, then the adversary would lose the game in one of Steps 1-3 (see Section 5.3). Therefore, Step 4 in Section 5.3 will be executed by Ch, which invokes $\texttt{C.Init}(inpt_{\tilde{\mathcal{C}}}, sk_{\tilde{\mathcal{C}}}, pk_{\tilde{\mathcal{C}}}, adr_{\tilde{\mathcal{S}}}, adr_{\mathcal{SC}}, st_{\tilde{\mathcal{C}}})$.
By the description of $\texttt{C.Init}$, if $\tilde{\mathcal{S}}$ is recorded as a bad actor or it has not been registered in $\mathcal{SC}$, then $\mathsf{Validate}^{(\mathcal{C})}(st_{\mathcal{C}_i}, adr_P, adr_{\mathcal{SC}})$ will return 0, and in turn, $\texttt{C.Init}$ will output $(\perp, 0, \perp)$, so the adversary will lose the game due to Step 5. Therefore, we assume that $\tilde{\mathcal{S}}$ is successfully registered via a valid $(\textsc{Registration}, msg_{\tilde{\mathcal{S}}})$ command which implies that $\mathcal{A}$ cannot win w.r.t. the condition of Step 6 in Section 5.3. Let $\mathtt{F}_{\tilde{\mathcal{S}}}$ be the service function and $\mathbf{V}_{\tilde{\mathcal{S}}}$ be the set of valid clients' inputs for the service that $\tilde{\mathcal{S}}$ provides. By the description of $\texttt{C.Init}$, the client $\tilde{\mathcal{C}}$ will verify the validity of $inpt_{\tilde{\mathcal{C}}}$. Thus, we assume that $inpt_{\tilde{\mathcal{C}}}$ is a valid input that parses as (α^*, X, Y) (otherwise $\texttt{C.Init}$ will output $(\perp, 0, \perp)$ and the adversary will lose the game due to Step 5 in Section 5.3). Therefore, $\mathcal{A}$ cannot win w.r.t. the condition of Step 7 in Section 5.3. In addition, Ch provides $\mathcal{A}$ with $(inpt^*_{\tilde{\mathcal{C}}}, pp_{\tilde{\mathcal{C}}})$, where $pp_{\tilde{\mathcal{C}}} = inpt^*_{\tilde{\mathcal{C}}} = (id\_inpt_{\tilde{\mathcal{C}}}, X, Y)$.
According to Step 8 in Section 5.3, $\mathcal{A}$ provides Ch with a bit $\tilde{e}$ and a response $resp_{\tilde{\mathcal{S}}}$. By Step 9 in Section 5.3, we may assume that $\tilde{e} = 1$ (otherwise $\mathcal{A}$ loses the game).
Thus, Ch will append $(id\_inpt_{\tilde{\mathcal{C}}}, \tilde{e}, resp_{\tilde{\mathcal{S}}}, adr_{\tilde{\mathcal{C}}})$ to $\mathcal{SC}$ and run protocol $\texttt{SendTransaction}(\tilde{\mathcal{C}}(id\_inpt_{\tilde{\mathcal{C}}}, sk_{\tilde{\mathcal{C}}}, inpt_{\tilde{\mathcal{C}}}, st_{\tilde{\mathcal{C}}}, \tilde{\alpha}, adr_{\tilde{\mathcal{S}}}, adr_{\mathcal{SC}}), \mathcal{SC})$. Initially, Ch runs $\mathsf{Decide}(st_{\tilde{\mathcal{C}}}, \tilde{\alpha}, inpt_{\tilde{\mathcal{C}}}, resp_{\tilde{\mathcal{S}}})$ that we may assume it outputs $\hat{q} = 1$ (otherwise the adversary loses the game). By the description of Decide, the above implies that (i) $resp_{\tilde{\mathcal{S}}}$ can be parsed as a pair (ρ, μ), (ii) $\tilde{\alpha} = \alpha^*$, (iii) $\mu \geq \tilde{\alpha} \cdot \rho$. Subsequently, the execution of $\texttt{CCE.C.Request}$ will produce a request $req_{\tilde{\mathcal{C}}} = (id\_inpt_{\mathcal{C}_i}, , \tilde{\alpha}, X, Y, \rho)$. Next, Ch computes the amount $\tilde{\gamma} \leftarrow \mathsf{CalPremium}(state_{\mathcal{SC}}, adr_{\tilde{\mathcal{S}}}, req_{\tilde{\mathcal{C}}}) + \tilde{\alpha}$ as described in the ICE protocol and checks if there is sufficient balance. Without loss of generality, we assume that $1 \leftarrow \mathsf{CheckBudget}(req_{\tilde{\mathcal{C}}}, adr_{\tilde{\mathcal{S}}}, adr_{\mathcal{SC}}, \mathsf{balance}, state_{\mathcal{SC}}, plc_\lambda, \tilde{\gamma}, pp, L_{\mathsf{pnd}})$ (otherwise, $\mathcal{SC}$ outputs $\perp$ and $\mathcal{A}$ loses the game). At the end of this step, Ch has computed a unique transaction identifier $\tilde{tx}$ along with $\tilde{\gamma}$ and $req_{\tilde{\mathcal{C}}}$. It records T_1 at the time at which $\tilde{tx}$ is registered. Ch provides $\mathcal{A}$ with $(\tilde{tx}, req_{\tilde{\mathcal{C}}})$.
According to Step 10 in Section 5.3, $\mathcal{A}$ can send a number of $\textsc{Advance\_Clock}$ commands of its choice. Then, it provides Ch with two bits $\tilde{f}$, $\tilde{g}$. By Step 11 in Section 5.3, we may assume that $\tilde{f} = 1$ (otherwise $\mathcal{A}$ loses the game). If $\tilde{g} = 1$ and the current time T is such that $T - T_1 \leq \Delta$, then the execution of $\texttt{Withdraw}(adr_{\tilde{\mathcal{C}}}, pp, \tilde{tx}, T, T_1)$ (Step 13 in Section 5.3) will output $\hat{g} = 1$ and $\mathcal{A}$ will lose the game (Step 14 in Section 5.3). Therefore, without loss of generality, we may assume that $\tilde{g} = 0$ or $T - T_1 > \Delta$ (i.e., $\texttt{Withdraw}$ will preserve the public state as $\hat{g} = 0$ consistently with step 15 in Section 5.3) holds. Hence, in any case, Ch runs the algorithm $\texttt{Transfer}(1, 1, 0, \tilde{tx}, \tilde{\alpha}, \tilde{\gamma}, \Delta, T, T_1)$ and Step 16 in Section 5.3 will be checked. Recall that $\tilde{\gamma} = \mathsf{CalPremium}(state_{\mathcal{SC}}, adr_{\tilde{\mathcal{S}}}, req_{\tilde{\mathcal{C}}}) + \tilde{\alpha}$. Besides, we assumed that $T - T_1 > \Delta$ and $\texttt{Transfer}$ runs on input $(1, 1, 0, \tilde{tx}, \tilde{\alpha}, \tilde{\gamma}, \Delta, T, T_1)$ (implicitly $\hat{e} = \tilde{e} = 1, \hat{f} = \tilde{f} = 1, \hat{g} = 0$). So, it holds that $\texttt{Transfer}$ outputs $\hat{h} = 1$ (the check in Step 16 in Section 5.3 does not pass). Thus, $\tilde{\alpha}$ amount of coins in X will be transferred to $\tilde{\mathcal{S}}$ and Ch provides $\mathcal{A}$ with $\hat{h}$ (Step 17 in Section 5.3). Let T_2 be the time that $\texttt{Transfer}$ algorithm is completed.
According to Step 18 in Section 5.3, $\mathcal{A}$ sends a number of $\textsc{Advance\_Clock}$ commands of its choice. Upon receiving any such command, Ch also checks if $T - T_2 > \Delta$ (and no service has been received). We distinguish the following cases:

- Case 1: $T - T_2 > \Delta$ (and no service has been received). Let $\tilde{\pi}$ be the proof that $\tilde{\mathcal{S}}$ has posted in $\mathcal{SC}$ (possibly empty). In this case, Ch runs the algorithm $\texttt{GenComplaint}(sk_{\mathcal{C}_i}, \perp, \tilde{tx}, adr_{\mathcal{SC}})$ and goes to Step 22. Since $service_{s\tilde{r}v} = \perp$, $\texttt{GenComplaint}$ will run $\mathsf{VerifyExchage}(sk_{\tilde{\mathcal{C}}}, req_{\tilde{\mathcal{C}}}, \tilde{\pi})$. By the soundness of transaction statements, $\tilde{\pi}$ is invalid ($\mathcal{A}$ cannot create a valid confirmation of a transaction for a service

that did not take place), so VerifyExchage will return $\hat{z} = 0$ and $\zeta_{\tilde{c}} = \mathit{AccountStatement}$. Therefore, `GenComplaint` outputs a complaint $\mathit{complaint}_{\tilde{c}}$ that includes $(\mathit{AccountStatement}, \tilde{tx})$, so $\mathcal{A}$ cannot win w.r.t. the condition of Step 22 in Section 5.3. Since $\mathcal{A}$ has corrupted less than a fraction of t registered auditors, at least a fraction of $1-t$ registered auditors will run the algorithmic steps, described in Section 7.1. Each such auditor will check that $\tilde{tx}$ is associated with a fresh transaction for which no missing information is missing (recall that $\hat{e} = \tilde{e} = 1, \hat{f} = \tilde{f} = 1, \hat{g} = 0, \hat{h} = 1$). In addition, since VerifyExchage was honestly executed, it holds that $\mathit{AccountStatement}$ is valid while the fact that no service took place implies that $\tilde{\pi}$ is invalid. By the description in Section 7.1, each honest auditor will decide 1. Therefore, by the description of the subroutine ExtractVerdict in Figure 10 the counter *counter* will be at least $R \cdot (1-t)$, where $R > 0$ is the number of successfully registered auditors, which implies that the output of ExtractVerdict will be $\hat{r} = 0$ and the adversary will lose the game.

