Peer-to-Peer Networking and Applications
https://doi.org/10.1007/s12083-020-00901-w

A post-quantum end-to-end encryption over smart contract-based blockchain for defeating man-in-the-middle and interception attacks

Amir Hassani Karbasi[1] · **Siyamak Shahpasand**[2]

Received: 4 March 2019 / Accepted: 28 February 2020
© Springer Science+Business Media, LLC, part of Springer Nature 2020

Abstract

Ethereum is a public, open-source, decentralized, and peer-to-peer blockchain-based computing network which is involving to the usefulness of smart contract. It gives a distributed Turing-complete virtual machine in which some codes can be executed by utilizing a worldwide and public network of nodes. The compelled certificate creation and Man-In-The-Middle (MITM) attacks are two major attacks on End-to-End Encryption (EEE) and SSL/TLS. A portion of the real attacks on end-to-end encryption and SSL/TLS is IP/ARP poisoning and the phishing attack. MITM attack makes the client difficult to understand, whether they are associated with a unique verified and secured connection or not. Since the certificate and public-key that is being passed during the connection setup is unreliable and insecure, the attacker can undoubtedly change the data in the certificate and leaves the endorsement of the certificate and public-key to the client. The purpose of this paper is to present a solution of providing the legitimacy and authenticity of freely shared and published online digital data, e.g., digital certificates, cryptographic keys, and common reference strings such as shared passwords using a mix of recently developed innovations which primary include blockchain, smart contract, InterPlanetary File System (IPFS), and quantum-resistant Password-based Authenticated Key Exchange (PAKE) protocol over rings and ideal lattices. Ethereum smart contract is utilized to manage, surveil, and give detectability and visibility into the history of digital data from its beginning to the most recent variant, in a way that it is decentralized and internationally accessed with high integrity, resiliency, and transparency, that we should thank to the immutability and irreversibility of the blockchain. The full code of our smart contract is given, with a discourse on the execution and testing of its key functionalities.

Keywords Blockchain · Decentralized app (DApp) · IPFS · Lattice-based key exchange · MITM attacks · Privacy · Smart contract

1 Introduction

Nowadays, the amount of data transmitted electronically over the web and network has continued with a quick development over the past years, and it is anticipated to grow much more forcefully. Essential communications, similar to business deals, monetary exchanges, restorative records, private material, cryptographic keys (certificates), and so forth depend upon a safe and confirmed channel.

In the wake of understanding the pervasiveness of network and Internet monitoring directed both by country state on-screen characters and criminal associations, numerous residents and organizations have reasoned that the best way to accomplish protection and secure correspondence over the network is to utilize end-to-end encryption. PGP [1] and OpenSSL [2], are the most outstanding amongst the most broadly utilized email and data encryption schemes, empower clients to send secure messages over insecure channels without dread of interception or modification of the digital data.

✉ Amir Hassani Karbasi
amirhassanikarbasi@gmail.com

Siyamak Shahpasand
shahpasand_siyamak@yahoo.com

1 Department of Computer Science, Khazar University, Baku, Azerbaijan

2 Department of Computer Engineering, University of Tabriz, Tabriz, Iran

Published online: 19 March 2020

In the same way, as other security components which are dependent on public-key encryption, these schemes have one essential deficiency that is the key exchange. in fact, before setting up a safe channel, the sender should retrieve the public key of the recipient initially. In any case, doing this action over an unreliable channel is seemingly not prescribed and as a result, an adversary could undoubtedly change the valid public-key with his own and it can empower a man-in-the-middle attack. Different methods such as hand exchange, web of trust, public key infrastructure, and authenticated key exchange protocols have been embraced for securely setting up a connection between a public-key and the identity of its owner.

For example, in the first structure, a messaging app used to utilize the TCP/IP protocol to send messages and data between clients. Encryption and digital signing have been added to the protocol, however just covering a piece of the journey, between the client and server, or between two servers, without any end-to-end encryption. It would be troublesome to TCP/IP protocols for making them secure because numerous nodes need to organize and receive a similar security standard, even those like service providers which have nothing to gain.

To solve this problem and the session key establishment, one of the most popular protocols is Password-based Authenticated Key Exchange (PAKE). For example, our protocol which is called *ring-PAKE* [3], is based on the algebraic rings and its security relies on ideal lattice assumptions in lattice-based cryptography.

In this paper, we will propose a new strategy for secure key establishment and 2-step password agreement. The method will include designing and building a self-ruling and decentralized client-side DApp as a key server which is bound up with Ethereum smart contracts. This framework will empower clients to store their public-keys on the Ethereum blockchain, rather than being dependent on a conventional key server. This will decrease the measure of trust that clients have to put on third-party services for their protected messaging needs. Beside trusting to different entities which is truly a risk under the security perspective and expanding the number of on-screen characters, it also prompts an expanded probability that one of them will be compromised by an attacker or might act maliciously.

Ethereum in its present structure isn't the best platform to store large amounts of data and public-keys. For such substance, we need to utilize an outer IPFS server [36]. Right now, there is no built-in component that can enable smart contracts in communicating with IPFS (or anything outside of Ethereum). To make those two things exchange the information better, first we have to save the public key in IPFS that will restore a hash, and then, the hash will be saved to an Ethereum node against a location for reference and recovery.

In this paper, we propose a consolidated IPFS-blockchain-based strategy to solve the credibility and authenticity of public-keys posted freely on the network or Internet. In the paper, we show how this issue can be comprehended in online distributed public-keys. We show how our method can follow the digital data or public keys, in which they are ensured that they are genuine duplicate and were made by the first creator or original client. The fundamental contributions of this paper can be abridged as follows:

- We propose an IPFS-blockchain-based arrangement and structure for giving validness and authenticity of online distributed public-keys and the passwords of the PAKE protocol. Our technique gives decentralized storage and administration with various copies of public-keys being stored, followed, and tracked with high integrity, respectability, and resiliency. Each user acts as a key server to create and revoke his own public and private keys.
- We present and examine the framework structure and design, alongside with sequence diagrams to show the connections among members which incorporate new user, key server, and existing users.
- We give the full code of the smart contract and examine key execution and testing details to exhibit the appropriate tasks and usefulness of the general framework.

We give the full code of the smart contract and examine key execution and testing details to exhibit the appropriate tasks and usefulness of the general framework.

The remainder of this paper is organized as follows. Section 2 surveys previous works in this field. Section 3 summarizes some background concepts which are necessary to understand the paper argument. Section 4 explains problem definitions and our proposed solution for improving credibility and authenticity of transmitted public-keys. Section 5 presents the concepts about the designation of our method. Section 6 describes the key aspects of implementation and analysis of the smart contract. Section 7 concludes the paper.

2 Related works

During the previous couple of years, lattice-based cryptography has been known for its various developments and

cryptographic schemes in making solid security proofs, protection from quantum assaults, adaptability for fully homomorphic encryption [4] and productivity with aggressive performance among traditional protocols which were built up by integer factoring problem (IFP) and discrete logarithm problem (DLP). Specifically, learning with errors over rings (ring-LWE) [5, 6], as a lattice problem, is utilized for improvement of secure and efficient lattice-based primitives based on rings.

Cramer and Shoup presented one protocol which is classified and called Smooth Projective Hash Function (SPHF) [7] in order to get hash proof frameworks for IND-CCA security. Gennaro and Lindell proposed a summed up SPHF [8] for its numerous appealing properties and purposes, like implicit designated verifier proofs of membership [9, 10]. Then, there is beneficial use of SPHF called Password-Based Authenticated Key Exchange (PAKE) protocol. It was introduced by Katz, Ostrovsky, and Yung [11] and additionally, Gennaro and Lindell [8] which is known as the KOY-GL primitive.

A PAKE protocol is established by a shared password for the entities of a particular key exchange. A PAKE gives security against offline dictionary attacks (formally, Bellare-Pointcheval-Rogaway (BPR) paradigm [12]), and this setting, even with low-entropy passwords keeps from clients impersonation. Besides, for a PAKE, a safe server outfitted with password-based authentication can give security against online dictionary attacks to such an extent that an adversary endeavors to impersonate a client which is utilizing every conceivable password. PAKE protocols, as a rare cryptographic primitive for real-world applications, have been standardized [13] and broadly conveyed and implemented [14].

In the light of the password and public-keys, a hybrid model is shown in [15, 16]. A password-only model dependent on just a secret phrase with heuristic security is started in [17]. Specifically, PAKEs with provable security in the random oracle model and formal models are appeared in [12, 18–20]. The primary inefficient PAKE in the standard model is exhibited in [19] and its enhancement with a more reliable thought of security is appeared in [21] but by comparing these two, it is still impractical. The principal effective PAKE in the standard model with provable security is proposed in [11] and its variations are given in [22–26]. A Common Reference String (CRS) is required for these conventions and a PAKE in the CRS model is shown in [27].

Building SPHF from lattice suppositions related to LWE and SIS issues has stayed open. The first PAKE and its special case which is utilizing SPHFs in the standard model related to lattices is proposed in [28]. This development has a crucial disadvantage: it works for an ad-hoc language of ciphertexts so the relating decryption technique should be changed.

As we mentioned before, fewer attention has been paid to key exchange protocols for real-world transmission from ideal lattice assumptions and as we can see practically all models for cryptographic primitives are still planned around classical systems, for example, RSA [29] and Diffie-Hellman [30]. However, some ongoing proposals are promising [31–33] which depend on random oracles and standard proofs.

It is clear that the public-key discovery for sending encrypted session keys or encrypted shared passwords using public and private keys in a distributed messaging app (DApp) is a challenging problem in asymmetric cryptography with the existence of MITM attacks. This would be so problematic into a DApp with application of digital currency transactions, posting news, videos, and voices as social network activities, and etc. Because, we might face with the manipulated certificate or session key and receive fake data. Notice that an Internet Service Provider (ISP), Certificate Authority (CA), Unified Threat Management (UTM), Routers, Virtual Private Network (VPN), proxy servers, and so many others act as a middleman. The next two presented solutions are web of trust and public key infrastructure (PKI).

