IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 19, 2019, accepted August 4, 2019, date of publication August 8, 2019, date of current version August 22, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2934049

Decentralized Distribution of PCP Mappings Over Blockchain for End-to-End Secure Direct Communications

ELIE F. KFOURY[1], (Student Member, IEEE), JOSE GOMEZ[1], JORGE CRICHIGNO[1], (Member, IEEE), ELIAS BOU-HARB[2], (Member, IEEE), AND DAVID KHOURY[3], (Member, IEEE)

[1]Integrated Information Technology Department, University of South Carolina, Columbia, SC 29201, USA
[2]Cyber Threat Intelligence Lab, Florida Atlantic University, Boca Raton, FL 33431, USA
[3]Computer Science Department, American University of Science and Technology, Beirut 16-6452, Lebanon

Corresponding author: Elie F. Kfoury (ekfoury@email.sc.edu)

This work was supported by the National Science Foundation under Grant 1822567.

ABSTRACT Network Address Translation (NAT) is a method that enables devices with private IP addresses to connect to the Internet by sharing a public IP address. Traversing the NAT device remains a challenge for a wide range of applications such as Voice over IP (VoIP) and Internet of Things (IoT). The Port Control Protocol (PCP) is a relatively new protocol standardized by the Internet Engineering Task Force (IETF) to solve the NAT traversal issues. It allows a NATed device to request and manage a mapping between its private IP address and transport-layer port to a public IP address and port. As PCP requires an application-dependent method for distributing the mappings to remote hosts, several attacks can target the distributing server and render the communication channel vulnerable. In this paper, we propose and implement a decentralized Blockchain-based approach for distributing PCP-mappings, enabling secure end-to-end (e2e) direct communications without any trusted third party server. NATed devices register their PCP mappings and public keys into the Blockchain, and other peers can then learn about these mappings to establish end-to-end secure direct communications. The implementation verifies that the system is feasible in terms of transactions fees, can simplify and secure end-to-end direct communications, and can interwork with conventional security methods.

INDEX TERMS Blockchain, mapping distribution, NAT traversal, PCP, secure communications, trust model.

I. INTRODUCTION

As the Internet's adoption started to grow dramatically in the late 1980s, researchers and practitioners aimed to develop new approaches to conserve public IPv4 addresses and resolve the addressing exhaustion problem. Consequently, Network Address Translation (NAT) was developed to enable devices with unregistered (private) IP addresses to share a public IP address, and connect to remote devices on the Internet [1]. Although this method solved IPv4 depletion, it introduced other concerns for a wide-range of end-to-end services such as Instant Messaging (IM), Voice over IP (VoIP), online gaming, and IoT applications in traversing the NAT Device [2]. The Internet Engineering Task Force (IETF) has standardized various methods for NAT traversal, namely, Session Traversal Utilities for NAT (STUN) [3], Traversal Using Relays around NAT (TURN) [4], Interactive Connectivity Establishment (ICE) [5], and others [6]. STUN allows NATed devices to discover their IP address and port number on-demand, with the help of a third party trusted server (STUN server) residing on the public Internet. The requirement of a hosted STUN server is not STUN's only limitation, given that its incompatibility with special types of NATs (symmetric NAT) is also an issue. TURN, on the other hand, allows symmetric NAT traversal by providing a relay server (TURN server) to which NATed clients can connect and receive application data through their `Relayed Transport Address`. However, TURN requires high server bandwidth as it relays the entire communication data. ICE simply relies on combining both STUN and TURN; hence, their problems are inherited as well. In 2013, IETF standardized the Port Control Protocol (PCP) as a new

The associate editor coordinating the review of this article and approving it for publication was Dr. Yogachandran Rahulamathavan.

promising attempt to solve the aforementioned NAT traversal problems [7]. In PCP, NATed hosts create their *mappings* and control how incoming packets are forwarded by the NAT device. PCP does not require an externally hosted server for packets' forwarding to internal hosts, however, it requires a mechanism to inform remote hosts about the mappings of the devices. According to RFC 6887 [7], this mechanism is executed in an *application-specific* manner, which again, requires a trusted third party server.

Trust in computational systems has always been a challenging problem. Recently, several incidents and breaches showed that the trust model offered by certificate authorities (CA) could be compromised [8]–[10]. A potential solution for trust is decentralizing the network, removing any intermediary (centralized) server. As the emerging Blockchain technology aims at redefining trust and achieving network decentralization, in this work, we leverage its properties to propose a Blockchain-based solution for distributing devices' PCP-mappings. Moreover, our solution stores the public keys of such devices in the Blockchain for simplifying end-to-end secure connections. In this context, we frame the paper's contributions as follows:

- Providing a novel method for PCP-mappings distribution without any application server.
- Distributing public keys without relying on trusted CAs.
- Simplifying the addressing scheme by assigning logical identities to devices.
- Developing and evaluating the system using Ethereum Blockchain [11] and an SDN (Software Defined Networking)-based implementation of a PCP server.

The rest of the paper is organized as follows: Section II provides a background on PCP and Blockchain. Section III describes the proposed system and its components. Section IV provides an analysis of the system's security. Section V discusses the system's implementation and the obtained results. Section VI pinpoints a few limitations related to the proposed work and offer some remediation methods. Finally, Section VII concludes by pinpointing a number of future work endeavors.

II. BACKGROUND

In this section, we provide a background on both Blockchain and PCP. We also discuss the benefits and the limitations of both technologies.

A. BLOCKCHAIN TECHNOLOGY

Blockchain is an emerging disruptive technology that provides a replicated secure *ledger* in a peer-to-peer network [12]. It was introduced in Satoshi Natakmoto's Bitcoin paper as the technology that solves the cryptocurrencies' double-spending problem without any intermediary third party (banks, servers, etc.) [13]. In a Blockchain network, *nodes* (participating peers) maintain a full copy of the *transactions'* ledger which consists of a sequence of cryptographically linked *blocks*. Each block holds a set of transactions and the *hash* of the previous block as shown in Figure 1.

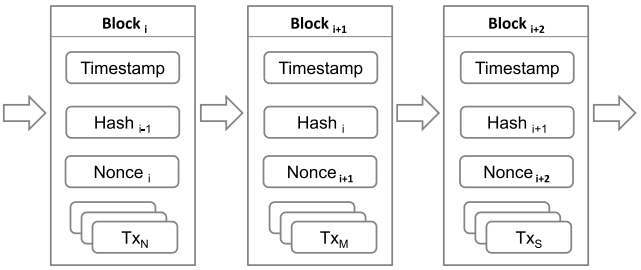

FIGURE 1. General Blockchain blocks sequence.

The nodes apply a *consensus* algorithm such as the Proof-of-Work (PoW) or the Proof-of-Stake (PoS) to agree on the current version of the Blockchain [14]. Hence, the integrity of the stored transactions' data is preserved among all nodes. PoW dictates nodes to compete by solving a computationally intensive mathematical puzzle, while PoS randomly selects a node to create a block, taking into account the wealth and the age (the stake). Users interact with the Blockchain network after creating a wallet, which essentially constitutes of a pair of public/private keys. The private key is used to sign transactions (e.g., transfer digital asset from Account_A to Account_B), while the public key is used to address the wallet and verify its performed transactions by other users.

1) ETHEREUM AND SMART CONTRACTS

While the applicability of Blockchain was originally limited to digital currencies, researchers discovered that this promising technology could be generalized and applied to decentralize a plethora of application system. Consequently, open-source platforms like Ethereum [11] were designed to provide *smart contract* scripting functionality, enabling developers to easily program decentralized applications (DApps) over an abstract layer, without reprogramming Blockchains' logic. Ethereum can be defined as a state machine, initialized with a genesis state, and modified after execution of *transactions* to reach the final state (i.e., accepted version of the Blockchain). Formally, this is expressed as:

$$\sigma_{t+1} \equiv \gamma(\sigma_t, T)$$

where γ is the state transition function that performs arbitrary computation, σ store arbitrary data between transactions, and T is the transaction. Figure 1 shows the general block headers of a Blockchain. Figure 2 shows the particular implementation and additional block headers of Ethereum Blockchain: parent hash, beneficiary, block number, state root, nonce, logs bloom, gas limit, transaction root, timestamp, difficulty, gas used, uncles hash, extra data, mix hash, and receipt root. More detailed information on block headers can be found in Ethereum's yellow paper [11]. According to Buterin [15], the original goal of Ethereum is to offer a high degree of abstraction when developing DApps. Rather than limiting users to a specific set of transaction types and applications, the platform allows anyone to create any kind of Blockchain application by writing a script and uploading

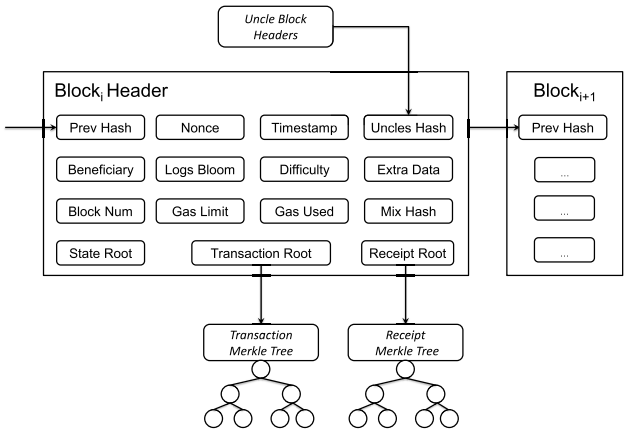

FIGURE 2. Ethereum block header.

it to the Ethereum Blockchain. Therefore, Blockchain components like block headers, block validation, peer-control, rewards, consensus, are DApp-agnostic.

