Policychain: A Decentralized Authorization Service With Script-Driven Policy on Blockchain for Internet of Things

E Chen, Yan Zhu, Zhiyuan Zhou, Shou-Yu Lee, *Member, IEEE*, W. Eric Wong, *Senior Member, IEEE*, and William Cheng-Chung Chu, *Senior Member, IEEE*

***Abstract*—The decentralization mechanism provides manufacturers and distributors with greater customization and flexibility they need through Internet of Things (IoT)-based industrial collaboration systems (IoT-ICS), but it has brought forward security concerns about the shared data-processing tasks and IoT-based access to services and resources. To address them, we propose a practical blockchain solution to achieve decentralized policy management and evaluation on attribute-based access control (ABAC). By offloading the responsibility of ABAC policy administration and decision making to blockchain nodes, a blockchain-based access control framework, called Policychain, is presented to ensure policy with high availability, autonomy, and traceability. To deliver a solid design, we first present a transaction-oriented policy expression scheme with a well-defined syntax and semantics. The scheme can translate ABAC policies into the blockchain transactions with JavaScript object notation (JSON) syntax and script-based logical expression. We further realize a script-driven policy evaluation by extending blockchain inherent scripting instructions to support attribute acquisition of ABAC entities. Furthermore, we propose a policy lifecycle management scheme from policy creation, renovation, to revocation, in which policies are verified by three validation principles at the transaction level. Finally, we provide sophisticated analysis and experiments to show that our framework is secure and practical for decentralized policy management on ABAC in IoT-ICS.**

***Index Terms*—Attribute-based access control (ABAC), blockchain, Internet of Things (IoT), policy script, scripting language, transaction-based policy.**

I. Introduction

AS GLOBAL markets continue to expand, more and more companies are choosing a decentralized strategy to meet the demands for customized products and decreased costs in manufacturing industry. The advantages of decentralized manufacturing include flexibility, better and timelier information, more motivated managers and employees, and the ability to take advantage of low labor costs in different areas [1]. Some new emerging technologies, such as blockchain and Internet of Things (IoT), make it easier for companies to build Industry 4.0 or industrial collaboration system (ICS) during their decentralized manufacturing processes.

Blockchain is a decentralized ledger technology that stores an immutable record of all transactions in a cryptographic way. The combination of blockchain and IoT, called blockchain IoT, not only offers various potential benefits, e.g., higher trust, efficient data exchange, high-level security, and fault tolerance, but also eliminates the need for a central location to handle communication among IoT devices. According to MarketsandMarkets Strategic Insights,[1] the global blockchain IoT market size is projected to grow from USD 258 million in 2020 to USD 2 409 million by 2026, at a compound annual growth rate (CAGR) of 45.1% during the forecast period.

The blockchain IoT technique is influencing manufacturing from sourcing raw materials to delivering the final product. It was clear that blockchain can increase transparency and efficiency at every stage of the industrial value chain. However, when the resource is shared through a decentralized blockchain network, the security risks faced by the industrial system have not been reduced due to a lack of continual monitor and authentication. Such security risks include the intrusion from compromised devices, data leakages, and hacker's controlling devices remotely. Therefore, it needs to reduce the risks about the sharing of data-processing tasks and the unauthorized access to services and resources in IoT-based ICS (in short IoT-ICS).

To ensure the security of resources (things, services, and applications), access control is usually the first line of defense to regulate who or what can view or use the resources in the IoT-ICS. In view of the distinctive characteristics of IoT, e.g., dynamic positioning, heterogeneity, and widely geographical

Manuscript received April 27, 2021; revised June 11, 2021; accepted August 26, 2021. Date of publication August 31, 2021; date of current version March 24, 2022. This work was supported in part by the National Key Technologies Research and Development Programs of China under Grant 2018YFB1402702, and in part by the National Natural Science Foundation of China under Grant 61972032. *(Corresponding authors: Yan Zhu; William Cheng-Chung Chu.)*

E Chen, Yan Zhu, and Zhiyuan Zhou are with the Department of School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing 100083, China (e-mail: chene5546@163.com; zhuyan@ustb.edu.cn; zhouzhiyuan999@icloud.com).

Shou-Yu Lee and W. Eric Wong are with the Department of Computer Science, University of Texas at Dallas, Richardson, TX 75080 USA (e-mail: sxl128630@utdallas.edu; ewong@utdallas.edu).

William Cheng-Chung Chu is with the Department of Computer Science, Tunghai University, Taichung 40704, Taiwan (e-mail: cchu@thu.edu.tw).