– Case 2: At time $T \leq T_2 + \Delta$, $\mathcal{A}$ provides Ch with a pair $(\mathit{service}_{\tilde{s}}, \tilde{\pi})$. By Step 20 in Section 5.3, we assume that $\mathit{service}_{\tilde{s}}$ does not specify a valid transfer of $\tilde{\alpha} \cdot \rho$ coins in Y to the account $\tilde{\mathcal{C}}$ (otherwise $\mathcal{A}$ loses the game). Therefore, according to Step 21, Ch will run the algorithm $\texttt{GenComplaint}(sk_{\mathcal{C}_i}, \mathit{service}_{\tilde{s}}, \tilde{tx}, adr_{\mathcal{SC}})$, for an incorrect service $\mathit{service}_{\tilde{s}}$. By the soundness of transaction statements, $\tilde{\pi}$ is invalid ($\mathcal{A}$ cannot create a valid confirmation of a transaction for an incorrect service), so VerifyExchage will return $\hat{z} = 0$ and $\zeta_{\tilde{c}} = \mathit{AccountStatement}$. The rest of the analysis is similar to Case 1. We conclude that in any case, $\mathcal{A}$ will lose the game. □

7.5 Proof of Server-Side Security

Let $\mathcal{A}$ be an adversary that participates in the Server-side security game defined in Section 5.4. After completing the **Initialization** and **Execution** phases of the game, $\mathcal{A}$ provides Ch with two challenge addresses, $adr_{\tilde{c}}$ and $adr_{\tilde{s}}$.

Proof. Without loss of generality, we may assume that:(i) there is at least one successfully registered auditor, (ii) $\mathcal{A}$ has corrupted less than t fraction of all successfully registered auditors, (iii) $\tilde{\mathcal{S}}$ is an honest registered server that has generated its key pair, and (iv) $\tilde{\mathcal{C}}$ is corrupted. Indeed, if any of these conditions do not hold, then the adversary would lose the game in one of Steps 1–4 (see Section 5.4). Therefore, upon receiving the triple $(\mathit{inpt}^*_{\tilde{c}}, \hat{a}, pp_{\tilde{c}})$ from $\mathcal{A}$ (Step 5 in Section 5.4), Ch will execute Step 6 in Section 5.4, which invokes $\texttt{S.Init}(\mathit{inpt}^*_{\tilde{c}}, adr_{\tilde{c}}, sk_{\tilde{s}}, pk_{\tilde{s}}, st_{\tilde{s}}, adr_{\mathcal{SC}})$.

By the description of `S.Init`, $\mathit{inpt}^*_{\tilde{c}}$ is parsed as $(\mathit{id\_inpt}_{\tilde{c}}, X, Y)$ and its validity is checked by calling $\texttt{CCE.S.Init}((X,Y), adr_{\tilde{c}}, st_{\tilde{s}})$. By the description of `CCE.S.Init`, if the exchange from X to Y is not supported by $\tilde{\mathcal{S}}$, then $\hat{e}$ will be set to 0 and $\mathcal{A}$ will lose the game (Step 7 in Section 5.4). Therefore, we assume that $(X,Y) \in pp_{\tilde{\mathcal{S}}}$ and that `S.Init` will output $\hat{e} = 1$ and a response $\mathit{resp}_{\tilde{s}} = (\rho, \mu)$ (i.e., Step 8 in Section 5.4 will be executed). Subsequently, Ch sends $(\mathit{id\_inpt}_{\tilde{c}}, \hat{e}, \mathit{resp}_{\tilde{s}}, adr_{\tilde{c}})$ to $\mathcal{SC}$ and provides $\mathcal{A}$ with $(\mathit{id\_inpt}_{\tilde{c}}, 1)$.

By Step 9 in Section 5.4, $\mathcal{A}$ provides Ch with a request $\mathit{req}_{\tilde{c}}$ an amount of $\tilde{\gamma}$ coins. In particular, we may assume that $\mathit{req}_{\tilde{c}}$ is parsed as a tuple of the form $(\mathit{id\_inpt}'_{\tilde{c}}, \tilde{\alpha}, C_\ell, C_{\ell'}, \rho')$, where $\tilde{\alpha}$ is an amount, C_ℓ, $C_{\ell'}$ are identifiers of currencies, ρ' is a rate, and $\mathit{id\_inpt}'_{\tilde{c}}$ is some input identifier. Otherwise, the CheckBudget algorithm would output 0 and $\mathcal{A}$ would lose the game according to Step 10 in Section 5.4. We assume that $\mathsf{CheckBudget}(\mathit{req}_{\tilde{c}}, adr_{\tilde{c}}, adr_{\mathcal{SC}}, \mathsf{balance}, \mathit{data}_{\mathcal{SC}}, plc_\lambda, \tilde{\gamma}, pp, L_{\mathsf{pnd}})$ outputs $\hat{w} = 1$. So, Ch provides $\mathcal{A}$ with a unique transaction identifier $\tilde{tx}$. Let T_1 be the time that $\tilde{tx}$ was registered.

Upon receiving an input $\mathit{inpt}_{\tilde{c}}$ from $\mathcal{A}$ (Step 11 in Section 5.4), Ch runs $\texttt{VerRequest}(adr_{\mathcal{SC}}, \tilde{tx}, \mathit{inpt}_{\tilde{c}})$ (Step 12 in Section 5.4). According to the description of `VerRequest`, we assume that $\mathit{inpt}_{\tilde{c}}$ and $\mathit{req}_{\tilde{c}}$ are consistent; namely, $\mathit{req}_{\tilde{c}}$ is parsed as $(\mathit{id\_inpt}_{\tilde{c}}, \tilde{\alpha}, X, Y, \rho')$ and $\mathit{inpt}_{\tilde{c}}$ as $(\tilde{\alpha}, X, Y)$. Also, since `VerRequest` invokes $\texttt{CCE.S.VerRequest}((\alpha, X, Y, \rho'), adr_{\tilde{c}}, st_{\tilde{s}})$, it should hold that $\rho' = \rho$. Indeed, if the above relation does not hold, then `VerRequest` will output 0 and $\mathcal{A}$ will lose the game (Step 13 in Section 5.4). For the same reason, we assume that the amount $\tilde{\alpha}$ will be such that $\mu \geq \tilde{\alpha} \cdot \rho$. So, `VerRequest` will output $\hat{f} = 1$ while $(\mathit{inpt}_{\tilde{c}}, \mathit{req}_{\tilde{c}}, pp)$ is a valid input of the service function $\mathtt{F}_{\tilde{s}}$) which means that $\mathcal{A}$ will not win the game w.r.t. the condition of Step 14 in Section 5.4. Moreover, Ch sends 1 to $\mathcal{SC}$.

According to Step 15 in Section 5.4, $\mathcal{A}$ can send a number of ADVANCE_CLOCK commands of its choice. Then, it will provide Ch with a bit $\tilde{g}$. We will show that for every value of $\tilde{g}$, Ch will run the algorithm $\texttt{Transfer}(1, 1, 0, \tilde{tx}, \tilde{\alpha}, \tilde{\gamma}, \Delta, T, T_1)$ (where T is the current time) or $\mathcal{A}$ will lose the game.

– If $\tilde{g} = 0$, then this is straightforward for the case (Step 16 in Section 5.4).
– If $\tilde{g} = 1$, Ch executes the algorithm $\texttt{Withdraw}(adr_{\tilde{c}}, pp, \tilde{tx}, T, T_1)$ (Step 17 in Section 5.4). By the description of `Withdraw`, we assume that $T - T_1 > \Delta$ otherwise `Withdraw` will output 1 and $\mathcal{A}$ will lose the game (Step 18 in Section 5.4). Besides, if $T - T_1 > \Delta$, then `Withdraw` will output 0 and, by Step 19 in Section 5.4, Ch will run the algorithm $\texttt{Transfer}(1, 1, 0, \tilde{tx}, \tilde{\alpha}, \tilde{\gamma}, \Delta, T, T_1)$.

By the description of `Transfer`, even if $\tilde{g} = 0$, we assume that $T - T_1 > \Delta$, else `Transfer` outputs $\perp$ and $\mathcal{A}$ loses the game (Step 20 in Section 5.4). Recall that implicitly, the algorithm runs on inputs $\hat{e} = 1$, $\hat{f} = 1$, and $\hat{g} = 0$. Therefore, in order for `Transfer` to output $\hat{h} = 1$, it is necessary and sufficient that $\tilde{\gamma} \geq \mathsf{CalPremium}(state_{\mathcal{SC}}, adr_{\tilde{\mathcal{S}}}, req_{\tilde{c}}) + \tilde{\alpha}$. The latter implies the transfer of $\tilde{\alpha}$ amount of coins in X to $\tilde{\mathcal{S}}$ and the execution of $\texttt{Serve}(sk_{\tilde{\mathcal{S}}}, adr_{\mathcal{SC}}, \tilde{tx})$ by Ch (Step 21 in Section 5.4).
By executing `Serve` on behalf of $\tilde{\mathcal{S}}$, Ch (i) completes $service_{\tilde{\mathcal{S}}}$, i.e., the transfer of $\tilde{\alpha} \cdot \rho$ amount of currency in Y and (ii) sends $\tilde{\pi}$ to $\mathcal{SC}$. Then, it provides $\mathcal{A}$ with $service_{\tilde{\mathcal{S}}}$ that, in turn, can send a number of ADVANCE_CLOCK commands of its choice and provides Ch with a complaint $complaint_{\tilde{c}}$ (Step 22 in Section 5.4).
Without loss of generality, we assume that $complaint_{\tilde{c}} \neq \perp$, otherwise $\mathcal{A}$ would lose the game (Step 23 in Section 5.4). Therefore, $\texttt{Reimburse}\Big(\langle \mathcal{D}_1(adr_{\mathcal{SC}}, sk_{\mathcal{D}_1}), \ldots, \mathcal{D}_m(adr_{\mathcal{SC}}, sk_{\mathcal{D}_m}),\ \mathcal{SC}(plc, \tilde{\alpha}, \tilde{\gamma}, complaint_{\tilde{c}}, pp)\rangle\Big)$ will be executed. Recall that a fraction of at least $1 - t$ registered auditors are honest, so they follow the algorithm in Section 7.1. By the description of this algorithm,

– If $complaint_{\tilde{c}}$ specifies an identifier tx that is either previously recorded in the auditors' states, or is invalid then the auditor will decide as 0.
– Note that the only new and valid identifier is $\tilde{tx}$. If $complaint_{\tilde{c}}$ specifies $\tilde{tx}$, then
  • If $\mathcal{A}$ has advanced the clock such that $T - T_1 > \Theta$ (insurance period has expired), then the auditor will decide as 0.
  • Otherwise, if $T - T_1 > \Theta$, then since $\tilde{\pi}$ is valid statement confirming the transaction, the auditor will decide 0.

