Fair Delegation of Digital Services Without Third Parties

Andreas Guillot
University of Strasbourg
Strasbourg, France
andreas.guillot@unistra.fr

Fabrice Theoleyre
CNRS
Strasbourg, France
theoleyre@unistra.fr

Cristel Pelsser
University of Strasbourg
Strasbourg, France
pelsser@unistra.fr

***Abstract*—The software architecture of most applications is more and more fragmented, and relying on micro-services. Moreover, some parts may be specialized, and a customer may choose to delegate a task to a service provider. In this situation, the customer must be sure to get results that comply with the task when they pay the service provider, and inversely. We propose a framework based on atomic swaps to enable such simultaneous exchanges. Our scheme is based on exchanging a transactional key during an atomic swap. Our framework protects both actors, and enables non-repudiation, from both sides, even in an asynchronous environment.**

***Index Terms*—Services; Secure Delegation; Atomic Swap;**

I. Introduction

Transactions involve two parties: a buyer wants to buy some good, and a seller wants to trade their work for a compensation. In face-to-face transactions, the exchange occurs almost simultaneously. If either party refuses to complete their part of the exchange, the transaction aborts, and both parties keep their belongings. This **fair exchange** is a transaction with only two possible outcomes:

1) *Successful* when both parties end the transaction with what they desire, i.e. correct data for correct amount of money;
2) *Cancelled*, where the transaction fails without any party losing anything.

Services, and particularly micro-services, have become popular: they rely on a fragmented architecture, where each service is in charge of a small part of a global software. This part may involve a *customer* that delegates a service to a *service provider*. In this situation, a fair exchange needs to secure the transaction, especially if no trusted third party can arbiter disputes.

The customer and the service provider agree on the terms of the service; a set of specifications that defines what is expected by the customer (e.g. a program with a specified accuracy), and by the service provider (e.g. currencies). Typically, the service provider is paid only if these specifications, or Service Level Agreements (SLAs) [1], are met.

If the customer and the service provider do not trust one another, then they need a transaction where they exchange both the data, and the payment, simultaneously. In other words, we must guarantee the transaction's atomicity for both actors, even when using an asynchronous, best-effort Internet infrastructure, or when an actor is malicious:

- if the customer sends the payment, they must receive SLA compliant data;
- if the service provider produces SLA compliant work, they must receive their payment. The customer should not be able to invalidate the payment if the work is SLA compliant (non-repudiation).

Most solutions rely on a trusted third party to enforce atomicity and SLA verification, as a fair exchange is impossible without a third party [2]. Human third parties suffer from two limitations: both parties (i) must trust the arbiter, and (ii) the arbiter must not disclose data. For instance, Zhou *et al.* reveal the data to a human witness, and mitigate the risk of getting a malicious arbiter by selecting it randomly [3]. Cryptographic operations can remove the need to disclose data to the arbiter [4], but the arbiter is still able to favor/damage an actor by disclosing secrets.

Recent works use the Blockchain to remove the limitations of a human third party [5]. Leveraging the Blockchain's *immutability* (i.e. unaltereable past states) with its *public visibility*, actors are able to create a smart contract to act as a third party. They can verify the code of the smart contract *before* the start of the transaction. The public visibility of the Blockchain makes it a bad candidate to transfer data because (i) the *private* data will be readable by everyone, and (ii) pushing data into the Blockchain is very costly. We can solve this by exchanging the encrypted data off-chain, and by exchanging a fixed-sized key for currencies using the third party smart contract [5].

However, this solution is unsuitable for the private delegation of a service because it (i) reveals part of the data in their exchange, and (ii) assumes that the target file is known in advance. We need a protocol where (i) the target file, the processing output, is not known in advance, (ii) we do not disclose its contents, and (iii) where the target file must be SLA compliant.

Let us illustrate the interest of these properties with an application that creates a multi-party privacy preserving

978-1-6654-2744-9/21/$31.00

deep learning model [6]. In their solution, participants (i.e. service providers) compute local models on private datasets, and the system (i.e. the customer) aggregates the models' parameters, and not the private data, to create a more accurate model.