A smart contract can be seen as an autonomous program distributed and executed on top of the Blockchain network [16]. Its operation is enforced by rules expressed as functions and conditional/repetition statements. Once deployed, the smart contract lives forever on the Blockchain without the possibility to modify its rules even by its creator. Each smart contract has an *address* used by other accounts for referencing the contract's instance and for invoking its functions. In Ethereum, smart contracts are Turing-complete, executed as instructions by a virtual machine called the Ethereum Virtual Machine (EVM) on each node. As the EVM instruction's execution consumes nodes' resources, a transaction fee has to be paid by an Externally Owned Account (EOA) invoking the contract's function. `Gas` is the execution fee used to pay Ethereum operations. Potential attacks or expensive infinite loops are prevented by using a field called `GAS_LIMIT`. Gas consumption depends on the calculations done by the miners to execute transactions; the more complex the transaction or the operation, the more gas it would cost. Unlike Bitcoin, a transaction in Ethereum can be created by either an external entity (EOA) or by a smart contract. Optionally, it includes the data field that accepts the smart contract's function encoding and its corresponding parameters.

2) LIGHT CLIENTS

A crucial element in Blockchain systems is the light client. It enables users to access the Blockchain in a secure and decentralized way, without the need to download and synchronize the whole Blockchain. Initially, light clients must download the *block headers* of the Blockchain to fetch data from full nodes (with the support of *Light Client Servers*) in a trustless manner. The Blockchain data, also referred to as `chaindata`, is downloaded once and can be used for all DApps running on the device. *Merkle proofs* are requested along with the data, and subsequently, the clients verify the integrity of the data by checking the path of nodes along a branch in the Merkle tree. More information on Merkle proofs is available in [17]. Ethereum uses the Light Ethereum Subprotocol (LES) to enable a light client to download block headers and fetch data from the Blockchain [18]. LES uses RLPx protocol, a TCP-based transport protocol used for communication among Ethereum nodes.

B. PORT CONTROL PROTOCOL (PCP)

Port Control Protocol is a relatively new protocol aiming to solve the NAT traversal issues [7]. It was standardized by the IETF in 2013 as an improvement for NAT Port Mapping Protocol (NAT-PMP) [19] to enable large-scale deployments, specially in Carrier Grade Networks (CGN). Therefore, PCP works in both residential NATs (small NAT) and operator's NAT (CGN). Moreover, it is compatible with legacy NAT devices, namely, IPv4/IPv4 (NAT44), IPv6/IPv4 (NAT64), and firewalls. An important advantage of PCP is the reduction of `keepalive`'s traffic overhead, as clients are no longer required to generate keepalive messages for maintaining their private/public IP mappings in the NAT device [7]. Consequently, PCP consumes less network resources and bandwidth (e.g., battery consumption for mobile devices and Internet of Things (IoT) devices). Figure 3 demonstrates a network with a PCP-enabled NAT.

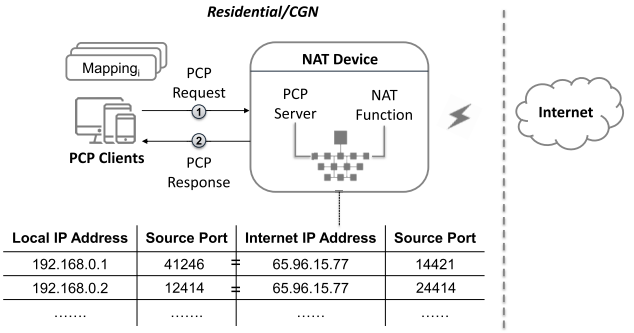

FIGURE 3. PCP-enabled NAT.

1) MECHANISM

The PCP protocol follows a UDP-based request-response approach, where a PCP client sends a request to a PCP server to create and manage its mapping. A PCP client is a software residing in internal NATed hosts, while a PCP server is a software that exists on the NAT device. A PCP request message includes a 7-bit *Opcode* which defines the request type (`MAP`, `PEER`, `ANNOUNCE`). When a NATed client wishes to become publicly available, it sends a MAP request to create an explicit dynamic mapping between its private IP address/port and a public IP address/port. External peers can send traffic destined to the internal host through this mapping. A `PEER` request enables a client to create a dynamic mapping to a remote peer, or to extend an outbound mapping's lifetime. An `ANNOUNCE` request is issued by the PCP server when

it loses its mappings (after reboot, halt, etc.); consequently, clients re-create their mappings with the server to remain publicly available.

2) LIMITATIONS

As PCP requires the transmission of additional messages to manage and create mappings, scalability issues might arise when a large number of PCP clients exist in a network. Another important limitation in PCP is security. Currently, PCP security is being investigated and RFC 7652 [20] is exploring an authentication mechanism. Finally, PCP requires clients and NAT devices to be PCP-aware.

3) ATTACKS AND ADVERSARY MODEL

PCP's RFC 6887 [7] states that the distribution of PCP mappings should be done in an application-specific manner. Therefore, a rendezvous server is required for each application to handle the distribution among peers. For instance, in a VoIP environment, a Session Initiation Protocol (SIP) proxy forwards SIP requests to manage the call control. Figure 4 illustrates a typical PCP distribution attack with a malicious server. After acquiring its mapping from the NAT device, Dev_A stores it in the rendezvous server. When a client on the Internet wishes to access a service hosted on Dev_A, it requests Dev_A's PCP mapping. However, the malicious server sends Dev_B's PCP mapping, leading to a Man-in-the-Middle (MITM) attack.

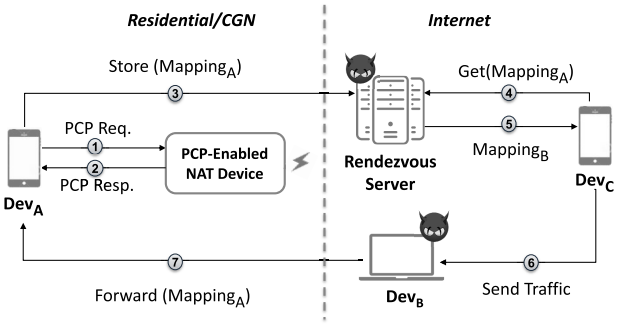

FIGURE 4. Mapping distribution attack scenario.

Moreover, the PCP messages sent from Dev_A to the rendezvous server can be intercepted and altered if the communication channel between these devices is not secured. This scenario will also lead to a MITM attack [6]. RFC 6887 [7] also suggests to use a Domain Name System (DNS)-Based Service Discovery (DNS-SD) [21] for acquiring a remote peer's PCP mapping. However, all attacks targeting a DNS server are also applicable here. These attacks might target the integrity of the stored records and/or the availability of the DNS server itself. The attacks include Cache poisoning [22], TCP SYN floods [23], DNS hijacking [24], Phantom Domain attack [25], etc. In the following section we describe our proposed system and we evaluate how it mitigates all the aforementioned attacks.

III. PROPOSED SYSTEM

In this section, we introduce our proposed system that aims at solving the barriers in existing solutions. In a nutshell, the system enables devices that are behind a PCP-enabled NAT device to publicly register their acquired PCP-mappings into the Blockchain. Other devices can then learn about these PCP-mapping and establish a communication channel with the registered devices. Moreover, devices' public keys are also stored in the Blockchain to enable secure communications without a trusted centralized authority. Due to the data immutability property of Blockchains, PCP-mappings and public keys are protected against alterations by adversaries.

A. ASSUMPTIONS AND DESIGN GOALS

From a practical perspective, we assume that any device can participate in the system and create an unlimited number of Blockchain wallets. We also assume that an adversary can register fake PCP-mappings (i.e., IP:Port that are not controlled by the adversary). However, standard cryptographic assumptions are considered, for instance, signatures cannot be forged without possessing the private key, one-way hash functions cannot be reversed, etc. Moreover, since the system relies on the Blockchain technology, we assume that an adversary does not control more than 51% of the mining nodes. Finally, we assume that a legitimate device, which is one that truly controls a claimed PCP-mapping, is not infected. The system's design goals are the following:

- **Queries' availability**. The system must always be available for devices querying PCP-mappings and public keys. There should be no single point of failure which is currently a major concern in existing name resolution services.
- **Man-in-the-Middle Mitigation**. The system must ensure that an adversary cannot mount a MITM attack and intercept session's traffic data.
- **Authentication and confidentiality**. When starting a communication session with a device, the initiator must verify the authenticity of the destination (i.e., the peer). Additionally, data must be end-to-end (e2e) encrypted between communicating parties, without having any central authority. Mutual authentication should also be considered, but optionally provided to clients.
- **Interworking with current security methods**. The system must be integrated with conventional security methods. This allows a smooth transition and simplifies the deployment in realistic environments.
- **Applicability on a variety of devices**. As PCP is designed to work on almost all types of devices, the system must allow resource-limited devices such as smartphones, Internet of Things (IoT) devices (non-constrained), laptops, etc. to register their mappings.

