IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received December 9, 2020, accepted December 22, 2020, date of publication December 25, 2020, date of current version January 7, 2021.

Digital Object Identifier 10.1109/ACCESS.2020.3047413

BorderChain: Blockchain-Based Access Control Framework for the Internet of Things Endpoint

YUSTUS EKO OKTIAN AND SANG-GON LEE
College of Software Convergence, Dongseo University, Busan 47011, South Korea

Corresponding author: Sang-Gon Lee (nok60@dongseo.ac.kr)

This work was supported in part by the Basic Science Research Program through the National Research Foundation of Korea, Ministry of Education, under Grant 2018R1D1A1B07047601, and in part by Institute for Information and Communications Technology Promotion, Korea government (MSIT) (development of prevention technology against AI dysfunction induced by deception attack), under Grant 2018-0-00245.

ABSTRACT The Internet of Things (IoT) providers serve better IoT services each year while producing more IoT gateways and devices to expand their services. However, the security of the IoT ecosystem remains an afterthought for most IoT providers. This action results in many cybersecurity breaches in the field, most likely due to the lack of access control mechanisms. In this paper, we propose BorderChain, an access control framework based on blockchain for IoT endpoints. The security protocol guarantees two properties. First, our proposal assures IoT users and services that they communicate with approved IoT gateways as endpoints, holding verified IoT devices that they need. Second, BorderChain also generates access tokens that the IoT service and users can use to query IoT resources legitimately inside the IoT domains. As a result, the protocol can convince IoT domain owners that the system will only authorize IoT requests that they approve. We realize our protocol in the form of a smart contract to allow many IoT entities such as IoT domain owners, IoT devices, IoT gateways, IoT vendors, IoT services, IoT users, and Internet Service Provider (ISP) to collaborate in a unified environment. We then implement entities in BorderChain as Node JS applications connecting to the Ethereum blockchain as our peer-to-peer platform. Based on our performance evaluation using several Raspberry Pi hardware and our private server, we show that BorderChain can process entities' authentication and authorization requests efficiently using all hardware resources. Finally, we release BorderChain for public use.

INDEX TERMS Blockchain, smart contract, access control, IoT, endpoint.

I. INTRODUCTION

The Internet of Things (IoT) is growing, in both quality and quantity, and has helped us to live a better life each day. For example, IoT can guide us to reach new places that we have never visited before, remind us to pick up our umbrellas when the day is going to rain, and drive us safely with sophisticated car safety mechanisms. The possible use cases are fastly spreading as researchers are developing IoT applications in many sectors. This growth has introduced a lot of new devices to connect to the Internet. Gartner estimates that 14.2 billion things were working actively in 2019, and the number will keep increasing and reach 25 billion by 2021 [1]. Moreover, Gartner predicts that the number of IoT endpoints will reach 5.8 billion in 2020 [2].

The associate editor coordinating the review of this manuscript and approving it for publication was Theofanis P. Raptis.

While IoT technology is ostensibly ameliorating our living standards, it is also becoming a platform for attackers to attack our life. Specifically, the presence of bulk IoT devices in the network is a sweet spot for attackers to blend in and do malicious activities. For starters, a hacker can steal sensitive data from NASA using a cheap and portable Raspberry Pi device [3]. Furthermore, attackers can also compromise mass IoT devices to launch DDoS attacks. One of the most massive DDoS attacks ever recorded in history originated from the IoT devices [4]. We could have prevented these calamities if all related organizations employ a proper access control mechanism in their network.

Unfortunately, implementing robust access control for IoT devices is challenging. An IoT device may not have the necessary computing resource to do heavy cryptography to secure itself; some devices cannot perform any cryptography at all. Moreover, the diverse nature of the IoT devices possesses

challenges when developing custom authentication protocol. The resulted scheme may work on a specific protocol but break in other protocols. Therefore, it has limited compatibility. Finally, the IoT architecture moves from a centralized model to a decentralized one (c.f. [5] for more detailed reflections on both models). Since solving access control issues involves the trust model's definition, it becomes more difficult to define the trust model in a decentralized environment, where no central party governs the system.

Blockchain, a promising technology behind the Bitcoin [6], brings the decentralization hype to many non-cryptocurrency sectors; one of them is IoT [7]. Researchers view the blockchain as the ''missing link'' to facilitate a decentralized platform for IoT, where many entities share data and resources. Furthermore, with the smart contract's invention [8], it allows adopters to enforce trust in a decentralized and verifiable way. It can also automate many time-consuming IoT workflows; thus, increasing the overall IoT environment efficiency. With all of these benefits, we argue that blockchain is a suitable platform candidate to build an access control system for IoT.

This paper proposes BorderChain, an access control framework for the IoT endpoint using blockchain. Before opening his endpoint to others, the IoT domain owner authenticates all gateways and devices in his endpoints. The corresponding Internet Service Provider (ISP) and IoT vendors examine those gateways and devices, then provide attestations of their authenticity. They then instruct the smart contract to put the gateway and device information in the trusted list. Once authenticated, other entities can begin requesting IoT resource access to the gateway. Through his gateway, the domain owner grants accesses only to the legitimate IoT services, which act as the IoT domain clients. He then activates the access token for the services in the smart contract. The services use this token to gain access to the IoT endpoint. While accessing the resource, both the services and the gateways also construct secure channels as additional security procedures.

We implement BorderChain as distributed applications equipped with a smart contract to allow IoT domain owners, IoT devices, IoT gateways, IoT vendors, IoT services, IoT users, and ISP to collaborate in a unified environment. Entities in BorderChain are built in Node JS applications connecting to the Ethereum blockchain as our peer-to-peer platform. Based on our performance evaluation using several Raspberry Pi hardware and our private server, BorderChain can process entities' authentication and authorization requests efficiently using all hardware resources.

Contribution: In summary, we made the following contributions.

- We propose a blockchain-based authentication protocol to provide verifiable identity and location guarantee of IoT devices and gateways. Using our approach, the IoT vendor verifies devices' authenticity while the ISP acts as a location attestation service for IoT gateways.
- We present the idea of blockchain-facilitated authorization protocol to allow IoT domain owners to authorize selectively IoT users or services that they permit to access their domains. We also provide an extension of our protocol to build a secure channel between IoT gateways and services.
- We analyze the security and trust properties of our proposal. We then implement our protocol and assess its feasibility through performance evaluations.
- To encourage reproducible research, we open our source code in a public repository.[1]

We describe the rest of the paper as follows. We present related work in Section II. Section III discusses problem statements and challenges of distributed access control in IoT. Then, we explain our design decisions to solve those issues in Section IV. We elaborates details of our proposed access control in Section V and evaluate its security, trust, and performance in Section VI. Section VII discusses our proposal's limitations. Finally, we conclude in Section VIII.

II. RELATED WORK

This section presents studies closely related to our proposal and elaborates on their similarities and differences.

A. IoT DOMAIN

The authors in [9] discuss access control's importance in distributing trusts among entities in the IoT environment. They propose using a hybrid architecture called Auth, locally centralized yet globally distributed architecture. With this approach, an IoT gateway governs IoT devices in the domain centrally, while the system manages accesses to different domains distributedly.

A similar design pattern also appears in LSB [10]. The authors take the same hybrid architecture idea to the blockchain realm. Specifically, they suggest using two blockchain networks. A local blockchain network exists on each of the IoT domains, with the IoT gateway serves as a central authority that mines the blockchain solely. Meanwhile, an overlay blockchain network oversees the governance among multiple domains in a decentralized manner.

Similar to previously mentioned proposals, we also employ a gateway-based architecture in this paper. In particular, the gateways manage the domain centrally while the blockchain maintains inter-domain communications governance.

B. IoT AUTHENTICATION

Certcoin [11] propose a decentralized authentication system that provides excellent key management features such as online and offline secret key, public keys binding to domains, public key lookups, key recovery, and key revocation. Moreover, this study also presents several strategies to cut down blockchain's storage requirements using accumulators and Distributed Hash Table. Together with SCPKI [12], these two

[1] https://github.com/mrkazawa/border-chain

papers can serve as extensions to our protocol. We can use them for our key and certificate management, which we do not discuss in this paper.

Another study uses blockchain to provide an out-of-band authentication [13]. Before getting access to IoT resources, IoT devices need the help of a nearby already-authenticated device to provide proof for their authentications. For example, the IoT server instructs a light bulb to perform a secret sequence that the target device has to decode and present back to the server through blockchain. In other words, the system resembles a two-factor authentication mechanism. Therefore, this study can serve as an alternative method for authentication between IoT devices and vendors in our protocol.

Bubbles of Trust (BoT) [14] propose an IoT authentication mechanism using virtual domains, called a bubble. A Master exists in their architecture to create bubbles in the blockchain and distribute tickets to the Follower. The Follower has to sign those tickets and deliver them to the blockchain for authentication to join a bubble. At the n-th transaction, when the Follower wants to transmit messages to other entities in the same bubble, he sends a transaction to the blockchain by attaching his previous tickets as proof for authentication.

There are several differences between the BoT and our proposal. First, BoT is a rigid system. It does not allow a particular group member to communicate with others who do not belong to the same group. Contrary to this paradigm, our authentication focuses on inter-domain relationships, which encourages IoT services and users (i.e., entities outside the domains) to trust IoT gateways and devices (i.e., entities inside domains). Second, we argue that BoT is costly. For example, when the Follower joins the bubble, the smart contract must verify the Master's signature on the tickets on-chain, which should be very expensive to perform. In our proposal, we outsource a similar signature verification mechanism off-chain to be more efficient and cheaper.

Trust List [15] proposes to use SDN and blockchain to provide authentication for IoT devices and services in an IoT domain. By default, IoT devices and services are untrustworthy. Therefore, the SDN controller drops all traffics that coming from them in the IoT domain. For authentication purposes, the SDN controller redirects their traffics to trusted validators. Once they are authenticated, the smart contract builds the Device Profile and Service Profile, serving as proof of authentication. These profiles also act as whitelists for the SDN controller to let traffics from these profiles go through the IoT domain.

Similar to this study, our proposal also uses trusted verifiers. However, we do not limit our implementation to only the SDN-enabled network. Hence, ours is network-agnostic. Moreover, in our proposal, IoT services are taking a more active role, similar to IoT users. Meanwhile, the IoT devices, which behave like servers, become passive and only respond according to their messages. Therefore, we put more concerns on the authenticity of IoT devices and gateways rather than IoT devices and services, as the authors propose.

C. IoT AUTHORIZATION

FairAccess [16] proposes the use of blockchain to store access token for IoT authorization. The resource owner sends an access token to the requester by creating a transaction and lock script in the blockchain. The requester then generates an unlocking script for the access token and then sends a reply transaction back to the blockchain. At this moment, the authorization is complete. Other parties can conduct verification by checking that the script from the requester can unlock the token. Because this study is one of the first blockchain-based IoT authorization schemes, the authors employ the scripting model of the Bitcoin with limited functions. However, our proposal makes use of a more modern approach by leveraging the smart contract.

Like the previous research, IoTChain [17] also stores proofs of access control in the blockchain. However, this study's distinguishable feature is the introduction of the Key Server, which serves as a proxy for the client and server to communicate securely. By default, IoT services encrypt all the IoT resources using a secret key. IoT users then have to get the key from the Key Server to decrypt the resource. However, the key server will check the proof in the blockchain before he distributes the keys to the users. Unlike this approach, clients and servers can build a secure channel without any proxy or centralized third party in our proposal.

LSB [10] presents a local access control mechanism in their gateway-based architecture. Specifically, the authors store access policies in the policy header of their custom blockchain architecture. The gateway intercepts every request coming to his domain and then allows or rejects requests based on the saved policy. The minor difference with our proposal is the type of blockchain platform that each of us uses. Our proposal can be used in any publicly available blockchain that supports smart contracts. Thus, we do not require custom blockchain architecture, as the authors propose.

III. PREREQUISITES

This section discusses problem statements and challenges that serve as our motivations in developing yet another access control for IoT.

A. PROBLEM STATEMENT

We can rephrase an access control problem to a trust issue. In a general sense, as humans, we tend to be more open to someone we trust than strangers. The same logic applies to the IoT, where we grant trustful entities more access to our IoT devices than, let say, hackers. Thereby, giving access control is closely related to distributing trust among entities.

A recent trend that is inevitable for IoT is the movement from centralized management to the distributed one. This move brings benefits to the IoT as it can reduce the overall latency of IoT workflows and enables real-time IoT processing. However, the decentralization of IoT is a disturbing maneuver for access control. In a centralized architecture, we gather IoT data from multiple devices in the field and store the data in a siloed and centralized database in the

Cloud. When we want to share this data with other parties, we open the Application Programming Interface (API) and apply access control centrally from the Cloud. By contrast, anyone can share resources and data arbitrarily in a decentralized environment. Thus, we have to disseminate the access control procedures across multiple actors, making the system complex.

Moreover, in distributed IoT, the roles of clients and servers are now upside down. With CoAP [18] or MQTT [19] protocol, IoT services and users now have to initiate the IoT data request to the IoT devices through gateways or brokers. Thus, the IoT device plays a passive role by responding to the gateway or the broker's queries. This condition highlights IoT gateways' importance and augments the need to apply a robust access control mechanism to each IoT endpoints in the network.

Use Cases: Let us assume that company *A* has built a weather monitoring application in a particular city. Meanwhile, company *B* has developed an IoT-enabled public transport application for the same city. A recent business meeting requires *B* to integrate weather data into its platform to serve citizens with better transit options. Instead of building its own siloed weather application, *B* tries to negotiate with *A* to share their IoT weather infrastructure. Let us assume that *A* approves *B*'s request and allows *B* to access weather data from various *A*'s devices through multiple *A*'s endpoints scattered all across the city. Now, *B* needs to initiate the process to *A*'s endpoints. However, *B* is not sure how to authenticate *A*. Thus, from this scenario, we define three problems:

P1 How to distribute and enforce trust among many participants in the IoT network?

P2 How do we know that we are communicating with a verified IoT endpoint?

P3 How do we assure that the endpoint truly holds the IoT devices they claim?

Assuming that *B* can identify *A*'s endpoints and devices correctly, a follow-up question now becomes how *A* can efficiently give *B* access. Recall that *A* has multiple IoT devices and endpoints that all may speak different protocols. It is convenient if *B* has a single token representing his access across multiple endpoints and devices without a protocol boundary. Furthermore, with the nature of IoT devices being constrained devices and IoT endpoints reside mostly in the field with low-security protection, attackers can most likely compromise them easily. Hence, the system should provide a revocation procedure that *A* can use in the aftermath of cybersecurity hacks. Thus, we define two additional problems:

P4 How do we provide universal access control across multiple IoT endpoints and devices?

P5 How do we provide a robust revocation mechanism to repeal our authentication and authorization?