Our contributions, which address all these constraints, are the following:

1) *Fair Service Delegation:* We propose a protocol where customers delegate services to service providers;
2) *Proofs of fairness:* We show that our solution is fair, i.e. a transaction either succeeds, or ends without any party gaining an unfair advantage over the other, and that we have **no possible disputes**.

II. Related Work

The first approach to fair exchange is to remove the need for a third party. In [7], the authors rely on zero knowledge proofs so that an actor is able to prove that it has what the other desires without revealing it. The service provider sends the encrypted data to the customer, and the customer deposits money in a smart contract that can only be unlocked if called with the secret that matches the zero-knowledge proof. Then, the service provider reveals the secret that matches the proof by calling the smart contract, which allows the service provider to get the smart contract's deposit, and the customer to decrypt the encrypted data.

However, creating zero knowledge proofs is so computationally intensive to create, and to verify, that their use is prohibitive.

The historical approach to *fair exchange* relies on the use of a trusted third party that settles disputes. Kupcu *et al.* [4] propose an optimistic protocol for fair exchange in p2p file sharing. A trusted third party called the *arbiter* only intervenes when there is a conflict between the customer and the service provider. If a conflict occurs, the arbiter uses cryptographic keys and signatures, and not the data, to determine who is malicious. The arbiter needs to be trusted, as they might advantage an actor by sending the other actor's key. Similarly, Zhou *et al.* [3] rely on a set of third-parties called *witnesses* that they randomly select from a large set. In doing so, they alleviate the chance that a malicious third party may be involved in the transaction. These witnesses receive incentives to verify service violations, which increases the cost of the transaction, and a malicious witness may still rule unfairly. However, involving witnesses has a cost in delay (selection, transmission), and the customer or the service provider needs to pay these witnesses, increasing the transaction's cost.

The main drawback of this family of solutions is that one may never fully trust the third parties, especially if they alternate, in which case one would not be able to reuse a witness they consider to be trustworthy.

It is also possible to use the blockchain as a deterministic third party. Such escrow combines the properties of the blockchain (*immutable*, *public knowledge*, and *imputable*) with smart contracts to design a protocol where a party gets their share of the transaction (e.g. cryptocurrencies) by revealing a secret to the contract (e.g. a cryptographic key). These smart contracts use high-level programming languages (e.g. Solidity [8] contracts on the Ethereum [9] blockchain) to execute programs on all the nodes of the blockchain. By combining the advantages of the blockchain with smart contracts, the research community developed a profusion of distributed applications, e.g. a voting system for Switzerland [10]. Involving a trusted third party that punishes malicious actors requires setting up an escrow [11] to draw from if a misbehavior occurs, or a reputation system [12].

FairSwap [13] uses *proofs of misbehavior*, where a party can punish the other party if it proves that the other misbehaved. They include a signature that uniquely identifies the file that is exchanged during the transaction, so that the customer can verify if the file is indeed the same. If it is not, then the duped party can prove the misbehavior to a judge smart contract, which will then punish the malicious actor. TrueBit [14] works in a similar fashion, where they use multiple miners from the Blockchain to determine if someone produced false data. In that case, the system itself penalizes the malicious user.

This family of solutions shares a common drawback, namely the fact that data needs to be sent to the Blockchain to prove or disprove the dispute. This adds a cost to the transaction, especially if the data is large.

Delgado-Segura *et al.* [5] rely also on the blockchain as a third party, but without any possible dispute. The service provider sends the key that decrypts the traded data *after* being paid by the customer. They protect an honest customer from a malicious service provider by revealing a subset of the unencrypted traded data and by proving that it is a part of the encrypted file. This solution relies on prior knowledge of the exchanged file, and reveals part of the data: in service delegation, the customer does not know a priori what it will receive.