We conclude that in any case, all honest registered auditors will decide 0. Therefore, less than a fraction of t registered auditors (all corrupted ones) can contribute a decision bit equal to 1. Given that $t \leq \frac{1}{2}$, we have that $t \leq 1 - t$ which implies that less than a fraction of $1 - t$ registered auditors can contribute a decision bit equal to 1. By the description of $\mathsf{ExtractVerdict}$ in Figure 10, the counter will never reach the critical value that will set the final verdict to $\hat{r} = 1$. Hence, $\hat{r}$ will be equal to 0 and the adversary will lose the game. □

8 Further Discussions

8.1 Fair Exchange Versus ICT

Fair exchange [65,2,20] is an interesting problem in which two mutually distrustful parties want to swap digital items such that neither party can cheat the other, in the sense that either each party gets the other's item, or neither party does. Solutions to the problem are often certain cryptographic schemes, called fair exchange protocols (or fair exchange for short). Fair exchange protocols can be used to deal with a variant of *Authorized Push Payment* (APP) fraud, called purchase fraud, where a service provider may wish to receive a certain amount of coin without delivering the service. We refer readers to [85] for further discussion about APP fraud.
Thus, fair exchange could be seen as closely related to the concept of insured cryptocurrency transactions. In this section, we provide a detailed comparison between these two concepts from various perspectives, highlighting their distinct characteristics and implications.

– **Timing of Transactions.** The timing of transactions plays a crucial role in differentiating fair exchange protocols from insured cryptocurrency transactions. In fair exchange protocols, neither party receives the counterparty's item unless both parties have fulfilled their obligations. This ensures an atomic and

fair exchange. In contrast, insured cryptocurrency transactions follow a different timing structure. For example, in financial transactions, once one party sends funds, there is no immediate requirement for the receiving party to provide a service or return an asset to the sender. Instead, the sender relies on insurance mechanisms to guarantee the return of funds if necessary, offering security without the need for an immediate reciprocal transaction.

- **Risk Involved.** Another key difference lies in the level and nature of risk associated with each type of transaction. In fair exchange protocols, the risk is minimal, as the exchange is atomic, ensuring that neither party can gain an advantage over the other. This built-in fairness limits the scope for risk. On the other hand, insured cryptocurrency transactions inherently involve risk. The value transfers between the involved parties do not always occur simultaneously or atomically. As a result, the party initiating the transfer depends on insurance mechanisms to mitigate potential risks, including malicious behaviors, such as failure to send the promised asset.
- **Coverage and Protection.** In fair exchange protocols, the coverage is generally limited; it only protects against the completion of unfair transactions and does not account for external risks, such as third-party fraud[12] or post-transaction losses[13]. In contrast, insured cryptocurrency transactions offer broader protection. This protection extends beyond the transaction conditions, encompassing risks such as third-party fraud, service failures, and losses not directly associated with the transaction execution itself.
- **Dispute Resolution.** Fair exchange protocols generally require minimal dispute resolution. Since the transaction is cryptographically enforced, both parties are automatically ensured of fulfilling their obligations without requiring intermediary intervention. On the other hand, insured cryptocurrency transactions involve a mechanism for claim evaluation and dispute resolution. This process is crucial to ensure that insurance claims are handled fairly, particularly when disagreements arise, such as when one party claims that the terms were not met or that a loss occurred due to factors beyond the transaction's execution.
- **Post-Transaction Processes.** In fair exchange protocols, once the exchange is completed, the transaction is considered final. No further actions are typically required, as the cryptographic enforcement ensures that both parties have met their obligations. Conversely, insured cryptocurrency transactions often involve post-transaction processes. These may include preparing and submitting claims, undergoing evaluation, and processing payouts. The insurance mechanism may require additional steps to resolve issues or provide compensation, depending on the terms of the insurance agreement and the specific circumstances surrounding the transaction.
- **Economic Impact.** The economic impact of fair exchange protocols is generally low to moderate due to their limited coverage. The scope of protection in fair exchange is confined to ensuring fairness in the transaction itself, without significantly affecting broader economic activities. In contrast, insured cryptocurrency transactions have a moderate to high economic impact. Their broader scope of protection promotes innovation, job creation, and business growth within the cryptocurrency ecosystem. By relying on insurance mechanisms, entities involved in these transactions contribute to greater participation and investment.
- **Fraud Prevention and Protection.** Both fair exchange protocols and insured cryptocurrency transactions offer measures to deal with fraud, but the extent of their protection differs. Fair exchange protocols *prevent* certain types of fraud, such as cryptocurrency purchase fraud [68]. However, insured cryptocurrency transactions provide a more comprehensive range of fraud *protection*. They cover victims of various types of fraud, including purchase fraud, investment fraud, cryptocurrency theft, and money exchange fraud. This extended coverage helps mitigate a wider range of fraudulent activities.

8.2 ICT Versus Traditional Insurance

ICT and accordingly ICE offer various advantages over traditional insurance companies, such as:

[12] For instance, situations where an external entity, not directly involved in the transaction, deceives one or both parties. For example, a third party may impersonate one of the counterparties and defraud the other.

[13] For instance, scenarios where one party suffers a loss after the transaction is completed, such as when a cryptocurrency wallet is hacked after the exchange.

– *Decentralized Dispute Resolution*: Disputes are resolved by a committee of external auditors who are selected based on explicit, publicly accessible policies. This approach mimics the real-world jury system and can enhance overall fairness, transparency, and trust in the system. In contrast to traditional insurance companies, where disputes are resolved either internally or by independent arbitrators or mediation firms [19] paid by the insurance companies, this method ensures greater impartiality. External auditors in the decentralized system who are selected according to explicit terms are less likely to be influenced by financial relationships or internal biases, thus providing a more trustworthy resolution process.
– *Real-Time Access and Transparency*: Clients and service providers can access transaction data in real time. This means they have immediate visibility into the status of their interactions and any insurance claims, allowing them to monitor progress and detect any issues promptly. In contrast, when dealing with a traditional insurance company, clients often receive periodic updates on the status of their transactions and claims, which can lead to delays in getting important information [48]. Insurance companies have a limited capacity to process claims, which may lead to a backlog and the claimant may need to contact the insurance company to request updates. The processes and criteria used by insurance companies to handle claims and transactions are often lengthy from the perspective of the claimant as insurers need to ensure that a claim is not fraudulent. Further, whilst most regulated insurance companies will be required to adhere to a declared standard of fairness, insurance buyers may have a perception that the claims process is subjective and not fully transparent [74].
– *Fixed Policy and Terms*: Once deployed, the insurance policy and terms encoded in a smart contract cannot be changed. This ensures that all parties are bound by the original agreement, providing certainty and clarity. However, traditional insurance policies can be modified by the insurance company, often through policy amendments or updates [5]. While these changes are typically communicated to policyholders, they can create uncertainty.
– *Minimized Human Intervention*: Automation through smart contracts can reduce the need for human intervention, minimizing the potential for errors, biases, or fraud. In contrast, traditional insurance processes rely heavily on human intervention for tasks such as policy management and claims handling. This reliance increases the likelihood of errors. The manual nature of these processes may lead to inconsistencies and delays [42], undermining the efficiency and reliability of the insurance system.

8.3 ICT as an Insurance Paradigm for Cryptocurrency Transactions

We briefly sketch out the relevance of the ICT protocol to conventional commercial insurance. There are many different types of insurance and markets for risk transfer.[14] The simple principle of insurance is that a customer makes a payment to an insurance company, known as a premium, in return for indemnification against a specified (usually financial) risk. It is common to divide insurance into personal and commercial lines. Personal lines are areas such as car, home, or travel insurance. Commercial lines provide cover for companies against specified perils. The most relevant application of this work would be insurance against financial liability arising from fraud [7,21].

It is unlikely that consumers would want to buy insurance from a third-party provider to cover the risk of fraudulent transactions. The most likely implementation of insurance is that the provider of transactional services (e.g., our ICT) agrees to indemnify the user against fraud. This poses an accumulation risk to the insurer in the event of widespread fraudulent activity. The insurer may elect to purchase insurance to cover the potential costs (direct, legal, or indirect) of fraudulent activity in excess of a certain amount. In commercial contracts, it is usual that the insured retains some level of losses, which is known as a deductible or a self-insured retention (SIR). It should be noted that within the realm of cybersecurity, an advantage of insurance is that it provides the insured with access to remedial services as part of the policy (see, for example, [9] and associated references). There is a parallel with ICT, in that those wishing to develop business ventures may gain higher trust if partnering with an insurance operator deploying ICT.

For insurance markets to function, there needs to be some level of confidence in the process of handling claims [69,4,25]. Our ICT provides a robust algorithmic framework for dealing with claims and fraud.

[14] For suitable introductory references see, among many, [71,73].

Often it is the case that a single insurance company insures multiple large corporations to spread the risk (diversification), as it is done in our ICT which provides coverage for different users against various servers. Alternatively, the coverage limits might be allocated between different insurers. For example, consider the scenario under which a user wants coverage of $30mn against fraud but the maximum line size available from any single insurer is $10mn. In this case, multiple insurers may collaborate to provide this coverage. In this case, the insurer bearing the first $10mn of losses is known as the primary carrier, the insurer bearing the next $10mn as the first excess and the insurer bearing the final $10mn as the second excess. In the commercial insurance markets, this is known as an insurance tower [6], which can be applied to ICT too.
Finally, it should be noted that insurance companies may have greater legal resources at their disposal to potentially recover misappropriated funds than individuals. The process whereby insurers attempt to recover the costs of claims from third parties is called subrogation [47]. Recovering cryptocurrency assets is not always straightforward [53] yet there are anecdotal examples of successful recovery of ransomware payments, for example [77].