B. CHALLENGES

We argue that blockchain is a suitable decentralized platform for IoT and plan to build our access control on top of it. However, after we investigate the literature regarding access control, IoT, and blockchain, we find several challenges that may hinder our objective. We describe them as follows.

C1 IoT devices have various resources to do cryptography.

A study by Ometov *et al.* [20] explores diverse IoT and wearable devices' feasibility to perform cryptographic algorithms. The result shows that modern IoT devices can do the cryptographic procedure with varied performance depending on their resources. Furthermore, some IoT devices, such as sensor networks, are designed to preserve battery. Thus, heavy cryptography, while it is viable [21], needs to be performed scarcely. Consequently, IoT manufacturers can take several alternatives to provide authentication for IoT devices [22], ranging from the most secure but requiring more processing to the least secure with least processing. Our access control should be compatible with many authentication options, such as:

- *PKI signature*, a challenge-response with IoT device's public key to prove IoT device's signature generated from its private key.
- *A pre-embedded root of trust*, IoT vendor stores a pre-shared secret in a secure place (possibly in Trusted Platform Module) embedded in the IoT device; then, the device performs symmetric signing with it.
- *Device fingerprinting*, IoT device generates a hash of selected files from its file system then compares it to the pre-computed hashes in the server.
- *MAC authentication*, the vendor previously built a whitelist record containing a list of eligible IoT device MAC addresses, and then she only approves requests that contain listed MAC addresses.

C2 IoT devices apply diverse IoT protocols.

IoT devices may use various protocols stacks in their architecture [23], which complicates the compatibility between one IoT application to another. For example, whether to use TCP or UDP in the Transport Layer, using IPv4 or IPv6 in the Network Layer, or to use RFID, Wi-Fi, Bluetooth, 6LowPAN, 3G, or 4G to communicate with other devices in the network. Our proposal should be able to operate with many IoT protocols.

C3 The number of IoT devices and endpoints are surging.

With the growing number of IoT devices and endpoints, the widely-adopted X.509 digital signature scheme [24] is not suitable for the distributed IoT environment, at least not in the current form. First, the system has a single point of failure problem. Since anyone must fully trust the Centralized Authority (CA), the CA may break the whole system once it starts to misbehave [25]. Second, the scheme does not scale well because CA now has additional jobs to sign a vast number of IoT devices or endpoints. Lastly, CA cannot provide a seamless revocation procedure when signatures are compromised [26]. Thus, we should design our proposal with the robustness, scalability, and flexibility that the IoT network needs.

C4 Blockchain node needs ample resources.

Becoming a blockchain node requires many processing power in CPU, memory, storage, and networking. The typical permissionless blockchain such as Bitcoin [6] or Ethereum [8] requires the node to solve a cryptographic puzzle to achieve consensus. We commonly define it as Proof-of-Work (PoW). This process is CPU-intensive, such that the majority of IoT devices cannot perform well enough. For example, a Raspberry Pi 2 takes 3082 seconds to solve a PoW puzzle and wastes 5,3 KJ of energy [27]. This metric should worsen on more constrained IoT devices.

Moreover, being a blockchain node also requires the node to sacrifice storage space to maintain a distributed ledger copy. The data size varies depending on two things, how long the blockchain has run and how many users are using it. For instance, we need at least 200 GB of free hard drive storage and 2 GB of RAM to run a full Bitcoin node [28]. Lastly, in terms of networking, we also need constant network communication with other peers to get up-to-date ledger information. Therefore, we should find a workaround to allow IoT devices to access the blockchain widely.

IV. DESIGN DECISION

We came up with several core design decisions for our access control that we believe can serve as direct and workaround solutions to the previously mentioned problems and challenges. We summarize them in Table 1 and elaborate them as follows.

TABLE 1. A summary of our solutions toward the problems and challenges in Section III.

Issues	Our Solutions
P1	Smart Contract is a secure and transparent entity that serves as the root of trust among many participants in the IoT environment.
P2	ISP acts as a trusted approver for IoT endpoints in the network.
P3	IoT vendor becomes a trusted approver for all of the IoT devices in the domain.
P4	IoT gateway conducts local access control rules to cope with various IoT devices and protocols while also maintaining global and consistent rules in the blockchain.
P5, C3	The access control state are maintained in the blockchain, which can scale the current X.509 certificates to meet IoT requirements and offer a seamless revocation experience.
C1	IoT gateway piggybacks and facilitates multiple types of authentication payloads for IoT devices.
C2	IoT gateway serves as a protocol bridge to ensure compatibility across many IoT protocols.
C4	IoT gateway is the blockchain node, which acts as a proxy for IoT devices to communicate to the blockchain network.

A. GATEWAY-BASED ARCHITECTURE

Throughout the rest of this paper, we employ a gateway-based architecture scattered across our IoT network. We depict our conceptual IoT architecture in Figure 1. In this architecture, IoT devices cannot communicate to the Internet directly. Instead, they rely on a centralized, trusted IoT gateway to relay their messages to the rest of the network. Likewise, anyone requires to access the IoT devices must go through the gateway as well. IoT devices and the gateway form an IoT domain or endpoint.[2] The gateway also serves many roles as follows:

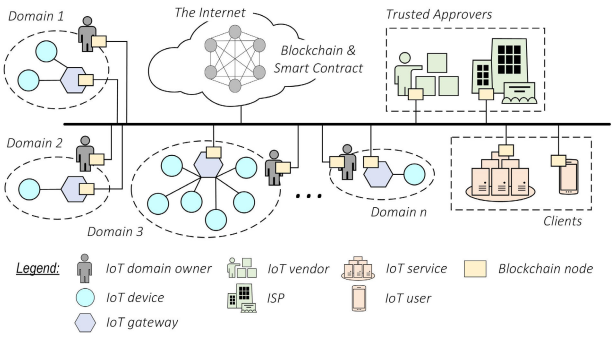

FIGURE 1. The IoT architecture for our proposed access control scheme. IoT gateway and the underlying IoT devices form an IoT domain, governed by the IoT domain owner. The IoT vendor and ISP exist in the system as trusted verifiers to approve the authenticity of the gateways and devices. Meanwhile, the IoT service and user serve as the IoT domain clients that query resources from the IoT domain.

- **The gateway becomes a protocol bridge**. It translates multiple IoT-related protocols (e.g., Bluetooth Low Energy (BLE), Zigbee, 6LowPAN) from the devices into TCP/IP protocol to communicate with entities outside of the domain. Thus, the gateway helps in overcoming **C2**.
- **The gateway provides local access control rules**. The gateway enforces local access control for all IoT devices in its endpoint. This scheme is feasible because the system routes every request through the gateway. Moreover, the gateway maintains a global rule for every requester across multiple domains with the blockchain network. Therefore, it realizes **P4**.
- **The gateway piggybacks security constraints of the IoT devices**. Due to the constrained nature of IoT devices, IoT manufacturers mostly implement weak or even no security at the IoT devices for the sake of communicating to the nearby gateway. The gateway transforms this unsecure communication into a secure channel using TLS or DTLS when communicating with the outside domain. Thus, to solve **C1**, we argue that the gateway can facilitate the transfers of various types of authentication payloads from IoT devices to the IoT vendor.
- **The gateway as a blockchain node**. Since the gateway is most likely to have higher processing power and storage than IoT devices, the gateway is more appropriate to become the blockchain node. Any IoT device that wants to communicate to the blockchain can contact the gateway as a proxy. Hence, this decision solves **C4**.

B. TRUSTED APPROVER

We use trusted approvers to authenticate and authorize IoT endpoints, devices, and resources. By doing so, we introduce centralization into our system. However, we argue that

[2] This paper uses the terms domain and endpoint interchangeably

centralization is inevitable in some parts of the IoT ecosystem. For example, we rely on IoT manufacturers to fabricate our IoT devices. Then, we lean on IoT gateways to relay messages from our endpoint to the Internet and vice versa. Lastly, we depend heavily on the Internet Service Provider (ISP) to route our IoT traffics worldwide. Our idea is to let the ISP, vendor, and gateway handle security verifications with blockchain serves as the backbone communication to perform the validations in a distributed, fair, and transparent manner.

- **ISP as the gateway approver**. To make the public believe that a given IoT endpoint belongs to a particular owner, we use ISP as a trusted verifier. Specifically, the owner has to present proof of his network registration to the ISP authentication server. Upon validation, the ISP maps the owner's gateway blockchain address to the owner's IP. The ISP then saves this mapping to the smart contract. Because everyone trusts the ISP, they will trust this information once recorded; thus, this solution solves **P2**.
- **Vendor as the device approver**. The manufacturing process inherits the trust relationship between IoT vendors and their devices. The gateway piggybacks the device authentication payload to the vendor authentication server. The message to the vendor also contains the gateway blockchain address. Thereby, one can relate the relationship between the gateway and the device. Upon validation, the vendor stores this relationship in the smart contract. Since everyone trusts the IoT vendor, this design resolves **P3**.
- **Gateway as the access verifier**. IoT domain owner is the one who opens the IoT endpoint and the underlying devices to the rest of the network. As a result, he has the full right to give access to anyone that he trusts. To grant access, he leverages his IoT gateway as an authorization server. The gateway intercepts all requests coming to the domain and only allows the authorized ones to go through. Thereby, this scheme contributes to **P4**.

C. DISTRIBUTED ACCESS CONTROL STATE

We take advantage of decentralization in blockchain to store our access control's state distributedly on all IoT entities. Specifically, after approvers conduct verifications on the gateway, device, or access, they store the corresponding state in the smart contract. Because the blockchain replicates the state to all nodes, lookup on this state becomes local processes on each associated node. This local process is faster than in the centralized architecture. As a result, it can scale the system further and solve **C3**. It also speeds up the revocation processes and resolves **P5**.

D. SMART CONTRACT AS THE ROOT OF TRUST

We have many entities in our proposal that comprise IoT domain owner, IoT device, IoT gateway, IoT vendor, IoT service, IoT user, and ISP. Each of them may have a conflict of interest, and the smart contract is a suitable candidate to unite them since it is open, transparent, and deterministic. All of the blockchain nodes can understand what data is being stored and how the contract will perform a particular method. Thus, they get a single unified truth. This mechanism then satisfies **P1**.

V. PROPOSED PROTOCOL

Our access control protocol comprises several steps. First of all, we perform gateway authentication, in which IoT domain owners validate their IoT domains to the ISP. Once validated, the owners perform authentication for each IoT device under each domain to its respective IoT vendor. Others can then begin to request access authorizations to the IoT gateways. The owners grant access to their IoT domains by giving access tokens to legitimate requesters. Finally, requesters can build secure channels with the gateways before querying for IoT resources using the granted access tokens.

A. NOTATIONS AND TOOLS

In the remainder of this paper, we use the following notations:

1) α_L refers to L's blockchain address that serves as L's identity in the blockchain (on-chain) and outside the blockchain (off-chain).
2) $PKE_{PK_x}(J)$ is the public-key encryption of J using x public key.
3) $PKD_{SK_y}(K)$ is the public-key decryption of K using y private key.
4) $E_z(J)$ is the symmetric encryption of J using z pre-shared key.
5) $D_z(K)$ is the symmetric decryption of K using z pre-shared key.
6) $PKSIGN_{SK_y}(J)$ generates a public-key digital signature for J using y private key.
7) $PKVERIFY_{\alpha_L}(K, J)$ verifies whether the blockchain address α_L signs data J and generates the public-key digital signature K.
8) $SIGN_z(J)$ generates a symmetric digital signature for J using z pre-shared key.
9) $VERIFY_z(K, J)$ verifies whether the sender signs the data J using z pre-shared key and generates the symmetric digital signature K.
10) $H(J)$ generates a hash of J.
11) $X \parallel Y$ represents the concatenation of X with Y.

Our access control scheme also includes multiple entities, which have different roles and interests from one another. We introduce them as follow:

1) D is the IoT device, it is a sensor or an actuator originated from a particular IoT vendor. Its role is to generate IoT data for consumers and execute assigned commands.
2) V is the IoT vendor. He is the manufacturer and the seller of the IoT devices.
3) ISP is the Internet Service Provider (ISP), facilitating the transmission of IoT data packets across many IoT endpoints.
4) GW is the IoT gateway. This entity relays all communications between entities outside the IoT endpoint to

the IoT devices within the domain and vice versa. It can be a protocol-specific peripheral (e.g., CoAP gateway or MQTT broker) or a general multi-purposes gateway, where it serves many protocols at once.

5) O is the IoT domain owner. He is the owner and the administrator of an IoT endpoint. He owns the gateway and IoT devices, which also is responsible for registering them to the corresponding ISPs and IoT vendors. He also opens his domain to the public.
6) S is the IoT service, which serves as the consumer of IoT data. He garners the IoT data from IoT devices through IoT gateways. He can also instruct commands to the IoT devices.
7) U is the IoT users. He has a similar role as the IoT services in collecting data from devices and instructing commands to devices. Moreover, he is also the consumer of the IoT data analytics that the IoT service provides.
8) SC is the smart contract. This entity resides in the blockchain and becomes the root of trust among all participants. It enforces trust between parties by presenting trusted Turing-complete methods for all participants.

B. GATEWAY OR ENDPOINT AUTHENTICATION

Premise: As the one who owns and controls the IoT domain, O first registers himself to *ISP* to connect his domain to the Internet. He then sets up GW for his domain.

1) O creates a pair of PK_O and SK_O. By doing so, O also produces α_O.
2) O generates a pair of υ_O and ρ_O, which are O's username and password. In this paper, we assume that *ISP* authenticates its subscribers by their username and password. However, adopters can change it to other authentication alternatives (e.g., public keys, biometrics, or pin) when necessary.
3) O contacts *ISP* and then submits υ_O, ρ_O, and α_O to the system. In this process, he also subscribes to the Internet service that *ISP* provides (we omit the details here as the subscription procedures may vary across many ISPs).
4) *ISP* creates a pair of PK_{ISP} and SK_{ISP}, which also produces α_{ISP}. She then validates the owner's registration. If the registration is successful, she returns PK_{ISP}, α_{ISP}, and γ_O (i.e, O's IP address) to O.
5) O then creates PK_{GW}, SK_{GW}, and α_{GW}. After that, he equips GW with γ_O so that the public can access his gateway through the given IP.
6) *ISP* deploys an authentication server equipped with the parameters of α_{ISP}.
7) We assume that a trusted third party (e.g., the government) exists to deploy the smart contract to the blockchain network. Once SC is deployed, O and *ISP* subscribe to SC's events.
8) SC is discoverable through α_{SC}, and we assume that all parties know this address as common knowledge.

Goal: By default, the public does not trust GW and the associated γ_O. The following protocol facilitates a transparent endpoint authentication by factoring the process in the blockchain. *ISP* will provide an attestation of γ_O possession to GW when the authentication is successful. We summarize the whole process in Figure 2.