III. Problem statement and assumptions

Modern applications often rely on a collection of tasks. A company may not have all the skills and needs to outsource some tasks to other entities. Such **service delegation** involves two parties:

1) a *customer* that offloads the service execution because of computational or knowledge constraints;
2) a *service provider* that exposes a service executable on demand by any customer on that customer's data.

The customer defines their expectations for the task with SLAs that the service provider must follow. Once the service provider completes their work, they send the data to the customer, and receive their payment in exchange. These two operations need to happen atomically.

A_0	Actors assume that the other is malicious
A_1	A file encrypted with k can only be decrypted using k' from the same key pair
A_2	$P()$ tests if two keys are in the same pair
A_3	$H()$ is non-reversible and collision free
A_4	The actors know the other before the transaction
A_5	Sharing $input$ with sp is not a privacy violation
A_6	The verfier's contents cannot be read
A_7	The task's logic is automatable
A_8	$\{sk, pk\}$ pairs are only used in one transaction
A_9	Actors communicate through a secured channel
A_{10}	Public Blockchain where all can read/write
A_{11}	The smart contract is already on the Blockchain
A_{12}	Actors know the smart contract's address
A_{13}	Actors trust the smart contract's code
A_{14}	Actors do not stall the transaction
A_{15}	sp eventually produces SLA compliant work

TABLE I: Assumptions

A. Scenario

In our party-party deep learning example, we assume the following. The customer has a set of SLAs that define (i) what the task consists in, and (ii) what are the criteria that the work must meet. They also have data on which to perform the task. The service provider has a set of skills that the customer lacks, and must produce SLA compliant work on the data they receive from the customer. Once complete, they send their work to the customer and expect to be paid in return.

B. Security requirements

The security properties that we need for a fair delegation of services are the following:

P_1**- Fair service delegation:** The customer receives data if they spend currencies to pay the service provider;

P_2**- SLA enforcement:** The customer receives work that is SLA compliant;

P_3**- Fair compensation:** The service provider receives the agreed upon wage;

P_4**- Protection from outsiders:** A third party cannot gain information by listening to communications or by reading the Blockchain.

C. Model and assumptions

This section summarizes all our assumptions regarding our cryptographic capabilities and the actors' behaviors. We list our variables and functions in Table II.

Public-key cryptography relies on a **private key** (abbrv. *sk*) and a **public key** (abbrv. *pk*) respectively. We take the following assumptions:

A_1**:** A key can only decrypt a message encrypted with the other key in the key pair;

	Notation	Description
Actors	c	Customer
	sp	Service provider
Knowledge before the transaction	$wage_{\{c,sp\}}$	Amount that c pays, and that sp receives
	$input$	c's input data used by sp
	$sk_{\{c,sp\}}$	Private key of actor $\{c, sp\}$
	$pk_{\{c,sp\}}$	Public key of actor $\{c, sp\}$
Data operations	$V_{sk_c, pk_{sp}}$	Tests work against SLA
	$E(k, data)$	Encrypts data using key k
	$D(k, data)$	Decrypts data using key k
	$H(data)$	Outputs data's hash
	$S(k, data)$	Outputs data's signature

TABLE II: Notations

A_2**:** We can determine if two keys belong to the same key pair by comparing the public key with the public key derived from the private one ($P()$);

A_3**:** We use a collision free, non-reversible hashing function [15] ($H()$);

We use $E(k, data)$ (resp. $D(k, data)$) to encrypt (resp. decrypt) data using a key. Note that the private (resp. public) key can be used to decrypt messages encrypted with the public (resp. private) key. $S(k, data)$ returns the signature of $data$ using the key k [15]. A signature validates the sender's identity.

Asymmetric operations are more computationally expensive than symmetric ones. However, we need a mechanism to verify if a key is valid without decrypting the data itself. Verifying the public key when having the private one is an essential feature in our protocol.

An actor always acts under the suspicion that the other is malicious (A_0). They always protect their own interest. Actors do not stall the transaction by never sending data (A_{14}), and that the service provider eventually produces SLA compliant work (A_{15}). We include these assumptions to make sure that the transaction eventually ends.