Figure 5 demonstrates a high-level system architecture and depicts how the system's components interact. The software components of the proposed approach consist of the following:

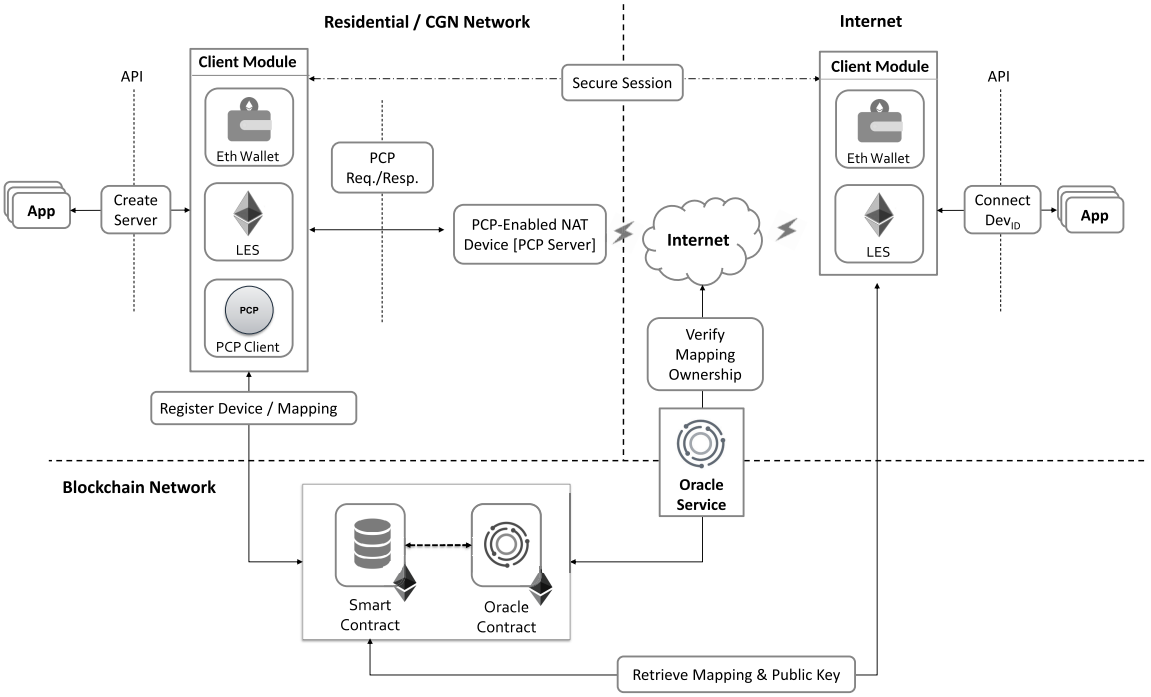

FIGURE 5. Proposed system architecture.

1) PCP-Server: Software residing on the PCP-Enabled NAT device which connects the internal private network to the public Internet. It can be installed on a residential or carrier-grade NAT, and responds to PCP requests issued by the client. In the proposed system, the PCP server performs its functionality without any modification to the standard.

2) Client Module: As depicted in Figure 5, the NATed device consists of a plugin module which comprises of three main software components, namely, Ethereum wallet, light Ethereum client (LES), and a PCP client. The PCP client is not modified and follows the standard specifications of PCP. The main tasks of the client module include device registration, and the mapping storage and retrieval from the Blockchain.

3) Smart Contract: Self-containing autonomous distributed software deployed on the Blockchain network, containing the major processing and storage functions, in addition to their rules.

4) Oracle: A service that enables a DApp to interact with the Internet. Its main role is to verify mappings' ownership by devices. The verification process is described in Section III-E.

The designed system is composed of four different phases: setup, device registration, PCP-mapping storage, and secure session establishment. Each step includes many atomic operations, as discussed in what follows.

B. SETUP

The configuration initiates with installing a plugin software module on an end device (NATed). Upon its first installation, the client module generates an empty Ethereum wallet to connect the device to the public Ethereum network using the LES client. Ethereum uses the Elliptic Curve Digital Signature Algorithm (ECDSA) [26], specifically, the secp256k1 curve. The wallet consists of a private key p_r, a 32-byte sized array in big-endian form ranging from $[1, secp256k1n - 1]$, and a public key p_u, a 64-byte sized array constructed by concatenating two positive integers less than 2^{256}. The wallet address is derived as follows:

$$A(p_r) = B_{160}(Keccak(ecdsa\_pub(p_r))),$$

where $A(p_r)$ is the wallet address generated from the private key p_r, B_{160} represents the rightmost 160-bits, *Keccak* is the Keccak-256 hash function, *ecdsa_pub* derives the 64-byte public key p_u from p_r. A client is required to possess enough Ether to be able to interact with the Blockchain and pay the miners' *gas* fees. Filling the client's wallet with sufficient Ether is oustide the scope of this paper, however, this can be achieved by buying Ether from an external cryptocurrency exchange service or through automated wallet filling providers as described in [27]. Subsequently, the device generates a new key pair (K_u, K_r) using the RSA public key cryptosystem with 2048-bit key size. K_u and K_p represent the public and private key respectively. Then, K_u is stored within a self-signed X.509 certificate $Cert_{self}$. K_u is later used to distribute a symmetric session key K_S with other end device.

C. DEVICE REGISTRATION

After filling the generated wallet with enough Ether to cover the transactions' fees, the client module prompts the user

to specify an identifier Dev_{ID} for the device. Afterwards, the wallet invokes a `constant` function (also referred to as `view`) to query the previous inclusion of Dev_{ID} into the smart contract. As Dev_{ID} uniquely identifies a device, it must not be previously allocated to other device. Since this function call does not modify the state of the Blockchain (i.e., it does not store data), it can be executed without spending Ether to pay gas fees. To this end, the wallet sends a registration transaction T_{reg} that contains the device's identifier Dev_{ID} and the certificate's public key K_u, along with other metadata added into a signed transaction. Ethereum's signed transaction is represented as follows:

$$T_r = \{F, T, G, P, V, D, N\}$$

where,

- F: `from` field which holds the address of the wallet which initiated T_r
- T: `to` field which holds the destination's address.
- G: `gas` field that holds an integer value representing the maximum amount of gas T_r is willing to spend for its execution. This value represents a limit on the total amount of computational steps performed on the transaction.
- P: `gasPrice` field that holds the value for each G. In other words, this is the cost of the computational steps priced in Ether.
- V: `value` field that holds the Ether value to be transferred with T_r to the wallet whose address is T.
- D: `data` field that holds the contract code (for creation) or the encoded function to be invoked with its parameters.
- N: `nonce` field that holds a nonce integer, incrementing by 1 at each transaction performed by an originating wallet, allowing the wallet to overwrite pending T_r.

The device registration transaction T_{reg} is represented as follows:

$$T_{reg} = \{A(p_r), W_{SC}, G_x, P_x, addClient(Dev_{ID}, K_u), N_i\}$$

The `from` field contains the device's wallet address $A(p_r)$, and the `to` field contains the smart contract's address W_{SC}. G_x and P_x are evaluated in Section V. The `data` field holds the hash of the *addClient*() method signature and its encoded parameters (Dev_{ID}, K_u), and N_i is used as T_{reg}'s nonce. Note that no IP address and port number are stored so far as this is only the device registration phase; this is not related to starting a service and publishing its PCP mapping. The requirements needed for the successful storage are 1) the possession of enough Ether, 2) specification of a valid 2048-bit RSA public key and 3) Dev_{ID} is not used by another device. After constructing the transaction, the wallet's private key p_r is used to sign the transaction which is then broadcasted to the Ethereum network. Every signed transaction in Ethereum has a global variable `msg.sender` which is implicitly set, containing the $A(p_r)$ which initiated the transaction. $A(p_r)$ is also stored in the client's registration record in a field called `owner`. This record is stored in a `mapping`– a data structure that is essentially like a hash-table that allows $\theta(1)$ storage and retrieval operations– (not to be confused with PCP-mapping) called *clients*, which is indexed by Dev_{ID}.

D. PCP MAPPING STORAGE

After the device is successfully registered in the smart contract (i.e., there is a record in the smart contract indexed by Dev_{ID}), it can issue other transactions to store as much PCP-mappings as it requires. We allow user-level applications to interact with the client module via an Application Programming Interface (API) to request and store the mapping information. The storage steps are as follows:

1) When an application wishes to host/update a server (i.e., host a publicly addressable service), it sends a PCP-MAP request to the PCP server and acquires a PCP-mapping.
2) If the PCP response is `SUCCESS`, it contacts the client module to store the PCP mapping in the Blockchain. The PCP-mapping to be stored in the smart contract is represented as follows:

$$PCP =< Serv, Proto, IP, Port, Val >$$

where, *Serv* is the service name to be started on this mapping (e.g., HTTP, SIP, etc.). *Proto* is the transport-layer protocol of *Serv* (e.g., TCP, UDP, etc.). *IP*: public IP address portion of the mapping. *Port* is the port number portion of the mapping, used in conjunction with *IP* to form a mapping. *Val* is a boolean value defaulting to false, used to indicate if the requested mapping is validated. Since a device might host several services, we refer to $PCP_i^{Dev_{ID}}$ as the i-th PCP-mapping of device Dev_{ID}. To store a mapping, Dev_{ID} must issue a signed transaction T_{map_reg}, similar to T_{reg}, but with the `data` field containing the *addMapping*() function with encoded parameters matching $PCP_i^{Dev_{ID}}$, excluding *Val*, and including Dev_{ID}.
3) The smart contract verifies that the storage transaction is sent from the wallet used to register the device by asserting the following condition: $T_{map\_reg}(msg.sender) = clients[Dev_{ID}].owner$. If this is not the case, then the transaction is `reverted`, and the mapping won't be stored successfully. Note that an unsuccessful storage transaction still consumes gas (which costs the device), and inherently mitigates against flooding the network with unused identities.

The smart contract then verifies the ownership of the claimed mapping, and approves the storage transaction on success by setting $mappings[Dev_{ID}||Serv].Val = true$, where *mappings* is a mapping data structure that stores all registered PCP mappings. The *addMapping*() function is also used to renew or update an existing mapping because changing the values of a mapping requires re-applying the ownership verification mechanism.