1) O forms:
   - η, a random string for entropy and replay attack protection
   - t, a current timestamp
   - $X_1 = \upsilon_O \parallel \rho_O \parallel \gamma_O \parallel \eta \parallel t$
   - $Y_1 = H(X_1)$

   X_1 is the whole gateway authentication payload with Y_1 as its corresponding hash. Y_1 holds an essential role because it also acts as an identifier for the authentication request. As a result, O must store Y_1 in his local storage since he will need this hash for the revocation use case later, which we explain at the end of this section.
2) O sends a transaction (tx) to SC by calling SC's method to record a log of the authentication request in the blockchain. Specifically, O includes the following information in the transaction:
   - *hash*, the hash of the gateway authentication payload, Y_1
   - *source*, the address that sends this transaction, α_O
   - *target*, the address of the authentication target, α_{GW}
   - *approver*, the address of the authentication approver, α_{ISP}

   In other words, O puts a log in the blockchain, indicating an instruction for *ISP* to validate his GW by presenting the authentication payload Y_1. This log acts as a factor in the blockchain, making the authentication process transparent and fair for everyone. This step ends with the SC returns a tx hash as proof of submission.
3) Upon receiving the previous transaction calls, SC stores all information to a list of authentication logs in its key-value storage. The *hash* serves as the key, with the *source*, *target*, and *approver* serves as the values. There are also two parameters in each entry of the list, *approved* and *revoked*, that SC sets to *False* by default. The former indicates that the authentication request has not been approved yet, and the latter tells that the request is not revoked.
4) After receiving the tx event from Step 2, which indicates Y_1 metadata have been inserted successfully in the blockchain, O then forms:
   - $C_1 = PKSIGN_{SK_O}(Y_1)$
   - $X_2 = PKE_{PK_{ISP}}(X_1 \parallel C_1)$

   C_1 is the signature of the authentication payload while X_2 is an encrypted authentication request for *ISP*.
5) O sends the authentication request X_2 off-chain to *ISP*'s authentication server.

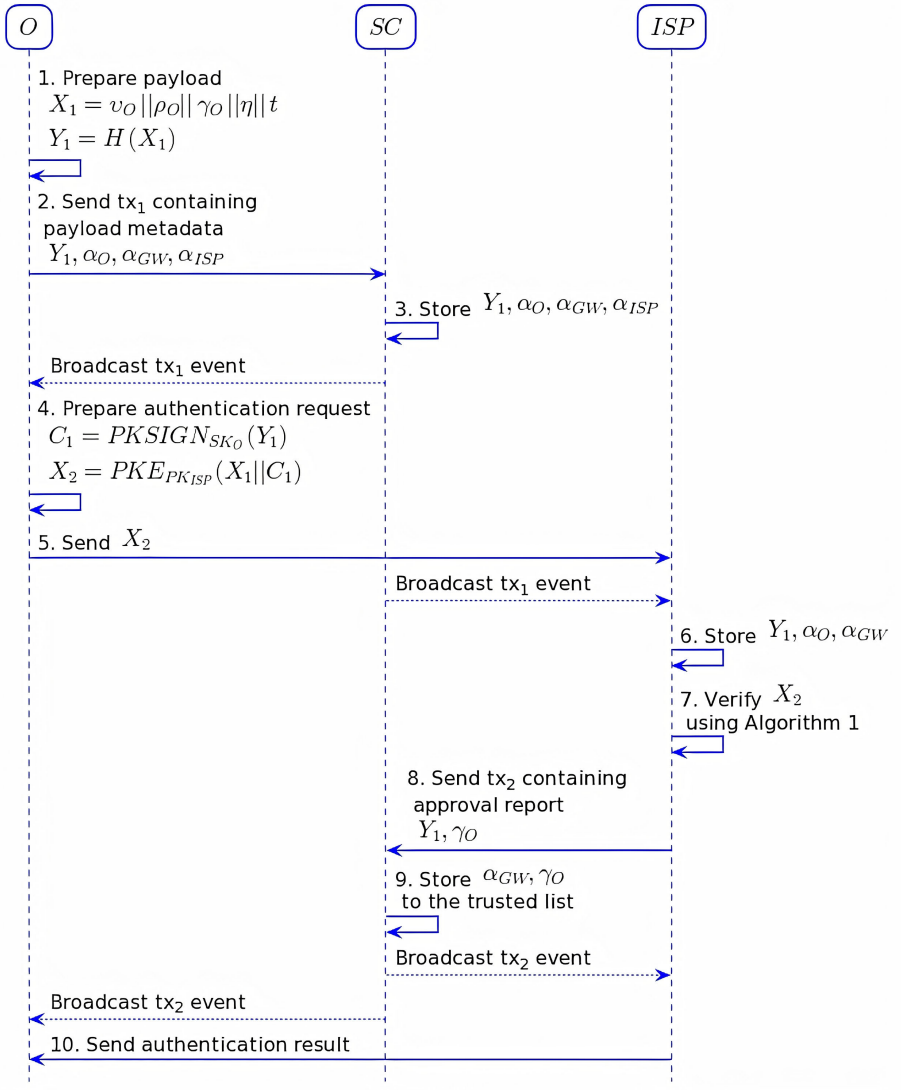

FIGURE 2. The sequence diagram for our proposed gateway authentication.

6) When *ISP* receives the event from Step 2, where *approver* equals α_{ISP}, she stores the corresponding *hash*, *source*, and *target* information to his local database. Note that the delivery of events is subject to the miner and network latency. Therefore, O may receive this event first before *ISP* and vice versa.
7) After receiving X_2, *ISP* begins the verification process, which we outline at Algorithm 1.
For starters, she decrypts the encrypted authenticaton request X_2 (line 1). The decryption reveals the gateway authentication payload X_1 as well as the signature C_1. She then calculates the hash of authentication payload Y_1 (line 2).
After that, she verifies whether Y_1 exists in her local database (line 3). If the authentication hash does not exist, it may imply two things. First, the owner sends a valid X_1, but the *ISP* has not received the tx event containing Y_1 yet. Thus, she cannot find it in her database. Second, the owner sends an invalid X_1.

Algorithm 1 The Verification of the Gateway Authentication Request in *ISP*

Input: X_2, SK_{ISP}
Output: True or False
1: $X_1 \parallel C_1 \leftarrow PKD_{SK_{ISP}}(X_2)$
2: $Y_1 \leftarrow H(X_1)$
3: **if** ! $exist(Y_1)$ **then return** False
4: $source \leftarrow getSource(Y_1)$
5: **if** ! $PKVERIFY_{source}(C_1, Y_1)$ **then return** False
6: $\upsilon_O \parallel \rho_O \parallel \gamma_O \leftarrow X_1$
7: $\upsilon'_O \parallel \rho'_O \parallel \gamma'_O \leftarrow getUserInfo(source)$
8: **if** υ'_O ! $= \upsilon_O$ **or** ρ'_O ! $= \rho_O$ **or** γ'_O ! $= \gamma_O$ **then return** False
9: **return** True

If the *ISP* finds the hash, she then gets the *source* of this authentication log from her database and verifies whether this *source* signs the received authentication payload (line 4-5). This procedure validates the entity

that sends X_1 off-chain is the same entity that transmits the authentication log Y_1 on-chain. Thus, this step is crucial to ensure that the *ISP* deals with the same domain owner.

The rest of the procedure is straightforward. The *ISP* queries details of the user's information from the database using *getUserInfo*(·) method and then checks if they match the one in the authentication payload (line 6-8).

The algorithm will return *False* if there is an issue or invalid checking during these verifications and return *True* when everything is valid.

8) Assuming that the verification is successful, *ISP* sends the authentication report back to *SC* by sending a transaction with this information:
   - *hash*, the hash of the approved gateway authentication payload, Y_1
   - *routerIP*, the approved IP from the authentication payload, γ_O

   She then receives tx hash as proof of submission.
9) In the blockchain, *SC* maintains a list of trusted *GW* that everyone should trust. Upon receiving the transaction in the previous step, *SC* sets the *approved* status of Y_1 to *True*. This action indicates that *ISP* already verified Y_1 payload. Then, *SC* saves α_{GW} along with γ_O in the trusted list.
10) *ISP* sends a message to *O* to let him know the result of his authentication request. *O* and *ISP* may also receive the event from Step 8, which tells that the gateway address and the IP have been successfully approved in the blockchain.

At this moment, *ISP* has approved that γ_O belongs to *O* legitimately. Furthermore, from the authentication request, we can also learn that *O* has equipped his gateway (i.e., α_{GW}) with γ_O. Therefore, γ_{GW} is equal to γ_O. At any given time, anyone can query γ_{GW} from the blockchain. They can also check if *SC* lists α_{GW} in the trusted list before deciding to trust *O*'s endpoint and making any further access. More importantly, one can also relate the authentication relationship between *O*, *GW*, and *ISP* from the recorded authentication log. Thus, any potential malicious entities or relationships can be audited easily.

Note that the username and password, υ_O and ρ_O in X_1 serves as the main authentication in *ISP*. Meanwhile, the α_O serves as the two-factor authentication. Moreover, *O* must provide γ_O in X_1 to cope with a possibility that *O* has multiple registered IPs in *ISP*. Therefore, he needs to pinpoint which IP he wants to use in association with the targeted gateway.

Alternative (Signature-Based Attestation): When we design this protocol, we also consider another alternative to achieve the same goal. First, *O* authenticates himself to the *ISP* off-chain. If the authentication is successful, *ISP* will sign α_{GW} and γ_O for *O*. Let us assume that this signature is C_0. Afterward, *O* sends a transaction to *SC* to store α_O, α_{GW}, α_{ISP}, γ_O, and C_0 in the blockchain. Other entities can look up this information and verify the associated signature. They trust this information if the signer is indeed coming from the *ISP*. However, we refrain from using this alternative version due to several reasons.

- Storing signature C_0 requires 64 bytes while storing the hash Y_1 only needs 32 bytes. Therefore, using our proposal, we can save 32 bytes in the blockchain storage. With a massive number of IoT endpoints is available in the network, this small difference can become significant on a large scale.
- The alternative hides the prior authentication requests from the public. One can only understand that a particular gateway has proof of signature from the ISP. By contrast, we anchor the whole authentication request and response in the blockchain. We argue that this decision leads to a more transparent and fair process. For instance, one can audit records of the authentication requests to point out malicious requests or responses and determine potential attackers.

C. DEVICE AUTHENTICATION

Premise: During the manufacturing of *D*, *V* inserts pre-defined secrets, which only *V* and *D* know, for authentication purposes. Afterward, *O* buys *D* from *V* and configures it to connect to his authenticated *GW*.

1) *V* produces PK_V, SK_V, and α_V.
2) *V* then creates PK_D, SK_D, and α_D for the IoT device that she manufactures. Note that *V* may also generate other device parameters that can be used to provide device authentication alternatives. For example, a pre-shared secret key z, device fingerprint f, or MAC address *mac*.
3) *V* signs α_D (i.e., $PKSIGN_{SK_V}(H(\alpha_D))$) and produces a signature C_D. This signature is proof that the corresponding device is coming from this vendor.
4) *V* embeds previously created secrets to the secure storage of *D*'s hardware. The following values are embedded for all of the authentication types: α_D, α_V, PK_V, and C_D. Additionally, *V* embeds PK_D and SK_D for *PKI signature*, z for *Pre-embedded root-of-trust*, f for *Device fingerprinting*, and *mac* for *MAC authentication*.
5) *O* purchases *D* from *V*.
6) *O* connects *D* to *GW*, which is identifiable by α_{GW}.
7) In this paper, we put more concerns at inter-domain communications. Therefore, we assume that *D* trusts *GW* by default. We also assume that communication between *D* and *GW* is secure. However, the link from *GW* to *V* is insecure.
8) *O* has conducted the endpoint authentication in Section V-B for α_{GW} before processing the following device authentication.
9) *V* deploys an authentication server in the network that is discoverable through α_V.
10) *GW* and *V* subscribe to the *SC*'s events.

Goal: By default, the public do not trust *D* inside *GW*. Like the gateway authentication, the following protocol facilitates a transparent device authentication by factoring the processes

in the blockchain. When the authentication is successful, V provides attestation of the legitimacy of D inside GW. We summarize the whole process in Figure 3.

1) D forms secret authentication payload for V.
   - $X_3 = \eta \parallel t$
   - $Y_2 = H(X_3)$

   For *PKI signature*, D sets $\tau = 1$. τ is a unique parameter to indicate the type of device authentication scheme to use. D then forms:
   - $C_2 = PKSIGN_{SK_D}(Y_2)$
   - $X_4 = \tau \parallel X_3 \parallel C_2$

   For *Pre-embedded root-of-trust*, D sets $\tau = 2$, then forms:
   - $C_2 = SIGN_z(Y_2)$
   - $X_4 = \tau \parallel X_3 \parallel C_2$

   For *Device fingerprinting*, D sets $\tau = 3$, then forms:
   - $X_4 = \tau \parallel X_3 \parallel H(f)$

   For *MAC authentication*, D sets $\tau = 4$, then forms:
   - $X_4 = \tau \parallel X_3 \parallel mac$

   X_4 is the authentication payload for V. Meanwhile, Y_2 is the corresponding hash.
2) D then prepares the authentication request:
   - $X_5 = \alpha_V \parallel PK_V \parallel \alpha_D \parallel C_D \parallel X_4 \parallel Y_2$

   X_5 contains the whole authentication request to GW, which he will then relay to V.
3) D sends X_5 to GW off-chain.
4) Upon receiving X_5, GW verifies C_D whether the signer is coming from α_V. Specifically, $PKVERIFY_{\alpha_V}(C_D, H(\alpha_D))$ must return *True*. This operation is to ensure that the device indeed belongs to the respective V. Furthermore, we assume that GW maintains a list of trusted V information locally, such that given a payload containing α_V, GW understands to which V's server he has to relay this payload.
5) Assuming that the previous verification is successful, GW sends a transaction to SC by calling a method to record a log of the D's authentication request in SC. GW includes the following information in the transaction:
   - *hash*, the hash of the device authentication payload, Y_2
   - *source*, the address that sends this transaction, α_{GW}
   - *target*, the address of the authentication target, α_D
   - *approver*, the address of the authentication approver, α_V

   In other words, GW puts a log in the blockchain, indicating an instruction for V to authenticate D by presenting the authentication payload Y_2. This step ends with GW receives a tx hash as proof of submission. Note that GW must store Y_2 in his local storage for device revocation usage later.
6) Like the gateway authentication scenario, SC saves the received information in authentication logs. SC also sets *approved* and *revoked* value for Y_2 to *False*.
7) After receiving the tx event from Step 5, which indicates Y_2 log has been inserted successfully in the blockchain, GW then prepares the authentication request for the vendor. Specifically, GW forms:
   - $X_6 = C_D \parallel X_4$
   - $Y_3 = H(X_6)$
   - $C_3 = PKSIGN_{SK_{GW}}(Y_3)$
   - $X_7 = PKE_{PK_V}(X_6 \parallel C_3)$