Digital Object Identifier 10.1109/JIOT.2021.3109147

[1]https://www.marketsandmarkets.com/Market-Reports/blockch-ain-iot-market-168941858.html

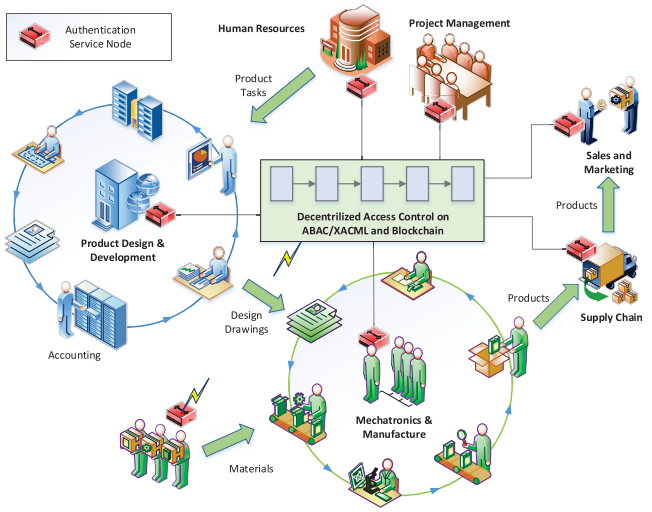


Fig. 1. IoT-ICS system with blockchain.

distribution, attributed-based access control (ABAC) [2] is the most appropriate access control model for protecting the shared resources in IoT-ICS with the outstanding features of high flexibility, scalability, manageability, and powerful expression ability. As a practical standard of ABAC for industrial applications, extensible access control markup language (XACML) has received increasing attention in recent years. Using a rich expression to log various policies and rules, XACML allows the resource owner (vendor or enterprise) to impose access restrictions upon a specified workflow of evaluation requests against policies. Meanwhile, XACML can help requester to enforce the authorized operations and strictly limit the behaviors on system resources [3].

More importantly, the ABAC/XACML model can be naturally integrated into the blockchain IoT system. As an example shown in Fig. 1, we describe an IoT-ICS with geographically distributed devices and services, including human & project management (H&PM), product design & development (PD&D), mechatronics & manufacture (M&M), supply chain, and marketing [4], as follows. First, project managers schedule and publish the product tasks to PD&D. Second, PD&D organizes product analysis and drawing design according to business process, and then conducts planning review and process proposal, and finally sends the design drawings to M&M, At last, M&M purchases raw materials from suppliers, makes molds, tests prototypes, produces and tests products, and then delivers the products via the supply chain to sales and market, and finally to users. To prevent unauthorized access, each of IoT devices in the system embeds a blockchain node, called the authorization service node (ASN), to manage the shared resources (e.g., design drawings, production tasks, product specifications, and sales information) [5].

Due to the large volume of protected resources, we advice that the access policy of resources, rather than the resources themselves, is stored in the blockchain. By this way, all authorized IoT devices can be allowed to access *off-chain* resources through access authorization on the *on-chain* policy. Therefore, through the ASN, the devices can share the resources by submitting the ABAC-type policy of owned resources into blockchain, or gain access authorization to protected resources by making consensus decisions on the submitted policy. Here, the consensus decision means all nodes make policy decisions with majority voting through the blockchain consensus mechanism.

However, the ABAC/XACML must face the decentralization problem of access policies when it is used in blockchain IoT-ICS. As ABAC is also known as policy-based access control, the architecture of ABAC/XACML is mainly designed and widely validated for an industrial environment having a set of policies, which are generally managed by a single trusted node. This node, as the center of the system, is called policy administration point (PAP) in ABAC architecture [6]. However, this architecture can not be easily applied to more decentralized IoT-ICS environments, where multiple nodes jointly manage the policies and make policy decisions together in a collaborative way.

Aiming at the actual demands across the lifecycle of policies (including generation, application, renovation, revocation, etc.), the policy decentralization of ABAC/XACML should highlight the following two challenges.

1) *Decentralized Policy Management:* Access policies delegated to the decentralized environment should be stored, distributed, and maintained by all nodes, and these policies should also be credible and trustworthy to the corresponding generators and consumers.
2) *Decentralized Policy Execution:* Access policies could be retrieved and executed by any node in the decentralized environment if needed, and the final decision is made by using a consensus process with a majority of nodes rather than an individual or a minority.

Although policy decentralization inevitably brings about a series of problems, the decentralized policy implementation also enjoys high availability and better autonomy for policy management and execution in the industrial processes. More specifically, the individual nodes have complete control of their own resources by manipulating policies. As a result, the administrative right of policies is vested in all nodes rather than a central node. In view of these advantages, it is necessary to explore a practical solution for the decentralized policy implementation of ABAC/XACML to develop global manufacturing enterprises.