The actors interact with a public Blockchain supporting smart contract (e.g. Ethereum [9]) that they can read and write to at all times (A_{10}). The smart contract used in this transaction is already in the Blockchain (A_{11}), and the actors know its address (A_{12}). Actors trust the smart contract's code, as they can verify it before the start of the transaction (A_{13}).

IV. Fair exchange of services

Both actors start with the initial information described in the previous section and summarized in Step (0) of Figure 1. Our protocol has 4 steps:

Initialization: the customer and the service provider must have a public (personal) key, and must decide the price to pay/get for a given service;

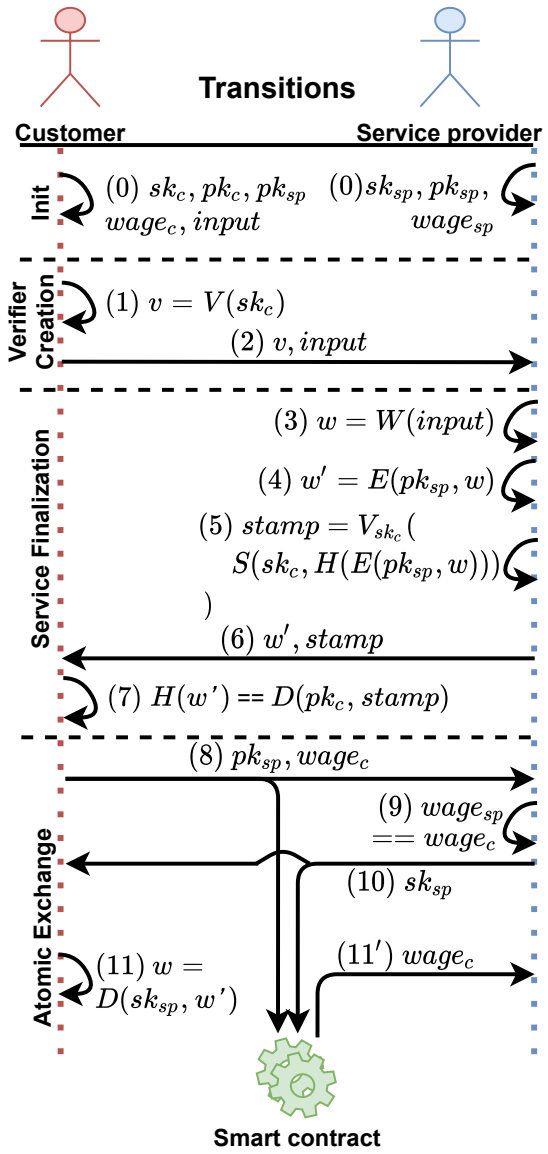

Fig. 1: Different steps required to delegate services fairly

Creation of the verifier: the customer creates a verifier checking if the work is SLA compliant, and creates the proof of compliance if it is;

Service finalization: the service provider creates SLA compliant work, encrypts it, and sends it to the customer. The customer can verify the SLA compliance, but cannot decrypt the work;

Atomic exchange: the customer and the service provider exchange currencies for the key that unlocks the service provider's encrypted work.

A. (Offline) pre-initialization

The customer and the service provider have the following initial knowledge before the start of the transaction. We index the variables with the actor in their possession, either c for the customer, or sp for the service provider:

$sk_{c|sp}$: the **private** part of their key pair;

$pk_{c|sp}$: the **public** part of their key pair, and the **public** part of the other's key pair;

$wage_{c|sp}$: the wage that they expect to pay (resp. receive) for the customer (resp. service provider).

We assume that the customer and the service provider know each other before the transaction (A_4). We assume the existence of a directory, describing the API of each service (e.g. [16]), so that the actors know each other, and the terms of the service.

The customer prepares $input$, the data that they will send to the service provider once the task begins. This data must be transmitted to the service provider (A_5) so that they can perform their task. However, the service provider are not allowed to share this data with any third party.