8.4 Claim Accumulation Risk

A key concern for an organization providing insurance coverage is for a catastrophic accumulation of claims that in a worst-case scenario results in the insurance company becoming insolvent (this is known as a ruin in the insurance literature). With a new protocol or insurance application where no historical data on claims exists, insurance pricing is difficult to achieve using conventional actuarial methods. Accordingly, insurers either solve the issue of catastrophic claims using reinsurance products or by adopting a test-the-water approach and granting only small limits in order to build claims data. Importantly, ICT is compatible with both of these strategies. In relation to reinsurance, it is possible that services that wish to build their customer base are willing to supply or pool capital to fund the insurance provided by ICT.
In terms of product development and market construction, the robustness and transparency of ICT reduce the potential for disputes to emerge during compiling claims — where the purchaser and provider of insurance disagree about the amount that should be paid on the claim. In conventional insurance, there is a difference in claims data between claims allegedly incurred by an insured and those paid after loss adjustment. ICT helps to mitigate that uncertainty. In essence, ICT is an algorithmic loss adjuster, which reduces the risk of perceived subjectivity in claims handling.

8.5 Relation of ICT to Economic Theory on Insurance Pricing

The classical economic formulation of insurance usually begins with an analysis of the utility function of the insurance buyer, see [78] for an extensive discussion. One starts with the basic concept of a utility function, $\mathsf{Utility}$, which must be continuous and twice differentiable. This represents the preferences of either insurance buyer or seller in respect of losses and is often referred to as risk tolerance [46].
Commonly used utility functions[15] exhibit specific properties, such as constant absolute risk aversion (CARA), such that the coefficient of absolute risk aversion $A(x) = -\frac{u''(x)}{u'(x)} = \bar{\alpha}$, or constant relative risk aversion (CRRA), where the coefficient of relative risk aversion $R(x) = -\frac{x \cdot u''(x)}{u'(x)} = \bar{\gamma}$. CARA utility functions are of the form $u(x) = 1 - \bar{\alpha} \cdot e^{-\bar{\alpha} \cdot x}$ and CRRA of the form $u(x) = \frac{x^{1-\bar{\gamma}}-1}{1-\bar{\gamma}}$ for $\bar{\gamma} \neq 1$ and $u(x) = ln(x)$ for $\bar{\gamma} = 1$. The utility function of each individual client, $\mathcal{C}$, can be defined as:

$$U_c = \bar{\pi}_c \cdot \mathsf{Utility}_c(W_c - \mu - l + ai) + (1 - \bar{\pi}_c) \cdot \mathsf{Utility}_c(W_c - \mu) \tag{1}$$

where U_c is the total utility of the client, $\mathsf{Utility}_c$ is the client utility function, W_c is the wealth of the client, l is the value of loss, $\bar{\pi}_c$ is the subjective probability of covered loss believed by client C, and μ is the amount of premium paid for an amount of cover ai. One possible formulation of the decision problem for the insurance seller may be framed in terms of wealth — the following discussion is abridged from [79]:

[15] These are very well-established and standard results and, accordingly, a detailed discussion is outside the scope of this paper. [46] is recommended as a good introductory text.

$$W = W_0 + \bar{\mu} - \int_0^A X \cdot dF(X) \tag{2}$$

In this case, W_0 is the initial capital of the insurance operator, $\bar{\mu}$ are the premiums received from clients purchasing insurance — the sum of μ in Equation 1, X is the total claims paid out over a period, and $F(X)$ is the subjective probability distribution (of the insurance operator) of such claims. Finally, A is the amount of claims at which both capital and premium income is fully depleted and the insurance operator faces insolvency. The problem for the insurance operator is to balance premium income with the probability of loss at a premium rate which is utility-enhancing for the insurance buyer relative to their baseline of not insuring against loss. The function CheckBudget in ICT optimizes Equation 2 in real time. Within the ICT protocol, the worst possible outcome is considered, i.e., the protocol assumes full funding for all claims. In insurance, this is rarely the case; insurers will be capitalized to be able to pay expected claims up to a certain probability threshold — for example, 99.5% in the Solvency 2 framework [27]. Over time, if losses prove to be infrequent, it might be possible to relax the capital restrictions of CheckBudget to a similar threshold.

9 Cost Analysis of ICE

9.1 Asymptotic Cost Analysis

In this section, we present a detailed evaluation of ICE's costs. Table 1 summarizes the complexities of different parties in ICE.

Table 1: Complexities of different parties in ICE.

Party	Computation Cost	Communication Cost
Operator $\mathcal{O}$	$O(\|db_{\mathcal{D}}\|+\|db_{\mathcal{S}}\|+\|data_{\mathcal{SC}}\|+k+m)$	$O(k+m)$
Client $\mathcal{C}_i$	$O(\|db_{\mathcal{D}}\|+\|db_{\mathcal{S}}\|+\|data_{\mathcal{SC}}\|)$	$O(1)$
Server $\mathcal{S}_j$	$O(1)$	$O(1)$
Smart contract $\mathcal{SC}$	$O(m)$	$O(1)$
Auditor $\mathcal{D}_i$	$O(1)$	$O(1)$
Total	$O(\|db_{\mathcal{D}}\|+\|db_{\mathcal{S}}\|+\|data_{\mathcal{SC}}\|+k+m)$	$O(k+m)$

Cost of Operator $\mathcal{O}$. The computation cost of $\mathcal{O}$ in Phase 1 is constant $O(1)$, with respect to the number of parties, as it develops a smart contract with a few functions. However, the dominant cost in this phase is $O(|db_{\mathcal{D}}|+|db_{\mathcal{S}}|+|data_{\mathcal{SC}}|)$ stemming from the invocation of Update.State. The complexity of $\mathcal{O}$ in Phase 3 is $O(k+m)$ as it needs to invoke $\mathsf{Validate}^{(\mathcal{O})}$ linearly with the number of servers and auditors. Thus, the total computation complexity of $\mathcal{O}$ is $O(|db_{\mathcal{D}}|+|db_{\mathcal{S}}|+|data_{\mathcal{SC}}|+k+m)$. The communication cost of $\mathcal{O}$ in Phase 1 is $O(1)$, with respect to the total number of parties involved. The communication cost of $\mathcal{O}$ in Phase 3 is $O(k+m)$.

Cost of Client $\mathcal{C}_i$. Overall, Phase 6 imposes a negligible overhead to $\mathcal{C}_i$ because it primarily involves (i) `CCE.C.Request` that itself relies on predicate Decide consisting of a few basic arithmetic and comparison operations, and (ii) CalPremium which also involves a few arithmetic operations. The cost of $\mathcal{C}_i$ in Phase 11 is dominated by the overhead of executing VerifyExchage. However, this cost is negligible as it mainly requires

$\mathcal{C}_i$ to log in to its account and read its most recent transactions. Hence, $\mathcal{C}_i$'s overall computation complexity is $O(|db_{\mathcal{D}}|+|db_{\mathcal{S}}|+ |data_{\mathcal{SC}}|)$.
Next, we estimate the communication cost of $\mathcal{C}_i$. In Phase 4, its cost is $O(1)$ with respect to the number of servers and auditors, as it sends only three messages (where the size of each message is at most 128-bit) to $\mathcal{S}_j$ and two messages to $\mathcal{SC}$. In Phase 6, step 6(a)v, it sends six messages to $\mathcal{SC}$ while in step 6c, it sends at most three messages to $\mathcal{S}_j$. In Phase 11, $\mathcal{C}_i$ sends at most two messages to $\mathcal{SC}$ to register its complaint. Therefore, the overall communication complexity of $\mathcal{C}_i$ is $O(1)$.

Cost of Server $\mathcal{S}_j$. The computation complexity of a server $\mathcal{S}_j$ in Phase 5 is low and $O(1)$ as it mainly involves reading its internal state, given a value. Its cost in Phase 7 is also low because it involves a few arithmetic operations as a result of executing `CCE.S.VerRequest`. The primary computation cost of $\mathcal{S}_j$ in Phase 10 stems from the invocation of $\mathsf{F}_{\mathcal{S}_j}$ and UpdateRate in step 10b. The computation cost of executing $\mathsf{F}_{\mathcal{S}_j}$ is low as it involves transferring a certain amount of coin/money to the client. Similarly, UpdateRate mainly involves updating its record of rates that requires a few arithmetic operations. Hence, the computation complexity of $\mathcal{S}_j$, for each client, is $O(1)$ with respect to the number of auditors.
We proceed to estimate the communication cost of $\mathcal{S}_j$. In Phase 3, it sends a constant number of messages to $\mathcal{O}$, where the size of each message is at most 256 bits. In Phase 5, it sends only five messages to $\mathcal{SC}$, where the size of each message is at most 256 bits. In Phase 7, it sends only a binary value to $\mathcal{SC}$. In Phase 10, it sends out a constant number of messages as a result of delivering the service. In the same phase, it transmits a single message to $\mathcal{SC}$. Thus, the communication complexity of $\mathcal{S}_j$ is $O(1)$.

Cost of the Smart Contract $\mathcal{SC}$. Overall, the computation overhead of $\mathcal{SC}$ is low, as the main operations that $\mathcal{SC}$ performs involve basic arithmetic operations, in Phases 6, 8, 9, and 12. However, unlike other parties in the protocol, the computation complexity of $\mathcal{SC}$ is linear with the number of auditors, $O(m)$. Next, we estimate its communication cost. In each of Phases 6 and 9, it transmits at most two messages. While in each of Phases 8 and 12, it transmits a single transaction. Thus, the party that involves $\mathcal{SC}$ will have at most $O(1)$ communication cost.

Cost of an Auditor $\mathcal{D}_i$. The computation complexity of $\mathcal{D}_i$ is $O(1)$, as in Phase 12, for each client, it processes the client's claim and computes a verdict. Its communication complexity is also $O(1)$ because in Phase 3 it sends a single message and in Phase 12 it sends two messages to $\mathcal{SC}$.