   GW strips the unrelated information from the original X_5 and makes a new request X_6 for V with C_3 as the signature of this authentication request. Notice that X_7 is encrypted, so it is safe to transfer it to an unsecured channel.
8) GW contacts V to deliver X_7 off-chain.
9) When V receives the event from Step 5, where *approver* is equals to α_V, V stores the corresponding *hash*, *source*, and *target* information to his local database. Like the gateway authentication case, the delivery of events is subject to the miner and network latency. Therefore, GW may receive this event first before V and vice versa.
10) Upon obtaining X_7, V verifies the device authentication request following the steps in Algorithm 2.
    First of all, V decrypts the authenticaton payload X_7 using SK_V, and traverses deep into the payload to generate the authentication payload hash Y_2 from $H(X_3)$ (line 1-4). She then makes sure that Y_2 exists in her local database (line 5). Similar to gateway authentication, when Y_2 is not found, it implies two things. First, the payload hash is valid, but V has not received the event from Step 5 yet. Second, X_6 is invalid.
    V only authenticates devices that she previously manufactured. Therefore, she verifies C_D using *target* information from the blockchain (line 7). Then, she checks if the *source* is indeed the sender of this payload by verifying C_3 (line 9). This verification ensures that the instance that delivers the payload Y_2 on-chain and X_6 off-chain is the same entity.
    Finally, the vendor can begin verifying the device authentication payload depending on the authentication type τ that the device uses. For *PKI signature*, the vendor checks whether C_2 is coming from the *target* (line 10-12). For *Pre-embedded root-of-trust*, the vendor validates if C_2 is signed using the correct key z (line 13-16). For *Device fingerprinting*, the vendor verifies that the hash of the fingerprint $H(f)$ is the same as the hash value in the database $H(f')$ (line 17-20). For *MAC authentication*, the vendor makes sure that the given *mac* is the same as the value in the database mac' (line 21-24).
    The algorithm will return *False* if there is an issue or invalid checking during these verifications and return *True* when everything is valid.
11) Assuming that the previous verification is successful, V sends the authentication report back to SC by sending a

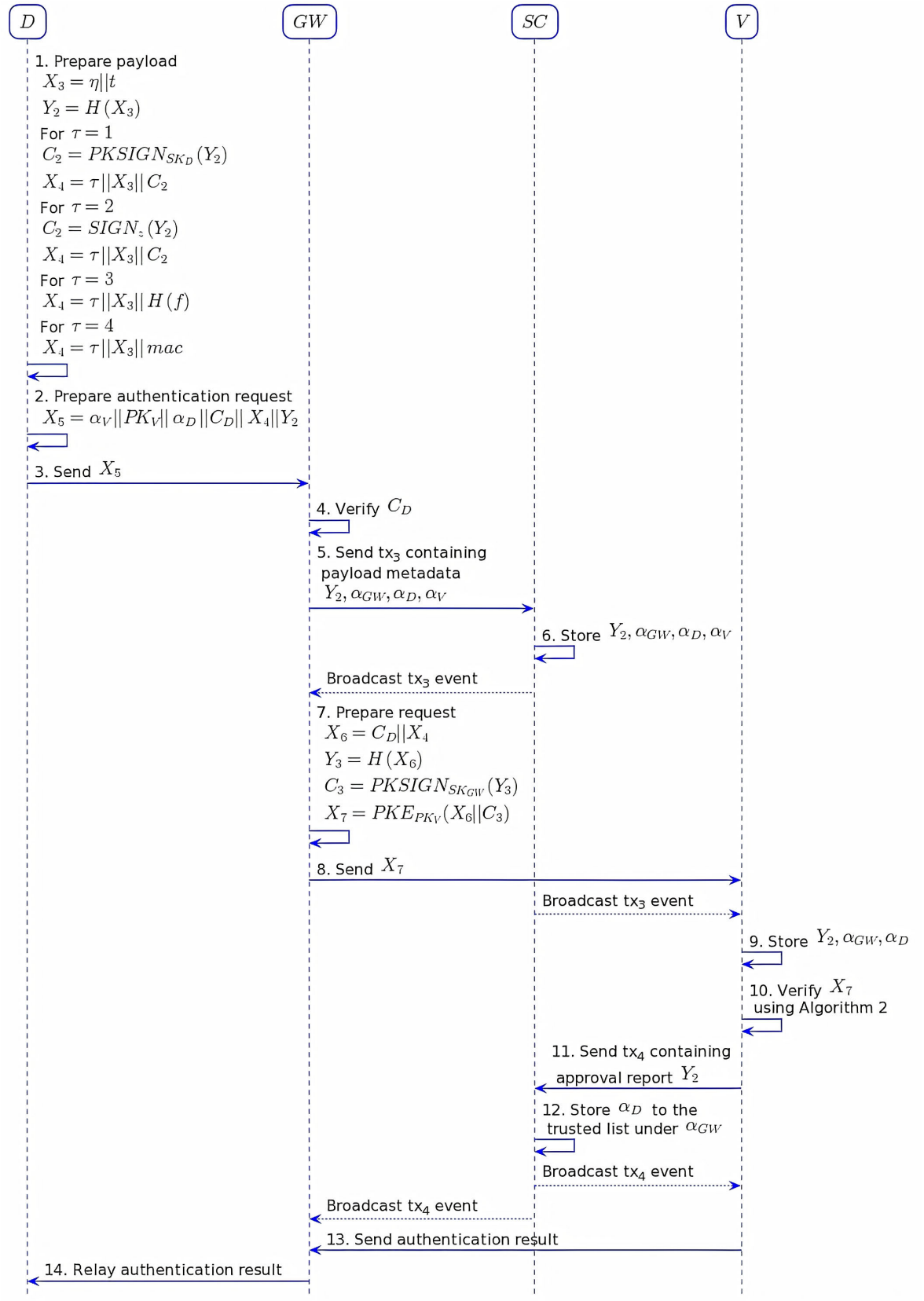

FIGURE 3. The sequence diagram for our proposed device authentication.

Algorithm 2 The Verification of the Device Authentication Request in V

Input: X_7, SK_V, α_V
Output: True or False

$X_6 \parallel C_3 \leftarrow PKD_{SK_V}(X_7)$
$C_D \parallel X_4 \leftarrow X_6$
$\tau \parallel X_3 \leftarrow X_4$
$Y_2 \leftarrow H(X_3)$
if ! $exist(Y_2)$ **then return** False
$source \parallel target \leftarrow getPayloadInfo(Y_2)$
if ! $PKVERIFY_{\alpha_V}(C_D, H(target))$ **then return** False
$Y_3 \leftarrow H(X_6)$
if ! $PKVERIFY_{source}(C_3, Y_3)$ **then return** False
if $\tau == 1$ **then**
    $C_2 \leftarrow X_4$
    if ! $PKVERIFY_{target}(C_2, Y_2)$ **then return** False
else if $\tau == 2$ **then**
    $z \leftarrow getDeviceInfo(target)$
    $C_2 \leftarrow X_4$
    if ! $VERIFY_z(C_2, Y_2)$ **then return** False
else if $\tau == 3$ **then**
    $f' \leftarrow getDeviceInfo(target)$
    $H(f) \leftarrow X_4$
    if $H(f) != H(f')$ **then return** False
else if $\tau == 4$ **then**
    $mac' \leftarrow getDeviceInfo(target)$
    $mac \leftarrow X_4$
    if $mac != mac'$ **then return** False
else
    return False {invalid device authentication type}
end if
return True

transaction using SC's method that include the following information:
- *hash*, the hash of the approved device authentication payload, Y_2

This step ends with V obtaining the tx hash as proof of reporting.

12) In the blockchain, aside from the list of trusted gateways, SC also maintains a list of trusted devices as a nested list of the former list. Upon receiving Y_2 from the previous step, SC set the *approved* value of Y_2's log to *True*. He also queries the value of *target* (i.e., α_D) and saves it to the trusted device list under the *source* (i.e., α_{GW}) as its parent gateway.
13) After V receives the tx event from Step 11, she returns the device authentication result to GW.
14) GW also receives an event from Step 11 indicating that V already validated his device. Upon receiving both the event and the response from V, GW notifies this information to D.

At this moment, V has approved that α_D is an authentic device originates from V. Furthermore, the mapping between α_{GW} and α_D in SC indicates that the device is indeed connected to the mentioned gateway because the GW is the one who processes D's authentication. Thus, anyone can safely assume that α_{GW} is the correct endpoint for D. Moreover, since the blockchain is open, any node can relate the authentication relationship between GW, D, and V. Thus, they can detect potential malicious entities easily.

Note that, in the production case, the contents of X_3 can be modified by adding more variables to match the required scenarios. For example, one can add the device's current software version so that the vendor can check whether the device is updated to the latest software version or not. The vendor can have a verification policy only to approve the up-to-date devices. Therefore, an out-of-date device will not be recognized by the vendor during the authentication. This procedure is useful to encourage the domain owners to update their devices regularly.

D. ENDPOINT AUTHORIZATION

Premise: O lets anyone discover his endpoint publicly. However, he wants only authorized parties to access the gateway. He owns the endpoint, so he has the power and rights to determine which party is legitimate to enter his gateway.

1) O has already conducted verification for his endpoint, described in Section V-B. Thus, GW is trusted.
2) GW has already performed authentication for all of the devices inside her endpoint, explained in Section V-C. Thereby, D is also trusted.
3) In general, S and U are eligible to request IoT accesses to the endpoint. However, we only use S as our example for the remainder of this paper.
4) S produces PK_S, SK_S, amd α_S.
5) We assume that S can get information about an IoT endpoint off-chain. From this procedure, S retrieves knowledge about α_{GW}, γ_{GW}, and PK_{GW}.
6) GW through γ_{GW} maintains an open channel for anyone to discover lists of IoT accesses that GW has.
7) GW already had a definite policy of giving access to S and what access should be given to S.
8) GW deploys an authorization server in the network, discoverable through α_{GW} and γ_{GW}.
9) Upon deployment, GW and S subscribe to SC's events.

Goal: By default, the public cannot legitimately access any D from the IoT endpoint because GW will reject all unauthorized accesses. The subsequent protocol describes a transparent access negotiation between GW and S. It outlines endpoint authorization for S by factoring the process in the blockchain. GW will store access tokens for S in SC upon successful authorizations. We summarize the process in Figure 4.

1) Using γ_{GW}, S can query for a list of accesses that is available in GW.
2) Upon receiving this request, GW forms:
   - $A = \{a_1, a_2, a_3, \ldots, a_n\}$, a list of open IoT accesses through GW
   - $Y_4 = H(A)$
   - $C_4 = PKSIGN_{SK_{GW}}(Y_4)$

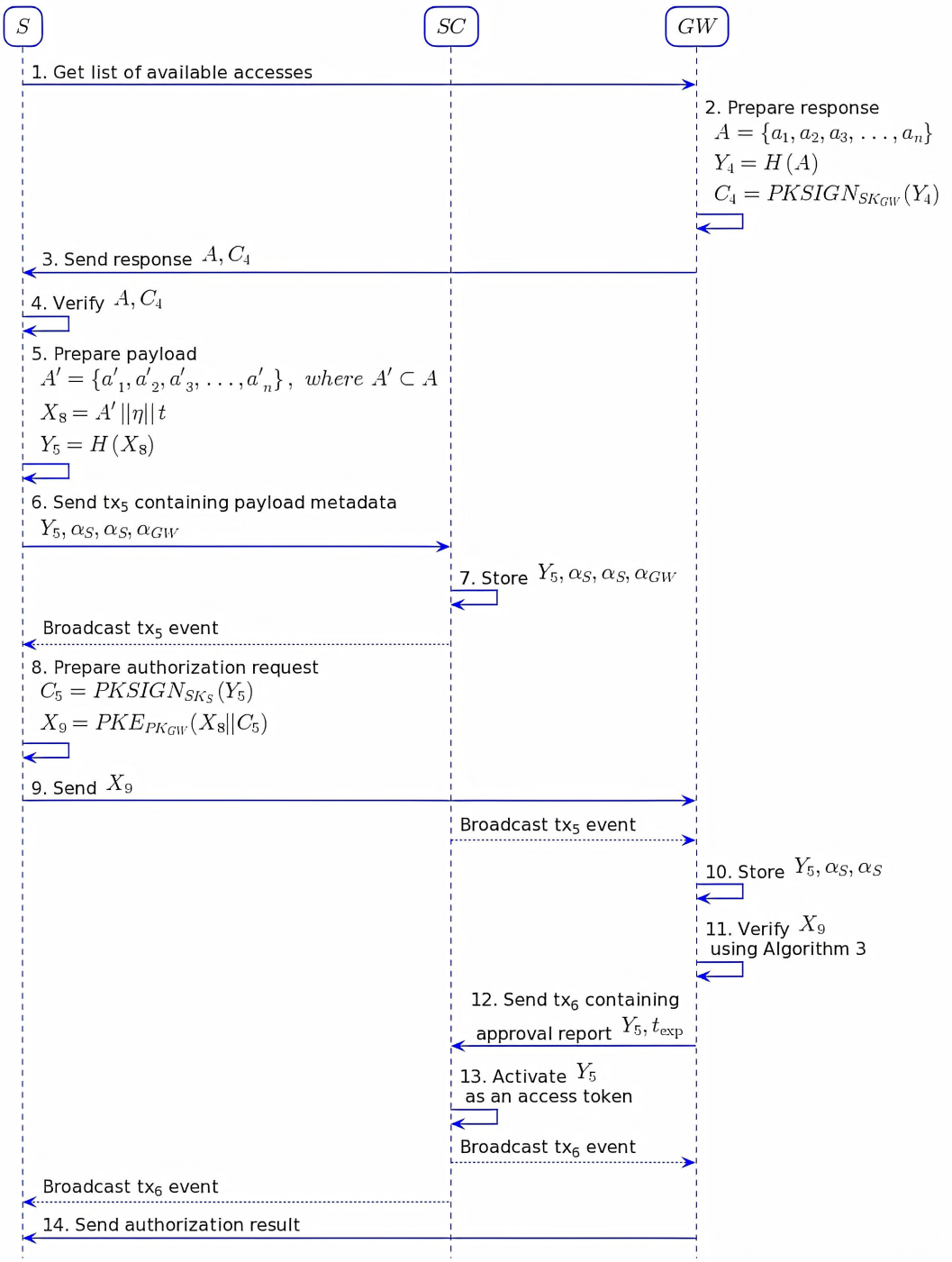

FIGURE 4. The sequence diagram for our proposed endpoint authorization.

The value of $\{a_1, a_2, a_3, \ldots, a_n\}$ varies depending on the type of protocol that *GW* employs in the domain. Thus, a can have the value of CoAP accesses using GET, POST, PUT, and DELETE commands, or it can also be MQTT publish and subscribe accesses.