II. Motivation and Approach

Presently, the blockchain technique may be the best choice to construct a decentralized IoT-ICS. Specifically, the blockchain is considered as a sequential, tamper proof, and decentralized ledger that stores records of transactions. These transactions are validated by all nodes, then reach the agreement through consensus protocol, and finally are appended to blockchain. Furthermore, the integration of existing techniques, including peer-to-peer networks, cryptographic algorithms, scripting language, and consensus protocols, makes blockchain in industrial settings of IoT-ICS with some salient features, such as immutability, transparency, fault-tolerance, traceability, low cost, and wide availability.

As shown in Fig. 2, the blockchain is a sequential chain for storing a back-linked list of blocks, each of which is composed

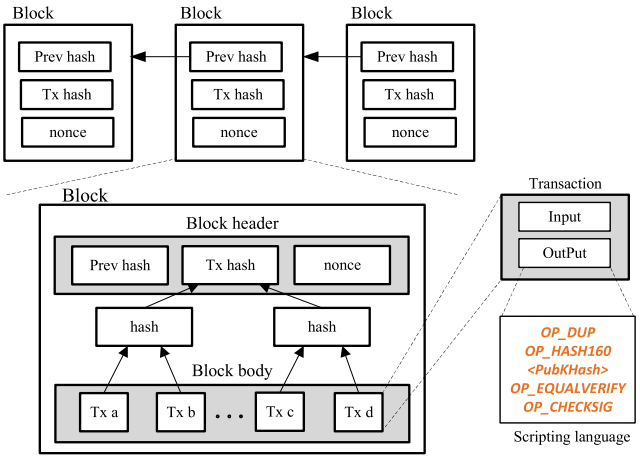


Fig. 2. Block structure of blockchain.

of a header and a body with batches of valid transactions. In the block header, the hash of the previous block (Prev hash) is recorded for linking the current block to the prior one, which confirms the integrity of the previous block. In the block body, all transactions are hashed into a Merkle tree (binary hash tree), and its root (Tx hash) is recorded in the header. Besides, the transaction may contain multiple inputs and outputs so that the new transaction's input can reference a previous transaction's output as its source. The validity of this reference can be verified by using a stack-based script settled in the input and output.

Aiming at these features, the blockchain designed for the IoT-ICS can provide a new solution for access policy management and decision making by integrating it into the ABAC architecture. The solution's advantages are listed as follows.

1) *Policy Unforgeability:* By using a cryptographic signature and Merkel hash tree, the policies stored in blockchain will be unforgeable and nontamperable.
2) *Policy Availability:* The decentralized storage allows nodes to independently obtain any policy stored in the blockchain, which improves the fault tolerance and wide availability of the access control system.
3) *Policy Consistency:* Policy decision results need to be agreed upon by consensus protocol throughout the whole network, which greatly enhances the validity of policy decision and reduces the impact of misjudgment.
4) *Policy Traceability:* Ledger-formatted structure provides traceability for access control policies recorded in transactions, such that auditors can retrieve the modification history of a specific policy by backtracking all related transactions.

Thus, the blockchain technology is a reasonable choice to carry out our research on decentralized policy management and execution. Moreover, as the most typical and earliest blockchain, bitcoin system has been running stably for more than ten years. Therefore, we adopt bitcoin architecture to design and implement decentralized IoT-ICS.

A. Related Works

The XACML was presented by Anderson *et al.* [11] as a language to define a schema for access control policies, access requests, and the associated response within a determined environment in an organization. However, the original XACML architecture [6] with centralized policy management cannot be so easily applied to more distributed or decentralized environments. For purpose of solving this problem, Listchka *et al.* [12] proposed a framework for the execution of XACML policies in a distributed way. This framework is applied in a context where an authorization decision in one local domain depends on the decisions of other remote domains. Then, Diaz-Lopez *et al.* [13] proposed an effective solution for distributed policy management where the master PAP assigns the management operations to the slave PAPs in collaborative environments composed of multiple security domains. The similar works include: the OHRM system [14] for XACML obligations handling, the extension of COPS protocol [15] for distributed policy transport between policy enforcement point (PEP) and policy decision point (PDP), and the JACPol policy language [16] for converting existing extensible markup language (XML) policies into JavaScript object notation (JSON) format (not specific to blockchain). These existing distributed access control schemes are insufficient in fairness and collaboration in comparison with decentralized systems, but provide meaningful references for us to implement the decentralization policy of XACML.