Web of trust is a decentralized framework to set up the credibility of the connection between a public-key and its owner's identity. The idea depends on the possibility that every client will sign the public keys of alternate clients which he trusts and then distributes them. Along these lines, keys are connected together and if the identity of a client is trusted then it is sensible additionally to confide in the identity of alternate clients that he vouches for. The web of trust idea was first presented in the cryptography field by PGP maker Phil Zimmermann [1] in 1992 in The official PGP client's guide.

According to this method, Alice can request Bob's public-key from Carol, Chuck, and Dave whom they are acting as clients that Alice has trusted for them and she has already confirmed their public-keys. They sign Bob's key with their public-keys, and Alice can choose what number of affirmations she needs to trust Bob. Note that this procedure is programmed, in light of the fact that each time two clients impart the information by utilizing PGP which is empowered by web of trust, and a rundown of each trusted contact is enclosed to the email and refreshed by the recipient.

Typically, key proprietorship between clients that trust and realize each other is checked in person, occasionally

at alleged key signing parties. Under this plan, the choice about who to trust is left to every individual thus the control is progressively granular. This decentralization method forgets the governments' rules and enormous enterprises that cannot impact the framework, and if somebody's key is compromised and the original client still has the private key, the compromised key can be revoked for the most part in any event. Because of the Small World Theory [34] the average path length between clients in a web of trust ought to be entirely little, enabling to convey safely even between outsiders that never checked their individual keys.

The most concerning issue with the web of trust is that, among PGP clients, almost no one uses it. This consideration depends on an examination of the strong set [35]. There are diverse reasons to say why the selection of a web of trust idea isn't greater. It is not practical to meet individuals face to face, and for some evident reasons, signing parties are incredibly uncommon in the cryptographic protocols. In addition, there are no feasibility motivations for clients to add another trusted key to their key-ring, and still it should be done just as a selfless exertion. At last, there is the likelihood for clients to lose their private key, to such an extent that they have no access to encrypted messages which has sent to them. For this situation, it is likewise difficult to make a revocation certificate, and the public-key stays on the web of trust until each client believes that he himself chooses to deny his trust. Thus, the choice of reaching an individual utilizing a public-key can't be founded just on the web of trust, however, it must be affirmed in some other ways, or otherwise it is conceivable to communicate something specific that the recipient will never have the capacity to peruse.

The other strategy that client need to get public-keys is through key servers or the PKI based on certificates. These servers are normally dealt with associations, for example, Comodo,[6] GoDaddy,[7] Certum,[8] PGP Corp key server,[9] MIT PGP Public Key Server,[10] or by people as a de facto model, and they are reachable through the HKP (HTTP Keyserver Protocol) or a web interface. As an example on PGP keys, They associate an email address with a public-key or with a key-ring (accumulation of public-keys) and makes them accessible. Everybody can include an email and a public-key, however, just the proprietor can revoke his very own key. A few servers likewise check the responsibility for the address sending an affirmation mail. Key servers can be utilized as a supplement of the web of trust if clients discover public-keys that are not affirmed by their network of contacts, or they can be the main confided source for keys. In the last case, there are some security disadvantages.

The side which controls a key server has unlimited authority over it, and it can change public-keys and addresses related to a specific client voluntarily. regardless of whether the association, similar to a college or an NGO, which is controlling it, tends to be viewed as trusted, but no one can ensure that it won't be compromised later on. Additional to this, as it is significantly stressing, an adversary which picks up control of a key server could demonstrate an alternate public-key just to the specific client, with the goal that the proprietor could never find the irregularity simply making a decision about a server reaction. For this situation, when a client needs to impart safely, other than confiding in the current PGP app implementation, his machine, and the recipient's machine, he additionally needs to believe that the key server is giving him the correct public-key, adding a frail spot to the idea of end-to-end encryption.

Finding an answer that wipes out or diminishes the measure of trust that clients of PGP arrangements need to stretch out to the key server would incredibly lessen the assault surface, expanding the security and reliability of secure correspondence through email and different channels.

3 Background

3.1 Blockchain, Ethereum, and smart contracts

Bitcoin [37] is the principal use of blockchain, it's a sort of digital cash dependent on blockchain innovations, which is utilized for commerce things on the network and Internet, like cash as we do in this present reality. We utilize blockchain services or toolkits in our daily life, and similarly cybercriminals are motivated to use their chance to take part in cybercrime commitments [38, 39].

Blockchain [37, 40, 41] innovations are not just one procedure, it contains cryptography, mathematics, algorithm, and financial model, joining peer-to-peer systems and besides, they are utilizing distributed consensus algorithm to tackle conventional distributed database synchronize problem, and it's an incorporated multi-field foundation development. The blockchain at this point is viewed as a distributed ledger that can be available internationally by anybody to verify stored information and data, with high trustworthiness, integrity, validity, and particularly its traceability.

Ethereum [42, 43] would like to solve the quandary between expanding over an officially settled blockchain and

[6] https://www.comodo.com

[7] https://www.godaddy.com

[8] https://www.certum.pl/pl

[9] https://keyserver.pgp.com/vkd/GetWelcomeScreen.event

[10] https://pgp.mit.edu

designing another protocol giving engineers a structure to fabricate a decentralized application over a blockchain that strengthens the entire case mentioned as Turning-complete smart contracts and the meaning of new transaction arrangements and conventions. Smart contracts are client-defined projects that determine rules administering transactions, and that is confirmed by a system of peers (assuming that the fundamental digital money is secure). In examination with traditional financial contracts, smart contracts convey the guarantee of low legitimate and transaction costs and can bring down the bar of entry for clients.

Different high-level languages have been created for Ethereum programming, which is then compiled to produce ASM code. The formally confirmed languages at contract programming are:

1. *Solidity*: a contract-oriented, high-level language whose syntax is similar to that of JavaScript and it is designed to target the Ethereum Virtual Machine (EVM). Solidity is statically typed, supports inheritance, libraries and complex user-defined types among other features. It is supported by Remix, an Ethereum IDE, that offers a runtime environment and a compiler [44, 45].
2. *Serpent*: a python-like language with domain-specific features for contract programming. The Serpent compiler is written in C++ [46].
3. *LLL*: a lisp-like language, meant to be very simple and minimalistic, essentially just a tiny wrapper over coding in ASM directly. It is still supported but used rarely.

3.2 IPFS

Blockchain, however, is a costly mechanism for information storage, particularly for extensive information and digital data. For the productive storage of large information and substance, we propose utilizing the IPFS file system. IPFS represents Inter-Planetary File System [36], which is a distributed, decentralized file system and an architecture to store information and documents with high integrity and resiliency. On a very basic level, IPFS is a peer-to-peer, open source, content addressable, and internationally dispersed file system that can be utilized for storing and sharing an extensive volume of data with high throughput. Our proposed system makes utilization of both blockchain smart contracts and IPFS, whereby the digital data are stored on the IPFS and the IPFS hashes are stored into the blockchain smart contracts to give traceability and authenticity. In particular, the hash produced on storing the archives to IPFS can be stored in the smart contracts viably and the data can be assigned to utilizing the hash. In the event that there is any change in the content of the digital data, the hash changes, to demonstrate that the first and original datum was changed and modified.

We can utilize a customary RDBMS to store information and save just hashes of the transactions to Ethereum. However, utilizing IPFS for storing information has the following advantages:

- *No duplication*
- *Compatibility with Ethereum*
- *Cost effective*
- *Bandwidth efficiency*

For installing and configuring IPFS, the reader can refer to the instructions on its website.[1]

3.3 PAKE

Bitcoin and each blockchain framework depend strongly on digital signatures (DS) [47]. A DS protocol, when connected to a message, can accomplish some objectives as follows:

- *Authentication*: The receiver of a message can trust the sender's identity;
- *Integrity*: The receiver of a message can trust that the content of the message have not been tampered;
- *Non-repudiation*: The sender of a message cannot later deny that message was sent.

Utilizing symmetric cryptography to send secure messages is to some extent unfeasible in light of the fact that both the sender and the receiver must know ahead of time that the secret key is being used to encrypt and then decrypt the message. So key must be exchanged through a safe channel of correspondence before the encrypted message is sent. However, a safe channel can be hard to get or unfeasible.

Public key cryptography, also called asymmetric cryptography, tackles this issue in an exquisite way. With public key cryptography is conceivable to execute both encrypted transmissions and digital signature systems.

- *Encrypted communication*: The recipient of the message shares publicly his public-key. The sender encrypts the message with the recipient's public-key. The encrypted message is sent and only the recipient who is owning the private key, can decrypt it.
- *Digital signature*: The sender of the message shares publicly his public-key. Then he encrypts the hash of the message with his private key and sends both the plain text message and the encrypted hash to the recipient. The receiver then proceeds to decrypt the hash with the public key of the sender. Finally he computes the hash of the received message and compare it with the decrypted hash. If the hashes match, then the message was indeed signed by the sender.

[1] https://ipfs.io/docs/install/

We have proposed two proficient ideal lattice-based schemes for crucial asymmetric interactions which are the ring-based smooth projective hash function (ring-SPHF) and password-based authenticated key exchange protocol (ring-PAKE) [3] that are appropriate for real-world applications like the Web, Internet, and session key agreement between entities in a network. Our recommendations could all be proved secure (in the standard model) in view of the trusted difficulty of the ring-LWE problem such that all elements are from the ideal lattices space and cyclotomic ring R or $R_q = R/qR$ where q is an integer, and the discretized gaussian error distribution is used for the secret sampling from $R^\vee$ which is the dual ideal of the cyclotomic ring R.

Now, our proposed new efficient password-based authenticated key exchange protocol over rings (ring-PAKE) on ideal lattices from ring-SPHF is presented that its structure is an enhancement and improvement of the Katz and Vaikuntanathan's structure [28] and its security is characterized dependent on the standard meaning of security for PAKE [19, 22, 26, 28]. Here, we show a high-level overview of the 3-round ring-PAKE protocol in Fig. 1.