E. PCP MAPPING OWNERSHIP VERIFICATION

To be able to store its PCP-mapping, the device must prove that the claimed mapping ($PCP_i^{Dev_{ID}}$) creates a direct connection to its service. Otherwise, any malicious device can inject fake mappings and flood the network with non-existent mappings, which renders the whole system vulnerable. As a pre-condition for $PCP_i^{Dev_{ID}}$ verification, we assume that the oracle service is not tampering with the data retrieved from outside the Blockchain. Although this might bring centralization back as the oracle is owned by a third party, we show in Section IV how to achieve a high-level of data integrity insurance with minimal trust. In a nutshell, the device hosts a temporary HTTPS server with the self-signed certificate $Cert_{self}$ created during device registration before issuing T_{map_reg}. The client's module facilitates this process by *transparently* creating the HTTPS server and installing the generated certificate upon requesting a new PCP-mapping storage. To verify the PCP-mapping ownership, the smart contract uses a challenge-response method with the help of an Oracle service (*Ora*) to prove that the HTTPS server is hosted on the $< IP : port >$ combination extracted from T_{map_reg}. This verification is somewhat similar to the one used in the recently standardized Automatic Certificate Management Environment (ACME) protocol [28], where a server automatically obtains a browser-trusted certificate, without any human intervention. The challenge-response method illustrated in Figure 6 is detailed in the sequel:

1) After hosting an HTTPS server, the client module issues a PCP-mapping storage transaction $T_{map\_reg}(PCP_i^{Dev_{ID}})$ and listens to the $N1Ready$ event. Events in Ethereum are signals that are fired by the smart contract and pushed to any listening device.

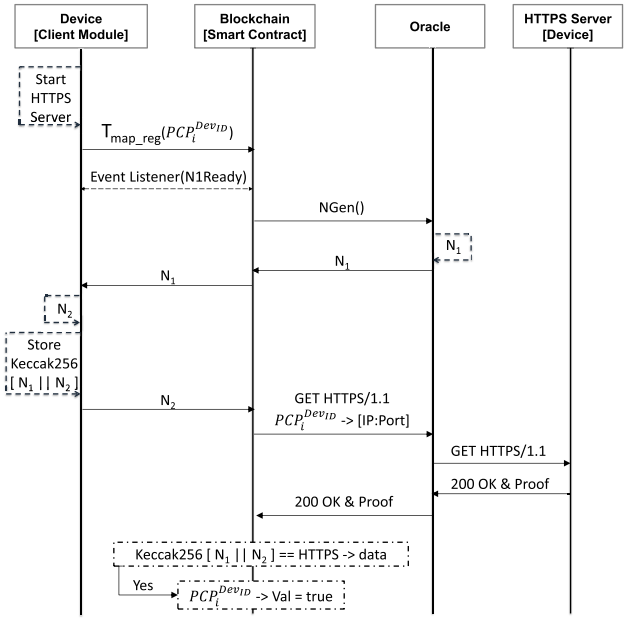

FIGURE 6. Mapping storage validation.

2) The smart contract contacts *Ora* to generate a random number N_1. The generation is not done on the smart contract itself since random number generation is not possible in a deterministic machine– EVM. Note that even if the generated random number is exposed and read by other parties, the security of the system won't be affected. In fact, this number is just used to provide "freshness" for the session, and to make sure that Dev_{ID} will use a fresh value to be stored on it's server's root directory. Moreover, Dev_{ID} can ensure the integrity of N_1 through light client's Merkle proofs, discussed further is Section IV.
3) N_1 is returned to the device through an `event`. As the client module is listening to $N1Ready$, it receives N_1.
4) The device generates a new random number N_2, and calculates $H(N_1||N_2)$, where H is a Keccak-256 hashing function. The device then stores the hashing result $H(N_1||N_2)$ in its web server's (HTTPS) root directory.
5) The device creates a new signed transaction $T_{validate}$ using the same wallet address used with T_{reg}, specifying the contract's address in the `to` field, and N_2 as parameter to the function *validatePCPMapping*().
6) The smart contract *SC* issues an HTTPS request through *Ora* to the web server's root directory hosted on $PCP_i^{Dev_{ID}}[IP : Port]$. It also requests an authenticity proof to verify that the request was truly issued to the requested server. The HTTPS response contains $H(N_1||N_2)$ which was stored previously by the device as payload.
7) *SC* calculates $H(N_1||N_2)$ (where H is a Keccak-256 hash function) using the previously stored values of N_1 and N_2, and compares the hashing result with the returned value of $H(N_1||N_2)$ from the server. If they match, this means that the device is legitimate, and the claimed address is truly owned by the device. Consequently, the mapping storage is *approved* by the smart contract by setting $PCP_i^{Dev_{ID}}[Val]$ to `true`, otherwise, the mapping is considered malicious, and is deleted from the Blockchain.
8) Finally, the client's module shuts down the HTTPS server, and allows the user-application to host its desired server on the requested port.

F. SECURE SESSION ESTABLISHMENT

Besides distributing PCP-mappings, a major goal of this work is to establish end-to-end security among communicating parties without any trusted third party (CA). As described in Section III-C, a public key K_u corresponding to a self-signed certificate $Cert_{self}$ is stored upon registering the device. The secure session setup mechanism is shown in Figure 7. To establish a secure session with a hosted server, the following steps are performed:

1) To determine the PCP-mapping's information of the destination, the initiator invokes the constant function (i.e., function that doesn't change the state of the Blockchain, and therefore, does not consume `gas`)

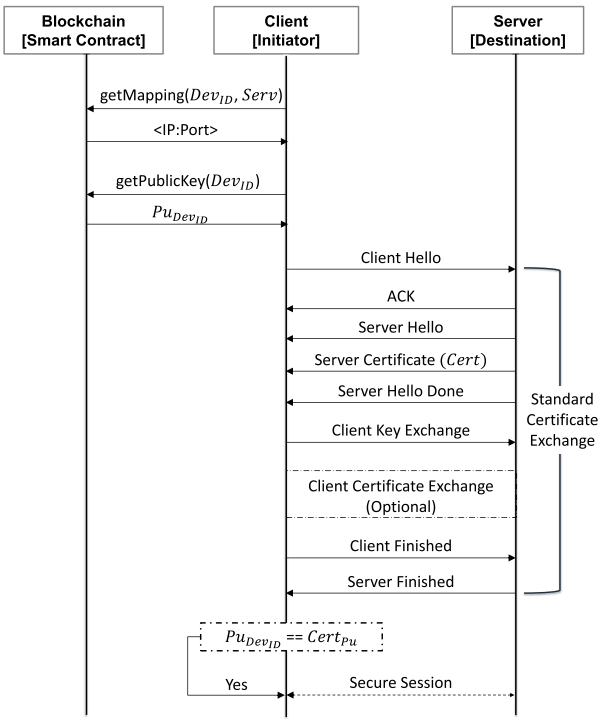

FIGURE 7. Secure session setup.

getMapping, and pass as parameters the destination's Dev_{ID} and its hosted service name.

2) The session initiator retrieves the destination's public key K_u from the smart contract by invoking the constant function *getPublicKey* with Dev_{ID} as parameter. This operation does not change the state of the Blockchain either, and therefore does not consume `gas`. The integrity of K_u is maintained due to the Merkle proof used by light clients.
3) Consequently, the standard client-server certificates exchanged in TLS/CA is applied between the initiator and the server hosted on the device. The client sends a `Client Hello` message to the server containing the TLS version, list of cipher suites, random number, compression methods, etc.
4) The server replies with an `ACK`, and a `Server Hello` message containing a random number, the mutually supported TLS version, cipher suite, compression method, etc.
5) The server also sends its certificate which contains the server's public key. Finally, it sends the `Server Hello Done` message to the client.
6) The client now starts the key exchange mechanism, and both parties computes the master secret key.
7) Then, the client finishes the TLS handshake.
8) Since the server's certificate is not signed by a trusted CA, the client will complain about its validity. To overcome this issue, we allow clients to accept the self-signed certificate (as done in browsers when connecting to a web site with a self-signed certificate). To verify the certificate's validity, the client compares the certificate's public key with K_u which was retrieved earlier from the Blockchain. If they match, the client authenticates the server, and a secure session is established. Note that the client can optionally exchange its certificate for mutual authentication. In this case, the server must also retrieve the client's public key from the Blockchain and compares it with the certificate's public key to authenticate the client.

Consequently, clients can ensure that they are communicating with an authenticated peer and that the self-signed certificate is valid.

G. PUBLIC KEY REVOCATION

Conventional X.509 certificates are revoked whenever they deemed to be no longer trustable. For example, certificates include a validity period that defines their expiration date and time; certificates that have an expired date (i.e., certificate used after the expiration date) are considered not valid, and therefore are not trusted by peers. Another important reason for revocation is having the certificate's encryption keys compromised; in our proposed system, public key revocation can be applied in such case. Essentially, only the owner of the device can update its public key in the Blockchain. This is done by calling the *updatePublicKey*() function which accepts a new public key and the Dev_{ID} as parameters. The smart contract verifies that the update transaction is sent from the wallet used to register the device by asserting the following condition: $T_{update}(msg.sender) = clients[Dev_{ID}].owner$.

IV. SECURITY ANALYSIS

In this section, an analysis of the proposed system's security is provided. We focus on explaining how we can achieve minimal trust when dealing with the oracle service, and how the data retrieval from the Blockchain, particularly with the light client, is secured.

A. ORACLE SECURITY

As discussed in Section III, an Oracle service is a component that connects a smart contract to the public Internet. It often connects Blockchain applications to the publicly hosted web services using a Representational State Transfer (Restful) API (i.e., issue HTTP to GET/POST/PUT/DELETE data). The entities involved in this model are: 1) The *application* (smart contract) that requires data from outside the Blockchain, 2) the *data source* which is the back-end that provides the data publicly through Restful API, and 3) the *Oracle* which is the entity that provides a data transport layer from the *data source* to the *application*. The reason for separating the data source from the Oracle is to enable easy reachability to data source providers without making them adapt to the Blockchain. Although adding an Oracle facilitates data transfer between the Blockchain and the public networks, another problem known as the ''Oracle Problem'' arises; there is a need to *trust* the Oracle for the outside-data integrity, hence, centralization still exists.