Along with the popularity of blockchain, several new types of researches have been proposed to solve the above problem by integrating the access control model and blockchain in recent years. For example, Maesa *et al.* [7] described a prototype of a blockchain-based access control system for XACML policies management in 2017. The system was implemented by using bitcoin transactions to store the original XML-based XACML policies, so that it could be considered as a decentralized policy database. The pity is that their work did not refer to decentralized policy execution and evaluation. In 2019, Maesa *et al.* [17] furthered proposed a scheme to codify XACML policy as a smart contract and deploy it on a blockchain, the scheme was designed on off-chain plus on-chain architecture and presented some conceptual descriptions. However, as a program stored and run in bytecode, the proposed policy is poor in readability, such that it is unfriendly to policy managers. Zhu *et al.* [18] also presented a transaction-based access control (TBAC) platform for digital asset management based on blockchain. This platform built four types of transactions as a bridge between ABAC and blockchain to describe the access control procedure, including subject registration, object escrow, access requirement, and authorization, for resource distribution and sharing. The decentralized approach in [18] provides a meaningful reference for the storage and usage of attributes.

We here compare our framework with previous schemes [7]–[10], and summarize the comparison results in Table I. In [8], the Dike multitenant access control is introduced, but it is just a distributed file management and does not belong to the ABAC framework. The paper [9] implements a decentralized policy evaluation and dataflow tracking system based on a data usage control model. Moreover, this system translates OSL policies into the Event-Condition-Action rule and makes policy decision locally. The paper [10] proposes a

TABLE I
COMPARISON OF OUR WORK WITH [7]–[10]

Frameworks	Decentralized	Access Control Model	Policy Expression	Policy Evaluation	
				Decision-Making	Consensus Protocol
[7]	√	ABAC	XML	×	×
[8]	×	Multitenant AC	×	×	×
[9]	√	Data Usage Control	Event-Condition-Action Rule	Distribution Management Point	×
[10]	×	Attribute-based Encryption	Policy Trees	×	×
Our Work	√	ABAC	JSON + Script	Policy Scripting Interpretor	√

collaborative access control scheme based on attribute-based encryption, but it still has centralized authorization nodes to achieve key management. Compared with these four frameworks in Table I, our framework has the following advantages.

1) ABAC model is more fine-grained and flexible.
2) The "JSON + Script" format used for policy expression ignores the process of constructing policy tree, and is more compact and simple in syntax.
3) In contrast with other execution and verification methods, the combination of policy scripting interpreters and consensus protocol can improve fault tolerance, be faster and easier to decide and edit the policies.

B. Technology Roadmap

We focus on a practically decentralized solution for policy management and evaluation of ABAC/XACML to achieve IoT-ICS in industrial settings. Compared with the traditional centralization methods, this solution should make remarkable improvements in reliability, availability, and security. However, the following technical challenges still need to be addressed.

1) How to concisely represent the ABAC/XACML policies in blockchain? To solve this challenge, we should consider the following aspects: a) a well-defined syntax and semantics should be specified to translate XACML policies into the JSON-formatted transactions in blockchain, to strip away the redundant meta data in the XML-formatted XACML policies and b) the hierarchical structure (including condition, rule, and policy) in ABAC/XACML policies should be organized into JSON syntax rules embodied in transactions, and their semantics are identical.
2) How to implement efficient decentralized policy evaluation? To solve it, we try to take advantage of the blockchain's powerful and efficient scripting system, then we consider the following aspects: a) the logical expressions of ABAC policies need to be converted into the scripting instructions for policy evaluation and b) the blockchain's inherent scripting interpreter should be improved to support attribute acquisition [from policy information points (PIPs)] and hierarchical JSON syntax evaluation (based on condition, rule, and policy).
3) How to implement policy lifecycle management? In order to manage policies at the transaction level, we consider the following aspects: a) since the blockchain are immutable, a policy maintenance mechanism is needed to satisfy the requirements of creating, updating, and revoking during policy lifecycle and b) each of the nodes in the decentralized system should be autonomous but the behavior of policy creation, renovation, and revocation must be restricted in an explicit and trustworthy way.

C. Our Contributions

Motivated by the three aforementioned challenges in decentralized IoT-ICS, we present a blockchain-based access control framework, called Policychain, for decentralized policy storage, evaluation, and lifecycle management. In this framework, a concise, lightweight, and interpretively executable policy model is integrated into blockchain by using a stack-based script, JSON-formatted transaction, and formalized ABAC/XACML language. We call this model "JSON + Script" format. Our main contributions are summarized as follows.