9.2 Concrete Gas Consumption in ICE

Experimental Setup. We have implemented [1] the smart contract $\mathcal{SC}$ of ICE in Solidity and estimated the gas consumption of its main functions. In the implementation, for the sake of simplicity, we assumed that the server always accepts (i.e., `S.Init` returns 1) the request a client sends to it (as a result of `C.Init` invocation). The implementation does not rely on any external library. In our experiment, we report the gas consumption of each function as estimated by the Remix IDE in "Gwei" during the execution of that function. To convert the gas consumption from Gwei (where each Gwei is approximately 10^{-9} ether) to US Dollars (USD), we used the price of each "ether" which, at the time of writing, was approximately 3183.44 USD according to Coinbase, a cryptocurrency trading website [22].
In the experiment, we have allocated seven different addresses to different auditors an allocated to each client, server and operator an address. The experiment was repeated an average of 10 times. We have implemented a version of $\mathsf{CalPremium}(data_{\mathcal{SC}}, adr_{\mathcal{S}}, req_c) \rightarrow \mu$ that computes the output as:

$$\mu = \alpha \times adjustedRiskFactor \times adjustedDuration \times coverage$$

where

$$adjustedRiskFactor = riskFactor \times 30\%$$

$$adjustedDuration = \Theta \times 30\%$$

$riskFactor$ is an integer indicating a risk factor of the server, $coverage$ is the percentage of the transaction amount α covered by the policy plc_λ already stored in $\mathcal{SC}$, $(riskFactor, \Theta, coverage) \in \mathcal{SC}$, and $\alpha \in req_c$. We multiplied $riskFactor$ and Θ by a rate (i.e., 30%) to normalize the original values and ensure they impact the premium amount in a controlled way. Without this scaling, the premium could grow fast with higher $riskFactor$ and Θ, even surpassing the transaction amount. The choice of 30% is based on a set of empirical observations. Depending on specific use cases, a different value may be adopted. We have also implemented a simple version of $\mathsf{CompRemAmount}(plc_\lambda, \gamma, req_c, pp) \rightarrow amount$ that generates an output as: $amount = \alpha \times coverage$.

Result. Table 2 presents the gas consumption and corresponding monetary cost (in USD) for deploying and executing the smart contract on the Ethereum test network. The listed costs reflect the amount of gas required by each function and its equivalent value in US dollars. As illustrated in Figure 13, there is a notable variation in gas usage across different smart contract operations.

Table 2: Gas consumption of primary functions of $\mathcal{SC}$ in ICE. Each Gwei is 10^{-9} ether.

Operation	Gas Consumption	
	Gwei	USD
Smart contract deployment	2,828,379	7.9
`Register`	78,763	0.2
`SendTransaction`	334,370	0.9
`VerRequest`	66,910	0.1
`Withdraw`	59,047	0.1
`Transfer`	71,670	0.2
`GenComplaint`	63,889	0.1
`Reimburse`	180,843	0.5
Total	3,683,871	10

The deployment of the smart contract incurs the highest gas cost, amounting to 2,828,379 Gwei (7.9 USD). This result is expected, as smart contract deployment is typically one of the most resource-intensive phases, as discussed in [29]. This suggests that optimizing this process could lead to considerable cost savings. Future efforts might focus on optimizing deployment by simplifying initialization procedures or using libraries.
Among the functions, `SendTransaction` consumes the highest amount of gas, 334,370 Gwei (0.9 USD), due to several factors: (i) multiple checks to validate conditions like Ether sufficiency, server registration, and contract budget, (ii) creation and storage of a multi-field transaction structure, and (iii) invocation of a hash function to generate a transaction ID. The other functions such as `Reimburse`, `Transfer`, and `GenComplaint` show lower gas consumption, ranging from 0.5 to 0.1 Gwei.

9.3 Transaction Latency

The transaction latency in our schemes primarily depends on the consensus protocol employed by the underlying blockchain. In Ethereum, block mining requires an average of 12 seconds per block [33]. However, to mitigate the risk of forks and ensure a high probability that a block remains part of the canonical chain, it is customary to wait for six subsequent blocks, resulting in an effective latency of approximately 72 seconds.

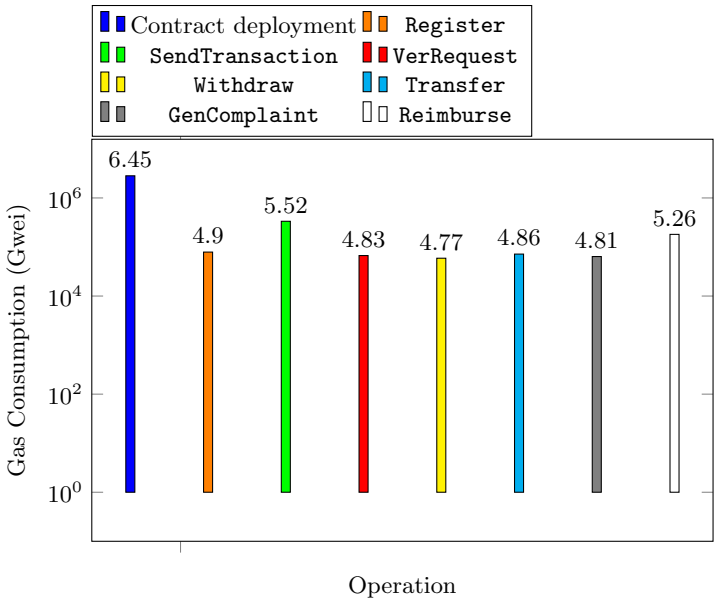

Fig. 13: Gas consumption (in Gwei) for $\mathcal{SC}$'s operations (logarithmic scale) in ICE.

By comparison, Byzantine Fault Tolerant (BFT) blockchains, such as Hyperledger Fabric, adopt a different approach by avoiding proof-of-work mining altogether. Instead, they leverage deterministic consensus protocols that enable faster finality. For example, Hyperledger Fabric can achieve transaction confirmation in around 35 seconds when operating with 20 nodes [50]. This significant reduction in latency makes BFT-based blockchains particularly attractive for applications requiring quick transaction finality.

10 Conclusion and Future Works

In this work, we introduced Insured Cryptocurrency Transactions (ICT), a novel decentralized insurance framework designed to safeguard users against fraud within the cryptocurrency ecosystem. By formalizing ICT and developing the Insured Cryptocurrency Exchange (ICE), we provided both a rigorous theoretical foundation and a practical demonstration of how decentralized insurance can enhance user protection in the context of centralized exchanges. Our implementation, accompanied by a cost analysis, demonstrates that such frameworks can offer robust security guarantees with minimal on-chain overhead, positioning them as practical solutions for contemporary financial technologies. These solutions not only mitigate financial losses but also have the potential to improve market stability and trust within digital currency ecosystems.
As future work, the integration of ICT within decentralized exchanges (DEXs) could be explored, along with the development of automated policy adaptation mechanisms, such as dynamic premium rates, coverage limits, and reimbursement criteria, using machine learning models for real-time risk assessment and adaptive decision-making.

Acknowledgments

Aydin Abadi was supported in part by UKRI grant: EP/V011189/1. We are grateful to Steven J. Murdoch for the initial conversations that led to this research.

References

1. Abadi, A.: Source code of ICE (2025), https://github.com/AydinAbadi/ICT/blob/main/Source_code.sol
2. Abadi, A., Murdoch, S.J., Zacharias, T.: Recurring contingent service payment. In: EuroS&P (2023)
3. Agarwal, R., Barve, S., Shukla, S.K.: Detecting malicious accounts in permissionless blockchains using temporal graph properties. Appl. Netw. Sci. (2021)
4. Ahmed, D., Xie, Y., Issam, K.: Investor confidence and life insurance demand: can economic condition limit life insurance business? International Journal of Emerging Markets (2021)
5. Alexander, N.: How to respond to policy changes. Journal of Accountancy (2002)
6. Allianz: What is D&O insurance? learn more about directors & officers insurance (2022), https://commercial.allianz.com/news-and-insights/expert-risk-articles/d-o-insurance-explained.html
7. American Insurance Group: (2024), https://www.aig.co.uk/home/risk-solutions/business/financial-lines#accordion-fea384af66-item-bf35d7eba9
8. Aon: Aon is in the business of better decisions (2024), https://www.aon.com/en
9. Arce, D., Woods, D.W., Böhme, R.: Economics of incident response panels in cyber insurance. Computers & Security (2024)
10. Ateniese, G., Magri, B., Venturi, D., Andrade, E.R.: Redactable blockchain - or - rewriting history in bitcoin and friends. In: EuroS&P (2017)
11. Aventus: Comprehensive & end-to-end blockchain-as-a-service for enterprises (Accessed in 2024), https://www.aventus.io
12. Avizheh, S., Nabi, M., Safavi-Naini, R., K., M.V.: Verifiable computation using smart contracts. In: CCSW (2019)
13. Bellare, M., Micciancio, D., Warinschi, B.: Foundations of group signatures: Formal definitions, simplified requirements, and a construction based on general assumptions. In: EUROCRYPT (2003)
14. Bellare, M., Shi, H., Zhang, C.: Foundations of group signatures: The case of dynamic groups. In: RSA (2005)
15. Bindseil, U.: Tiered cbdc and the financial system (2020), https://www.ecb.europa.eu/pub/pdf/scpwps/ecb.wp2351~c8c18bbd60.en.pdf
16. BIS: Bank of canada, european central bank, bank of japan, sveriges riksbank, swiss national bank, bank of england, board of governors of the federal reserve, and bank for international settlements. central bank digital currencies: foundational principles and core features, (2020), https://www.bis.org/publ/othp33.htm
17. BitGo: Insurance (2024), https://www.bitgo.uk/resources/insurance
18. Boneh, D., Boyen, X., Shacham, H.: Short group signatures. In: CRYPTO (2004)
19. Bourova, E., Ramsay, I., Ali, P.: Cause to complain? consumer experiences of internal and external dispute resolution in the context of general insurance. Consumer Experiences of Internal and External Dispute Resolution in the Context of General Insurance. Australasian Dispute Resolution Journal (2020)
20. Campanelli, M., Gennaro, R., Goldfeder, S., Nizzardo, L.: Zero-knowledge contingent payments revisited: Attacks and payments for services. In: CCS (2017)
21. Chubb: Crime insurance for financial institutions (2024), https://www.chubb.com/uk-en/business/products/crime-insurance-financial-company.html
22. Coinbase: coinbase (2024), https://www.coinbase.com/en-gb/converter/eth/usd
23. CoinCodex: Crypto trading volume tracker (2024), https://coincodex.com/trading-volume/
24. Coincover: Coincover (2024), https://www.coincover.com
25. Cox, K.F.: Insurance bad faith refusal to settle: What's an insured to do. JL & Com. (1987)
26. Deb, S., Raynor, R., Kannan, S.: STAKESURE: proof of stake mechanisms with strong cryptoeconomic safety. CoRR (2024)
27. Doff, R.: A critical analysis of the solvency ii proposals. The Geneva Papers on Risk and Insurance-Issues and Practice 33, 193–206 (2008)
28. Dolev, D., Dwork, C., Naor, M.: Non-malleable cryptography (extended abstract). In: STOC (1991)
29. Dong, C., Wang, Y., Aldweesh, A., McCorry, P., van Moorsel, A.: Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing. In: CCS (2017)
30. Dtravel: The world's first home sharing economy for the people, by the people (2024), https://polkastarter.com/projects/dtravel
31. Dzhondzhorov, D.: Indian student committed suicide after becoming a victim to a bitcoin scam (2023), https://cryptopotato.com/indian-student-committed-suicide-after-becoming-a-victim-to-a-bitcoin-scam-report/
32. Eigenmann, D.: Reversible token (2018), https://github.com/MediciDAO/ReversibleToken/blob/master/contracts/ReversibleToken.sol