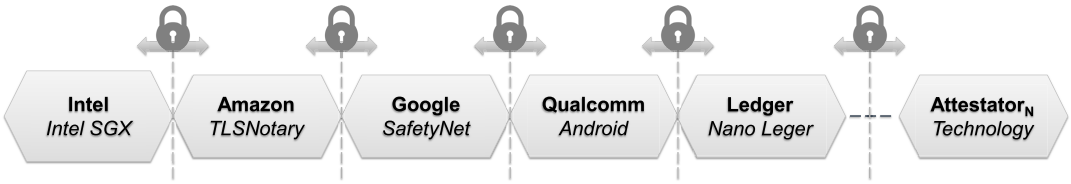

FIGURE 8. Attestators chain of trust.

Several approaches to solve the Oracle problem exist; such approaches provide an "authenticity proof", which is essentially a cryptographic-based evidence that the data is not tampered with, and the data integrity property is maintained. Oracle services send to the smart contract not only the result of the request, but also the authenticity proof, which ideally, can be checked on-chain by the smart contract. The cryptography is based on either software or hardware attestations, or signatures created by the data source (e.g., IETF proposal *"cavage-http-signatures-06"* [29]). Obviously, the most trivial scenario is when the data source signs the data; nevertheless, there exist numerous standards to sign data, and the smart contract should be able to handle all, which is not efficient nor scalable. Software-based techniques used for verification include the TLSNotary proof [30] which is a new digital auditing algorithm that uses cryptographic functions to provide an evidence that certain web traffic have occurred. Hardware-based techniques include the TownCrier project, which leverages trusted computing (Intel SGX) to provide a strong guarantee that the data came from a web service [31]. Other hardware-based techniques include Qualcomm TEE, Android SafetyNet, Ledger Nano S attestation, Samsung Knox, etc.

Each one of the aforementioned techniques is bound to a major company (or the attestator). For example, Oraclize uses the TLSNotary proof method, and stores the secret in an Amazon Web Services (AWS) Virtual Machine (VM). Therefore, you need to trust Amazon for not tampering with the stored secret and faking the authenticity proof. TownCrier uses Intel SGX technology for trusted computing which is designed by Intel, therefore, you need to trust Intel for designing the chip according to what they claim.

Oraclize provides several alternatives for providing authenticity proofs when integrating it with the smart contract. Therefore, it is possible to combine several technologies to minimize trust to a negligible level as shown in Figure 8. If four of the aforementioned technologies were used simultaneously, then the authenticity proof is valid as long as all four parties managing these technologies (e.g., Amazon, Intel, Google and Qualcomm) do not continuously coordinate for every requested proof.

B. LIGHT CLIENT SECURITY

As the Blockchain size is increasingly growing, it is not possible nor practical to download the Blockchain data on all types of clients. Therefore, light clients were introduced to solve this problem: initially, Bitcoin used the Simplified Payment Verification (SPV) method to verify if a particular transaction is included in a block without downloading the entire chain data. Afterwards, other Blockchain platforms like Ethereum started to develop light clients to enable low-powered constrained devices (e.g.: Internet of Things (IoT), mobile devices, etc.) to interact with the Blockchain.

The key element of light clients is the Merkle tree which provides evidence of the inclusion of data in a large dataset without downloading/storing the whole dataset. Merkle trees use extensively one-way hash functions, and require these functions to be collision free. The value of inner node is a one-way function of the values of its children [32]. When retrieving the leaf node's corresponding pairs up to the root, the client can verify the integrity of the leaf's value. Since the light client cannot download all Blockchain's data, it requires interacting with a full node which will provide the block header and the transaction path in the tree. This mechanism is illustrated in Figure 9; to prove that transaction C is in the block, the full node will send C's corresponding pairs denoted in gray– $H(D)$, $H(A, B)$, $H(E, F, G, H)$, and the block header. Having this information, the light node can calculate the Merkle root (starting with $H(C)$, then hashing it with $H(D)$ to get $H(C, D)$, then hashing this value with $H(A, B)$ to get $H(A, B, C, D)$, and finally, hashing the latter with $H(E, F, G, H)$ to get the root (i.e., $H(A, B, C, D, E, F, G, H)$).

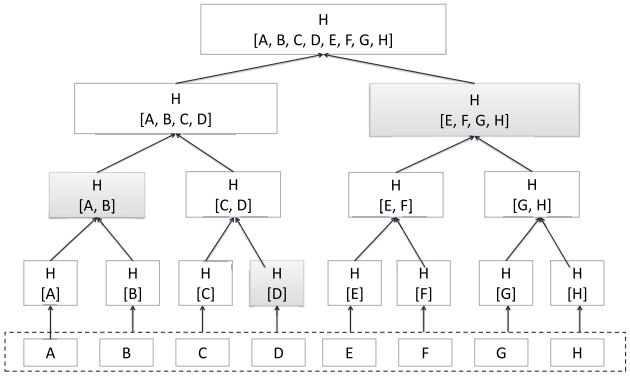

FIGURE 9. Merkle tree.

C. ATTACKS AND ADVERSARY MODEL MITIGATION

In Section II, we listed potential attacks on existing approaches. One possible attack is having a malicious (or hacked) rendezvous server handle the PCP-mappings

distribution. As we used the Blockchain in our approach, there is no central server responsible for informing remote peers about PCP mappings; therefore, the MITM and Denial of Service (DoS) attacks are no longer applicable. Other attacks targeting DNS-based servers, are also not possible in a Blockchain-based system. For instance, DNS Cache poisoning is prevented using the PCP-mapping verification method and the immutable property of Blockchains. DNS amplification, which is a reflection-based DDoS attack, is also not applicable in this system. More recently, Bushart et. al. introduced a clever attack that targets DNS infrastructure by carefully chaining `CNAME` records to force DNS resolvers to perform deep name resolutions [33] and achieve amplification; this is not applicable in this system as we do not allow chaining Dev_{ID} records in the Blockchain. Other attacks, mainly DDoS attacks, are implicitly mitigated due to the distributed nature of the Blockchain network.

V. IMPLEMENTATION AND EVALUATION

For its proper functioning, PCP requires the NAT device and the NATed hosts to be PCP-aware. Integrating a PCP client in a user-application enables the host to issue PCP requests and acquire PCP mappings from the NAT device. This is relatively simple compared to integrating the PCP server in middleboxes as some NAT devices are not PCP-aware. Since PCP is not supported on some existing networking devices, we rely in our implementation on Software Defined Networking (SDN), a new networking paradigm which enables the programming of network devices and increases vendors' devices compatibility [34]. Burda *et al.* [35] succeeded in implementing PCP over SDN, aiming at reducing the keepalives traffic generated in a non-PCP environment. Detailed instructions for running their system is available at [36]. In our proposed system, we intend to re-use their implementation and extend it to support mapping distributions over Blockchain. Figure 10 illustrates the implementation's topology and the interaction between the system's components. The following software components are employed:

- RYU: a component-based software defined networking framework [37].
- ofsoftswitch: OpenFlow software switch [38].
- libpcp: PCP client library [39]. It features a Command Line Interface (CLI) to issue PCP requests.
- Geth light client: Ethereum node implemented in Go language with `--light` option switch activated [18]. This software is installed on the client module.
- Web3j: lightweight, reactive Java and Android library for integrating applications with Ethereum Blockchain. [40]. This library is embedded into user applications that interact with the client module.
- Ropsten Testnet: a testing network for Ethereum Blockchain [41], used by DApps developers before deploying to the main network.
- Solidity: Javascript-based smart contract programming language [42]. Used in the online Remix Integrated Development Environment (IDE). The code listing is available in the Appendix.
- Oraclize: Oracle service that enables smart contracts to issue HTTP requests with TLSNotary proofs [43].
- Bouncy Castle library [44]: Cryptography library for Java and Android.

The mapping registration and session setup steps can be summarized as follows. First, the device registers itself to the smart contract by sending its ID (RPIa) and its public key using a `signed transaction` through the web3j library. Consequently, the PCP client uses libpcp to send a PCP request (1). On `SUCCESS` PCP response, the web3j library initiates a `signed transaction` containing the function name "addMapping", with the desired parameters (DevID: RPIa, serviceName: http, proto: tcp, ip: 198.xx.156.167, port: 2456) (2). If the mapping verification succeeded, DevA can start listening on TCP/2456 HTTP traffic. RPIb can retrieve the IP:port combination by sending a `get transaction`, containing the device name (RPIa) and the service name (HTTP). Finally, RPIb can send HTTTP requests to the retrieved IP:port combination.

A. RESOURCES CONSUMPTION EVALUATION

The above components were installed and configured on a Raspberry Pi III Model B. It uses a Quad Core 1.2GHz Broadcom BCM2837 64bit CPU and has a RAM of 1 Gigabytes (GB).

Figure 10 (a) shows the percentage of CPU and RAM used by the Raspberry Pi while synchronizing the Blockchain using the LES light client on Ropsten testnet. The synchronization process took approximately 30 minutes with relatively heavy processing and RAM consumption. However, this synchronization is done once only upon installing the

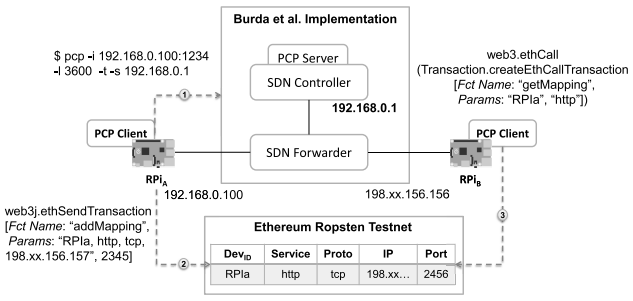

FIGURE 10. Implementation topology.