1) Present a *transaction-oriented policy expression* scheme, where the responsibilities of PAP and PDP in ABAC are offloaded to each blockchain node as a foundation of decentralized policy management and evaluation services. Moreover, a well-defined set of semantics over *attribute, expression, rule, target,* and *policy* is proposed to translate XACML policies into blockchain transactions with hierarchical JSON syntax and script-based logical expression.
2) Realize a *script-driven policy evaluation* scheme by extending Bitcoin's scripting instructions to support attribute acquisition of the *subject, object, action,* and *environment* entities in the script execution process. Based on this, a new policy scripting interpreter is constructed on three evaluation algorithms (*evalScript, evalRule, and evalPolicy*) to parse the hierarchical JSON syntax.
3) Propose a *policy lifecycle management* scheme based on policy issuing transactions (PITs), in which bidirectional checking between protected resource and issued policy is enforced by complying with three validation principles of policy verification. Meanwhile, the traceability of three policy lifecycle stages, from *policy creation, renovation,* to *revocation*, could be guaranteed by designing an uninterrupted, verifiable transaction stream through continual references between transactions' inputs and outputs.

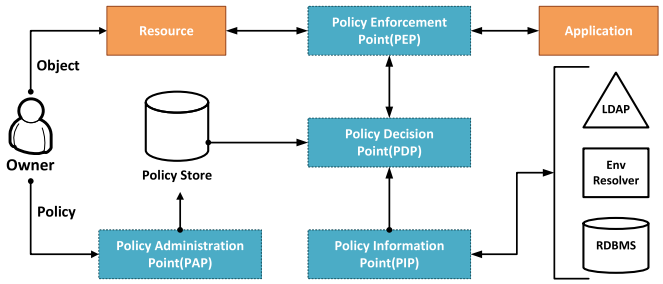


Fig. 3. ABAC reference architecture of our design for IoT-ICS.

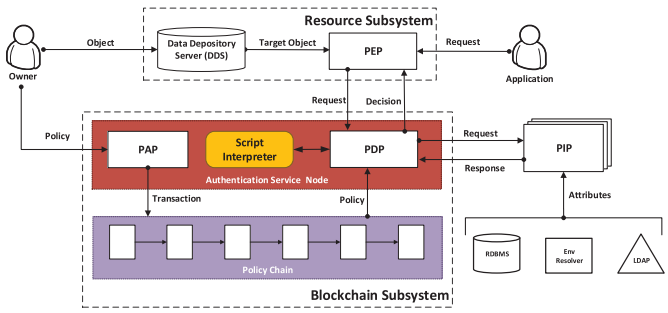


Fig. 4. System model with integration of blockchain and ABAC.

We also analyze the security of our framework at three aspects of policy storage, evaluation, and management, and then evaluate the performance of a Policychain prototype system.

Organization: The remainder of the paper is organized as follows. We describe our Policychain framework in Section III and then illustrate how to extend script's instructions and improve script interpret for policy evaluation in Section IV. The policy representation and the corresponding management scheme are presented in Sections V and VI, respectively. We provide security and performance evaluation in Section VII. This article concludes in Section VIII.

III. System Framework

We address the problem of protecting the shared resources in decentralized IoT-ICS. Our approach is to integrate blockchain with the ABAC model [19] for applying and managing policies as a reliable authentication service in industrial environments. We design a new transaction form to represent and store ABAC-type policy in the blockchain, and further provide a new perspective to make policy decisions on the blockchain's scripting interpreter. In this way, all authorized IoT devices will be allowed to access *off-chain* resources through access authorization on the *on-chain* policy. Moreover, to offload the responsibility of ABAC policy administration and decision making to blockchain nodes, we allow each node to play two roles, both being as a policy administrator and as a performer.

A. ABAC/XACML Model

To date, the most prevalent ABAC standard is OASIS's XACML [19]. It defines a declarative fine-grained, XML-based policy language, and a request/response processing model, which describes the way of evaluating access requests according to the rules defined by the policy language. In an ABAC model for industrial communication of IoT-ICS, there are four entities, *subject, object, action,* and *environment*. The characteristics of these entities are defined as *attributes*. Based on attributes, access policy extracted from common rules can be used to determine whether a *subject* should be allowed to perform expected operations (*actions*) on *objects* under a specific *environment*.

Fig. 3 shows an ABAC's reference architecture for the IoT environments. The architecture includes four main service nodes.