33. Eyal, I., Gencer, A.E., Sirer, E.G., van Renesse, R.: Bitcoin-ng: A scalable blockchain protocol. In: NSDI (2016)
34. Fabusola, V.: Crypto suicides: Blood on crypto's hands? (2023), https://dailycoin.com/blood-on-cryptos-hands-real-suicide-rates-due-to-cryptocurrencies/
35. Farao, A., Panda, S., Menesidou, S.A., Veliou, E., Episkopos, N., Kalatzantonakis, G., Mohammadi, F., Georgopoulos, N., Sirivianos, M., Salamanos, N., et al.: Secondo: A platform for cybersecurity investments and cyber insurance decisions. In: TrustBus (2020)
36. Farao, A., Paparis, G., Panda, S., Panaousis, E., Zarras, A., Xenakis, C.: Inchain: a cyber insurance architecture with smart contracts and self-sovereign identity on top of blockchain. International Journal of Information Security (2024)
37. Federal Bureau of Investigation: Internet crime report (2023), https://www.ic3.gov/media/pdf/annualreport/2023_ic3report.pdf
38. Federal Trade Commision : What to know about cryptocurrency and scams (2022), https://consumer.ftc.gov/articles/what-know-about-cryptocurrency-and-scams
39. Franco, M., Berni, N., Scheid, E., Killer, C., Rodrigues, B., Stiller, B.: Saci: A blockchain-based cyber insurance approach for the deployment and management of a contract coverage. In: GECON (2021)
40. Franco, M.F., Berni, N., Scheid, E.J., Killer, C., Rodrigues, B., Stiller, B.: Saci: A blockchain-based cyber insurance approach for the deployment and management of a contract coverage. In: GECON (2021)
41. Garay, J.A., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol: Analysis and applications. In: EUROCRYPT (2015)
42. Gharaeiyan, M.: How do manual vs automated claims handling processes compare (Accessed in 2024), https://insly.com/en/blog/how-do-manual-vs-automated-claims-handling-processes-compare/
43. Go Superscript: Blockchain insurance (2024), https://gosuperscript.com/advised/blockchain-insurance
44. Goldreich, O.: The Foundations of Cryptography - Volume 2, Basic Applications. Cambridge University Press (2004)
45. Goldwasser, S., Micali, S.: Probabilistic encryption. J. Comput. Syst. Sci. (1984)
46. Gollier, C.: The economics of risk and time. MIT press (2001)
47. Greenblatt, J.A.: Insurance and subrogation: When the pie isn't big enough, who eats last? The university of Chicago law review (1997)
48. Harton, O.: Insurance company bad faith tactics and examples (2024), https://www.findlaw.com/consumer/insurance/insurance-company-bad-faith-tactics-and-examples.html
49. House of Commons Treasury Committee: Regulating crypto: Fifteenth report of session 2022–23 (2023)
50. Hyperledger Foundation: Hyperledger blockchain performance metrics (2018)
51. IBM Food Trust: About ibm food trust (Accessed in 2024), https://www.ibm.com/downloads/cas/8QABQBDR
52. Jung, E., Tilly, M.L., Gehani, A., Ge, Y.: Data mining-based ethereum fraud detection. In: Blockchain (2019)
53. Kamps, J., Trozze, A., Kleinberg, B.: Cryptocurrencies:: Boons and curses for fraud prevention. In: A Fresh Look at Fraud (2022)
54. Katz, J., Lindell, Y.: Introduction to Modern Cryptography, Second Edition. CRC Press (2014)
55. Kiayias, A., Kohlweiss, M., Sarencheh, A.: PEReDi: Privacy-enhanced, regulated and distributed central bank digital currencies. In: CCS (2022)
56. Kumar, S., Dohare, U., Kaiwartya, O., et al.: Flame: Trusted fire brigade service and insurance claim system using blockchain for enterprises. IEEE Transactions on Industrial Informatics (2022)
57. Labs, P.: Filecoin: A decentralized storage network (2017), https://filecoin.io/filecoin.pdf
58. Lepoint, T., Ciocarlie, G., Eldefrawy, K.: Blockcis—a blockchain-based cyber insurance system. In: (IC2E) (2018)
59. Lloyd's: Lloyd's (2024), https://www.lloyds.com
60. Lockton: Cryptocurrency insurance: Best practices for custodians (2024), https://global.lockton.com/gb/en/news-insights/cryptocurrency-insurance-best-practices-for-custodians
61. Loukil, F., Boukadi, K., Hussain, R., Abed, M.: Ciosy: A collaborative blockchain-based insurance system. Electronics (2021)
62. MakerDAO: The maker protocol: Makerdao's multi-collateral dai (mcd) system (2024), https://makerdao.com/en/whitepaper/#abstract
63. Manda, V.K., Yamijala, S.: Peer-to-peer lending using blockchain. International Journal Of Advance Research And Innovative Ideas In Education (2019)
64. Marsh: Marsh (2024), https://www.marsh.com/en/services/financial-professional-liability/expertise/innovative-insurance-protection-for-digital-assets.html
65. Maxwell, G.: Zero knowledge contingent payment (2011)
66. MORLEY, K.: My father attempted suicide after crypto fraudsters tricked him out of £144k (2022), https://www.telegraph.co.uk/money/katie-investigates/father-attempted-suicide-crypto-fraudsters-tricked-144k/

67. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. Tech. rep. (2019)
68. NatWest: Purchase scams (2024), https://www.natwest.com/business/security/purchase-scams.html, accessed: 2024-11-09
69. Njegomir, V.: Marketing in insurance: The importance of efficient insurance claims management. Civitas (2018)
70. OneInfinity: Wallet (2024), https://oneinfinity.global/wallet
71. Parodi, P.: Pricing in general insurance. Chapman and Hall/CRC (2023)
72. Propy: Buy and sell homes - faster, easier and more securely (Accessed in 2024), https://propy.com/browse/
73. Rees, R., Wambach, A., et al.: The microeconomics of insurance. Foundations and Trends® in Microeconomics 4(1–2), 1–163 (2008)
74. Reurink, A.: Financial fraud: A literature review. Contemporary topics in finance: A collection of literature surveys (2019)
75. Romanosky, S., Ablon, L., Kuehn, A., Jones, T.: Content analysis of cyber insurance policies: How do carriers write policies and price cyber risk? SSRN (2017)
76. Sarencheh, A., Kiayias, A., Kohlweiss, M.: PARScoin: A privacy-preserving, auditable, and regulation-friendly stablecoin. Cryptology ePrint Archive (2023)
77. Security Intelligence: (2002), https://securityintelligence.com/articles/recovering-ransomware-payment/
78. Skeoch, H.R.: Expanding the gordon-loeb model to cyber-insurance. Computers & Security (2022)
79. Skeoch, H.R., Ioannidis, C.: The barriers to sustainable risk transfer in the cyber-insurance market. Journal of Cybersecurity (2024)
80. Sotiropoulou, A., Ligot, S.: Legal challenges of cryptocurrencies: Isn't it time to regulate the intermediaries? European Company and Financial Law Review (2019)
81. Statista: Largest cryptocurrency exchanges based on 24h trade volume in the world on june 3, 2024 (2024), https://www.statista.com/statistics/864738/leading-cryptocurrency-exchanges-traders/
82. Statista: Uniswap trading volume (2024), https://coinmarketcap.com/exchanges/uniswap-v3/
83. The Bee Token: Decentralizing short-term housing rentals (2024), https://www.thebeetoken.com/
84. Toby Foster: Understanding crypto insurance: How to mitigate risks (2024), https://cryptomarketcap.com/learn/crypto-insurance
85. UK Finance: Half year fraud report, 2024 (2024), https://www.ukfinance.org.uk/system/files/2024-10/Half%20Year%20Fraud%20Report%202024.pdf
86. USDCoin: Centre whitepaper (2021), https://whitepaper.io/coin/usd-coin
87. Vakilinia, I., Badsha, S., Sengupta, S.: Crowdfunding the insurance of a cyber-product using blockchain. In: UEMCON (2018)
88. Venkataramakrishnan, S.: UK crypto fraud losses jump 40% (2023), https://www.ft.com/content/88602223-004b-4e27-b9eb-c45e1f850f9f
89. Wang, K., Wang, Q., Boneh, D.: Erc-20r and erc-721r: reversible transactions on ethereum. arXiv preprint arXiv:2208.00543 (2022)
90. Wang, K., Wang, Q., Cai, C., Boneh, D.: R-pool and settlement markets for recoverable ERC-20R tokens. In: DeFi (2023)
91. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper (2014)
92. Zeggari, M., Abadi, A., Lambiotte, R., Kassab, M.: Safeguarding physical sneaker sale through a decentralized medium. CoRR (2023)
93. Zhang, Y., Kang, S., Dai, W., Chen, S., Zhu, J.: Code will speak: Early detection of ponzi smart contracts on ethereum. In: SCC (2021)
94. Zhou, L., Wang, L., Sun, Y.: Mistore: a blockchain-based medical insurance storage system. Journal of medical systems (2018)
95. Zuckerman, A.: Insuring crypto: The birth of digital asset insurance. U. Ill. JL Tech. & Pol'y (2021)

A Survey of Related Works

In this section, we explore relevant blockchain-based solutions for insurance. Several studies have proposed integrating blockchain technology to enhance traditional insurance models by leveraging smart contracts and transparent, decentralized ledgers. These approaches aim to detect, reduce, and prevent fraud, streamline insurance processes, and ensure data integrity. Works in this area lack general formal analysis, modeling,

and security proofs, leaving potential security and reliability concerns unaddressed. For brevity, we do not mention this limitation for each work in the following. To the best of our knowledge, our work is the first to formally model and analyze insurance for cryptocurrency transactions. We provide a detailed and formalized definition of insurance within the context of blockchain, offering a comprehensive and precise conceptual framework that can be critically assessed and practically applied within the industry.