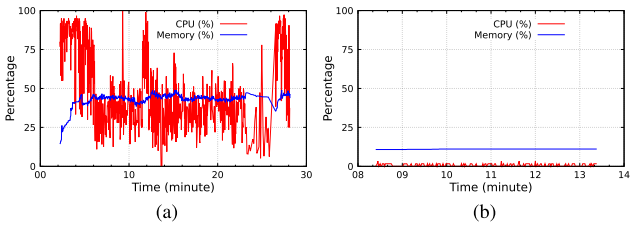

FIGURE 11. CPU and memory percentage during blocks synchronization (a) and after blocks synchronization (b).

TABLE 1. Smart contract's gas usage and fees (Ether and USD) for a $|Dev_{ID}|$= 30 with 1 ether ≈ $252.

Functions		Gas	Ether Value for Oracle	Ether Total	USD
Register Mapping	addMapping()	200500	N/A	0.0040	1.0133
	Generate Random Number (+TLSNotary)	200000	0.000239074	0.004239074	1.0712
	Oracle's Callback				
	validatePCPMapping()	200907	N/A	0.00401814	1.0154
	Send HTTPS (+TLSNotary) in Validation	200000	0.000239074	0.004239074	1.0712
	Oracle's Callback				
	Total	801407	0.000478148	0.016506288	4.1713
Register Device	addClient()	265047	N/A	0.0053	1.3396
Public Key Revocation	updatePublicKey()	89070	N/A	0.0017	0.4498

light client. Another alternative to save time and resources is to have the `lightchain` data ready and deployed on the device's secondary memory; if there is another machine which is trusted and already has the Blockchain's data downloaded, it is possible to export the data from that machine and import it to the current machine. This is particularly important for Internet of Things (IoT) devices as most of the time they are limited in resources. For instance, in a smart city, an administrator can simply copy the lightchain data from an already synchronized machine to the IoT device before installing it in the field. This would eliminate the need of downloading block headers on the IoT device. But of course, it is critically important to trust the synchronized machine otherwise another malicious version of the Blockchain might be used.

On the other hand, Figure 10 (b) shows the CPU/RAM percentage after finishing the block headers synchronization. As the light client did the validations off all previous blocks, the lightchain data now contains the last version of the Blockchain. At this point, negligible CPU/RAM usage are observed on the Raspberry Pi, specifically, the CPU percentage is often less than 5% and the RAM usage is constant at approximately 10%.

B. GAS AND COST EVALUATION

Evaluating a smart contract involves estimating the required `Gas` for executing its transactions [45]–[47]. The required `Gas` value for a transaction depends on the implementation's complexity of the function and the amount of data to be stored in the Blockchain. As the size of data to be stored increases, the Gas value increases. Similarly, heavy processing in a function's implementation results in a dramatic increase of Gas consumption. Table 1 represents the estimated costs in both Ether and USD of the smart contract's tasks execution for a $|Dev_{ID}|$= 30. As of May 2019, 1 Ether ≈ $252. As shown Table I, creating a mapping requires ≈ 0.0165 ETH, which is equal to $4.1713. Registering a device costs 0.0053 ETH, equivalent to $1.3396. Retrieving a PCP-mapping or a public key from the smart contract does not cost Ether as their corresponding functions do not alter or store data in the Blockchain.

Figure 12 demonstrates the Gas requirements for the smart contract's functions versus the functions' total parameters length. The graph shows that the required transaction fees are acceptable even for very large data size (mainly Dev_{ID}). Compared to obtaining a certificate from a trusted CA, the results show that the system offers an affordable solution even on the long term (i.e., when most of the short Dev_{ID}s are consumed).

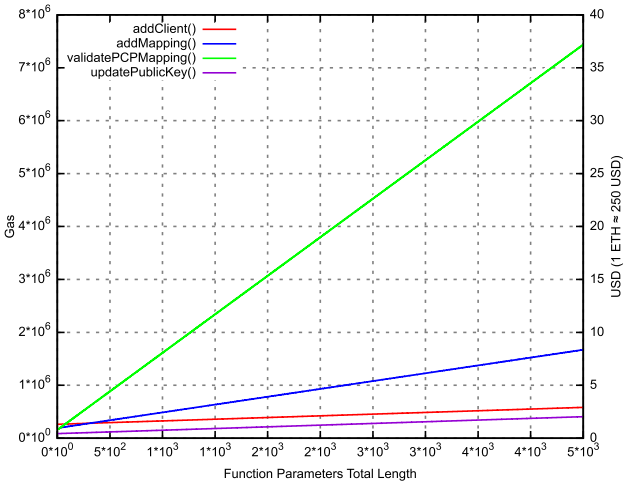

FIGURE 12. Smart contract's function GAS/USD requirements vs. total parameters length.

C. COMMUNICATION LATENCY EVALUATION

The secure session establishment discussed in Section III-F consists of four main phases: 1) retrieving a device's PCP-mapping from the smart contract, 2) retrieving a device's public key from the smart contract, 3) exchanging certificates using the standard TLS certificate exchange mechanism (excluding CA's signature verification), and 4) comparing the certificate's public key against the public key retrieved from the Blockchain. Table 2 shows the average time required to setup a secure session. TCP's handshake (SYN, SYN-ACK, ACK) requires on average 56ms while TLS

TABLE 2. Secure session establishment time in milliseconds (proposed system vs. TLS/CA).

	TCP-Handshake (SYN, SYN-ACK, ACK)	Client/Server Hello/Certificate	ClientKeyExchange ChangeCipherSpec	Retrieve $PCP_i^{Dev_{ID}}$	Retrieve $P_u^{Dev_{ID}}$	Verify $P_u^{Dev_{ID}}$	Total
Standard Certificate Exchange	56	56	56	N/A	N/A	N/A	168
Proposed System	56	56	56	36	36	5	245

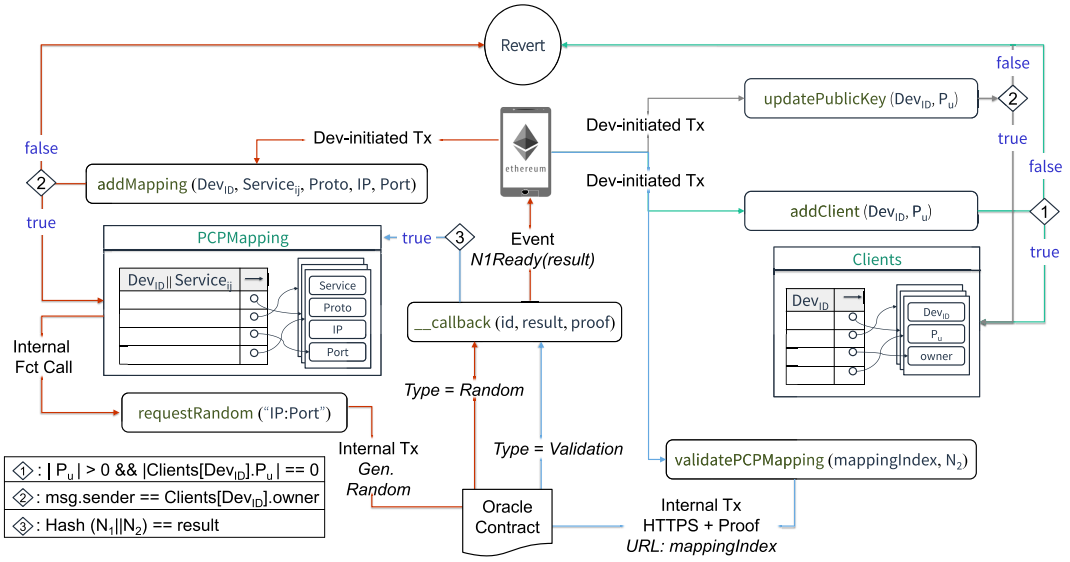

FIGURE 13. Smart contract's data structures and flows.

(Client/Server Hello/Certificate and ClientKeyExchange/ ChangeCipherSpec) requires 112ms [48]. The time needed to retrieve data from the Blockchain through the light client is 36ms; functions were executed 1000 times and the median value was selected. 36ms is spent twice (PCP-mapping retrieval and public key retrieval), and 5ms is spent to extract the certificate's public key and compare it against the one retrieved from the Blockchain. It is worth mentioning that the negligible increase of latency in the proposed system (77ms) is only considered when setting the session; the communication latency afterwards is not affected as the system is not modifying the encryption process.

D. DESIGN GOALS FULFILLMENT

The design goals targeted in Section III-A are fulfilled as follows: 1) Queries' availability is ensured due to the decentralization nature of the Blockchain (i.e., there is no single point of failure as in DNS-based servers). 2) Man-in-the-Middle attack is mitigated because a verification mechanism is executed autonomously in order to store a device's PCP-mapping. Since retrieving data from Blockchain is guaranteed to be secure (see Section IV), an adversary cannot mount an MITM attack. 3) Authentication and confidentiality are ensured by comparing a device's public key stored in the Blockchain with the certificate's public key before starting a secure session (see Section III-F). 4) Interworking with current security methods is also ensured because X.509 certificates are used in conjunction with the Blockchain, which makes all certificate-based secured servers compatible with the proposed system. 5) We evaluated the system on a Raspberry Pi, and we showed that the resources are almost idle when the Blockchain is fully synchronized. This makes the system work on almost all smartphones today, personal computers, non-constrained IoT devices.