1) *Policy enforcement point* is a module that receives requests from applications and performs access control by making decisions on requests and enforcing authorization decisions.
2) *Policy decision point* is a module that evaluates applicable policy and renders an authorization decision.
3) *Policy administration point* is a system module that creates and manages access policies, which are specified by the resources' owners, in a policy repository.
4) PIP consists of many distributed modules that act as a source of attribute values. It is shown as one logical repository but may comprise multiple physical repositories.

In this model, PDP and PEP can be distributed or centralized, but PAP is usually centralized to simplify policy management. For access requests, the workflow of this architecture is described as follows.

1) The PAP stores the resource owner's policies and makes them available to the PDP via a policy repository.
2) The PEP intercepts the access request from an authenticated subject and sends the request to the PDP.
3) The PDP makes access decision according to access policy generated by PAP and the attributes of *subject, object,* and *environment* obtained by querying the PIP.
4) The final decision result given by the PDP is sent to the PEP, and then the PEP fulfills the access request according to the decision of PDP (either permit or deny).

B. Our Policychain Framework

To apply the ABAC/XACML model, we present a novel blockchain-based ABAC framework, called *Policychain*, for enhancing availability, unforgeability, consistency, and traceability of policy management and execution under the decentralized IoT-based environment. The framework consists of two components: 1) resource subsystem (RS) and 2) blockchain subsystem (BS), as shown in Fig. 4.

The RS component is responsible to store and manipulate the shared resources in public storage environments (e.g., cloud server). Specifically, it contains two modules: 1) data depository server (DDS) and 2) PEP. The DDS must be trusted to support storing the shared objects hosted by their owners, and providing access service for these objects. Furthermore, the virtualization technique could be used to support the dynamic deployment of the PEP servers.

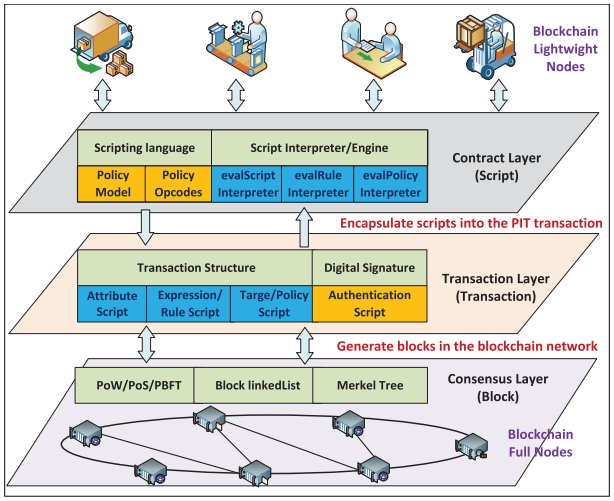


Fig. 5. Blockchain architecture for Policychain framework.

The BS component adopts the blockchain to manage all policies in the whole system and provides an authentication service based on policy evaluation. This component is constructed by a complete blockchain network, where many ASNs work together to maintain an access policy ledger called a policy chain. The policy chain and ASN are described as follows.

1) *Policy Chain:* It refers to a decentralized, always available, irreversible, tamper-resistant and replicated growing list of access policies, and each policy is considered as a transaction on the blockchain.
2) *Authentication Service Nodes:* All of them make up the blockchain network, jointly manage the policy chain on blockchain, and provide a collaborative authentication service based on policy decision-making for an access request.

The ASN is essentially a billing node in the original blockchain network, but the functions of PDP and PAP are appended to conveniently store and retrieve the policies on the policy chain. In addition to PAP and PDP of XACML, the ASN adds a new module, called policy script interpreter (PSI). This module is an extended Bitcoin's script interpreter that validates the policy-oriented scripts in the above-mentioned transactions (see Section VI-A). In Fig. 4, to clearly focus on the main components of our model, we do not describe some details related to network and consensus protocol.

C. Our Policychain Architecture

The key component of the blockchain IoT platform is the embedded ASN in IoT devices. The ASN is a "plug and play" solution that allows users to send various transactions to the blockchain network. All nodes may be permitted to access the transactions in blockchain, so as to realize communication and sharing among IoT devices. Our Policychain framework can be constructed by using the existing blockchain techniques. In terms of software architecture, the Policychain adopts a hierarchical structure, as shown in Fig. 5. More specifically, it can be divided into the following three layers.