The ring-based dual-style cryptosystem $\sum'$ with associated ring-SPHF is used and an ideal lattice-based CCA-secure encryption system is denoted by $\sum$. In the ring-PAKE, there is a common reference string (CRS) containing of public keys $\vec{a}$, $\vec{a'}$ for $\sum$, $\sum'$, respectively. Moreover, as we know, the ring-SPHF associated with $\vec{a'}$ is shown by:

$$(K, G, \mathbb{H}=\{H_k : X \to R^\vee\}_{k\in K}, S, \alpha : K \times (pR^\vee \times C) \to S).$$

In this paper, we will propose a novel technique for secure key establishment and 2-step password agreement for reliable and secure communications using credible and authentic public-keys. That is, at first, our messaging DApp generates its public and private keys and distributes the public-key based on blockchain smart contract and IPFS. Then, the DApp receives a random part of shared password by an encrypted message based on its authentic public-key from the server and the other part is received by a text message (SMS) which is sent by the server and finally, the DApp integrates the two parts of password together and creates a session key which has a immunity against the passive and active MITM/interception attacks. If the

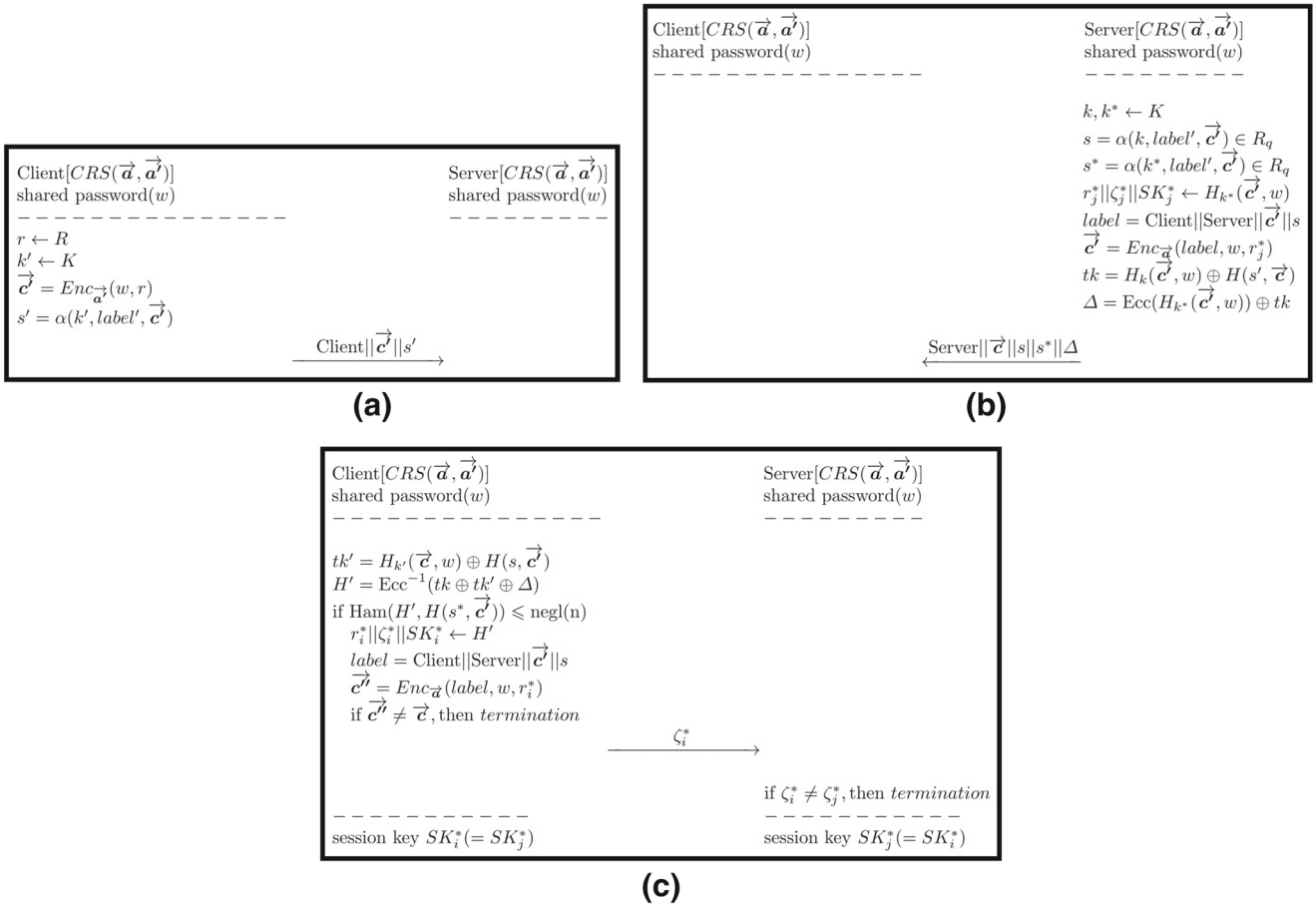

Fig. 1 The ring-PAKE protocol. **a** The first round. **b** The second round. **c** The third round and the generated session key

password wouldn't be correct, the generated session key by this password cannot decrypt the server's messages and the messages encrypted by this session key wouldn't be decrypted in server-side and the server gets noticed that there is an intervention. Therefore, the main issue here is authenticity of public-keys for a secure end-to-end encryption. Notice that the communications in the ring-PAKE's rounds will be done based on this authentic and original public-keys among the DApps and the server. Hence, according to this approach, all of the DApps in the network create the session key for one specific transaction session. An implemented version of a PAKE protocol is presented in [14]. In order to make a balance between performance and security, we aim to build a hybrid encryption shceme using asymmetric and symmetric cryptography based on credible and traceable keys.

3.4 Decentralized App (DApp)

In this section, we will explain the procedure that drove us to the real execution of the decentralized application and its outcomes.

The user-side application can be made out of a progression of web pages produced by the server and rendered by the browser which is utilizing a blend of HTML, CSS and JavaScript language. We can utilize the Bootstrap[2] system as a library for the UI.

The server-side application can be composed of utilizing a conventional LAMP (Linux, Apache, MySQL, PHP) stack as the environment running on the server machine. Alternatively, we can use the MEAN (MongoDB, Express, Angular.js, Node.js) stack which is a decent option for our necessities.

The primary choice we need to make with respect to the development of the Ethereum part of the scheme is picking the smart contract's programming language. We chose to implement a smart contract utilizing Solidity. The principle reason is that Solidity is the most well known high-level language for contracts, thus it has been tried and has the most definite documentation [44]. Another reason is that, while being contract-situated, it has a syntax structure like JavaScript, that we utilized only for the web part.

WebStorm from JetBrains[3] has Solidity through an official module, however, we chose to utilize the Remix as IDE since it was explicitly made for Ethereum smart contracts. The remix is browser-based and offers a Solidity coordinated compiler and a runtime environment without server-side segments. The deployment of a smart contract is also provided by Remix. The development should be possible in a virtual runtime, which permits the trial of straightforward techniques that don't require connection with different contracts, or it may be associated greatly with a Web3 compatible Ethereum client, local, or remote. Remix incorporates a debugger for both Solidity and EVM compiled code and can complete a gauge of the measure of gas expected to execute a transaction call.

Furthermore, we can pick *Geth* as our Ethereum node running on the server. Since testing on the Ethereum mainnet require spending ETHs, which are very costly, so, we implemented our smart contract using the Remix internal runtime.

Etherscan is another web equipment that is utilized for Ethereum interactions. It is a blockchain explorer, it may be utilized so well to see insights concerning the blockchain, similar to addresses, transactions, blocks and tokens, both on the mainnet and on some testnets. A valuable component is that it can likewise demonstrate the code of a particular contract, and can be utilized to interface with it.

Notice that the user, during the delivery phase and the revocation phase, must send Ethereum transaction to the smart contract via the DApp. One method for doing that is to utilize an Ethereum-aware browser, similar to *Mist*, or utilizing a Chrome module, as *Metamsk*,[4] that can get transaction demands from the web application through Web3.

3.5 MITM

In this section, we present a brief overview of Man-in-the-Middle attacks against a secured session in the Internet or a network. For better understanding, we describe the security of Secure Socket Layer (SSL)/Transport Layer Security (TLS) protocol [48] that is similar to a public-key distribution protocol in a DApp. At the center of SSL are various Certificate Authorities (CAs), each of which is in charge of checking the identity of the entities to whom they allow SSL certificates. It is in light of the confidentiality and authenticity given by the CA-based Public Key Infrastructure (PKI) that clients around the globe can bank on the web, participate in e-business and e-commerce, and speak with their friends about the most sensitive objects without hosting to stress over harmful third parties blocking and deciphering their communications. While not totally self-evident, the CAs are altogether trusted similarly in the SSL PKI, an issue which is intensified in a way that the significant internet browsers trust many diverse firms to issue certificates for any site.

SSL was created remembering to give data protection and privacy, but at the same time, this scheme has a few restrictions. Most importantly, the SSL pursues the feeble model of PKI: web-model [49]. Web-model is best for

[2] http://getbootstrap.com/

[3] https://www.jetbrains.com/webstorm/

[4] https://metamask.io/

expansive scale execution for the SSL, however, the issue is that there are 2 sorts of Trust Roots in the web-model: CA, and Client. Here the client can likewise pass judgment on regardless of whether they ought to permit some other certifications that are not in CA. Furthermore, the assaults that the SSL face is significantly from MITM attack, chiefly ARP poisoning, wherein the assailant can capture the verified session and can get the secured accreditations.

In order to protect clients from these powerful attacks, we introduce a lightweight defensive technique using blockchain smart contract.

Any site verified utilizing SSL/TLS can be impersonated utilizing a rebel certificate issued by a maverick CA. This is regardless of which CA issued the site's actual certificate and of any property of that certificate [50].

While a thorough clarification of MITM attacks against SSL is beyond the extent of this article, we give a short introduction to the subject. In the course of recent years, the SSL scheme has been subject to a series of prosperous assaults by security specialists, and some are abusing defects in conveyed frameworks while others made utilization of social engineering and different types of deception [51–55].

Since SSL ensured web associations stream over various other insecure protocols that it is practical for adversaries to capture and hijack a connection with an SSL secured server (these are known as a MITM assault like Fig. 2 (a)). It is just once the browser has gotten and checked a site's SSL certificate that the client can make sure that his connection is secure.

In any case, this progression alone is regularly insufficient to protect clients. Web sites that supply self-signed certificates, or that exploit unpatched vulnerabilities in the certificate code in the browsers can, in any case, trigger the presentation of the SSL lock symbol, yet without giving the client the related security insurances that they would ordinarily anticipate.