3) *GW* then returns A and C_4 to S off-chain.
4) S forms $Y_4 \leftarrow H(A)$ then verifies C_4 to check if $PKVERIFY_{\alpha_{GW}}(C_4, Y_4)$ equals *True*. These actions ensure that there is no data tampering on the returned message from *GW*.
5) If everything is valid, S creates:
   - $A' = \{a'_1, a'_2, a'_3, \ldots, a'_n\}$, where $A' \subset A$, a list of IoT access that S wants to access
   - $X_8 = A' \parallel \eta \parallel t$
   - $Y_5 = H(X_8)$
6) S sends a transaction to *SC* to record a log of the authorization request in *SC*. Four parameters are included in the transaction:
   - *hash*, the hash of the endpoint authorization payload, Y_5

- *source*, the address that requests the authorization, α_S
- *target*, the address of the authorization target, α_S
- *approver*, the address of the authorization granter, α_{GW}

In other words, this log indicates an authorization request for S, which S sends to GW using the payload Y_5. This step ends with S receives the tx hash as proof of submission.

7) Similar to the authentication cases, SC maintains a list of authorization log. Upon receiving the previous transaction, SC stores its parameters in the list. For each element in the list, SC sets *approved* and *revoked* value to *False* by default. Note that Y_5 also serves as an access token for the corresponding authorization request. When S queries the IoT resource from GW, he must specify this access token. Furthermore, S and GW must also mention Y_5 during revocation use cases.
8) After receiving smart contract event from Step 6, which indicates Y_5 log have been inserted successfully in the blockchain, S then forms an endpoint authorization request for GW:
   - $C_5 = PKSIGN_{SK_S}(Y_5)$
   - $X_9 = PKE_{PK_{GW}}(X_8 \parallel C_5)$

   C_5 is the authorization payload signature, while X_9 is the encrypted authorization request.
9) S transmits X_9 to GW off-chain.
10) When GW receives the event from Step 6, where its *approver* equals α_{GW}, GW stores the corresponding *hash*, *source*, and *target* information to the local database. Like the authentication cases, the delivery of events is subject to the miner and network latency. Therefore, GW may receive this event first before S and vice versa.

Algorithm 3 The Verification of the Endpoint Authorization Payload in GW.

Input: X_9, SK_{GW}
Output: True or False
1: $X_8 \parallel C_5 \leftarrow PKD_{SK_{GW}}(X_9)$
2: $Y_5 \leftarrow H(X_8)$
3: **if** ! $exist(Y_5)$ **then return** False
4: $source \leftarrow getSource(Y_5)$
5: **if** ! $PKVERIFY_{source}(C_5, Y_5)$ **then return** False
6: $A \leftarrow getAccessInfo(source)$
7: $A' \leftarrow X_8$
8: **if** $A' \not\subset A$ **then return** False
9: **return** True

11) Upon receiving X_9, GW conducts a formal verification by following steps shown in the Algorithm 3.
   First of all, GW decrypts the authorization request X_9 (line 1). This decryption reveals the S's authorization payload X_8 as well as the associated signature C_5. She then calculates the hash of authentication payload Y_5 (line 2).
   Afterward, GW checks whether Y_5 exists in the local database (line 3). If the authorization hash does not exist, it may imply two things. First, S sends a valid X_8 but GW has not received the smart contract event containing Y_5 yet. Second, S delivers an invalid X_8.
   GW takes the *source* information from the database and verifies whether this *source* indeed signs the received authorization payload Y_5 (line 4-5). This checking ensures that the entity that sends the payload Y_5 on-chain and X_8 off-chain is the same.
   The rest is pretty straightforward, GW queries detail of authorization information from the database using $getAccessInfo(\cdot)$ method (line 6). Then, GW checks if the requested access A' is a subset of the access that S owns in the endpoint A (line 8). We use a subset comparison instead of the equal one because it is possible that S only wants to leverage a small part of accesses from his overall privilege. Thus, to make the system secured, we adapt by only giving the least access that S requires.
   The algorithm will return *False* if there is an issue or invalid checking during these verifications and return *True* when everything is valid.
12) Assuming that the verification is successful, GW saves the requested access A' in her local database. She then sends a report back to SC by sending a transaction with the following parameter.
   - *hash*, the hash of the approved endpoint authorization payload, Y_5
   - t_{exp}, the expiry time of this access

   This step ends with GW obtains the tx hash as proof of reporting.
13) Upon receiving the previous transaction, SC sets the *approved* state of Y_5 to *True*. Note that this action activates the second role of Y_5 as an access token.
14) After getting the event from Step 12, GW returns the endpoint authorization result to S. S can also get the same event from Step 12, so he can be assured that GW already authorized his request. He can then start using Y_5 as an access token.

E. ACCESSING ENDPOINT

Premise: After requesting authorization to access O's endpoint and obtain an access token Y_5, S now begins to access IoT resources in the endpoint through GW.

1) GW and all of D inside the endpoint are trusted.
2) GW has given an authorized access list A' to S, which is associated to an access token Y_5.
3) GW maintains an open channel for accessing IoT resources in her endpoint. However, she only allows requests with valid access tokens to go through.

Goal: GW will intercept and validate all requests to her endpoint by default. Specifically, she strips the access token from each request and matches the information to the one in her local database and the blockchain. The following protocol presents an example of accessing IoT resources through GW.

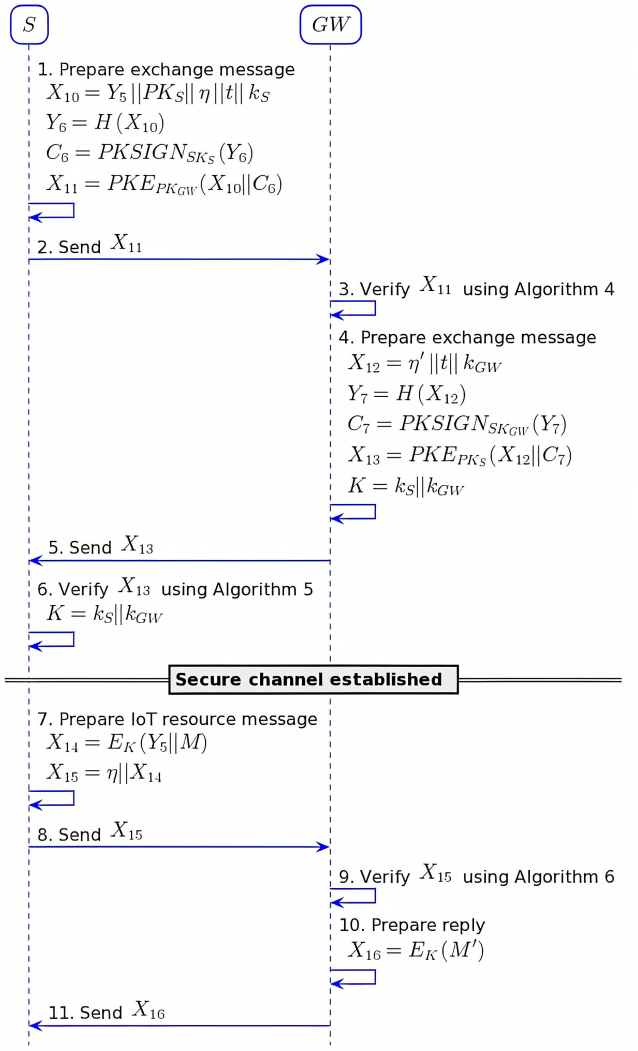

FIGURE 5. The sequence diagram for accessing IoT endpoint resource.

We also describe a potential extension of our protocol to create a secure channel between GW, as the authorization server, and S, as the requester. We detail the process in Figure 5.

1) S initiates the process by forming a request:
   - k_S, a random secret from S for GW
   - $X_{10} = Y_5 \parallel PK_S \parallel \eta \parallel t \parallel k_S$
   - $Y_6 = H(X_{10})$
   - $C_6 = PKSIGN_{SK_S}(Y_6)$
   - $X_{11} = PKE_{PK_{GW}}(X_{10} \parallel C_6)$
2) S transfers X_{11} to GW off-chain.
3) Upon receiving X_{11}, GW verifies it by following the Algorithm 4 procedures. She decrypts the message using her private key SK_{GW} (line 1). She then queries the nonce η and checks whether it exists in the cache (line 2-3). If the same nonce is found, this request may be a replay attack. Therefore, she rejects it.
   After that, GW conducts access control verification. Specifically, she checks if Y_5 exists in the database (line 4). Then, GW validates if she already approved this Y_5 before (line 5). She checks that the token is not revoked (line 6). Finally, she makes sure that Y_5 is not expired (line 7).
   The final verification is to check whether the access request X_{11} is coming from the authentic owner of the access token Y_5 (line 8-10). GW first hashes the payload to generate Y_6, and gets the *source* information in the database using *getSource*$(\cdot)$ method. Finally, she validates S's signature C_6.
4) When previous validations are success, GW forms a reply message:
   - k_{GW}, a random secret from GW for S
   - η', the η from X_{10}
   - $X_{12} = \eta' \parallel t \parallel k_{GW}$
   - $Y_7 = H(X_{12})$
   - $C_7 = PKSIGN_{SK_{GW}}(Y_7)$
   - $X_{13} = PKE_{PK_S}(X_{12} \parallel C_7)$

   At this moment, GW builds an ephemeral secret key K using k_S and k_{GW}. Specifically, $K \leftarrow k_S \parallel k_{GW}$. The gateway then stores this secret key temporarily in the database using η' as its keys. As a result, the gateway can serve many different secret keys from multiple users simultaneously.
5) GW transmits the reply X_{13} back to S off-chain.
6) S verifies the secret key exchange from GW using steps in Algorithm 5.

Algorithm 4 The Secret Key Exchange Verification in GW

Input: X_{11}, SK_{GW}
Output: True or False
1: $X_{10} \parallel C_6 \leftarrow PKD_{SK_{GW}}(X_{11})$
2: $\eta \leftarrow X_{10}$
3: **if** *exist*(η) **then return** False
4: **if** ! *exist*(Y_5) **then return** False
5: **if** ! *approved*(Y_5) **then return** False
6: **if** *revoked*(Y_5) **then return** False
7: **if** *expired*(Y_5) **then return** False
8: $Y_6 \leftarrow H(X_{10})$
9: *source* $\leftarrow$ *getSource*(Y_5)
10: **if** ! $PKVERIFY_{source}(C_6, Y_6)$ **then return** False
11: **return** True

Algorithm 5 The Secret Key Exchange Verification in S

Input: $X_{10}, X_{13}, \alpha_{GW}, SK_S$
Output: True or False
1: $X_{12} \parallel C_7 \leftarrow PKD_{SK_S}(X_{13})$
2: $\eta' \leftarrow X_{12}$
3: $\eta \leftarrow X_{10}$
4: **if** $\eta' != \eta$ **then return** False
5: $Y_7 \leftarrow H(X_{12})$
6: **if** ! $PKVERIFY_{\alpha_{GW}}(C_7, Y_7)$ **then return** False
7: **return** True

S decrypts the received message X_{13} (line 1), and queries the gateway's nonce η' (line 2). The service then searches its previous nonce η from X_{10} (line 3) and makes sure that it matches the replied nonce (line 4). Afterward, S validates the signature C_7 to ensure that the gateway is indeed the sender (line 5-6).
When all verification is successful, the method will return *True*. Otherwise, it gives back *False*.

7) At this moment, S can construct a secret key $K \leftarrow k_S \parallel k_{GW}$. He then forms:
   - M, an IoT resource request message (e.g., CoAP or MQTT message)
   - $X_{14} = E_K(Y_5 \parallel M)$
   - $X_{15} = \eta \parallel X_{14}$
8) S then sends X_{15} to GW off-chain.

Algorithm 6 The Verification of IoT Resource Access at GW

Input: X_{15}
Output: True or False
1: $X_{14} \parallel \eta \leftarrow X_{15}$
2: $K \leftarrow getSecretKey(\eta)$
3: $Y_5 \parallel M \leftarrow D_K(X_{14})$
4: **if** ! $exist(Y_5)$ **then return** False
5: **if** ! $approved(Y_5)$ **then return** False
6: **if** $revoked(Y_5)$ **then return** False
7: **if** $expired(Y_5)$ **then return** False
8: **return** True

9) Upon receiving X_{15}, GW verifies it by following Algorithm 6. GW retrieves the secret key K from the database (line 1-2). GW then decrypts the message with the retrieved key (line 3). She then must verify the validity of the access token Y_5 (line 4-7). First, she checks that this Y_5 exists in the database. She then makes sure that it is already approved, it is not revoked, and it is not expired. The algorithm returns *True* when it does not find any anomalies in the request. Otherwise, it returns *False*.
10) Assuming that no error occurs, GW can encrypt the reply message for S by forming M' using K, $X_{16} \leftarrow E_K(M')$. M' denotes an IoT resource response message.
11) GW then delivers this reply X_{16} to S off-chain.

At this moment, both S and GW have established the secure channel. They can keep exchanging the subsequent M and M' back and forth using the same key K. However, at each interaction, S still has to include Y_5 on his request messages to GW as proof of authorization.

Access Token Compatibility: This paper does not detail how the gateway obtains resources from the devices after the secure channel establishment. This action is intentional as we want our protocol to be compatible with many IoT gateways. Interested adopters can extend this protocol further by referring to the details of authorization strategies from other studies. For example, one can use [29] for CoAP or [30] for MQTT.

Delegating Existing Access Token: Moreover, we can leverage the blockchain as distributed storage to realize a *one-time grant* access control across multiple gateways. Hence, enabling *locally enforced yet globally available* access token. Let us assume that O has a new gateway GW'. He wants to give S the same access as the one in Y_5 in this new gateway. Both old and new gateway is a blockchain node; therefore, they can synchronize the access token's state from the smart contract. Instead of S and GW' negotiating a new authorization request, S can present his old Y_5, which gives access to A', empowered with S's signature to this new GW'. GW' will then first verify the token's validity and then grant access directly by storing the policy for Y_5 in her local database. Thus, we can avoid creating additional transactions in the blockchain.

F. REVOCATION

We have provided two security verifications: endpoint authentications (i.e., the gateway and device authentication) and endpoint authorization. These procedures need to be revocable as the nature of the IoT environment is dynamic and easy-to-compromise. In the blockchain, SC maintains a variable called *revoked* for gateway, device, and access object to indicate whether they have been repealed or not. By default, this variable has a value of *False* (not revoked).

Access Revocation: O as the owner of IoT endpoints and resources (i.e., GW and the underlying D) has the full right to revoke access to his endpoint. Similarly, S as the access authorization requester can also remove its previously granted access. To do so, they follow the subsequent course of action.