Actors have an asymmetric key pair: sk_c and pk_c for the customer, and sk_{sp} and pk_{sp} for the service provider. Note that these are disposable keys that the actors **only use for one transaction**, and for nothing else (A_8), as it would damage them. This implies that the actors must generate new key pairs for every transaction. We argue that this is acceptable because the generation of ECDSA 256 bits key pairs (classical size used in Ethereum) is fast, and that they only have to do it once for each transaction. Note that they do not need to create new wallet addresses for each transaction, which would be much more costly. We assume that the actors already share a secure channel for communications (A_9; created using e.g. mTLS [17]). This simplifies explanations, as actors only need one disposable key pair that way. As such, these keys can be disclosed without compromising the secure channel, or the actors' identities.

Both actors have their own $wage_{c|sp}$ variables to ensure that one actor does not pay or receive less than the agreed amount. If the two wages do not match, then the transaction aborts.

Thus, we initialize the protocol in the following state (Fig. 1):

(0): the service provider has their private/public key pair $\{sk_{sp}, pk_{sp}\}$, and their wage $wage_{sp}$. The customer has their private/public key pair $\{sk_c, pk_c\}$, and their wage $wage_c$. They also have the service provider's public key pk_{sp}, and the input to be sent to the service provider.

B. Online initialization — creation of the verifier

We assume that the task is verifiable without human intervention (A_7). While we agree that creating SLAs for any real-world applications might be difficult (e.g. determine artistic beauty), we also argue that this is the only realistic way to fully automate task verification. The formal specification of a service delegation has already been defined for various arguably different applications such as cancer treatment [18], or earthquake predictions [19].

The verifier (i) receives data, (ii) checks if the input is SLA compliant, and (iii) outputs the result. It contains a secret: sk_c, that enables non-repudiation, as explained in Section IV-C.

This verification acts in the following way (Fig. 1):

(1): the customer creates the verifier with the task's logic, the test data, and the secret;

(2): the customer sends the verifier and *input* to the service provider. The service provider will then be ready to start the work.

The service provider uses the verifier as a proof of compliance to the customer. Thus, we make it impossible to create a counterfeit proof. Additionally, this proof must not contain sensitive information from any of the actors, as the other would gain access to private data. We consider that the verifier (binary program) is unreadable by the service provider (A_6): they cannot access its data, or its secrets. They can just execute the binary with a set of inputs and receive a proof of compliance, or an error. Code obfuscation can further hide the contents of the verifier [20].

C. Service finalization

After the service provider has finished working, they have generated an output that needs to be sent to the customer to trigger the payment. However, the output cannot be sent directly since the exchange with the customer has to be atomic. Thus, we need first to prepare the atomic exchange: we need to transmit enough information to start the atomic exchange, but no sensitive information.

Our scheme relies on the following steps (Fig. 1):

(3): the service provider produces work using the customer's *input*;

(4): the work cannot be sent directly to the customer and is encrypted with the service provider's public key, pk_{sp}. Still, this key is disposable and will only be used for this transaction.
Because the encryption is asymmetrical, the customer is not able to decode the work, even if they are in possession of the key that encrypted this data; they must know the service provider's private key to achieve that. Since this private key will only be disclosed during the atomic exchange, the encryption protects the service provider: the customer will only be able to decrypt the data once they send the payment.

(5): the service provider executes the verifier with the work they created. If the verification fails, then the service provider goes back to Step (3). Otherwise, the verifier returns the stamp, whose purpose is twofold:

1) it **uniquely identifies the work** for non repudiation. It uses the hash of $E(pk_{sp}, w)$, the service provider's encrypted work, as illustrated in Figure 2.