Security scientists have repeatedly attacked the SSL-based chain of trust, uncovering misuses that influence both browser configuration imperfections and social engineering assaults against end-clients, for instance, sslsniff [56] and sslstrip [57] toolkits automate the assignment of executing an MITM attack, and when provided with a valid SSL certificate (got by means of a maverick CA for instance), can be utilized to block clients' communications without setting off any browser alerts.

Furthermore, the major type of MITM attacks on the SSL can be either ARP poisoning attack such as algorithm rollback attacks [48], cipher suite rollback attack [58], or fake certificate attacks such as compelled certificate creation attacks [59], etc. The primary issues that we are managing are ARP Poisoning; and Fake Certificate Attack.

In ARP assault, the attacker, above all, endeavors to catch a few packets to know that which one is the gateway which nodes are going to be associated with it [60]. When he finds the victim and the gateway IP addresses, it sends an ARP reply to the victim expressing that the gateway MAC address is now the MAC of the attacker and the same packet expresses to the gateway that the victim's MAC address is presently changed to that of the attacker's MAC. With this assault, the adversary can hijack the session regardless of whether it is secured by SSL/TLS as appeared in Fig. 2 (b). Here the victim contacts the server through the gateway by means of the adversary. The adversary gets every one of the packets that are going through the network and can see everyone's information.

On the other hand, fake certificate attack (Phishing Attack) can be utilized. Phishing [61] is the kind of assault in which the client is compelled to enter clients' qualifications into the phony sites. These sites seem to be like the genuine and valid sites and make the trick to their clients to enter their data. However, in a more modern way, the adversary can hijack the session between the user and the server.

According to the handshake convention of the SSL which underlies dealing with the server wherein, it sends its public-key data hence this data isn't secured. So the adversary can catch this message and change the data in the certificate. The adversary can change the public-key data to that of the assailant and after that sends it to the victim [62, 63].

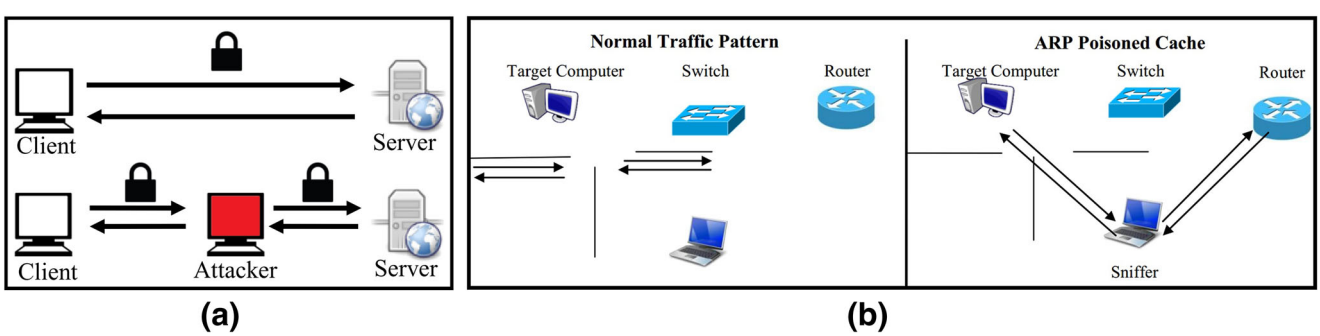

Fig. 2 The interception attacks. **a** The MITM attack. **b** The ARP poisoning

For the victim, it will be a verified connection as he is safely connected to the victim's PC, which thus it is safely associated with the server. This is the principle burden of the PKI model that the SSL underpins. This sort of assault gives a client a warning about the certificate being manufactured and gives the client to either acknowledge the certificate or reject the connection. Numerous clients ignore the warning and push forward with the connection.

4 Problem definition and proposed solution

The computer networks and the digital era have introduced one kind of accessibility to data. Based on the convenience of data sharing, the authenticity of openly posted and distributed digital data is constantly flawed and challenging. As those of as asymmetric and symmetric keys, the authenticity of cryptographic keys is a noteworthy challenge for today's cyberspace users, and cryptographic keys in general. The public-keys which are accessible on the web, can be altered, reproduced, re-distributed, and reformatted during their lifetime. There is a monstrous requirement for a suitable authenticity with the capacity to follow, trace and track the distribution history of posted online material to the original user, node, or network, with a high level of trust, credibility, and integrity.

As a solution, a hardcopy or cold version of a user's public-key can be printed and then scanned for digitizing (like a QR-code) to get utilized in different Apps but again here there is the mentioned challenge and multiple versions of the original public-key which can be printed by different network entities or individuals. That is, the public-key of a user available across printed sheets which can indeed be subjected to illegitimate alteration with the hand-delivered manner by the other users that ultimately leads to tainted information access. Therefore, the other users act as a middle-man. Also, there is a lack of a strict audit to ensure that the public-key is verifiable, complete and accurate. While the public-key is collected and printed from diverse sources, the authenticity and integrity of paper data are at stake. A certain printed version of a public-key cannot be traced back to an original user, as information is usually defragmented and not available to all users to verify and examine the authenticity and originality of published public-key. Hence, each user must deliver his printed public-key by himself to the network users all over the world and this is impossible and not applicable.

Now we get back to the digital world. To date, there exists an absence of authenticity and credibility of digital data made accessible online. Freely posted and distributed online public-keys are not tampered-proof and their production history can't be effectively tracked in a valid, trusted, open, and decentralized way.

The blockchain is a recently developing and emerging innovation that can be a key in giving an answer for the authenticity of digital data. At this point the blockchain is the basic technique of the digital money bitcoin, however, viewed as a distributed ledger that can be taken by anybody to check stored information and substance comprehensively, with high integrity, resiliency, credibility, and above all traceability [37]. The majority of this is done in a decentralized way and without middle-man.

The point of our proposition is to reveal a web application ready to confirm and store on the Ethereum blockchain information like PGP or OpenSSL public-keys, rather than utilizing a conventional key server. Along these lines, because of the immutable properties of the blockchain, when the information is composed on it, we can expect that no adversary will be able to change or remove it. Saving information isn't the main obligation of our proposed application. The other, a considerably vital procedure, is confirming and verifying that data. This implies assessing that the client owns the private key corresponding to the public-key. Above all, this can be accomplished in different ways by having the verification logic on-chain, in a smart contract, it will be the evaluation for the feasibility of our solution using the verification of client submitted data off-chain, on an appropriately configured server.

5 Design

Our strategy depends on utilizing IPFS and smart contracts of Ethereum blockchain. IPFS is utilized to store the digital data (of the public-key or documents) in a decentralized and distributed way that is available freely and all-inclusive by all of the hashes using IPFS. This IPFS hash is utilized by the smart contract of Ethereum blockchain to guarantee honesty, authenticity, and originality. The hash value remains the same if the content of the public-key remains intact. If there is a modification of information during the distribution phases, the IPFS hash for the public-key changes, and then would not coordinate the hash saved inside the smart contract. Along these lines, each party can track back, and check the correctness and history of public-keys being saved in the file system and being guaranteed that the public-key accessed is a genuine duplicate of the client's information.

Figure 3 shows the general framework and architecture for automating the online public-keys credibility, authenticity, and integrity utilizing IPFS and Ethereum smart contracts. The proposed scheme utilizes smart contracts to trigger events that are logged to tell the participating entities to monitor transaction and events' information. The figure illustrates the cooperations of the smart contract with

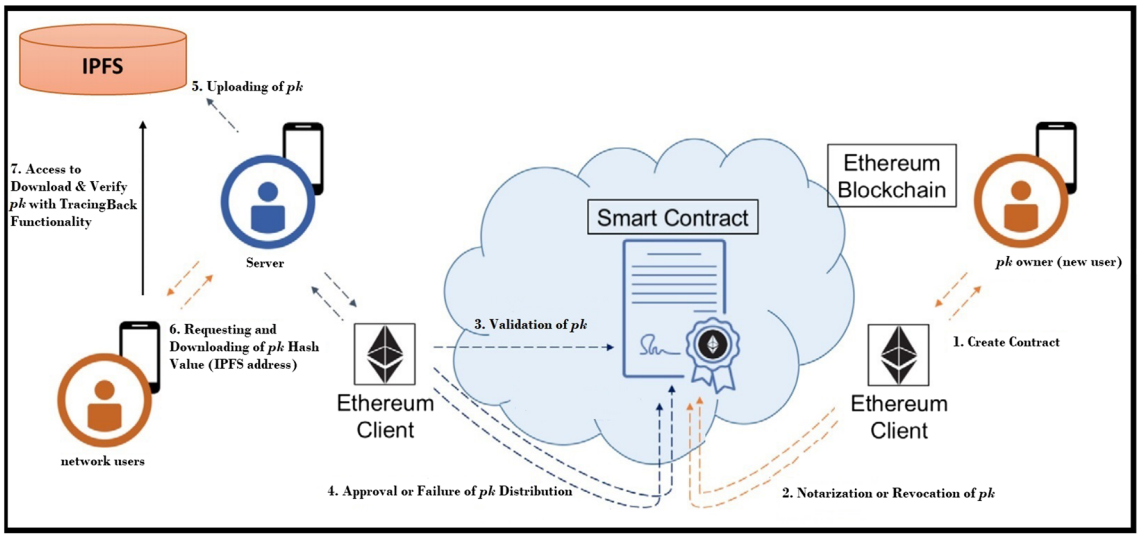

Fig. 3 The architecture and design for automating the authenticity of public-keys (*pk*) based on IPFS and Ethereum smart contract.

principle members that involve the public-key owner, server, and other users.

The members of the smart contract can be clarified as follows:

New user: The new user is the individual who possesses the IP address, and his/her unique and original file of the public-key. The new user creates the smart contract and gives authorization to the server to distribute his public-key. The server is also engaged with approving, validating and notarizing the public-key presented to it by the other network users. The server gets the opportunity to transfer the public-key on IPFS, only if the notarization by the new user is successful. The new user additionally keeps up the first IPFS hash of the public-key. In addition, like the PKI system, each public-key has one expiration date that after this period the user is putting the current public-key and its smart contract's address inside a revocation list, then it generates new public and private keys and creates a new smart contract and sends the list to the server via an encrypted message by a generated session key. Then, the server updates its own revocation list.