1) O (through GW) or S sends a transaction to the blockchain by calling the *revokeAccess*(·) method in SC. This function takes Y_5 as an argument, which is the hash of the prior endpoint authorization request that also serves as an access token.
2) For O's case, the transaction reveals α_{GW} as the transaction's sender. SC then checks previous authorization records in the list of authorization requests to determine that the *approver* of Y_5 is indeed α_{GW}. This verification is required to make sure that only the original granter is the one who can revoke the access.
   For S's case, SC ensures that the previous authorization logs Y_5 mention α_S as its *sender*. This action ensures that the original proposer can also revoke access.
   If all validations are correct, SC sets the *revoked* state of Y_5 to *True*, and the access token now becomes inactive.

Device Revocation: D is mostly a constrained device that O puts in a place with a lenient security environment. Therefore, attackers may compromise D successfully with a high probability. Our protocol facilitates a device revocation procedure for O to help stop the impact of attacks when D is under the attacker's control. Note that GW is not compromised in this scenario.

1) O, through GW, sends a transaction to the blockchain by calling a *revokeDevice*(·) function in SC.

This function takes Y_2 as an argument, the hash of the previous D's authentication request.

2) SC then browses through preceding authentication records to find out whether the *sender* of Y_2 is α_{GW}. This check is to make sure that only the original requester (i.e., α_{GW}) is the one who can revoke D. Hence, GW cannot revoke D that is not under his control. When everything is successful, SC sets the *revoked* state of Y_2 as *True*, and other parties will consider D as untrusted.

It is possible that, after GW revoked D, S tries to perform a legitimate access to D using his A'. As no relationship information maps D to A' in the blockchain (maintaining such info is very costly in terms of storage), SC cannot automatically adjust A' to exclude D. To modify A' in SC, it involves both S and GW to renegotiate on the new access terms and send transactions on the blockchain. It is costly and inefficient when many revocations occur. Thus, we prefer GW to enforce such revocation locally in his machine instead of updating the blockchain. In other words, when S is trying to access a revoked D, GW must return an error code telling that S is accessing a revoked device.

Upon receiving such error, S can double check the status of D by calling *isTrustedDevice*(·) function in SC while also presenting the associated α_D as an argument. The returned *False* value should affirm S that GW has indeed revoked D.

Endpoint Revocation: GW is a crucial entity due to its central role in relaying many inward and outward IoT traffics in IoT endpoints. Thus, attackers will most likely set GW as their first-priority target. To cope with a disastrous event such as attackers are taking over GW, we present a total revocation to close the endpoint entirely. In this scenario, GW is already under the control of attackers. Thus, anyone should not trust GW and all of D in that endpoint. However, O is not compromised.

1) O sends a transaction to the blockchain by calling a *revokeGateway*(·) function in SC. This method takes Y_1 as an argument, which is the hash of the past GW's authentication payload.
2) SC gathers previous authentication records and validates that the *sender* of Y_1 is α_O. Hence, only the true authentication requester can revoke a verified endpoint. When the verification is valid, SC sets the *revoked* parameter of Y_1 as *True*, and removes GW from the trusted list.

SC maintains a mapping between GW and D such that when we remove GW from SC's trusted list, all of the underlying D will also become untrusted automatically. In this case, query of both *isTrustedDevice*(·) and *isTrustedGateway*(·) methods will return *False*.

SC also has a mapping between GW and A' through Y_5. When S calls *isValidAccess*(·) function by presenting Y_5 as an argument, SC checks the contents of *revoked* variable of Y_5 and also validates if the *target* of Y_5 (i.e., α_{GW}) is revoked or not. The method will automatically return *False* if SC finds out that GW is repealed.

VI. EVALUATION

In the following section, we evaluate our proposed access control in three categories. First, we reinvestigate the trust models for each of the entities involved in our scheme. Second, we conduct a security evaluation to assess possible threat models for attackers. Lastly, we implement our protocol and measure the performance through a benchmark to analyze its feasibility.

A. TRUST EVALUATION

By default, entities in our access control act with a complete distrust of one another. We analyze our system's trustworthiness and investigate whether each entity can cheat our proposal.

Smart Contract as the Root of Trust: A trusted administrator (e.g., a government) must initiate our proposal by deploying the smart contract to the blockchain. This party can become malicious over time. However, we argue that our system remains secure. Because the blockchain is transparent, other nodes can validate the smart contract's source code to determine whether it is safe or harmful. Furthermore, the blockchain is also hard-to-tamper; thus, everyone can rest assured that the smart contract operations will remain deterministic. Finally, once the administrator deployed the smart contract, our access control can run independently without further intervention. There is no backdoor in our implementation for the admin to take over the system.

ISP and IoT Vendor as Trusted Approvers: Anyone is free to create an identity in the blockchain. As a result, it is challenging for a particular entity to distinguish valid approvers as others can claim themselves as ISPs or IoT vendors. To alleviate this issue, the administrator can act as a trusted mediator by signing the identity of the ISPs and vendors. In this case, the admin behaves like CA and bootstrap the trust in the approvers. Moreover, ISPs and vendors can also maintain credibility scores using a reputation system. This score should encourage them to behave honestly at all times.

Domain Owner and IoT Service as Requesters: The IoT domain owner may bribe the ISP to provide fake approval for his endpoint. However, because the authentication payload logs are recorded in the blockchain and visible to other nodes, it eases fraud detection. Moreover, the stakes are high for the ISP if she gets caught. The community can reduce the ISP's reputation score, and eventually, she may lose the credibility to become a trusted approver. Similarly, the owner will most likely be unable to perform the same malicious plot for device authentication. The IoT vendor may also lose its trustworthiness if she becomes dishonest. Finally, the service cannot perform a fake authorization approval plot for the same reason.

IoT Gateway as the Access Granter: As the IoT resource keeper, the domain owner can approve or deny any inward or outward access to his domain through his IoT gateway. Therefore, the gateway can discriminate against a particular service if the owner does not trust the service. We allow this absolute control scheme to protect the rights of the

resource owner. Before transmitting the request, the service can check the credibility of the gateway through the smart contract. Specifically, the service can verify whether the ISP has approved the gateway or not. He can also ensure the IoT vendor has endorsed the IoT device that he wants to access. Moreover, all gateways should also maintain reputation scores to punish malicious behaviors, such as distributing invalid resources that contain malware. Thus, from the reputation scores, the service can determine whether to trust this gateway or not.

Device and IoT Gateway Relationship: In this paper, we focus on the security outside an IoT domain. As a result, we assume that the device and gateway's communication channel remains secure and trusted. The device fully trusts the gateway in relaying all payloads from and to the device correctly.

Throughout our trust discussion, we acknowledge the benefit of a reputation system in our proposal. Therefore, we consider adding it to our future works.

B. SECURITY EVALUATION

We follow the security guideline and threat modeling that Microsoft develops, called STRIDE [31]. It is an acronym of Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege.

Spoofing: We use public keys (and associated blockchain addresses) to identify all entities; they are usable on-chain and off-chain. As a result, to spoof a target, attackers have to steal the private key. Furthermore, we also use pre-shared secrets to authenticate IoT gateways and devices. The ISP verifies the gateway's identity by examining the domain owner's username and password in the ISP system. On the other hand, the IoT vendor validates their devices by using the embedded asymmetric key, symmetric key, device fingerprint, or MAC address. Because of these approval rules, attackers have to compromise extra secrets to spoof the gateways and devices. Finally, attackers can potentially generate arbitrary addresses and claim themselves as IoT vendors or ISPs. As we mentioned in the previous subsection, the administrator can act as a CA for vendors and ISPs to mitigate this issue.

Tampering: We leverage the key-value storage of the smart contract to store essential information distributedly across all nodes. Mainly, the log of authentication requests, the log of authorization requests (the access tokens), the trusted gateways list, and the trusted devices list. As long as the blockchain remains secure, the system can guarantee that all of this information stays tamper-free. Aside from storing data in the smart contract, entities also save credentials and other information in their local storage. We refrain from providing tamper-proof guarantees in the local database as it is each of the entity's responsibility to keep their storage safe.

Repudiation: When we call methods in the smart contract by sending transactions in the blockchain, we equip them with the digital signatures. The signature protects the transactions against the repudiation attacks while also preventing malicious modification through our system. Furthermore, we also use digital signatures during the off-chain authentication, authorization, and accessing resources. As a result, we make it very challenging for attackers to repudiate and tamper with our protocol's exchanged messages.

Information Disclosure: Across the whole protocol, we leverage the public-key encryption scheme to protect the off-chain messages' confidentiality. Only the authorized entities can decrypt and understand the messages. Moreover, before accessing the IoT resources through the gateways, the IoT services build secure channels. Our secure channel works similarly to the Diffie-Hellman key exchange protocol, which generates secret keys at each session. Therefore, we keep all of the off-chain transmissions private.

All blockchain nodes can see what the smart contract saves in the blockchain. Attackers, disguised as one of the valid nodes, can access the authentication or authorization payload hash. They can then try to brute force fake payloads to find the hash that matches the recorded payload. If successful, attackers may figure out the secret contents of the authentication or authorization request. However, the addition of timestamps and nonces in our requests should complicate the attackers' attempt to successfully perform this action.

Denial of Service: Aside from protecting against the information disclosure, the use of nonces in our authentication and authorization payload is to defend against replay attacks, which can become one of the possible Denial of Service attack types. Furthermore, all of the off-chain authentication and authorization requests must have corresponding logs in the blockchain. Sending transactions in the blockchain requires the sender to pay a small transaction fee. Thus, spamming blockchain with faulty logs is costly for attackers.

Elevation of Privilege: All of the validations are based on the pre-determined secret information in the approvers' database. If a particular entity can present a valid secret, only then the approver will grant the requests. Therefore, to gain an elevation of privilege, attackers must compromise the approvers' servers and modify the storage maliciously. The security of each entity's machine is out of our paper's scope.

C. PERFORMANCE EVALUATION

We implement our protocol, then present our evaluations on its performance and assess its usability. First of all, we investigate the gas-used property of each method in our smart contract. Then, we measure the throughput of the cryptographic tools that we use in our system. Finally, we benchmark the client-side and server-side implementations from our proposal.

1) SMART CONTRACT COMPLEXITY

We use Ethereum as our peer-to-peer (P2P) blockchain platform because it has a widely-used smart contract feature. Based on the Ethereum guideline [8], users must pay a tx fee when they want to call a smart contract method that will change the smart contract's storage state across all nodes. As a rule of thumb, the more complex the called method is, or the

TABLE 2. The evaluations of writable methods in our smart contract. The starred column is statistic taken from ETH Gas Station [32] on Oct 7, 2020. The average gas price at that time is 69 Gwei. The transaction (Tx) fee is in US dollars, while the confirm time is in seconds.

Method	Description	Gas Used	Tx Fee*	Confirm Time*
`approveAccess`	Approving the access authorization.	56,742	1.32	366
`approveDevice`	Approving the device authentication.	59,401	1.39	366
`approveGateway`	Approving the gateway authentication.	54,667	1.27	366
`revokeAccess`	Revoking a valid access.	35,086	0.82	366
`revokeDevice`	Revoking a trusted device.	25,705	0.59	366
`revokeGateway`	Revoking a trusted gateway.	24,960	0.58	366
`storePayload`	Storing authentication or authorization payload.	90,105	2.10	366
`deployment`	Deploying the smart contract.	2,186,294	50.99	2079

TABLE 3. The list of hardware used in our performance evaluations. The Private Server's CPU cores vary depending on the configuration in the Virtual Machines (VMs).

Model	CPU Specification	# CPU Cores
Raspberry Pi Zero W Rev 1.1	Broadcom BCM2835 ARM 1176JZF-S @ 1 GHz	1
Raspberry Pi 3 Model B Rev 1.2	Broadcom BCM2837 ARM Cortex-A53 @ 1.2 GHz	4
Raspberry Pi 4 Model B Rev 1.1	Broadcom BCM2711 ARM Cortex-A72 @ 1.5 GHz	4
Private Server (in VMs)	Intel(R) Xeon(R) Gold 6136 CPU @ 3.00 GHz	1, 2, 4, or 8

more data the method stores, the more expensive the tx fee becomes.

In Table 2, we measure the used gas from each smart contract method, determining the tx fee and the confirm time. Note that we only put the writable methods, which modify the state of the smart contract. The most complex operation is the contract's deployment, which consumes about 24,26 times more gas than other methods, resulting in a costly tx fee. The deployment also takes 5.68 more times to confirm than the rest of the methods. However, we expect this behavior and argue that we can take it as an investment since it only happens once for all. The storing authentication or authorization payload scheme is also wasting more gas than the rest of the methods. During this step, the smart contract must store the payload metadata; therefore, more data is stored in this method. Meanwhile, approving authentication or authorization payload scenario drains relatively fewer gas. However, the ISP, vendor, and gateway may need to frequently call these methods for each of the payloads they receive. Thus, they should be aware of their economic resources. Finally, revoking scenarios are cheap operations, as they are mainly only performing simple negation operations.

Note that the tx fee is only required in the public Ethereum blockchain, and the gas price will vary depending on the market. As an alternative, adopters can apply our protocol in a private blockchain network. In this case, the gas cost has a less meaningful purpose as the tx fee may not exist in a private network. Nevertheless, the gas-used property is still useful to assess the complexity of our smart contract code.

2) CRYPTOGRAPHIC COMPLEXITY

We use two crytpography libraries in this paper, the crypto [33] and eth-crypto [34] modules; all is based on Node-JS. The crypto module contributes to the following operations. The `sk-sign` and `sk-verify` are the implementation of $SIGN(\cdot)$ and $VERIFY(\cdot)$ functions using HMAC algorithm. Meanwhile, the `sk-encrypt` and `sk-decrypt` realizes the $E(\cdot)$ and $D(\cdot)$ methods using AES-256. For the rest of the cryptographic processes, we use the eth-crypto module. The `hash` implements $H(\cdot)$ function to do the KECCAK-256 hash operation. The `pk-sign` and `pk-verify` are the applications of $PKSIGN(\cdot)$ and $PKVERIFY(\cdot)$ methods for ECDSA algorithm. Finally, the `pk-encrypt` and `pk-decrypt` realizes the $PKE(\cdot)$ and $PKD(\cdot)$ functions using ECC-based encryption.

To evaluate our chosen cryptographic tools' feasibility, we build four REST API servers using express module [35] in multiple Raspberry Pis hardware and our private server. We summarize the specification details of our hardware in Table 3. The REST API servers expose nine endpoints, one for each of the previously mentioned cryptographic operations. We then benchmark those servers by running the autocannon module [36]. We set the number of client connections to 10 and perform the benchmark for 30 seconds. We repeat the process ten times for each of the cryptographic operations. In total, we invoke the autocannon module 360 times, 90 times for each server. Finally, we plot the results in Figure 6.