1) *Contract Layer:* supports the description and evaluation of predefined policies or rules in the form of executable scripts. The scripting language is designed to fulfill policy expressions in the Policychain by establishing a script-driven policy model (Section V-A) and supporting policy opcodes (Section IV-A). Moreover, three kinds of interpreters, including evalScript (Section IV-B), evalRule (Section V-B), and evalPolicy (Section V-C), are inserted into the script interpreter/engine to support policy decision-making.
2) *Transaction Layer:* supports encapsulation of the contents of the contract layer in the form of transactions, and establishes the reliable connection between transactions. The encapsulation syntax of attribute script, expression/rule script (Section V-B), and target/policy script (Section V-C) will be used to realize the scripting representation of access policies. Moreover, a reliable connection between policy-related transactions (Section VI-C) is built on cryptography-based authentication scripts (Sections VI-A and VI-B) in order to implement the Policychain management and security.
3) *Consensus Layer:* supports the sharing of transactions in blockchain in the form of blocks. The existing blockchain's consensus mechanism is used to package the valid transactions in a certain period of time into the block, and ensure the authenticity and validity of shared transactions.

The reason why the script is adopted by Policychain is that it is simpler and more efficient than smart contracts based on virtual machines and Docker for the IoT system. Moreover, the Policychain architecture is not limited to blockchain architecture. The reason is that the Policychain is designed above the transaction layer (involving transactions and scripts), rather than block structure (consensus mechanism), blockchain network, and other underlying techniques.

In order to increase the applicability of blockchain IoT devices, the consortium blockchain will be recommended as the best choice of our Policychain. As shown in Fig. 5, the blockchain nodes can be divided into two categories: 1) full nodes and 2) lightweight nodes. The full node is responsible for storing a complete list of all transactions and uses consensus algorithms to build valid transactions into blocks. The lightweight node is not responsible for a consensus mechanism (such as mining) that consumes more resources but instead needs to connect to a full node in order to synchronize to the current state of the network and be able to participate. Therefore, the data encapsulated in the form of transactions can be submitted or the data authorized to access can be obtained.

D. Workflow of Policy Decision-Making

The workflow of our Policychain framework can support a more flexible access control for decentralized resources. Before proceeding to the policy decision making, we describe the process of resource submission and policy publishing as follows.

1) *Submitting Resource:* While intending to share a resource, the owner sends the object and its attribute information to a DDS and further, he/she can specify the corresponding access policy for the object and submit it to the ASN's PAP.
2) *Issuing Policy:* After receiving the access policy submitted by the owner, the PAP converts it into a transaction (called PIT) and stores it into the policy chain by using consensus protocol with the *access policy validation* (see Section VI-B).

We here explain how the ASN responds to the application's access request on the object stored in the DDS. At the beginning, the access requester sends a request to the PEP, and then the PEP transfers the request to a certain ASN's PDP. For such an access request, our framework works as follows.

1) *Retrieving Policy:* Upon receiving the request, the PDP queries the policy chain about the applicable access policy. The policy chain returns the policy[2] whose target matches the request (see Section V-C).
2) *Acquiring Attributes:* once receiving the returned policy, the PDP queries PIPs (via LDAP, environment resolver, RDBMS) for the verifiable attribute values specified by the attribute scripts (see Section V-B) in the policy. We note that the process is quite similar in the XACML.
3) *Making Decision:* After validating the attribute values, the PDP invokes the PSI to execute the script in the policy (see Section V-C). For the script's execution result (either permit or deny), the PDP calls consensus protocol to verify its correctness by collaborating with other nodes.
4) *Enforcing Request:* After the whole network reaches a consensus on the result, the PEP fulfills the requested action to access the resources in the DDS if permitted.

The above consensus protocol is a multiparty process used to achieve agreement on the result of a single computation or decision problem among the distributed and decentralized networks in the domain of IoT-ICS. In the existing blockchain consensus protocol, this process is divided into two steps: 1) each of the nodes verifies the correctness of the transaction (as an encapsulation of the above problem) and broadcasts its own determination and 2) the whole network agrees with the decisions of the majority as the final result of transaction verification. There exist various consensus protocols for different scenarios, such as Proof of Work (PoW), Proof of Stack (PoS), and practical Byzantine fault tolerance (PBFT) [20]. We here do not specify which one is used in our framework. Actually, our framework can support all the above consensus protocols, and we recommend readers to choose one matching their actual needs for the implementation.

Generally speaking, our framework is similar to XACML but provides three advanced features: 1) we transfer the policy storage location from the centralized database to the blockchain; 2) both PDP and PAP are integrated into the blockchain node so that all of the nodes jointly support policy management and evaluation; and 3) we employ the blockchain's script interpreter to perform the policy decision-making in a more efficient manner.

[2]In this article, we will only consider the case of single policy matching. We might deal with multipolicy matching by using the policy-combining method which is similar to the rule-combining method proposed in Section V-A.