Server: The server is the party which acquires permission from the new user to distribute the IPFS hash value of the public-key in different sources, for example, DApps, web

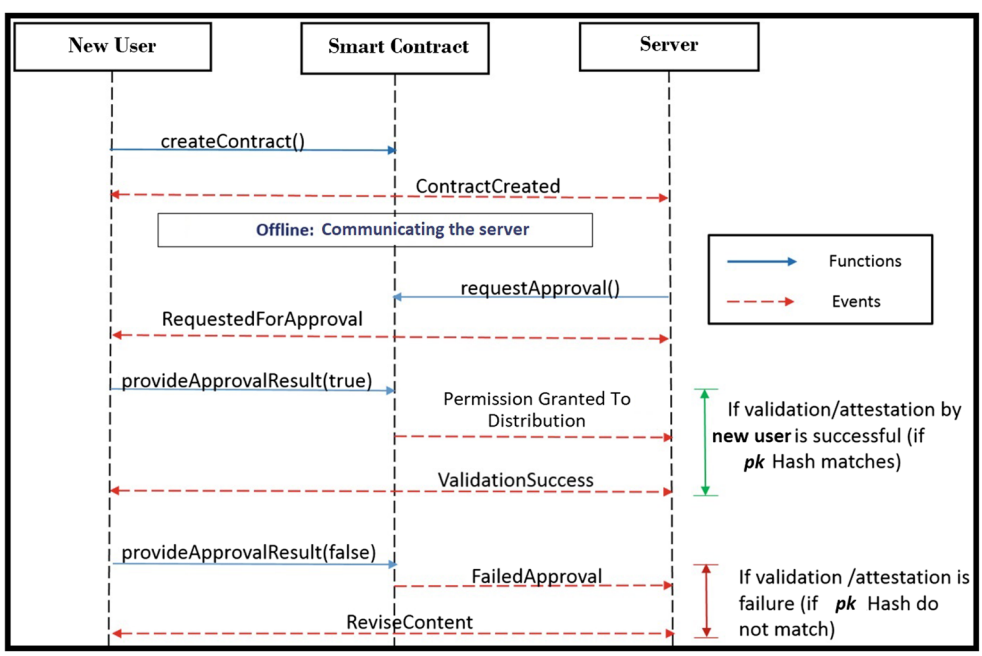

Fig. 4 Message sequence diagram appearing scenarios of successful validation and failed validation

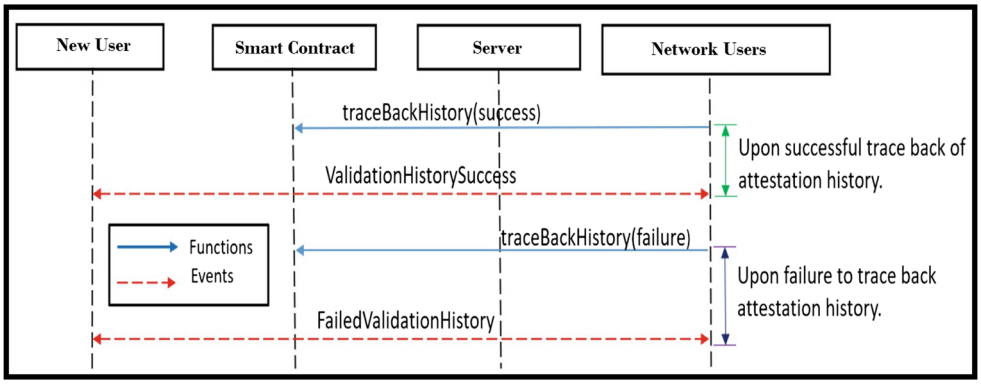

Fig. 5 Message sequence diagram appearing scenarios of successful validation traceback and history of failed validation traceback

pages, and printed sheets. The principle responsibility of a server is to keep up the public-key and digital data intact as reflected by the users. Furthermore, a server transfers the public-key to IPFS and the IPFS hash is stored in the smart contract with the user's verification and validation.

Network users The network users are the individuals who ask for access for genuine public-keys. The smart contract gives the users traceback functionality to check the originality of the public-key. The users can access the historical backdrop of notarizations with respect to the integrity, originality, and authenticity of the public-key.

In our proposed scheme, the new user first generates the public and private keys, creates the contract which incorporates key characteristics about the public-key which are the public-key ID, unique hash of the public-key and new user ID. In an off-chain way, the server looks for permission from the new user after finalizing the checking terms by the user. The server at that point demands approval from the user before transferring the public-key to IPFS, which restores a hash. The new user at that point looks at the stored IPFS record (with the given IPFS hash) to the original data submitted to the server. The new user infers that the public-key is uncorrupted and attests the servers' duplication. The network users have a list of public-keys confirmed by them.

The network users can also ask for data approval history, to know about the authenticity and legitimacy of the new user's public-key. When the public-key is made accessible on the IPFS, a user can ask for a history of validation proof. This can be expected to ensure that the public-key has the notarization of the server from whom the distributing consent was conceded and the confirmation of the original user who approved for the server to distribute the content. The smart contract can show the list of validations for that the public-key alongside the address of the requesting entity, the address of the server who gave access to the public-key and the address of the original user. The validation and approval history helps to track and trace back the history of attestation provided for the public-keys. Furthermore, as mentioned, we will have the revocation phase after expiration of the public-keys and the users create new smart contracts for new keys. As a result, by using our key distribution scheme, we can execute 2-step ring-PAKE's passwords discovery as mentioned in Section 3.3 for generating session keys for secure peer-to-peer communications.

6 Implementation and analyzed scenarios

Our smart contract was executed and analyzed utilizing Remix IDE.[5] Here, we give the inside information of execution and concentrate basically on analyzing the right communication and functionality among network members. Remix IDE offers rich highlights that make it conceivable to analyze and troubleshoot smart contracts preceding by deploying them.

6.1 Implementation details

The code was written in Solidity utilizing the internet browser-based IDE, Remix. There are three substances taking an interest in the contract, new user, server, and network users. Each of the substances has an Ethereum address and can participate by calling functions inside the smart contract at certain timestamps. Figure 4, outlines the message sequence diagram for conceding distributing permission with a successful verification and a failed confirmation by the new user. It demonstrates the communication between the server, the new user and the smart contract.

Figure 5 illustrates the message sequence diagram for successful and failed traceback of validation history between the requesting network users, the server and the new user. Figure 5 outlines the stream of *traceBackHistory*()

[5] http://remix.ethereum.org

```
pragma solidity ^0.4.11;

contract Authenticity_of_PublicKeys  {

address public newUser;
string public pkID;
string IPFShash_newUser;
string newUserID;

enum contractState { NotReady,Created,WaitingForServer,WaitingToProvideApproval, Aborted}
contractState public contState;
enum serverState {ReadyToSubmit, SubmittedForApproval, ValidationSuccess, FailedValidation}
serverState public serveState;

mapping (address=> bool) public recordList; //addresses of server and results (true or false)
mapping (address => bool) public approvedBynewUsers;
mapping(address=>string) public pkHashes; //hashes provided by server
mapping(address=>serverState) public server;

 uint  numberOfRequestsByServer;
 uint numberOfApprovalsBynewUser;

//constructor

function Authenticity_of_PublicKeys(){
    pkID = "publicKey1";
    newUser = msg.sender;
    newUserID = "Amir Hassani Karbasi"; //example
    IPFShash_newUser = "QmaozNR7DZHQK1ZcU9p7QesehCpP4jjFMTpvGamc1p";
    contState = contractState.NotReady;
    numberOfRequestsByServer = 0;
    numberOfApprovalsBynewUser = 0;
}
    //modifiers
     modifier  OnlynewUser(){
        require(msg.sender == newUser);
        _;
    }
    modifier NotnewUser(){
        require(msg.sender != newUser);
        _;
    }

    //events
    event ContractCreated(address networkUser , string info);
    event RequestedForApproval(address server, string info);
    event PermissionGrantedToShare(address newUser, string info);
    event ValidationSuccess(address server, string info);
    event FailedApproval(address newUser, string info);
    event ReviseContent(address server, string info);
    event ValidationHistorySuccess(address server,string info, address newUser);
    event FailedValidationHistory(address server,string info, address newUser);

    //functions
    function createContract()OnlynewUser{
       require(contState == contractState.NotReady);
       contState = contractState.Created;
       ContractCreated(msg.sender, "Waiting for Server...");
       }

  function requestApproval(address serverAddress, string publicKeyHash) NotnewUser  {
   require(contState==contractState.Created && server[serverAddress] == serverState.ReadyToSubmit);
        server[serverAddress] = serverState.SubmittedForApproval;
        contState = contractState.WaitingToProvideApproval;
        pkHashes[serverAddress] = publicKeyHash; //update the mapping
        RequestedForApproval(msg.sender, "Attest and Validate publicKey to proceed for sharing");
        numberOfRequestsByServer += 1;
    }


function provideApprovalResult(address serverAddress) OnlynewUser public {

require(contState==contractState.WaitingToProvideApproval && (server[serverAddress] == serverState.SubmittedForApproval));

  if(keccak256(pkHashes[serverAddress]) == keccak256(IPFShash_newUser)) //compare hashes
  {
      PermissionGrantedToShare(msg.sender, "publicKey Verified by NewUser. ");
      server[serverAddress] = serverState.ValidationSuccess;
      recordList[serverAddress] = true;
      approvedBynewUsers[serverAddress] = true;
      numberOfApprovalsBynewUser += 1;
      ValidationSuccess(serverAddress, "Proceed to share publicKey on IPFS");
  }
  else if(keccak256(pkHashes[serverAddress]) != keccak256(IPFShash_newUser))
  {
      FailedApproval (msg.sender, " Content Modified / Corrupted: Hash does not match . Failed to be approved by newUser");
      recordList[serverAddress] = false;
      server[serverAddress] = serverState.FailedValidation;
      ReviseContent(serverAddress, " Amend publicKey and request for attestation again.");
  }
}
function traceBackHistory(address networkUser ,address serverAddress, address newUserAddress ) public {

 require(contState == contractState.WaitingToProvideApproval && (server[serverAddress] == serverState.ValidationSuccess));
 if(recordList[serverAddress] == true && approvedBynewUsers[serverAddress] == true )
    if(newUserAddress == newUser) {
        //event showing publicKey is verified by the previous address
        ValidationHistorySuccess(serverAddress,"The publicKey is verified by:", newUserAddress);
    }
   else {
            FailedValidationHistory(serverAddress,"The publicKey is Not Verified by:", newUserAddress);
        }
      }
}
```

Fig. 6 Our Ethereum smart contract code for the authenticity of the public-keys

```
mapping (address=> bool) public recordList; //addresses of server and results (true or false)
mapping (address => bool) public approvedBynewUsers;
mapping(address=>string) public pkHashes; //hashes provided by server
mapping(address=>serverState) public server;
```

Fig. 7 Mapping of the server and the public-keys attested by the new user

function and events *ValidationHistorySuccess* and *FailedValidationHistory* on successful and failed traceback history of verifications respectively.