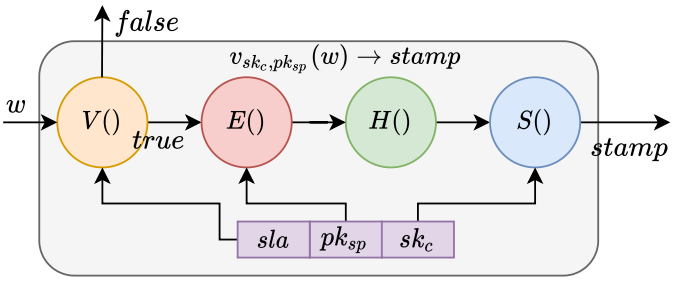

Fig. 2: Creation of the stamp

2) it certifies the work is SLA compliant, with a **proof of compliance**. By signing the hash described above with sk_c, the verifier's secret, the stamp certifies that the work is SLA compliant.

The service provider cannot fake the stamp because the verifier signs it with sk_c, their secret key (see Figure 2). A malicious customer is unable to include sensitive information inside the stamp (e.g. using steganography) since its content is readable and verifiable. The service provider has enough knowledge (i.e. the public key of the customer) to interpret the content of the stamp;

(6): the service provider sends to the customer the encrypted work (w') and the stamp generated by the verifier.
It is worth noting that the encrypted work is transmitted through a secure channel: no observer is able to get it. Thus, an observer will not be able to decrypt the work, even after the atomic exchange, when the key becomes public on the Blockchain.

(7): the customer checks whether the hash of w' matches that of the stamp. If it does not, then the customer aborts the transaction: it means that the verifier did not validate the work. At this point, the customer knows that the service provider produced SLA compliant work, but is not able to decrypt it until they get the service provider's private key, sk_{sp}.

At the end of this step, the customer and the service provider have enough knowledge to trigger the atomic exchange.

D. Atomic exchange

The atomic exchange consists in atomically exchanging the service provider's secret key sk_{sp} to decrypt the work (by the customer) and the payment. To perform the exchange, we use the smart contract present in the Blockchain (A_{11}).

The atomic exchange consists in 4 steps (Fig. 1):

(8): the logic of the contract is that an entity can only claim the customer's wage if they send a key that matches with the one published by the customer. The customer pushes a block containing the service provider's public key and the wage they want to

pay. The service provider sees the customer's message since they know the customer's identity on the Blockchain;

(9): the service provider can check that the wage in the contract is equal to what they expect ($wage_c = wage_{sp}$). If $wage_c$ is lower, then it means that the customer wants to pay them for less than initially agreed. Similarly, they check if the public key pushed to the Blockchain by the customer in Step (8) is really theirs (pk_{sp}). The transaction only proceeds if both wages and both keys are equal;

(10): to validate the transaction, the service provider sends their private key to the Blockchain. By pushing a block containing the private key to the Blockchain, they indirectly communicate this information to the customer, who is reading new blocks from the public Blockchain (A_{10}).
The contract tests whether the keys pushed by both actors match ($P(pk_{sp}, sk_{sp})$). The transfer of the customer's currencies ($wage_c$) to the service provider only occurs if the two keys belong to the same key pair. Thus, the service provider **must** send their private key sk_{sp} to trigger the payment;

(11): the customer can read the Blockchain to extract the service provider's private key, and to decrypt the work they received in Step (6). By construction, the private key is the one matching with the public key (pk_{sp}) that the consumer possesses since Step (0), so it must be the key that is able to decrypt the work. Any observer is able to read the service provider's private key, so they could theoretically decrypt the work. This is why we sent it over a secured channel. Moreover, the Blockchain is never used to store the data itself, so outsiders have no use for this key.

(11'): finally, the service provider receives their wage from the smart contract since the keys matched.

V. Security Proof

A. Fair compensation

Theorem V.1. *The service providers always receive their wage if the customers obtain their work.*

Proof. We consider each situation which would be unfair for the service provider.

Case 1: *the customer cannot decrypt the work before the atomic exchange.* The customer doesn't have sk_{sp}, the key that would decrypt the work. They have pk_{sp}, the key that encrypted the work, but it cannot decrypt it since we use asymmetrical cryptography.

Case 2: *If the customer pays less than the agreed amount, it cannot decode the work.* If the customer pays less, the condition for Step (8) will not hold, and the transaction will stop.