MIStore [94] introduces a blockchain-based medical insurance storage system in which data is stored on the blockchain and it is publicly verifiable. The key participants include a medical institution, a patient, an insurance firm, and several servers. A medical institution implements a threshold protocol involving a patient, an insurance company, and several servers. Initially, the institution securely stores confidential data on the blockchain. The design ensures that the servers cannot make sense of the data if (at maximum) up to t out of n servers are corrupted (it is impossible for anyone, including the insurance company, to derive any information with fewer than t responses). The servers are able to perform efficient homomorphic operations (e.g., additions and multiplications within a finite field) which are optimized for performance. Following this, when the insurance company submits a request to the blockchain and gathers t accurate replies from servers, it can then access the information it seeks.

Each server can confirm if its specific share, provided by the medical institution, has been accurately calculated. Moreover, the insurance company is able to verify if the responses it receives from the servers are correctly generated, and the patient also has the ability to ensure that their data has been properly processed by the relevant institution. The recipient of a transaction does not need to repeat the verifications carried out by the chain verifiers. Instead, the recipient only needs to conduct minimal, specific verifications unique to them (this is because of the inclusion of key data within the transaction's payload, much of which is accessible for public verification). This approach reduces the computational effort required for verification by users. While this system offers support for health insurance billing by enabling insurers to accurately assess patients' total medical expenses, it is primarily tailored to this specific application domain. The approach relies on hospitals uploading medical cost data, which insurers can query through the blockchain, yielding responses that outline patients' expenses. However, this design is highly specialized and has not been demonstrated in other contexts, making its applicability to broader insurance (e.g., financial applications) uncertain.

Romanosky et al. [75] provide a comprehensive analysis of the cyber insurance market through the examination of over 100 cyber insurance policies filed with state insurance commissioners. Romanosky et al. [75] highlight significant shortcomings in the cyber insurance market, revealing difficulties in accurately assessing the security posture of customers. The study identifies static assessment methods, such as security questionnaires, lack of attention to technical infrastructure, and simplistic premium computation formulas as key issues, emphasizing a critical lack of data for accurate risk assessment in the cyber insurance markets.

BlockCIS [58] introduces a blockchain-based cyber insurance system designed to address the challenges mentioned by Romanosky et al. [75]. BlockCIS establishes a continuous feedback loop among insurers, customers, third-party services, and auditors using a private, permissioned blockchain based on the open-source Hyperledger framework. BlockCIS has an incentive structure, which motivates entities in need of insurance to join the system. It enables insurers to customize premiums based on a company's cybersecurity measures, allowing companies to demonstrate coverage for potential cyber incidents. The system provides tailored premiums, and proof of coverage for cyber incidents (e.g., policyholders can show that their cyber insurance would cover them if a cyber incident happened). The system lacks a mechanism to verify the accuracy of the information provided by policyholders or monitor changes in their infrastructure. This could be problematic because it means the system might not detect false information or important changes that could affect security (BlockCIS has not addressed measures to prevent false insurance claims or ensure that all parties have access to the same information, which is essential for fairness and trust).

Vakilinia et al. [87] introduce a cyber insurance crowdfunding framework based on blockchain, which records all transactions and information, ensuring data integrity and preventing malicious misuse of the system. Their framework addresses the challenge of traditional cyber-insurance models which mainly cover businesses against liabilities related to digital assets, data breaches, and operational disruptions. These traditional

models, however, struggle to accurately assess the cyber-risks due to the vast diversity and complexity of IT services.
They focus on insuring a specific cyber-product, thereby narrowing the scope of potential threats and simplifying risk estimation. The associated smart contract developed for this purpose manages crowdfunding initialization, bidding, wrapping, and reimbursement. The system involves four main participants: vendor, customer, auditor, and insurer. The process begins when a vendor requests insurance for a cyber-product. Interested insurers then participate in a sealed-bid auction, submitting their bids to provide the insurance service at their preferred premium rates. The insurers who win the auction are chosen to cover the product. In the event of a claim, an auditor verifies the legitimacy of the request. Following this verification, the claim function is activated to allocate the appropriate amount from the collected funds to address the indemnity request.
They lack the necessary security measures, in their approach to cybersecurity data collection and fraud prevention, to guard against cyber threats, such as identity theft and the loss of sensitive information. Additionally, the proposed system's current analysis lacks a thorough evaluation of the processes used by insurance companies to verify policyholder attributes. It overlooks the need for a system that can adapt to changes in the policyholder's infrastructure.
Authors of [52] discuss the issue of Ponzi schemes and fraudulent activities on blockchain platforms, particularly focusing on Ethereum. A Ponzi scheme is a fraudulent investment scam promising high rates of return with little risk to investors, which pays earlier investors with the capital from newer investors until it inevitably collapses. The authors aim to address the problem of detecting and preventing such scams using data mining-based methods. Ponzi scheme detection can have two types of features that can be used for detection: 0-day features (available as soon as a smart contract is uploaded) and behavior-based features (related to the actions of the contract). They incorporate new features and classification models for detecting Ponzi schemes (with both 0-day and behavior-based features).
The SECONDO framework [35] supports organizations with decisions related to cybersecurity investments and cyber-insurance pricing. It is a specialized platform designed for evaluating and efficiently managing cybersecurity risks. It takes a quantitative approach that considers both technical and non-technical factors, including user behavior, which impact cyber exposure. The project provides analysis to enhance risk management by suggesting optimal investments in cybersecurity controls. It assesses residual risks, calculates cyber insurance premiums based on the insurance company's business strategy, and reduces information gaps between the policyholder and the insurance company.
To securely store information, SECONDO incorporates Blockchain and utilizes smart contracts, ensuring transparency, tracking, and verifying adherence to agreed-upon insurance policies in instances of disputes. The smart contracts automate the processing of agreements, notify the involved parties when an agreement is established, and streamline premium and commission payments. The SECONDO project actively gathers cybersecurity data from policyholders and records it on the blockchain. The collected data is crucial for refining future cyber insurance processes, such as underwriting. SECONDO reduces the information gap between policyholders and insurance companies by using cyber insurance policy ontology. The system lacks a proactive mechanism to prevent fraud by ensuring only eligible policyholders can submit claims. It does not have a strategy to verify the eligibility of policyholders before they submit a claim, nor does it ensure the accuracy of the policyholders' data collected during the cyber insurance process.
CioSy [61] is a collaborative blockchain-centered insurance system designed for the management and execution of insurance transactions. This paper addresses the oversight that has been present in current methodologies regarding the concept of collaborative insurance in pursuit of an automated, transparent, and tamper-resistant resolution. CioSy is specifically oriented towards the automation of insurance policy management, claims processing, and disbursements through the utilization of smart contracts. Insurance policies and claims are both machine-readable and self-executing based on voting mechanisms and external oracles (e.g., the interactions with external oracles automatically initiate claims and transfer the claimed amounts to the policyholders).
Specifically, CioSy utilizes smart contracts to enable insurers—such as individuals, banks, and insurance companies—to collaborate, and pool their resources. These smart contracts encapsulate the insurance poli-

cies, representing agreements between insurers and the insured. They present an experimental prototype based on the Ethereum blockchain to demonstrate the practicality of the proposed approach in terms of both time and expenses. The adoption of blockchain in this model ensures adherence to the collaborative insurance paradigm by all participants, enhances transaction transparency, enforces the non-repudiation principle among potentially distrustful entities, automates and accelerates insurance business processes from registration to claim resolution, and reduces costs by minimizing manual interactions.

Franco et al. [39] present SaCI, a Blockchain-powered method designed to boost trust and automation in the communication between a policyholder and its insurer. They assessed SaCI's efficacy by conducting a proof of concept deployed on Ethereum. This approach synchronizes critical aspects of customers with the requirements of cyber insurance companies, including business information, contractual limitations, and security considerations. They employ smart contracts to manage various aspects of the cyber insurance procedure, such as handling premium payments, updating contracts, processing damage coverage requests, resolving disputes, and verifying contract information and integrity. SaCI is capable of automatically transferring funds between stakeholders to fulfill payment obligations, including premium payments and compensations for losses arising from cyberattacks, provided that funds are accessible and contractual conditions are met. It also provides a reliable record of contract coverage and all subsequent modifications which serves as an impartial record or evidence in dispute situations, such as when a customer claims for a loss due to a cyberattack that the insurer has refused to cover. This system lacks a method for verifying the legitimacy of policyholders before they submit a claim request. There is no analysis of how insurance companies verify the attributes of policyholders.

Wang et, al. [89] propose reversible versions of ERC-20 and ERC-721 token standards[16] (denoted by ERC-20R and ERC-721R respectively), allowing for a temporary reversal of transactions during a defined dispute period, with the consensus of decentralized judges. The paper highlights the increasing frequency of thefts in both ERC-20 and ERC-721 contracts, underlining the need for mechanisms that can reduce the impact of such incidents. In cases of theft or accidental losses, the inability to reverse transactions has resulted in substantial losses in the blockchain ecosystem. These attacks were often discovered soon after the theft took place. Hence, the authors looked at ways to reverse the offending transaction(s) within a short dispute period (e.g., four days) –as in traditional finance– to reduce the damage. The affected party submits a request to freeze the transaction to a governance contract, along with relevant evidence and a stake.

A decentralized panel of judges decides whether to approve or deny the freeze request (only the party directly impacted by the transaction can initiate this freeze request.). If approved, they command a governance contract to execute the freeze function on the relevant ERC-20R or ERC-721R contract. As a result, the assets in question are immobilized and cannot be transferred. They consider scenarios where stolen assets may be dispersed or transferred through various accounts. The involved parties present their evidence to the judges, who then make a final decision. Depending on this decision, the governance contract is instructed to either execute the *reverse* function, returning the disputed assets to their original owner, or the *rejectReverse* function, leaving the assets in their current position. This trial could extend over several weeks or even months.