Now, we show the code of our smart contract in Fig. 6 and describe the important code snippets of the smart contract. To track the condition of the server and it's approval results we use mapping, which reflects a key-value pair. We also keep up a mapping to record a list of approvals by the new user and the hash given by the server during the validation procedure. Figure 7 demonstrates mapping where each Ethereum address focuses on the address of the server who presented a demand for approval and hashes given by the server. Figure 7 also illustrates a mapping which comprises of a list of public-keys affirmed by new users and a mapping comprising of the condition of the server.

Figure 8 shows constructor *Authenticity_of_PublicKeys*() which demonstrates that the new user is the owner of the contract. It involves initialization, for example, the *userID*, *pkID* and in particular comprises of the original public-key hash. The condition of the contract at first is *NotReady*. The new user creates the contract and executes the *CreateContract*() function and invokes *ContractCreated* event.

Figure 9 shows the *requestApproval*() function for the server asking for validation from the new user. At this stage, the condition of the contract is *Created* and the server state will be *ReadyToSubmit*. The server state changes to *SubmittedForApproval* and the contract state changes to *WaitingToProvideApproval*. The event *RequestedForApproval* is activated and the server waits for the approval results (True/False) from the new user.

We have considered two scenarios dependent on the approval results that are given by the new user to the server which is asking for a confirmation or attestation before transferring the public-key to IPFS as well as we have considered two other scenarios dependent on the validation history results requested by the network users. The four scenarios are as follows:

1. In the event that the public-key approval results, that is, *provideApprovalResult*() is *"True"*, the transaction is successful, and the smart contract guides the server to transfer the real public-key to the IPFS and the public-key is now freely available for the network users to access.
2. In the event that *provideApprovalResult*() yields *"False"*, the new user does not approve the public-key as the hash submitted by the server while asking for authentication does not match the first hash of the public-key which is saved in the smart contract. The change in hash plainly shows that the first content was changed/altered. The new user triggers an event requesting the server to alter the content according to the first original file and to resubmit for validation.
3. In the event that the authentication history traceback, that is, *traceBackHistory*() is successful, the list of verification made for the public-key is given to asking for entity.
4. Whenever *traceBackHistory*() is a failure, it demonstrates that the accessed public-key isn't approved and subsequently the requesting entity is provided with the detail that the public-key isn't a valid and authentic file of the new user.

Fig. 8 Constructor of The Smart Contract

```
//constructor

function Authenticity_of_PublicKeys(){
    pkID = "publicKey1";
    newUser = msg.sender;
    newUserID = "Amir Hassani Karbasi"; //example
    IPFShash_newUser = "QmaozNR7DZHQK1ZcU9p7QesehCpP4jjFMTpvGamc1p";
    contState = contractState.NotReady;
    numberOfRequestsByServer = 0;
    numberOfApprovalsBynewUser = 0;
}
```

```
function requestApproval(address serverAddress, string publicKeyHash) NotnewUser  {
 require(contState==contractState.Created && server[serverAddress] == serverState.ReadyToSubmit);
      server[serverAddress] = serverState.SubmittedForApproval;
      contState = contractState.WaitingToProvideApproval;
      pkHashes[serverAddress] = publicKeyHash; //update the mapping
      RequestedForApproval(msg.sender, "Attest and Validate publicKey to proceed for sharing");
      numberOfRequestsByServer += 1;
 }
```

Fig. 9 The function for the server asking for verification from the new user

6.2 Analyzed scenarios

In this section, we test the accuracy of the cooperation among members and the right functionality of the general system. Testing of the smart contract guaranteed that the stream of the contract pursued the right sequence, approvals, and refusals of submitted public-key were analyzed accurately and the validation history trackback functionality executed effectively. In this work, the smart contract is analyzed for 4 different scenarios. Figure 10 reflects the different analyzing situations. The first case is on the approval of the new user to distribute the public-key. It is additionally grouped for analyzing two scenarios – Validation success and Failed Validation. The second analysis group shows the network users asking for validation history details. The second case has two scenarios: Successful traceability and Failed traceability. Every one of the scenarios is described in detail in the following subsections.

6.2.1 The scenario 1

At first, we analyze the smart contract for a successful validation scenario. For analyzing reason, we consider the Ethereum address of the new user to be *0xca35b7d915458ef540ade6068dfe2f44e8fa733c*, the address of the server who presents the demand for validation or attestation to be *0x14723a09acff6d2a60dcdf7aa4aff308fddc160c*, and the original IPFS hash of the public-key is *QmaozNR7DZHQK1ZcU9p7QesehCpP4jjFMTpvGamc1p*. We analyzed, as appeared in Fig. 11, the situation when the server asks for validation to sharing the public-key from the new user. The event *RequestedForApproval* is activated and the server waits that the new user to validate its content.

Firstly, we analyze the scenario of successful attestation/validation of the public-key by the new user with the server. Upon successful attestation, events *PermissionGrantedToShare* and *ValidationSuccess* are activated and the server is allowed the consent to transfer the public-key on IPFS. Figure 12 demonstrates the scenario of successful validation gave to the server by the new user.

6.2.2 The scenario 2

Secondly, we analyze the smart contract for the scenario of a failed validation with the server. The validation fails due to change in hash given by the server while asking for attestation to share. The new user rejects approval and asks the server to resubmit the public-key after making the necessary amendment. We analyzed a case, where the server presented an alternate hash - *QmbpyMT7DZHQK1ZcU9p7QesehCpP4jjFMTpvGamc1p*. Figure 13 demonstrates that the approval failed, and the new user suggests for correction of content before resubmitting for validation once more. Events *FailedApproval* and *ReviseContent* are invoked to tell the server to change the public-key content.

6.2.3 The scenario 3

Next, we analyzed for successful traceability of the verification history of the public-key with a network user whose Ethereum address is *0x4b0897b0513fdc7c541b6d9d7e929*

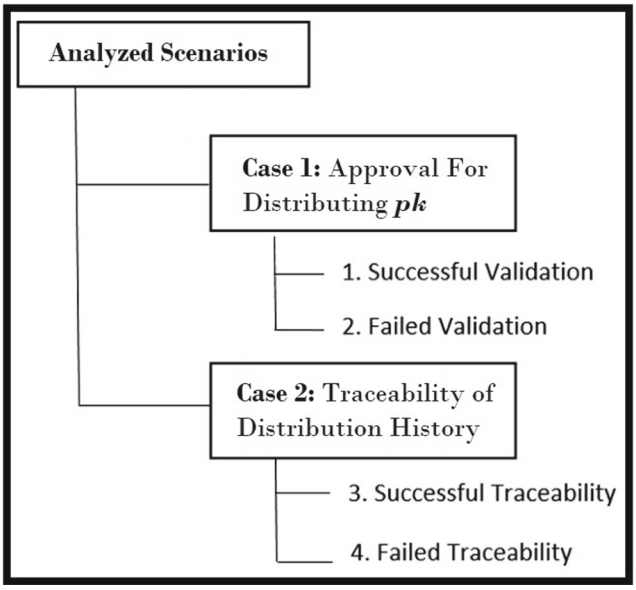

Fig. 10 The smart contract analyzed scenarios for the public-key (*pk*)

status	0x1 Transaction mined and execution succeed
transaction hash	0x876a4ccd5444006e45d40ec06bb1574bbdbd181495e49a1faf0b4d5867878ad0
from	0x14723a09acff6d2a60dcdf7aa4aff308fddc160c
to	Authenticity_of_PublicKeys.requestApproval(address,string) 0x692a70d2e424a56d2c6c27aa97d1a86395877b3a
gas	3000000 gas
transaction cost	136268 gas
execution cost	110260 gas
hash	0x876a4ccd5444006e45d40ec06bb1574bbdbd181495e49a1faf0b4d5867878ad0
input	0x1bd...00000
decoded input	{ "address serverAddress": "0x14723A09ACff6D2A60DcdF7aA4AFf308FDDC160C", "string publicKeyHash": "QmaozNR7DZHQK1ZcU9p7QesehCpP4jjFMTpvGamc1p" }
decoded output	{}
logs	[{ "from": "0x692a70d2e424a56d2c6c27aa97d1a86395877b3a", "topic": "0x0b7bbff0dc5fdbec48d0ad182e48548f4c5606c90668bec82ac980eceaf2f773", "event": "RequestedForApproval", "args": { "0": "0x14723A09ACff6D2A60DcdF7aA4AFf308FDDC160C", "1": "Attest and Validate publicKey to proceed for sharing", "server": "0x14723A09ACff6D2A60DcdF7aA4AFf308FDDC160C", "info": "Attest and Validate publicKey to proceed for sharing", "length": 2 } }]