Case 3: *the service provider always receives currencies after they disclose their secret key* . The first part of the smart contract is engaged before pushing the secret key to decrypt the work: the customer must have sent the money, as well as the public key of the transaction to identify the work. We assume that the service provider will never send a private key (sk_{sp}) that does not match the public one, since it would abort the transaction, which would only be prejudicial for themselves. □

Theorem V.2. *If the customers spend the wage, they receive a SLA compliant work.*

Proof. We split the proof in

Property 1: *The service provider cannot send encrypted work that is not SLA compliant*. If the verifier generates a stamp, the work must be SLA compliant, else the stamp would not be generated by the verifier. Besides, the stamp contains the signature of the hash of the encrypted work. Thus, the customer can verify that the stamp corresponds to the encryption of the same work (w) when receiving w'.

The service provider cannot generate a fake stamp, since it has no access to the private key sk_c, and we assume that the verifier is opaque (A_6).

Property 2: *If the customer sends currencies, it is able to decrypt the work that has been sent*. With Property 1, we know that the encrypted work is SLA compliant.

The customer publishes pk_{sp} in the Blockchain, and provisions the wage associated with the transaction. The service provider will receive the money only if they published sk_{sp}, and if sk_{sp} is the private key that pairs with pk_{sp}. If not, then the contract is cancelled and the money is returned to the customer. The Blockchain is immutable, so they cannot tamper with the smart contract's code (A_{13}). Plus, neither the servider provider's work sent in Step (6), nor pk_{sp} can be changed by the service provider. Thus, the customer will receive eventually sk_{sp} and will be able to decode the work. □

B. Protection from outsiders

Theorem V.3. *Both the customer and the service provider are protected from outsiders, i.e. outsiders cannot learn any knowledge that the actors did not intend to disclose.*

Proof. We assume that there is an attacker that knows who the customer and the service provider are. They have two attack vectors: network communications, and data available on the Blockchain.

Case 1: *Attackers steal information from network communications*. The customer and the service provider exchange everything through a secure channel (A_9).

Case 2: *Attackers steal information on the Blockchain*. An attacker may acquire the secret key sk_{sp} from the Blockchain, as well as the public key pk_{sp}. However, w' is not accessible in the Blockchain, and has been transmitted from the service provider to the customer through the secure channel. Thus, the keys are useless to decode the work.

Besides, an attacker cannot acquire the money of the customer before the service provider. For this purpose,

it must publish the secret key sk_{sp}, that it doesn't have. Moreover, A_8 states that actors use a different disposable key pair for each transaction. □

VI. Conclusion and future work

In this paper, we presented a solution that tackles the fair exchange of digital services where a customer delegates the completion of a service to a service provider in accordance with existing specifications. We do not rely on a trusted third party but instead use a public Blockchain as a neutral intermediary, and the properties of smart contracts to trade currencies for data atomically. Compared to related work, our solution does not presume that the traded file is known in advance, and only that it matches some specifications. We do not reveal part of the data during the transaction, and do not rely on complex zero-knowledge proofs. Plus, the cost of the transaction is constant, and does not depend on the size of the traded file.

In the future, we plan to generalize our model to multi-party transactions, not just transactions involving two parties. Chaining the smart contracts for all the parties may be challenging, particularly when using an unreliable communication infrastructure. Finally, we expect to exploit the Lightning Network to reduce the cost of using the Blockchain. More specifically, we would reduce the cost of the transaction by using the Blockchain only at initialization, and when a dispute has to be settled.

Acknowledgments

This project has been made possible in part by a grant from the Cisco University Research Program Fund, an advised fund of Silicon Valley Foundation (grant number 1318167).

References

[1] A. Gamez-Diaz, P. Fernandez, and A. Ruiz-Cortes, "SLA-driven governance for RESTful systems," in *International Conference on Service-Oriented Computing (ICSOC)*, 2017, pp. 352–356.