We can see from the figure that the public-key operations generate more overheads than symmetric-key processes. The `sk-sign` can produce up to 2.38 more signatures than the `pk-sign`. Meanwhile, we can verify up to 3.17 times more signatures if using `sk-verify` rather than `pk-verify`. For encryptions, the `sk-encrypt` is about 9.49 times more efficient than the `pk-encrypt`. Similar trend happens in decryptions with the `sk-decrypt` can process 9.11 times more payloads than the `pk-decrypt`.

3) CLIENT-SIDE BENCHMARKING

In the following two subsections, we present the off-chain performance evaluation of our proposed access control. This part describes the client-side implementation benchmark while the server-side counterpart is available in the next subsection. We outline the number of cryptographic operations

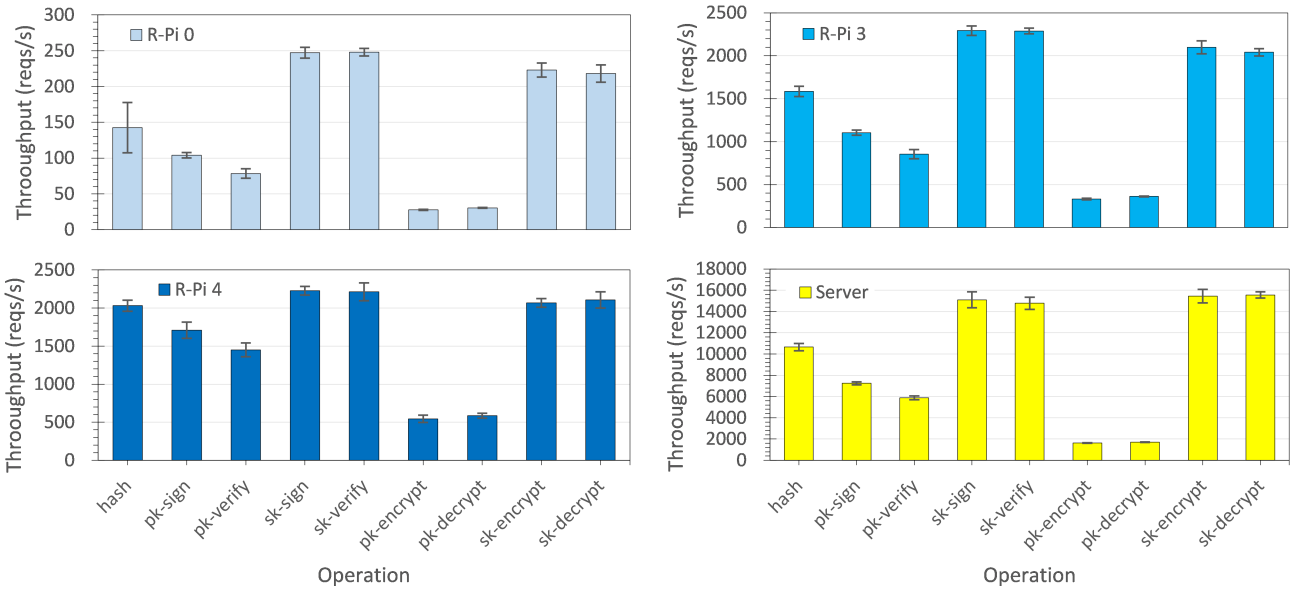

FIGURE 6. The average throughput benchmark result (in requests per second) of Node-JS cryptographic tools used in our implementations. The cryptographic operations includes hashing (`hash`), signing and verification for both public key and symmetric key (`pk-sign`, `pk-verify`, `sk-sign`, and `sk-verify`), as well as encryption and decryption process for public key and symmetric key (`pk-encrypt`, `pk-decrypt`, `sk-encrypt`, and `sk-decrypt`). We measure the performance in multiple hardware environments: Raspberry Pi Zero (R-Pi 0), Raspberry Pi 3 (R-Pi 3), Raspberry Pi 4 (R-Pi 4), and our private server with 2 CPU cores.

TABLE 4. The comparison regarding the number of cryptographic operations that each entity performs during our access control scenarios. AuthN = Authentication, AuthZ = Authorization, Ent = IoT entity, Ro = Role, Clt = Client, Srv = Server, $a = H(\cdot)$, $b = PKSIGN(\cdot)$, $c = PKVERIFY(\cdot)$, $d = SIGN(\cdot)$, $e = VERIFY(\cdot)$, $f = PKE(\cdot)$, $g = PKD(\cdot)$, $h = E(\cdot)$, and $i = D(\cdot)$.

Scenario	Ent	Ro	a	b	c	d	e	f	g	h	i
Gateway	O	Clt	1	2	-	-	-	1	-	-	-
AuthN	ISP	Srv	1	1	1	-	-	-	1	-	-
Device AuthN	GW	Clt	1	2	1	-	-	1	-	-	-
≻ PKSIG	D	Clt	1	1	-	-	-	-	-	-	-
≻ PKSIG	V	Srv	2	1	3	-	-	-	1	-	-
≻ SKSIG	D	Clt	1	-	-	1	-	-	-	-	-
≻ SKSIG	V	Srv	2	1	2	-	1	-	1	-	-
≻ Fingerprint	D	Clt	2	-	-	-	-	-	-	-	-
≻ Fingerprint	V	Srv	3	1	2	-	-	-	1	-	-
≻ MAC	D	Clt	1	-	-	-	-	-	-	-	-
≻ MAC	V	Srv	2	1	2	-	-	-	1	-	-
Endpoint	S	Clt	1	2	-	-	-	1	-	-	-
AuthZ	GW	Srv	1	1	1	-	-	-	1	-	-
Handshake	S	Clt	2	1	1	-	-	1	1	-	-
	GW	Srv	2	1	1	-	-	1	1	-	-
Accessing	S	Clt	-	-	-	-	-	-	-	1	1
Resource	GW	Srv	-	-	-	-	-	-	-	1	1

used in our access control in Table 4. The table also details the role of each entity, whether they are clients or servers.

Depending on the protocol flow, clients can interact with servers, the smart contract, or both. As a result, the network and miner latency may influence the results of our client-side performance evaluation. To alleviate this issue, we modify our code to ignore the network transmission parts and assume that the client receives the server and smart contract responses directly. Thus, we focus on the client's internal processing, including cryptographic operations and local database processes, implemented using Memcached [37].

We run our client-side scenarios in multiple hardware for several epochs. For Raspberry Pi Zero, 3, and 4, we run for 10,000, 50,000, and 100,000 iterations respectively. We also run the same scenario in our private server, wrapped in Virtual Machines (VMs). For VM with 1 and 2 CPU cores, we perform the scenario for 500,000 iterations. Meanwhile, we do it using 1,000,000 epochs for VM with 4 and 8 cores. We differentiate the number of iterations to match the capabilities of the hardware. Therefore, we can gain enough samples while keeping the benchmark running time short. Finally, we depict the results in Figure 7.

Our code supports multi-threading using the cluster module [38]. As a result, in general, we can achieve better performance in multi-core hardware. We can see this trend in all of the charts in the figure. For device authentication scenarios, the public-key signature scheme (PKSIG) suffers the highest overhead. This result matches our previous observation that public-key cryptographic operations are less efficient than symmetric ones. If we switch to the symmetric key signature scheme (SKSIG), we can increase the performance up to 2.56 times. Interestingly, the Fingerprint and MAC address authentication does not have significant gaps with the SKSIG option.

Note that the presented numbers from the figure are to assess the multi-threading and cryptographic tools feasibility. In the production case, the client may only perform the authentication or authorization request once for a particular device, gateway, or access. Thus, the client does not need to perform hundreds or thousands of requests per second. As a result, the given metric should be more than enough to conduct daily cases, even for the most constrained device

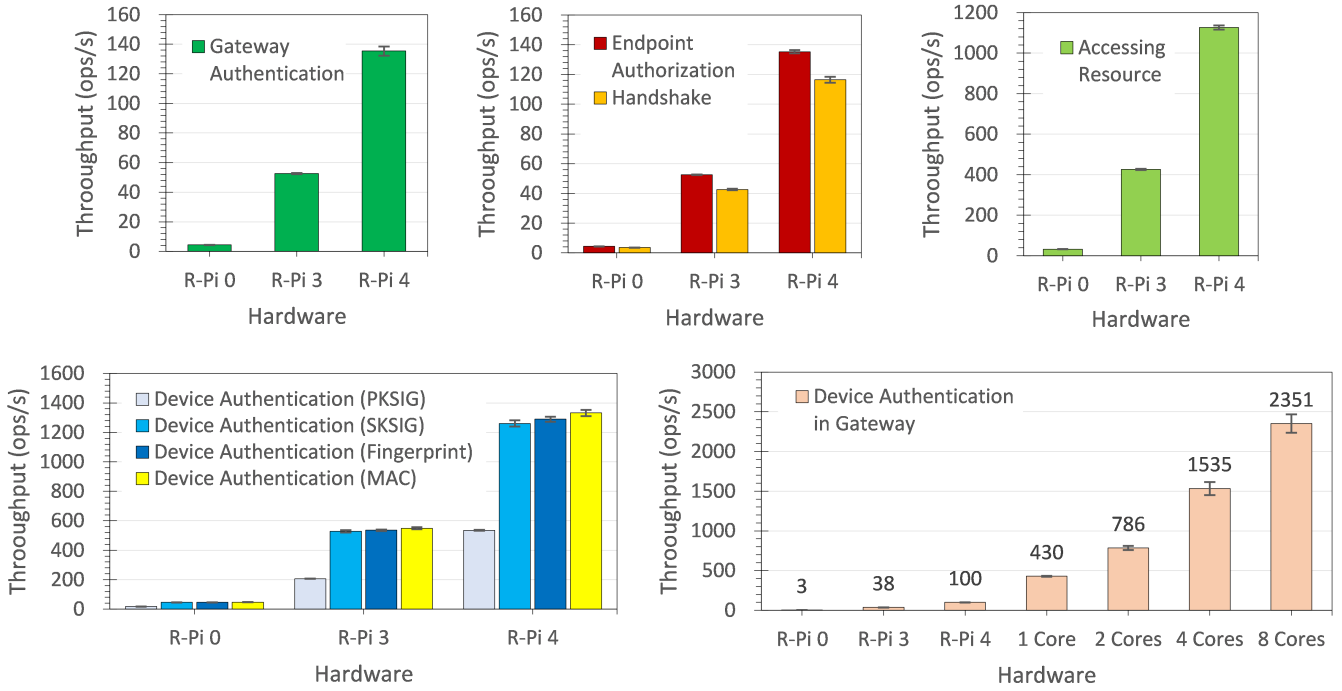

FIGURE 7. The average throughput benchmark results (in operations per second) of client-side implementations in our proposed access control scenarios. We measure the performance in multiple hardware environments: Raspberry Pi Zero (R-Pi 0), Raspberry Pi 3 (R-Pi 3), Raspberry Pi 4 (R-Pi 4), and our private server (implemented in VM with 1, 2, 4, and 8 Cores).

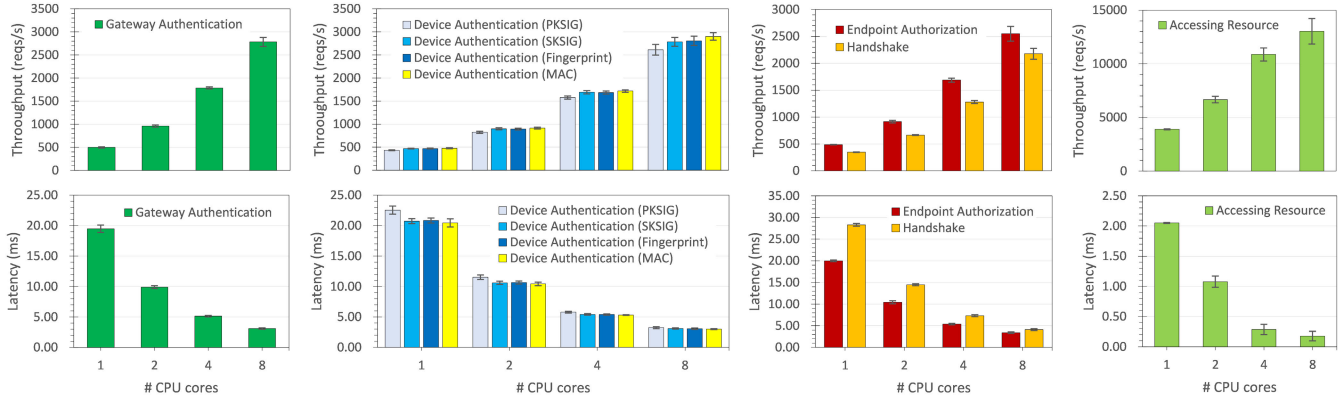

FIGURE 8. The average throughput (in requests per second) and latency (in milliseconds) measurements of server-side implementations in our proposed access control scenarios. We conduct the benchmark in our private servers using VMs by varying the number of CPU cores resources (1, 2, 4, and 8 cores).

(R-Pi 0). However, this rule does not apply to the gateway during the device authentication scenario because she piggybacks the device authentication payload from her domain to the vendor. In this case, we must experiment with the gateway using the most diverse hardware to find the upper-bound limit. Adopters can then use our results as pointers to choose the appropriate hardware for the gateway. In particular, they should consider the number of devices in their domain and how frequently they will send authentication requests.

4) SERVER-SIDE BENCHMARKING

We implement the authentication, authorization, and resource servers as REST API endpoints using express module [35]. Meanwhile, the blockchain network is simulated using ganache-cli [39]. Like the client-side case, depending on the protocol flow, the server may contact the smart contract. Thus, the network and miner latency also plays an essential role in our server-side implementations. For our server-side evaluation, we omit this factor as they tend to produce stochastic results. Instead, we focus only on the server's internal processing to assess its feasibility to carry out the client requests.

We perform the benchmark using the autocannon module [36]. We set the number of connections to 10 and run each benchmark for 30 seconds. After that, we run the autocannon ten times for each of the access control scenarios. Once they finish, we measure the average throughput and latency, and depict the results in Figure 8.

From the figure, we can see the charts' trend that the throughput increases as we use more CPU cores from the server. Consequently, the latency also decreases as we add more CPU cores. We argue that the given results should be enough to cope with the sheer number of IoT gateways and devices since they most likely perform authentication and authorization requests once in a while. However, the possible bottleneck may happen in the handshaking scenario. At this step, the server must conduct a pair of public-key decryption and encryption. Based on our previous cryptographic tools evaluation, those operations are the most expensive ones. Therefore, we expect this result. Fortunately, once the handshake completes, the server can process much more throughput during the accessing resource scenario. Handshaking happens less frequently than accessing resources. Thus, this bottleneck should be manageable.

VII. DISCUSSION

We discuss the possible weaknesses of our proposed approach and the future research directions in the following section.