Fig. 11 Logs indicating the activated event *RequestedForApproval*

c4e5364d2db. The network user requests to demonstrate that the public-key is the original key of the new user and has a notarization proof. Figure 14 reflects the scenario where the validation history of the public-key is acquired effectively showing the address of the server from whom the network user acquired the public-key access, or the address

status	0x1 Transaction mined and execution succeed
transaction hash	0x9f7a861e5b400e88e36aef96cb6bcfe52f0a723c7ef4d9406c38929fc595eeaa
from	0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to	Authenticity_of_PublicKeys.provideApprovalResult(address) 0x692a70d2e424a56d2c6c27aa97d1a86395877b3a
gas	3000000 gas
transaction cost	96493 gas
execution cost	73813 gas
hash	0x9f7a861e5b400e88e36aef96cb6bcfe52f0a723c7ef4d9406c38929fc595eeaa
input	0xcea...c160c
decoded input	{ "address serverAddress": "0x14723A09ACff6D2A60DcdF7aA4AFf308FDDC160C" }
decoded output	{}
logs	[{ "from": "0x692a70d2e424a56d2c6c27aa97d1a86395877b3a", "topic": "0x828b740933d998783e34263a76a0ed7901f702fe2436ce69b07155ff46781cfe", "event": "PermissionGrantedToShare", "args": { "0": "0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c", "1": "publicKey Verified by NewUser. ", "newUser": "0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c", "info": "publicKey Verified by NewUser. ", "length": 2 } }, { "from": "0x692a70d2e424a56d2c6c27aa97d1a86395877b3a", "topic": "0x6a8631a55e9a051559f93898d7bab45981eb82c36fecbc77256bc93af8826dfe", "event": "ValidationSuccess", "args": { "0": "0x14723A09ACff6D2A60DcdF7aA4AFf308FDDC160C", "1": "Proceed to share publicKey on IPFS", "server": "0x14723A09ACff6D2A60DcdF7aA4AFf308FDDC160C", "info": "Proceed to share publicKey on IPFS",

Fig. 12 Logs appearing the successful attestation

status	0x1 Transaction mined and execution succeed
transaction hash	0xbe9b20c96747018f41e08c85db3a883666703076df94040f90723445feb892de
from	0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to	Authenticity_of_PublicKeys.provideApprovalResult(address) 0x08970fed061e7747cd9a38d680a601510cb659fb
gas	3000000 gas
transaction cost	43439 gas
execution cost	20759 gas
hash	0xbe9b20c96747018f41e08c85db3a883666703076df94040f90723445feb892de
input	0xcea...c160c

decoded input	{
	"address serverAddress": "0x14723A09ACff6D2A60DcdF7aA4AFf308FDDC160C"
	}

decoded output	{}

logs	[
	{
	"from": "0x08970fed061e7747cd9a38d680a601510cb659fb",
	"topic": "0x159da0d7bb81fb987a7b3facd641bdbec279ff4b036a257fc585d8ef94d5052c",
	"event": "FailedApproval",
	"args": {
	"0": "0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c",
	"1": " Content Modified / Corrupted: Hash does not match . Failed to be approved by newUser",
	"newUser": "0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c",
	"info": " Content Modified / Corrupted: Hash does not match . Failed to be approved by newUse
	r",
	"length": 2
	}
	},
	{
	"from": "0x08970fed061e7747cd9a38d680a601510cb659fb",
	"topic": "0x78ac3d60d60fbf6cbd0cf9816e237d6eba02853d97d20da8fc8f50094c4dc37f",
	"event": "ReviseContent",
	"args": {
	"0": "0x14723A09ACff6D2A60DcdF7aA4AFf308FDDC160C",
	"1": " Amend publicKey and request for attestation again.",
	"server": "0x14723A09ACff6D2A60DcdF7aA4AFf308FDDC160C",

Fig. 13 Logs indicating the validation failure with activated events *FailedApproval & ReviseContent*

of the real new user. Event *ValidationHistorySuccess* is activated and the smart contract guarantees the network user that the received duplicate is a genuine file. The logs additionally demonstrate that the public-key was at first verified by the new user and the network users can affirm that the received public-key is authentic.

status	0x1 Transaction mined and execution succeed
transaction hash	0xf3e50ff17a5c4fee136c85884b7b9c3627b783f60a2793a0c80a97198d733dcc
from	0x4b0897b0513fdc7c541b6d9d7e929c4e5364d2db
to	Authenticity_of_PublicKeys.traceBackHistory(address,address,address) 0x5e72914535f202659083db3a02c984188fa26e9f
gas	3000000 gas
transaction cost	29765 gas
execution cost	4269 gas
hash	0xf3e50ff17a5c4fee136c85884b7b9c3627b783f60a2793a0c80a97198d733dcc
input	0x668...a733c

decoded input	{
	"address networkUser": "0x4B0897b0513fdC7C541B6d9D7E929C4e5364D2dB",
	"address serverAddress": "0x14723A09ACff6D2A60DcdF7aA4AFf308FDDC160C",
	"address newUserAddress": "0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c"
	}

decoded output	{}

logs	[
	{
	"from": "0x5e72914535f202659083db3a02c984188fa26e9f",
	"topic": "0x86f0ce6b61b086024ebd298eabfc3371cc2856a61f13b81008575c78cf5b0bac",
	"event": "ValidationHistorySuccess",
	"args": {
	"0": "0x14723A09ACff6D2A60DcdF7aA4AFf308FDDC160C",
	"1": "The publicKey is verified by:",
	"2": "0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c",
	"server": "0x14723A09ACff6D2A60DcdF7aA4AFf308FDDC160C",
	"info": "The publicKey is verified by:",
	"newUser": "0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c",
	"length": 3
	}
	}
	]

Fig. 14 Logs indicating the activated event *ValidationHistorySuccess* to demonstrate the content validation history of the public-key

status	0x1 Transaction mined and execution succeed
transaction hash	0x9d2ddcd26a41c66a0aa5261393c8dda31dca5db3912830b1ab2903aeacf8c073
from	0x4b0897b0513fdc7c541b6d9d7e929c4e5364d2db
to	Authenticity_of_PublicKeys.traceBackHistory(address,address,address) 0x9dd1e8169e76a9226b07ab9f85cc20a5e1ed44dd
gas	3000000 gas
transaction cost	30029 gas
execution cost	4533 gas
hash	0x9d2ddcd26a41c66a0aa5261393c8dda31dca5db3912830b1ab2903aeacf8c073
input	0x668...40225
decoded input	{ "address networkUser": "0x4B0897b0513fdC7C541B6d9D7E929C4e5364D2dB", "address serverAddress": "0x14723A09ACff6D2A60DcdF7aA4AFf308FDDC160C", "address newUserAddress": "0x583031D1113aD414F02576BD6afaBfb302140225" }
decoded output	{}
logs	[{ "from": "0x9dd1e8169e76a9226b07ab9f85cc20a5e1ed44dd", "topic": "0xc4d5e89505bf4870181d9a196536a305478516def34a330fb98f6560f48054da", "event": "FailedValidationHistory", "args": { "0": "0x14723A09ACff6D2A60DcdF7aA4AFf308FDDC160C", "1": "The publicKey is Not Verified by:", "2": "0x583031D1113aD414F02576BD6afaBfb302140225", "server": "0x14723A09ACff6D2A60DcdF7aA4AFf308FDDC160C", "info": "The publicKey is Not Verified by:", "newUser": "0x583031D1113aD414F02576BD6afaBfb302140225", "length": 3 } }]

Fig. 15 Logs indicating the invoked event *FailedValidationHistory*

6.2.4 The scenario 4

Finally, we analyze for the scenario of validation traceback failure of the network user with address *0x4b0897b0513fdc7c541b6d9d7e929c4e5364d2db*, who demands for verification history traceback from another user with address *0x583031d1113ad414f02576bd6afabfb302140225*. Figure 15 demonstrates the analyzing results. Event *FailedValidationHistory* is invoked as the attestation isn't finished by the genuine new user. This event shows that the public-key requested by the network user isn't verified by the server to whom the demand for validation history was made.

7 Conclusions

We perceive that the blockchain is a progressive innovation, however, it is still hard to discover use cases outside crypto-finance and notarization of public data. We feel that the reason is that we can't make a blockchain cooperate with this present reality without losing its trustless nature since collaborating with external entities which requires third-parties.

In this work, we have proposed a solution to solve the authenticity and credibility of freely distributed online public-keys using IPFS and Ethereum smart contracts to generate session keys for secure end-to-end encryption in messaging DApps for defeating MITM/Interception attacks. The technique has been analyzed for the four scenarios: successful validation, failed validation, successful traceability, and failed traceability. The system enables clients to confirm and store their public-keys like PGP or OpenSSL public-keys on the public Ethereum blockchain instead of depending on a traditional key server.

We believe that our application can be a useful tool in addition to the Web-of-trust and traditional key servers and we could have realized a progressively autonomous service. In any case, in the field of network security it is better to diminish the number of trusted clients need to put in third-party services, and with the proposed application we make a step forward through that path.

As a future work, we intend to deploy the PAKE protocols and the smart contracts on the genuine IPFS and develop the DApps on the Ethereum mainnet.