[2] H. Pagnia and F. C. Gärtner, "On the impossibility of fair exchange without a trusted third party," Darmstadt University of Technology, Technical Report TUD-BS-1999-02, 1999.

[3] H. Zhou, X. Ouyang, Z. Ren, J. Su, C. de Laat, and Z. Zhao, "A Blockchain based Witness Model for Trustworthy Cloud Service Level Agreement Enforcement," in *INFOCOM*. Paris, France: IEEE, Apr. 2019, pp. 1567–1575.

[4] A. Küpçü and A. Lysyanskaya, "Usable optimistic fair exchange," *Computer Networks*, vol. 56, no. 1, pp. 50–63, 2012, publisher: Elsevier.

[5] S. Delgado-Segura, C. Pérez-Solà, G. Navarro-Arribas, and J. Herrera-Joancomartí, "A fair protocol for data trading based on Bitcoin transactions," *Future Generation Computer Systems*, vol. 107, pp. 832–840, 2020, publisher: Elsevier.

[6] X. Ma, F. Zhang, X. Chen, and J. Shen, "Privacy preserving multi-party computation delegation for deep learning in cloud computing," *Information Sciences*, vol. 459, pp. 103–116, 2018, publisher: Elsevier.

[7] M. Campanelli, R. Gennaro, S. Goldfeder, and L. Nizzardo, "Zero-Knowledge Contingent Payments Revisited: Attacks and Payments for Services," in *SIGSAC Conference on Computer and Communications Security (CCS)*. Dallas Texas USA: ACM, Oct. 2017, pp. 229–243.

[8] C. Dannen, *Introducing Ethereum and Solidity*, 1st ed. USA: Apress, 2017.

[9] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[10] C. Killer, B. Rodrigues, R. Matile, E. Scheid, and B. Stiller, "Design and implementation of cast-as-intended verifiability for a blockchain-based voting system," in *Symposium on Applied Computing (SAC)*. Brno Czech Republic: ACM, Mar. 2020, pp. 286–293.

[11] L. Eckey, S. Faust, and B. Schlosser, "OptiSwap: Fast Optimistic Fair Exchange." *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 1330, 2019.

[12] S.-w. ZHENG and L. FAN, "Credit Model based on P2P Electronic Cash System Bitcoin [J]," *Information Security and Communications Privacy*, vol. 3, p. 040, 2012.

[13] S. Dziembowski, L. Eckey, and S. Faust, "FairSwap: How To Fairly Exchange Digital Goods," in *SIGSAC Conference on Computer and Communications Security*. Toronto Canada: ACM, Jan. 2018, pp. 967–984.

[14] J. Teutsch and C. Reitwießner, "A scalable verification solution for blockchains," *arXiv preprint arXiv:1908.04756*, 2019.

[15] B. A. Forouzan, *Cryptography & network security*. McGraw-Hill, Inc., 2007.

[16] M. Stocker, O. Zimmermann, U. Zdun, D. Lübke, and C. Pautasso, "Interface Quality Patterns: Communicating and Improving the Quality of Microservices APIs," in *European Conference on Pattern Languages of Programs (EuroPLoP)*. Irsee Germany: ACM, Jul. 2018, pp. 1–16.

[17] J. Bradley, B. Campbell, T. Lodderstedt, and N. Sakimura, "OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens," IETF, Tech. Rep. rfc8705, 2020.

[18] N. Sadhasivam, R. Balamurugan, and M. Pandi, "Cancer Diagnosis Epigenomics Scientific Workflow Scheduling in the Cloud Computing Environment Using an Improved PSO Algorithm," *Asian Pacific journal of cancer prevention: APJCP*, vol. 19, no. 1, p. 243, 2018.

[19] P. Maechling and *et al.*, *Workflows for e-Science*. Springer, 2007, ch. SCEC CyberShake workflows—automating probabilistic seismic hazard analysis calculations, pp. 143–163.

[20] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, "Code obfuscation against symbolic execution attacks," in *Annual Conference on Computer Security Applications (ACSA)*. ACM, 2016, pp. 189–200.