Off-Chain Scalability: We notice possible performance bottlenecks in our clients and servers implementation due to Node JS. Node JS is natively single-threaded. However, we achieve multi-threading through the cluster module [38], which we can say is a workaround solution by spawning many Node JS processes simultaneously. The module does not allow shared memory access among processes, which usually exist in native multi-threading languages. Instead, the module creates inter-process communication (IPC) that we deduce to be one of the bottleneck reasons. Another possible reason is that our chosen Node JS cryptographic libraries perform slower than those in other programming languages. For production cases, others may choose to apply the protocol's logic using high-performance languages (e.g., C, Java, or Go), which allow native multi-threading support and access to faster cryptographic libraries.

On-Chain Security and Scalability: Since we depend on the blockchain to decentralize our platform, we inherit the selected blockchain's security and scalability properties. The blockchain system's security issues and challenges may include byzantine faults, 51% attacks, and selfish mining attacks [40]. Meanwhile, the blockchain is also known for its slow processing. For example, Bitcoin can only handle about 7 txs per second, while Ethereum can manage up to 20 txs per second [41]. Those numbers are still far behind VISA, which can process about 24,000 tx per seconds [41]. Researchers can build the private blockchain, which uses the classic consensus from the distributed system, such as Practical Byzantine Fault Tolerance (PBFT), to produce high throughput in favor of a lower number of nodes [42]. However, this approach tends to make the blockchain centralized. Therefore, on-chain processing's security and scalability remain continuous research efforts.

Blockchain Nodes: Depending on the underlying P2P network, the blockchain nodes may perform CPU demanding tasks such as mining the correct nonces for the block hash. The nodes may also need to have a lot of storage space to cope with the blockchain network size. Due to the IoT entities' divergence nature, they may not be able to conduct such tasks. Therefore, developers must adapt by leveraging the light nodes, proxies, or even switching to the private blockchain whenever necessary during production.

Leaked Content: Generally, the stored content in the blockchain is visible to all blockchain nodes. All entities acting as blockchain nodes in our protocol can see the authentication and authorization requests log plus the trusted gateways and devices list. By examining this trusted list, the node can search for a hotspot domain, an IoT gateway with lots of trusted IoT devices in her possession. The IP address of that gateway is also observable from the list. We make this information public because we want the domain to be accessible by others. However, the downside is that the given information can ease the attackers' efforts to find a target and start hacking the targeted gateway. We argue that this leak is not a severe vulnerability as attackers can also obtain similar information using alternative tools such as Shodan [43]. Moreover, because we use the hash log of authorization requests as access tokens, attackers then know the tokens. However, attackers cannot use them without the corresponding private key. Recall that the token grantee must sign the access token during the handshake before accessing the resource. Nevertheless, adopters must understand these privacy issues when deploying our protocol in production cases.

Trusting Approvers: Our protocol assumes that entities fully trust the ISP and IoT vendor as verifiers of the IoT gateway and IoT device. However, in production cases, adopters must enforce such approvers' trust by using other schemes such as PKI [24] or PGP equipped with a reputation system [44].

IoT Devices' Mobility: In the current state of the protocol, the device can only connect to one gateway at a time. If the device moves to another gateway, the device must perform another device authentication to be recognized in the new gateway. The smart contract will then override the old gateway with the new one in the trusted list. When many devices frequently move from one endpoint to another (e.g., in the Vehicular Ad-hoc Network (VANET)), it can create an issue as it may increase authentication traffics. One solution for this problem is to have the gateway covers a wide operation area to limit the handover process.

VIII. CONCLUSION

We proposed BorderChain, an access control framework for the IoT endpoint using blockchain. The protocol comprised multiple scenarios ranging from gateway authentication, device authentication, endpoint authorization, and accessing endpoint. We have implemented our protocol in Node JS applications using Ethereum as our P2P platform. We then provided trust, security, and performance evaluation of the protocol. The results showed that adopters could use our framework in various Raspberry Pi devices and server-grade

computers. We also demonstrated that the authentication and authorization servers could use the multi-threading feature to boost the system throughput. Finally, we have discussed our protocol's possible limitations and future work, which mostly related to the general blockchain issues.

ACKNOWLEDGMENT

The authors want to thank the anonymous reviewers for their useful comments.

REFERENCES

[1] G. Omale. (Nov. 2018). *Gartner Identifies Top 10 Strategic IoT Technologies and Trends*. Accessed: Oct. 15, 2020. [Online]. Available: https://gtnr.it/2IxPtfx.

[2] L. Goasduff. (Aug. 2019). *Gartner Says 5.8 Billion Enterprise and Automotive IoT Endpoints Will be in Use in 2020*. Accessed: Oct. 15, 2020. [Online]. Available: https://gtnr.it/31afWGj

[3] A. Mamiit. (Jun. 2019). *NASA Hacked: 500 MB of Mission Data Stolen Through a Raspberry Pi Computer*. Accessed: Oct. 15, 2020. [Online]. Available: https://bit.ly/3j8ox2F

[4] K. Kochetkova. (Oct. 2016). *How to Not Break the Internet*. Accessed: Oct. 15, 2020. [Online]. Available: https://bit.ly/3iXPcyT

[5] R. Roman, J. Zhou, and J. Lopez, "On the features and challenges of security and privacy in distributed Internet of Things," *Comput. Netw.*, vol. 57, no. 10, pp. 2266–2279, Jul. 2013.

[6] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Satoshi Nakamoto Inst., Tech. Rep., 2008. [Online]. Available: https://nakamotoinstitute.org/bitcoin/

[7] K. Christidis and M. Devetsikiotis, "Blockchains and smart contracts for the Internet of Things," *IEEE Access*, vol. 4, pp. 2292–2303, 2016.

[8] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, pp. 1–32, Apr. 2014.

[9] H. Kim and E. A. Lee, "Authentication and authorization for the Internet of Things," *IT Prof.*, vol. 19, no. 5, pp. 27–33, 2017.

[10] A. Dorri, S. S. Kanhere, R. Jurdak, and P. Gauravaram, "LSB: A lightweight scalable blockchain for IoT security and anonymity," *J. Parallel Distrib. Comput.*, vol. 134, pp. 180–197, Dec. 2019.

[11] C. Fromknecht, D. Velicanu, and S. Yakoubov, "Certcoin: A namecoin based decentralized authentication system," Dept. Comput. Netw. Secur., Massachusetts Inst. Technol., Cambridge, MA, USA, Tech. Rep. 6.857, 2014, vol. 6. [Online]. Available: https://courses.csail.mit.edu/6.857/2014/files/19-fromknecht-velicann-yakoubov-certcoin.pdf

[12] M. Al-Bassam, "SCPKI: A smart contract-based PKI and identity system," in *Proc. ACM Workshop Blockchain, Cryptocurrencies Contracts*, Apr. 2017, pp. 35–40.

[13] L. Wu, X. Du, W. Wang, and B. Lin, "An out-of-band authentication scheme for Internet of Things using blockchain technology," in *Proc. Int. Conf. Comput., Netw. Commun. (ICNC)*, Mar. 2018, pp. 769–773.

[14] M. T. Hammi, B. Hammi, P. Bellot, and A. Serhrouchni, "Bubbles of trust: A decentralized blockchain-based authentication system for IoT," *Comput. Secur.*, vol. 78, pp. 126–142, Sep. 2018.

[15] K. Kataoka, S. Gangwar, and P. Podili, "Trust list: Internet-wide and distributed IoT traffic management using blockchain and SDN," in *Proc. IEEE 4th World Forum Internet Things (WF-IoT)*, Feb. 2018, pp. 296–301.

[16] A. Ouaddah, A. A. Elkalam, and A. A. Ouahman, "Towards a novel privacy-preserving access control model based on blockchain technology in IoT," in *Europe and MENA Cooperation Advances in Information and Communication Technologies*. Cham, Switzerland: Springer, 2017, pp. 523–533.

[17] O. Alphand, M. Amoretti, T. Claeys, S. Dall'Asta, A. Duda, G. Ferrari, F. Rousseau, B. Tourancheau, L. Veltri, and F. Zanichelli, "IoTChain: A blockchain security architecture for the Internet of Things," in *Proc. IEEE Wireless Commun. Netw. Conf. (WCNC)*, Apr. 2018, pp. 1–6.

[18] Z. Shelby, K. Hartke, and C. Bormann, *The Constrained Application Protocol (COAP)*, document RFC 7252, 2014.

[19] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "MQTT-S—A publish/subscribe protocol for wireless sensor networks," in *Proc. 3rd Int. Conf. Commun. Syst. Softw. Middleware Workshops (COMSWARE)*, 2008, pp. 791–798.

[20] A. Ometov, P. Masek, L. Malina, R. Florea, J. Hosek, S. Andreev, J. Hajny, J. Niutanen, and Y. Koucheryavy, "Feasibility characterization of cryptographic primitives for constrained (wearable) IoT devices," in *Proc. IEEE Int. Conf. Pervas. Comput. Commun. Workshops (PerCom Workshops)*, Mar. 2016, pp. 1–6.

[21] A. S. Wander, N. Gura, H. Eberle, V. Gupta, and S. C. Shantz, "Energy analysis of public-key cryptography for wireless sensor networks," in *Proc. 3rd IEEE Int. Conf. Pervas. Comput. Commun.*, 2005, pp. 324–328.

[22] N. Desai. (Jul. 2018). *Identifying the Internet of Things—One Device at a Time*. Accessed: Oct. 15, 2020. [Online]. Available: https://bit.ly/3nULqd8

[23] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of Things: A survey on enabling technologies, protocols, and applications," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 4, pp. 2347–2376, 2015.

[24] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams, *X. 509 Internet Public Key Infrastructure Online Certificate Status Protocol-OCSP*, document RFC 2560, 1999.

[25] I. Arghire. (May 2019). *Sectigo Revokes Certificates Used to Sign Malware Following Recent Report*. Accessed: Oct. 15, 2020. [Online]. Available: https://bit.ly/3k2yPCt

[26] S. Helme. (Jul. 2017). *Revocation is Broken*. Accessed: Oct. 15, 2020. [Online]. Available: https://bit.ly/3nUE0qn

[27] J. Kreku, V. Vallivaara, K. Halunen, and J. Suomalainen, "Evaluating the efficiency of blockchains in IoT with simulations," in *Proc. 2nd Int. Conf. Internet Things, Big Data Secur. (IoTBDS)*, 2017, pp. 216–223.

[28] Bitcoin Core. (2020). *Running a Full Node Bitcoin—Minimum Requirements*. Accessed: Oct. 14, 2020. [Online]. Available: https://bit.ly/2GZX3im

[29] P. P. Pereira, J. Eliasson, and J. Delsing, "An authentication and access control framework for CoAP-based Internet of Things," in *Proc. 40th Annu. Conf. IEEE Ind. Electron. Soc. (IECON)*, Oct. 2014, pp. 5293–5299.

[30] L. Cruz-Piris, D. Rivera, I. Marsa-Maestre, E. de la Hoz, and J. Velasco, "Access control mechanism for IoT environments based on modelling communication procedures as resources," *Sensors*, vol. 18, no. 3, p. 917, Mar. 2018.

[31] S. Hernan, S. Lambert, T. Ostwald, and A. Shostack, "Threat modeling-uncover security design flaws using the stride approach," *MSDN Magazine-Louisville*, vol. 2006, pp. 68–75, Nov. 2006.

[32] Concourse Open Community. (2020). *ETH Gas Station*. Accessed: Sep. 22, 2020. [Online]. Available: https://bit.ly/3kx8rjY

[33] Node-JS. (2020). *Crypto*. Accessed: Sep. 22, 2020. [Online]. Available: https://bit.ly/35VT9Bp

[34] Pubkey. (2020). *ETH-Crypto: Cryptographic Javascript-Functions for Ethereum and Tutorials on How to Use them Together With Web3js and Solidity*. Accessed: Sep. 22, 2020. [Online]. Available: https://bit.ly/3ckkfDm

[35] T. Holowaychuk. (2020). *Express: Fast, Unopinionated, Minimalist Web Framework for Node*. Accessed: Sep. 22, 2020. [Online]. Available: https://bit.ly/3hSIZ6L

[36] M. Collina. (2020). *Autocannon: A HTTP/1.1 Benchmarking Tool Written in Node, Greatly Inspired by Wrk and Wrk2, With Support for HTTP Pipelining and HTTPS*. Accessed: Sep. 22, 2020. [Online]. Available: https://bit.ly/3cn9X5f

[37] Dormando. (2020). *Memcached: A Distributed Memory Object Caching System*. Accessed: Sep. 23, 2020. [Online]. Available: https://bit.ly/3kHiVO0

[38] Node-JS. (2020). *Cluster*. Accessed: Sep. 23, 2020. [Online]. Available: https://bit.ly/32Sz1y0

[39] Truffle Suite. (2020). *Ganache: Your Personal Blockchain for Ethereum Development*. Accessed: Sep. 23, 2020. [Online]. Available: https://bit.ly/2RNz5ZA

[40] X. Li, P. Jiang, T. Chen, X. Luo, and Q. Wen, "A survey on the security of blockchain systems," *Future Gener. Comput. Syst.*, vol. 107, pp. 841–853, Jun. 2020.

[41] R. Amoros. (2020). *Transactions Speeds: How do Cryptocurrencies Stack Up to Visa or Paypal?* Accessed: Oct. 6, 2020. [Online]. Available: https://bit.ly/30Ttzd5

[42] M. Vukolić, "The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication," in *Proc. Int. Workshop Open Problems Netw. Secur.* Cham, Switzerland: Springer, 2015, pp. 112–125.

[43] Shodan. (2020). *The Search Engine for the Internet of Things*. Accessed: Oct. 6, 2020. [Online]. Available: https://bit.ly/2GsSb5e

[44] R. Dennis and G. Owen, "Rep on the block: A next generation reputation system based on the blockchain," in *Proc. 10th Int. Conf. Internet Technol. Secured Trans. (ICITST)*, Dec. 2015, pp. 131–138.

YUSTUS EKO OKTIAN received the bachelor's degree in electrical engineering from Petra Christian University, Indonesia, in 2013, and the master's degree in computer engineering from Dongseo University, South Korea, in 2016, where he is currently pursuing the Ph.D. degree. His research interests are on the topics of network security, distributed computing, blockchain, the Internet-of-Things, and software-defined networking.

SANG-GON LEE received the B.Eng., M.Eng., and Ph.D. degrees in electronics engineering from Kyungpook National University, Republic of Korea, in 1986, 1988, and 1993, respectively. He is currently a Professor with the Division of Computer Engineering, Dongseo University, Busan, Republic of Korea. He was a Visiting Scholar with QUT, Australia, from 2003 to 2004 and the University of Alabama at Huntsville, USA, from 2012 to 2013. His research areas include information security, network security, wireless mesh/sensor networks, and the future Internet.