VI. LIMITATIONS

In the proposed system, there are two main limitations: As clients' addresses are publicly available in the Blockchain, a malicious user might initiate a DoS attack on the device and render it unavailable. This limitation is not addressed in this paper as it is out of scope. Nevertheless, there are several ongoing research projects to solve the DoS attack using Intrusion Detection/Prevention Systems (IDS/IPS) that can reside on either the host or the NAT device or both [49]. Another extracted limitation is the `gas` fees required to store/update mappings in the Blockchain. This problem is inevitable as clients need to pay the transactions fees for miners. Since PCP mappings can be expired, the device might need to register a newly acquired mapping after timeout, *only if* the mapping's information (IP and port) has changed. When static IP addresses are assigned, PCP-mappings remain permanently unchanged without having to update the state of the Blockchain. When dynamic IP addresses are assigned, fixed devices (devices with no mobility) rarely change their assigned IP addresses, and therefore, the frequency of updating the PCP-mapping in the Blockchain is very low.

For example, a report [50] indicates that the same IP address is used by ISP's customers for multiple years and the average customer has the same IP address for nine months. For mobile devices, the frequency is higher. However, it is possible for customers of an MNO or an ISP to request a static IP for their equipment (residential border router or mobile device).

VII. CONCLUSION AND FUTURE WORK

In this paper, we have proposed a Blockchain-based approach for decentralized distribution of PCP-mappings with end-to-end secure communications. Our aim is to resolve the attacks that occur on third party servers (rendezvous, DNS-SD, etc.) in a PCP-environment, and eliminate the trust concerns by using the consensus algorithms of Blockchains. The solution offers a novel way to establish secure sessions without the need to have a CA-signed certificate for each server. Our implementation which is based on SDN and Ethereum Testnet showed that the solution has minimal impact on the existing network devices while providing security of cost efficiency. For future work, we intend to standardize this system and extend it to act as a complete PKI with automated domain verification without having any centralized third party.

APPENDIX
SMART CONTRACT

```
pragma solidity ^0.4.12;
import "github.com/oraclize/ethereum-api/oraclizeAPI_0.4.sol";

contract PCP_Dist  is usingOraclize {
  enum Query_Type {NULL, RANDOM, VERIFY}
  mapping(string => Client) clients;
  mapping(string => PCPMapping) pcp_mappings;
  mapping(bytes32 => string) Mapping_Identity_Random_QueryID;
  mapping(bytes32 => string) Mapping_Identity_Validation_QueryID;
  mapping(string => string) Mapping_N1_Indices;
  mapping(bytes32 => string) Mapping_N2_QueryID;
  mapping(bytes32 => Query_Type) queriesType;

  event N1Ready(string N1);

  function PCP_Dist() public{
    oraclize_setProof(proofType_TLSNotary | proofStorage_IPFS);
  }
  struct Client{
    string devId;
    bytes publicKey;
    address owner;
  }

  struct PCPMapping{
    string serviceName;
    string protocol;
    string ip;
    string port;
    bool validated;
  }

  function addClient(string _devId, bytes _publicKey)
  payable public {
    require(_publicKey.length > 0);
    if(clients[_devId].publicKey.length == 0)
      //check if client exist
      clients[_devId] = Client(_devId, _publicKey, msg.sender);
    else
      revert();
  }

  function addMapping(string _devId, string _serviceName, string
       _protocol, string _ip, string _port) {
    require(clients[_devId].owner == msg.sender);
    string memory mappingIndex = strConcat (_devId, _serviceName);
    pcp_mappings[mappingIndex] = PCPMapping(_serviceName, _protocol
         , _ip, _port, false);
    string memory addressable = strConcat (_ip, ":", _port);
    requestRandom(addressable);
  }

  function validatePCPMapping (string _mappingIndex, string _N2)
       payable public {
    string memory query=strConcat("json(https://", _mappingIndex);
    query = strConcat(query, "/well-known/pki-validation/");
    query = strConcat(query, _mappingIndex);
    query = strConcat(query, ").val");
    bytes32 id = oraclize_query("URL",query);
    queriesType[id] = Query_Type.VERIFY;
    Mapping_Identity_Validation_QueryID[id] = _mappingIndex;
    Mapping_N2_QueryID[id] = _N2;
  }

  function getPublicKey(string _devId) public constant returns (
       bytes publicKey) {
    return clients[_devId].publicKey;
  }

  function updatePublicKey(string _devId, bytes _publicKey) public
  {
    require(clients[_devId].owner == msg.sender);
    clients[_devId].publicKey = _publicKey;
  }

  function getMapping(string _devId, string _service) public
       constant returns (string map) {
    if(pcp_mappings[strConcat(_devId, _service)].validated)
      return strConcat(pcp_mappings[strConcat(_devId, _service)].ip
           , ":", pcp_mappings[strConcat(_devId, _service)].port);
  }

  function requestRandom(string _addressable)  payable{
    bytes32 myid = oraclize_query("WolframAlpha", "random number
         between 1000 and 9999");
    Mapping_Identity_Random_QueryID[myid] = _addressable;
    queriesType[myid] = Query_Type.RANDOM;
  }

  function __callback(bytes32 myid, string result, bytes proof){
    if (msg.sender != oraclize_cbAddress()) revert();
    if(queriesType[myid] == Query_Type.RANDOM){
       N1Ready(result);
       Mapping_N1_Indices[Mapping_Identity_Random_QueryID[myid]] =
            result;
       queriesType[myid] = Query_Type.NULL;
       delete Mapping_Identity_Random_QueryID[myid];
       return;
    }
    string N1 = Mapping_N1_Indices[
         Mapping_Identity_Validation_QueryID[myid]];
    string N2 = Mapping_N2_QueryID[myid];
    bytes32 res = keccak256(strConcat(N1,N2));
    if(res == fromHexToBytes32(result)){
       pcp_mappings[Mapping_Identity_Validation_QueryID[myid]].
            validated = true;
       delete Mapping_Identity_Validation_QueryID[myid];
       delete Mapping_N2_QueryID[myid];
       delete Mapping_N1_Indices[
            Mapping_Identity_Validation_QueryID[myid]];
       queriesType[myid] = Query_Type.NULL;
    }
  }

  // Convert  an hexadecimal character to their value
  function fromHexChar(uint c) public  returns (uint) {
    if (byte(c) >= byte('0') && byte(c) <= byte('9'))
      return c - uint(byte('0'));
    if (byte(c) >= byte('a') && byte(c) <= byte('f'))
       return 10 + c - uint(byte('a'));
    if (byte(c) >= byte('A') && byte(c) <= byte('F'))
       return 10 + c - uint(byte('A'));
  }

  // Convert an hexadecimal string to raw bytes
  function fromHex(string s) public  returns (bytes) {
    bytes memory ss = bytes(s);
    require(ss.length%2 == 0);
    bytes memory r = new bytes(ss.length/2);
    for (uint i=0; i<ss.length/2; ++i)
```

```
        r[i] = byte(fromHexChar(uint(ss[2*i])) * 16 + fromHexChar(
            uint(ss[2*i+1])));
      return r;
    }

    function fromHexToBytes32(string s) public  returns (bytes32){
      bytes memory b = fromHex(s);
      return bytesToBytes32(b, 0);
    }

    function bytesToBytes32(bytes b, uint offset) private  returns (
        bytes32) {
      bytes32 out;
      for (uint i = 0; i < 32; i++)
        out |= bytes32(b[offset + i] & 0xFF) >> (i * 8);
      return out;
    }
}
```

REFERENCES

[1] P. Srisuresh and K. Egevang, *Traditional IP Network Address Translator (Traditional NAT)*, document RFC 3022, Network Working Group, 2000.

[2] H. Khlifi, J. C. Gregoire, and J. Phillips, "VoIP and NAT/firewalls: Issues, traversal techniques, and a real-world solution," *IEEE Commun. Mag.*, vol. 44, no. 7, pp. 93–99, Jul. 2006.

[3] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing, *Session Traversal Utilities for NAT (STUN)*, document RFC 5389, Network Working Group, 2008.

[4] R. Mahy, P. Matthews, and J. Rosenberg, *Traversal Using Relays Around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)*, document RFC 6062, Internet Engineering Task Force (IETF), 2010.

[5] J. Rosenberg, *Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols*, document RFC 5245, Internet Engineering Task Force (IETF), 2010.

[6] P. Srisuresh, B. Ford, and D. Kegel, *State of Peer-to-Peer (P2P) Communication Across Network Address Translators (NATS)*, document RFC 5128, Network Working Group, 2008.

[7] D. Wing, S. Cheshire, M. Boucadair, R. Penno, and P. Selkirk, *Port Control Protocol (PCP)*, document RFC 7220, Internet Engineering Task Force (IETF), 2013.

[8] D. Fisher. (2012). *Final Report on Diginotar Hack Shows Total Compromise of CA Servers. Threatpost*. Accessed: Jul. 9, 2019. [Online]. Available: https://threatpost.com/final-report-diginotar-hack-shows-total-compromise-ca-servers-103112/77170/

[9] N. Leavitt, "Internet security under attack: The undermining of digital certificates," *Computer*, vol. 44, no. 12, pp. 17–20, Dec. 2011.

[10] S. Gangan, "A review of man-in-the-middle attacks," Apr. 2015, *arXiv:1504.02115*. [Online]. Available: https://arxiv.org/abs/1504.02115

[11] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," Yellow Paper, 2014. Accessed: Jul. 2, 2019. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[12] S. Underwood, "Blockchain beyond Bitcoin," *Commun. ACM*, vol. 59, no. 11, pp. 15–17, 2016.