References

1. Zimmermann PR (1995) The official PGP User's Guide., MIT Press, Cambridge
2. OpenSSL. http://www.openssl.org
3. Karbasi AH, Atani RE, Atani SE (2019) A new ring-based SPHF and PAKE protocol on ideal lattices. ISC Int J Inf Secur (ISeCure) 11(1):75–86
4. Gentry C (2009) Fully homomorphic encryption using ideal lattices. In: 41st ACM STOC. ACM Press, Bethesda, pp 169–178

5. Lyubashevsky V, Peikert C, Regev O (2013) On ideal lattices and learning with errors over rings. J ACM 60(6):43:1–43:35
6. Lyubashevsky V, Peikert C, Regev O (2013) A toolkit for ring-LWE cryptography. In: Johansson T, Nguyen PQ (eds) Eurocrypt 2013, Vol. 7881 of LNCS. Springer, pp 35–54
7. Cramer R, Shoup V (2002) Universal hash proofs and a paradigm for adaptive chosen ciphertext secure public-key encryption. In: Eurocrypt. Springer Press, Amsterdam, pp 45–64
8. Gennaro R, Lindell Y (2003) A framework for password-based authenticated key exchange. In: Eurocrypt. Springer Press, Warsaw, pp 524–543
9. Abdalla M, Chevalier C, Pointcheval D (2009) Smooth projective hashing for conditionally extractable commitments. In: CRYPTO. Springer Press, Santa Barbara, pp 671–689
10. Blazy O, Pointcheval D, Vergnaud D (2012) Round-optimal privacy-preserving protocols with smooth projective hash functions. In: TCC. Springer Press, Taormina, pp 94–111
11. Katz J, Ostrovsky R, Yung M (2001) Efficient password-authenticated key exchange using human-memorable passwords. In: Eurocrypt. Springer Press, Innsbruck, pp 475–494
12. Bellare M, Pointcheval D, Rogaway P (2000) Authenticated key exchange secure against dictionary attacks. In: Eurocrypt. Springer press, Bruges, pp 139–155
13. SPEKE: RFC5931, RFC6617, IEEE P1363.2, U.S. Patent 6,226,383
14. J-PAKE Implemented in OpenSSL, NSS, used by FIREFOX-SYNC. https://wiki.mozilla.org/Services/KeyExchange
15. Gong L, Lomas TMA, Needham RM, Saltzer JH (1993) Protecting poorly chosen secrets from guessing attacks. IEEE J Sel Area Comm 11(5):648–656
16. Halevi S, cryptography H. Krawczyk. (1999) Public-key cryptography and password protocols. ACM Trans Inf Syst Secur 2(3):230–268
17. Bellovin SM, Merritt M (1992) Encrypted key exchange: Password-based protocols secure against dictionary attacks. In: IEEE Symposium on security and privacy. IEEE Press, pp 72–84
18. MacKenzie PD, Patel S, Swaminathan R (2000) Password-authenticated key exchange based on RSA. In: Asiacrypt. Springer Press, pp 599–613
19. Goldreich O, Lindell Y (2006) Session-key generation using human passwords only. J Cryptol 19(3):241–340
20. Boyko V, MacKenzie PD, Patel S (2000) Provably secure password-authenticated key exchange using Diffie-Hellman. In: Eurocrypt. Springer Press, pp 156–171
21. Nguyen MH, Vadhan S (2008) Simpler session-key generation from short random passwords. J Cryptol 21(1):52–96
22. Benhamouda F, Blazy O, Chevalier C, Pointcheval D, Vergnaud D (2013) New techniques for SPHFs and efficient one-round PAKE protocols. In: Crypto. Springer Press, Santa Barbara, pp 449–475
23. Gennaro R (2008) Faster and shorter passwordauthenticated key exchange. In: TCC. Springer Press, pp 589–606
24. Katz J, MacKenzie PD, Taban G, Gligor VD (2005) Two-server password-only authenticated key exchange. In: 3Rd international conference on applied cryptography and network security (ACNS). Springer Press, pp 1–16
25. Canetti R, Halevi S, Katz J, Lindell Y, MacKenzie PD (2005) Universally composable password-based key exchange. In: Eurocrypt. Springer Press, pp 404–421
26. Gennaro R (2006) Y. Lindell. A framework for password-based authenticated key exchange. ACM Trans Inf Syst Secur 9(2):181–234
27. Jiang S, Gong G (2004) Password based key exchange with mutual authentication. In: 11Th annual international workshop on selected areas in cryptography (SAC). Springer Press, pp 267–279
28. Katz J, Vaikuntanathan V (2009) Smooth projective hashing and password-based authenticated key exchange from lattices. In: Asiacrypt. Springer Press, Tokyo, pp 636–652
29. Rivest RL, Shamir A, Adleman LM (1978) A method for obtaining digital signatures and public-key cryptosystems. Commun ACM 21(2):120–126
30. Diffie W, Hellman ME (1976) New directions in cryptography. IEEE Trans Inf Theory, IT- 22(6):644–654
31. Zhang J, Zhang Z, Ding J, Snook M, Dagdelen O (2015) Authenticated key exchange from ideal lattices. In: Oswald E., Fischlin M (eds) Eurocrypt 2015. LNCS, vol 9057. Springer, Berlin, pp 719–751
32. Ding J, Alsayigh S, Lancrenon J (2017) Provably secure password authenticated key exchange based on RLWE for the Post-Quantum world. CT-RSA, pp 183–204
33. Benhamouda F, Blazy O, Ducas L, Quach W (2018) Hash proof systems over lattices revisited. Public-Key Cryptography (PKC), pp 644–674
34. Travers J, Milgram S (1967) The small world problem. In: Phychology today 1, pp 61–67
35. Penning HP Analysis of the strong set in the PGP web of trust. https://pgp.cs.uu.nl/plot/ (visited on 02/16/2019)
36. Pors M Understanding the IPFS White Paper part 2. url=https://decentralized.blog/ understanding-the-ipfs-white-paper-part-2.html. (visited on 02/16/2019)
37. Nakamoto S Bitcoin: a peer-to-peer electronic cash system. White paper
38. Opara EU, Soluade OA (2015) Straddling the next cyber frontier: the empirical analysis on network security, exploits, and vulnerabilities. Int J Electron Inf Eng 3(1):10–18
39. Singh J (2014) Cyber-attacks in cloud computing: a case study. Int J Electron Inf Eng 1(2):78–87
40. Garay J, Kiayias A, Leonardos N (2015) The bitcoin backbone protocol: analysis and applications. Springer, Berlin, pp 281–310
41. Gervais A, Karame GO, Capkun V, Capkun S (2014) Is bitcoin a decentralized currency? IEEE Secur Privacy 12:54–60
42. Buterin V, et al. (2013) Ethereum white paper
43. Wood G (2014) Ethereum: a secure decentralised generalised transaction ledger. In: Ethereum project yellow paper 151
44. Solidity Language. https://solidity.readthedocs.io/en/develop/ (visited on 02/18/2019)
45. Remix-Solidity IDE. https://remix.readthedocs.io/en/latest/ (visitedon02/18/2019)
46. Serpent Language. https://github.com/ethereum/wiki/wiki/ Serpent (visited on 02/18/2019)
47. Schneier B (2007) Applied cryptography: protocols, algorithms, and source code in C. Wiley, New York
48. McKinley HL (2003) SSL And TLS: a beginners guide SANS institute
49. Huawei Z, Ruixia L (2009) A scheme to improve security of SSL. In: Proceedings of the Pacific-Asia Conference on Circuits, Communications and System, (PACCS' 09)
50. Stevens M, Sotirov A, Appelbaum J, Lenstra A, Molnar D, Osvik DA, Weger B (2009) Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In: Proceedings of the 29th Annual International Cryptology Conference on Advances in Cryptology. Springer, Berlin, pp 55–69
51. Kaminsky D, Patterson ML, Sassaman L (2010) PKI Layer cake: new collision attacks against the global x.509 infrastructure. In: Proceedings of Financial Cryptography and Data Security - 14th International Conference (FC 2010)

52. Sotirov A, Zusman M. Breaking the Security Myths of Extended Validation SSL Certificates. BlackHat USA, 2009. www.blackhat.com/presentations/bhusa-09/SOTIROV/BHUSA09-SotirovAttackExtSSL-SLIDES.pdf (visited on 02/23/2019)
53. Marlinspike M. More Tricks for Defeating SSL in Practice. BlackHat USA, 2009. www.blackhat.com/presentations/bh-usa-09/MARLINSPIKE/BHUSA09-MarlinspikeDefeatSSL-SLIDES.pdf (visited on 02/23/2019)
54. Ray M, Dispensa S. Renegotiating TLS. 2009. https://kryptera.se/Renegotiating%20TLS.pdf (visited on 02/23/2019).
55. Schechter SE, Dhamija R, Ozment A, Fischer I (2007) The emperor's new security indicators. In: SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy, pp 51–65, Washington, DC, USA
56. Marlinspike M. sslsniff, 2009. https://moxie.org/software/sslsniff/ (visited on 02/23/2019)
57. Marlinspike M. sslsniff, 2009. https://moxie.org/software/sslstrip/ (visited on 02/23/2019)
58. Lee Y, Hur S, Won D, Kim S (2009) Cipher suite setting problem of SSL protocol and it's solutions. In: Proceedings of the International Conference on Advanced Information Networking and Applications Workshops, (WAINA '09)
59. Christopher S, Stamm S (2011) Certified lies: Detecting and defeating government interception attacks against SSL (short paper). International Conference on Financial Cryptography and Data Security. Springer, Berlin
60. Wagner R, Bryner J (2006) Address resolution protocol spoofing and MITM attacks SANS institute
61. Joshi Y, Das D, Saha S (2009) Mitigating man in the middle attack over secure sockets layer. In: Proceedings of the International Conference on Internet Multimedia Services Architecture and Applications, (IMSAA '09)
62. Cheng K, Gao M, Guo R (2010) Analysis and research on HTTPS hijacking attacks. In: Proceedings of the Second International Conference Networks Security Wireless Communications and Trusted Computing, (NSWCTC '10)
63. Jiang D, Xinghui L, Hua H. (2011) A Study of Man-in-the-Middle Attack Based on SSL Certificate Interaction. In: Proceedings of the 2011 First International Conference on Instrumentation, Measurement, Computer, Communication and Control, (IMCCC '11)

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Amir Hassani Karbasi was born in Tabriz, Iran. He studied his BSc in Applied Mathematics at the University of Tabriz in Tabriz, Iran. He got his BSc degree in 2010. He was accepted to follow his masters' study at the University of Guilan in Rasht, Iran. He received the MSc degree in Information Technology Engineering in Computer Networks in 2013 and he was accepted to pursue his studies as a full-time Ph.D. candidate at the University of Guilan. He received Ph.D. in 2018 and work on "Design, Analysis, and Implementation of Lattice-based Cryptographic Structures". Now, He is a part-time faculty member at some Universities. His research interests include Cryptology and cybersecurity. For example, Public Key Cryptography, Lattice-based Cryptography, Cryptographic Structures and primitives, Digital Signatures, Digital Certificates, Computer and Network Security, Blockchain, IoT and Wireless Sensor Network Security, Steganography, Application of Algebraic structures to cryptography such as Rings and Semi-Ring Theory, Modules and Semi-Module Theory, Ideal Theory, Finite Fields, Pullback Rings. Moreover, Intrusion Detection Systems, Penetration Tests, E-Commerce Security and ISMS.

Siyamak Shahpasand was born in Tabriz, Iran. He studied his masters' study in computer engineering at the University of Tabriz in Tabriz, Iran.