In the event of a dispute concerning an ERC-721 NFT, a freeze is imposed on the current holder, whether it is the original attacker or an honest user who bought the stolen NFT from the attacker. If the judges determine that a theft occurred, the ERC-721R contract returns the NFT to the pre-theft owner, resulting in the current owner losing the NFT. In cases of disputes involving stolen ERC-20 tokens, the complexity arises from the dispersed nature of the funds across various downstream accounts, some of which may be honest while others are dishonest. To address this, a proposed algorithm assigns fractional responsibility to each downstream account that received a portion of the stolen funds, allowing for a partial freeze to be applied to those specific accounts. If the judges determine a theft occurred, the ERC-20R contract then transfers the frozen tokens from the obligated accounts to the pre-theft account. A reversible-token to reversible-token

[16] ERC-20 and ERC-721 are Ethereum token standards with different purposes. ERC-20 defines fungible tokens, where each token is identical and interchangeable, ideal for currencies or utilities. ERC-721, on the other hand, defines non-fungible tokens (NFTs), where each token is unique, making it suitable for digital collectibles, artwork, and one-of-a-kind assets.

exchange can settle instantly, but a reversible-token to a non-reversible-token exchange may need to be delayed (e.g., four days) until the reversible tokens are sufficiently old.
FLAME [56] introduces a framework aimed at improving fire detection and insurance claim processes through the use of blockchain. The authors present a system model that integrates multiple subsystems, including smart fire detection mechanisms, a monitoring station, a fire department management system, and an insurance company. This integration aims to enhance the efficiency and reliability of fire brigade services and insurance claims handling. The proposed sensing network and connectivity model leverages a variety of sensors to ensure accurate fire detection and efficient communication with the monitoring station, thereby facilitating prompt responses from fire brigade services.
The study emphasizes the automation of critical processes through the implementation of smart contracts. These contracts automate the dispatch of fire brigade services and the processing of insurance claims, thereby reducing response times and mitigating the risk of insurance fraud. The paper also includes a security analysis, highlighting the system's robustness against potential cyber threats. The FLAME framework is specifically designed to address the context of fire detection and fire-related insurance claims, the current implementation focuses on the unique challenges associated with fire emergencies. Thus, the FLAME framework is not intended as a general-purpose insurance solution for arbitrary applications but rather as a specialized system for fire-related services and insurance claims.
INCHAIN [36], addresses challenges in the cyber insurance industry by introducing a novel architecture leveraging blockchain. The authors identify key issues such as data insufficiency, manual processing inefficiencies, fraudulent claims, and identity theft. They propose the INCHAIN architecture to enhance data transparency and traceability through blockchain, automate processes with smart contracts, and ensure identification using self-sovereign identity (SSI). This integrated approach aims to mitigate fraudulent claims, streamline operations, and secure sensitive information, thereby providing a more efficient and trustworthy cyber insurance ecosystem. It offers an overview of the existing challenges within the cyber insurance sector, evaluates the current research that incorporates blockchain and smart contracts, and proposes a novel architecture to address these issues.
While the INCHAIN architecture provides an approach to tackling challenges in the cyber insurance sector, it lacks a formal definition and detailed formal treatment of cyber insurance concepts. INCHAIN offers a non-technical and high-level description of its proposed solutions without delving into the rigorous formal analysis that is necessary for a comprehensive understanding and implementation.
In the domain of cryptocurrency insurance services and blockchain insurance providers, several platforms and entities claim to offer solutions that protect users against potential losses due to cyber attacks, technical failures, or other unforeseen events [24,43,70,60,59,8,64,17]. Despite their claims, a critical examination reveals a significant gap in the formal documentation and clarity of their methodologies and operational frameworks from a cryptographic point of view. The information available on the associated websites and promotional materials often describes their services in broad and general terms without delving into the formal specifics of their risk assessment models, underwriting processes, the exact nature of the coverage provided, and so forth. This lack of formal treatment raises concerns about the robustness and reliability of their insurance solutions. Consequently, users and researchers alike are left without a clear understanding of the detailed procedures and standards these providers adhere to, making it challenging to evaluate the efficacy and trustworthiness of their offerings comprehensively.

B Additional Definitions

B.1 Distributed Ledgers

We adopt the definitions of *persistence* and *liveness* for public transaction ledgers as described in [41]. These properties are critical for ensuring the consistency and progress of a distributed ledger. To formalize these concepts, an oracle Txgen is introduced, which manages a set of accounts and generates transactions on their behalf. Txgen is defined as follows: (i) $\mathtt{GenAccount}(1^\kappa)$: It generates an account a. (ii) $\mathtt{IssueTrans}(1^\kappa, \tilde{\mathsf{tx}})$: It returns a transaction tx provided that $\tilde{\mathsf{tx}}$ is some suitably formed string, or $\perp$.

The notation $C(\cdot,\cdot)$ denotes when two transactions, tx_1 and tx_2, are in conflict. A valid ledger is defined as one that does not include any pair of conflicting transactions. An oracle Txgen is said to be *unambiguous* if, for every PPT adversary $\mathcal{A}$, the probability that $\mathcal{A}^{\mathsf{Txgen}}$ generates a transaction tx' such that $C(\mathsf{tx}', \mathsf{tx}) = 1$, for some tx issued by Txgen, is negligible in the security parameter κ. In this context, only unambiguous Txgen oracles are considered. A transaction tx is called *neutral* if $C(\mathsf{tx}, \mathsf{tx}') = 0$ for any other transaction tx'.

Definition 5 (Persistence). *It ensures that once a transaction is incorporated into the blockchain of an honest participant at a depth of more than k blocks from the chain's end, it will, with overwhelming probability, be included in the blockchain of every other honest participant. Furthermore, the transaction will occupy a fixed and permanent position in the ledger. For a depth parameter $k \in \mathbb{N}$, if, during a given round, an honest participant reports a ledger containing a transaction tx located in a block more than k blocks away from the ledger's end (rendering tx "stable"), then every honest participant will report tx at the same position in the ledger from that round onward.*

Definition 6 (Liveness). *It guarantees that all transactions initiated by honest account holders will eventually reach a depth greater than k blocks in the blockchain of at least one honest participant. Consequently, the adversary is unable to execute a selective denial-of-service attack against transactions originating from honest account holders. For parameters $u, k \in \mathbb{N}$ (representing "wait time" and "depth" respectively), any transaction that is either (i) issued by Txgen, or (ii) neutral, and is continuously provided as input to all honest participants for u consecutive rounds, will eventually be reported by all honest participants at a position more than k blocks from the end of the ledger, i.e., all participants will report it as stable.*

B.2 Digital Signatures

Definition 7 (Digital Signature Schemes). *Let $\Gamma = (\mathcal{K}, \mathcal{S}, \mathcal{V})$ denote a digital signature scheme. This scheme is characterized by the following components:*

- *$\mathcal{K}$ **(Key generation):** A probabilistic algorithm that, given a security parameter λ, outputs a key pair $(\mathsf{sk}, \mathsf{vk})$, where sk is the private signing key and vk is the public verification key.*
- *$\mathcal{S}$ **(Signature generation):** A (potentially probabilistic) algorithm that, given the private key sk and a message μ from the message domain $\mathcal{M}$, produces a signature $\sigma \in \Xi$, represented as $\sigma \leftarrow \mathcal{S}_{\mathsf{sk}}(\mu)$.*
- *$\mathcal{V}$ **(Signature verification):** A deterministic algorithm that, given the public key vk, a message $\mu \in \mathcal{M}$, and a purported signature $\sigma' \in \Xi$, outputs a binary decision $\beta \in \{0, 1\}$. The output $\beta = 1$ indicates that σ' is a valid signature for μ under vk, while $\beta = 0$ indicates rejection.*

It is required that $\mathcal{V}_{\mathsf{vk}}(\mu, \mathcal{S}_{\mathsf{sk}}(\mu)) = 1$ for all messages $\mu \in \mathcal{M}$ and all key pairs $(\mathsf{sk}, \mathsf{vk})$ generated by $\mathcal{K}$.

Definition 8 (EU–CMA Security). *A digital signature scheme $\Gamma = (\mathcal{K}, \mathcal{S}, \mathcal{V})$ is said to be existentially unforgeable under a chosen message attack (*EU-CMA*) if, for every PPT adversary $\mathcal{A}$, the success probability $\mathrm{Adv}_{\mathcal{A}}^{\text{EU-CMA}}(\lambda)$ in the following experiment is negligible in the security parameter λ:*

$$\mathrm{Adv}_{\mathcal{A}}^{\text{EU-CMA}}(\lambda) \leq \mathrm{negl}(\lambda).$$

The EU-CMA *experiment, denoted $\mathsf{Exp}_{\Gamma}^{\text{EU-CMA}}(\mathcal{A}, \lambda)$, is defined between a PPT adversary $\mathcal{A}$ and a challenger as follows:*

1. *The challenger runs $\mathcal{K}(1^{\lambda})$ to produce a key pair $(\mathsf{vk}, \mathsf{sk})$ and sends the public key vk to $\mathcal{A}$.*
2. *The adversary $\mathcal{A}$ queries a signing oracle $\mathcal{S}_{\mathsf{sk}}(\cdot)$, providing messages $\mu \in \mathcal{M}$. For each query, the challenger computes and returns the signature $\sigma = \mathcal{S}_{\mathsf{sk}}(\mu)$. All queried messages are logged in a set $\mathcal{Q}$ (the query history).*
3. *Eventually, $\mathcal{A}$ outputs a forgery attempt consisting of a message-signature pair (μ^*, σ^*), where $\mu^* \in \mathcal{M}$ and $\sigma^* \in \Xi$.*
4. *The adversary $\mathcal{A}$ succeeds if:*

– $\mu^* \notin \mathcal{Q}$ *(i.e.,* μ^* *was not previously queried to the signing oracle), and*
– $\mathcal{V}_{\mathsf{vk}}(\mu^*, \sigma^*) = 1$ *(i.e.,* σ^* *is a valid signature for* μ^* *under* vk*).*

The adversary's advantage $\mathrm{Adv}_{\mathcal{A}}^{\text{EU-CMA}}(\lambda)$ *is defined as:*

$$\mathrm{Adv}_{\mathcal{A}}^{\text{EU-CMA}}(\lambda) = \Pr[\mathsf{Exp}_{\Gamma}^{\text{EU-CMA}}(\mathcal{A}, \lambda) = 1].$$