[13] S. Nakamoto. (2008). *BitCoin: A Peer-to-Peer Electronic Cash System*. Accessed: Jul. 4, 2019. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[14] D. K. Tosh, S. Shetty, X. Liang, C. Kamhoua, and L. Njilla, "Consensus protocols for blockchain-based data provenance: Challenges and opportunities," in *Proc. IEEE 8th Annu. Ubiquitous Comput., Electron. Mobile Commun. Conf. (UEMCON)*, Oct. 2017, pp. 469–474.

[15] (2015). *On Abstraction, Ethereum Blog*. [Online]. Available: https://blog.ethereum.org/2015/07/05/on-abstraction/

[16] S. Omohundro, "Cryptocurrencies, smart contracts, and artificial intelligence," *AI Matters*, vol. 1, no. 2, pp. 19–21, 2014.

[17] R. C. Merkle, "Method of providing digital signatures," U.S. Patent 4 309 569 A, May 1, 1982.

[18] V. Buterin. (2016). *Ethereum Light Client Protocol*. Accessed: Jul. 9, 2019. [Online]. Available: https://github.com/ethereum/wiki/wiki/Light-client-protocol

[19] S. Cheshire and M. Krochmal, *NAT Port Mapping Protocol (NAT-PMP)*, document RFC 6886, Independent Submission, 2013.

[20] T. Reddy, S. Hartman, and D. Zhang, *Port Control Protocol (PCP) Authentication Mechanism*, document RFC 7652, Internet Engineering Task Force (IETF), 2015.

[21] S. Cheshire and M. Krochmal, *DNS-Based Service Discovery*, document RFC 6763, Internet Engineering Task Force (IETF), 2013.

[22] S. Son and V. Shmatikov, "The hitchhiker's guide to DNS cache poisoning," in *Proc. Int. Conf. Secur. Privacy Commun. Syst.* Berlin, Germany: Springer, 2010, pp. 466–483.

[23] W. Eddy, "TCP SYN flooding attacks and common mitigations," Tech. Rep., 2007.

[24] A. Dinaburg, "Bitsquatting: DNS hijacking without exploitation," in *Proc. BlackHat Secur.*, Las Vegas, NV, USA, 2011.

[25] K. Schomp, T. Callahan, M. Rabinovich, and M. Allman, "Assessing DNS vulnerability to record injection," in *Proc. Int. Conf. Passive Act. Netw. Meas.* Los Angeles, CA, USA: Springer, 2014, pp. 214–223.

[26] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ECDSA)," *Int. J. Inf. Secur.*, vol. 1, no. 1, pp. 36–63, Aug. 2001.

[27] E. F. Kfoury and D. J. Khoury, "Secure end-to-end VoLTE based on ethereum blockchain," in *Proc. 41st Int. Conf. Telecommun. Signal Process. (TSP)*, Jul. 2018, pp. 1–5.

[28] R. Barnes, J. Hoffman-Andrews, D. McCarney, and J. Kasten, *Automatic Certificate Management Environment (ACME)*, document RFC 8555, Internet Engineering Task Force (IETF), 2019.

[29] M. Cavage and M. Sporny, *Signing HTTP Messages, Version 11*, document, Network Working Group, Internet-Draft, 2018.

[30] (2014). *Tlsnotary—A Mechanism for Independently Audited Https Sessions*. [Online]. Available: https://tlsnotary.org/TLSNotary.pdf

[31] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town crier: An authenticated data feed for smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 270–282.

[32] E. F. Jesus, V. R. L. Chicarino, C. V. N. de Albuquerque, and A. A. de A. Rocha, "A survey of how to use blockchain to secure Internet of Things and the stalker attack," *Secur. Commun. Netw.*, vol. 2018, Apr. 2018, Art. no. 9675050.

[33] J. Bushart and C. Rossow, "DNS unchained: Amplified application-layer DoS attacks against DNS authoritatives," in *Proc. Int. Symp. Res. Attacks, Intrusions, Defenses*. Crete, Greece: Springer, 2018, pp. 139–160.

[34] N. McKeown, "Software-defined networking," *INFOCOM Keynote Talk*, vol. 17, no. 2, pp. 30–32, 2009.

[35] K. Burda, M. Nagy, and I. Kotuliak, "Reducing keepalive traffic in software-defined mobile networks with port control protocol," in *Information and Communication Technology*. Daejeon, South Korea: Springer, 2015, pp. 3–12.

[36] K. Burda. (2015). *Port Control Protocol in Software Defined Networks*. [Online]. Available: https://github.com/unifycore/pcp-sdn

[37] (2016). *RYU SDN Framework*. [Online]. Available: https://github.com/osrg/ryu

[38] (2014). *Openflow 1.3 Software Switch*. [Online]. Available: https://github.com/unifycore/ofsoftswitch13

[39] (2018). *PCP Client Library*. [Online]. Available: https://github.com/libpcp/pcp

[40] C. Svensson. *Web3j—Lightweight Java Library for Integration With Ethereum Clients*. Accessed: Aug. 1, 2019. [Online]. Available: https://github.com/web3j/web3j

[41] K. Iyer and C. Dannen, "The ethereum development environment," in *Building Games with Ethereum Smart Contracts*. Berkeley, CA, USA: Springer, 2018, pp. 19–36.

[42] C. Dannen, *Introducing Ethereum and Solidity*. Berkeley, CA, USA: Springer, 2017.

[43] T. Bertani. (2016). *Understanding Oracles*. Accessed: Jul. 1, 2019. [Online]. Available: https://medium.com/oraclize/understanding-oracles-99055c9c9f7b

[44] *Bouncy Castle Crypto APIs*. Accessed: Jul. 15, 2019. [Online]. Available: https://www.bouncycastle.org/

[45] K. Toyoda, P. T. Mathiopoulos, I. Sasase, and T. Ohtsuki, "A novel blockchain-based product ownership management system (POMS) for anti-counterfeits in the post supply chain," *IEEE Access*, vol. 5, pp. 17465–17477, 2017.

[46] H. R. Hasan and K. Salah, "Proof of delivery of digital assets using blockchain and smart contracts," *IEEE Access*, vol. 6, pp. 65439–65448, 2018.

[47] S. Wang, Y. Zhang, and Y. Zhang, "A blockchain-based framework for data sharing with fine-grained access control in decentralized storage systems," *IEEE Access*, vol. 6, pp. 38437–38450, 2018.

[48] I. Grigorik, *High Performance Browser Networking: What Every Web Developer Should Know About Networking and Web Performance*. Newton, MA, USA: O'Reilly Media, 2013.

[49] A. Fuchsberger, "Intrusion detection systems and intrusion prevention systems," *Inf. Secur. Tech. Rep.*, vol. 10, no. 3, pp. 134–139, 2005.
[50] *E3 Marketing Group, How Often Do Ip Addresses Change?* Accessed: Jul. 22, 2019. [Online]. Available: https://e3marketinggroup.com/digital-bullseye-blog/2018/1/9/how-often-do-ip-addresses-change

ELIE F. KFOURY is currently pursuing the Ph.D. degree in computer science with the University of South Carolina, USA. For the past three years, he was a Research and Teaching Assistant with the Computer Science and ICT Department, American University of Science and Technology, Beirut. He has published several research papers and articles in international conferences and peer-reviewed journals. His research interests include telecommunications, network security, blockchain, the Internet of Things (IoT), software-defined networking (SDN), and data plane programming.

JOSE GOMEZ is currently pursuing the Ph.D. degree in computer engineering with the University of South Carolina, USA. For the last three years, he was a Researcher and a Teaching Assistant with the School of Engineering, Catholic University, Asuncion.

JORGE CRICHIGNO received the bachelor's degree in electrical engineering from the Catholic University of Paraguay, in 2004, and the Ph.D. degree in computer engineering from the University of New Mexico, in 2009. He is an Associate Professor with the College of Engineering and Computing, University of South Carolina (USC). He has over 15 years of experience in academic and industry sectors. His work was supported by Google, the National Science Foundation (NSF), and the U.S. Department of Energy. His research interests include practical implementation of high-speed networks, network security, TCP optimization, the experimental evaluation of congestion control algorithms tailored for friction-free environments, the IoT security, and flow-based intrusion detection systems.

ELIAS BOU-HARB received the Ph.D. degree in computer science from Concordia University, Montreal, Canada. He is currently an Assistant Professor with the Computer Science Department, Florida Atlantic University. Previously, he was a Visiting Research Scientist with Carnegie Mellon University. He is also a Research Scientist with the National Cyber Forensic and Training Alliance (NCFTA), Canada. His research and development activities and interests include the broad areas of operational cyber security, including attacks detection and characterization, the Internet measurement, cyber security for critical infrastructure, and mobile network security.

DAVID KHOURY received the M.E. degree in telecommunications from ESIB, in 1983. He has more than 30+ years of experience in the field of telecommunications and technology. He held different positions at Matra and Ericsson mainly in France and Sweden in Research and Development, and Product and System management. He created and led a group to develop a generic ISDN platform in the Ericsson main exchange AXE. He was involved in early studies of the GSM and the evolution toward an IP-based network and the early studies of 3G/WCDMA, HSPA, and LTE. In 2005, he became a Technology and Business Consultant of the sales unit of the Middle East and Africa region driving new business opportunities and introducing new systems. In 2010, he has established his own start-up company (Secumobi) developing advanced military grade secure communications system and security solutions based on Ethereum blockchain and backed by hardware encryption and trusted execution environments (TEE) in Stockholm. The blockchain-based secure communications platform has received the Second Prize of the Ericsson Garage Startup Challenge among 189 startups in Stockholm, in 2017. For the past five years, he was a full-time Instructor and a Researcher with the Computer Science Department, American University of Science and Technology (AUST), Beirut. He holds four U.S. patents. He has published several research papers in international and local conferences. His research interests include the IoT, information security, and blockchain technology.