E. Technical Challenges

Before the detailed description of our design for IoT-ICS, we identify the technical challenges in implementing the Policychain framework. Some existing techniques provide solid building blocks for our work: 1) the inherent mechanisms of blockchain, e.g., flexible transaction structure, consensus mechanism, hash/signature verification, can provide a secure, trustworthy, efficient support for access policy storage, sharing, and maintenance; 2) the XACML defines the specification of request/response between PEP and PDP, as well as that between PDP and PIP, to help us simplify our design; and 3) we use some elegant technologies, such as the operation on visitors and resources in the cloud needs to be authenticated by signature, to ensure that the cloud is able to provide a secure platform to perform object storage and decision enforcement.

Due to the help of the above techniques, we can only focus on the problem of how to integrate PAP and PDP with blockchain nodes, which reflects the part of the red box in Fig. 4. We state that solving this problem can yield the benefit that the interface of the Policychain node can be invoked to implement effective services of policy management and execution. We can unfold the above problem into two aspects.

1) *Transaction-Oriented Policy Expression:* To avoid the cumbersome format of XML-based policy expression in XACML, we will design a new JSON-formatted transaction for a more lightweight policy encapsulation. In addition, Bitcoin's script language will be used to further simplify policy expression.
2) *Script-Driven Policy Evaluation:* To reduce the computational overhead of XML-based policy parsing in XACML, we will develop a new PSI for a faster evaluation of access policy defined by three hierarchical elements (including policy, rule, and condition).

In addition to the above problems, we still need to address two other problems. One is how to manage policies in the view of their lifecycle from creation through use, renewal, and finally to revocation; and the other is how to ensure the security of policy storage, decision making, and management.

IV. Script Improvement for Policy Evaluation

Our goal is to enable access policies to be stored and executed in the blockchain of an IoT-ICS. To achieve this goal, we will do some preparatory work by improving the blockchain scripting system to support the expression and execution of access policies. We will review the typical scripting language in blockchain, then illustrate our improvements on new attribute-oriented opcodes that are applied to our framework. We will also explain the script evaluation and its workflow in detail.

A. Policy Script Language

The scripting mechanism [21] is the inherent capability of current blockchain systems (e.g., Bitcoin, Multichain, and

TABLE II
OPCODE EXAMPLES FOR ACCESS CONTROL

Word	Opcode	Input	Output	Description
OP_EQUAL	0x87	a, b	True/ False	Returns 1 if the inputs are exactly equal; otherwise 0.
OP_NUMEQUAL	0x9D	a, b	True/ False	Returns 1 if the numbers are equal; otherwise 0.
OP_BOOLAND	0x9A	a, b	True/ False	If both a and b are not " " (null string), the output is 1; otherwise 0.
OP_BOOLOR	0x9B	a, b	True/ False	If a or b is not " " (null string), the output is 1; otherwise 0.
OP_LESSTHAN	0x9F	a, b	True/ False	Returns 1 if a is less than b; otherwise 0.
OP_GREATER THAN	0xA0	a, b	True/ False	Returns 1 if a is greater than b; otherwise 0.
OP_SUBATTR	0x83	entity. attr	value	Returns the attribute value responding to the requested "attr" from subject entity.
OP_OBJATTR	0x84	entity. attr	value	Returns the attribute value responding to the requested "attr" from object entity.

Litecoin) to validate transactions. Similar to the FORTH language, the script is a stack-based language based on reverse polish notation (RPN). RPN is a method to represent expressions in which the operator symbol (called opcode) is placed after the arguments being operated on. And, it is a string of lists of instructions recorded with each transaction. The core of scripts is a set of opcodes which starts with "OP_" and is concatenated with a specific operation name, e.g., OP_EQUAL, OP_DROP, and OP_SHA256. Table II shows the opcodes associated with logical and numerical comparisons in the scripting interpreter. For example, the script "$\langle a \rangle$ $\langle b \rangle$ OP_EQUAL" is used to verify if a is equal to b, where the string between $\langle\rangle$ denotes the operand.

The blockchain's scripting mechanism has a relatively complete instruction set, but it is not specifically designed for the access control process. Therefore, we need to introduce new instructions, called attribute acquisitions, to support the policy script. We define four new opcodes, including OP_SUBATTR, OP_OBJATTR, OP_ACTATTR, and OP_ENVATTR, to get the attribute values of subjects, objects, actions, and environment conditions for a specified attribute, respectively. Each of opcodes takes an attribute name as input and outputs a corresponding attribute value, for example, "$\langle age \rangle$ OP_SUBATTR" will return the subject's age as a result. In Table II, we show the details for these instructions.

B. Policychain's Script Interpreter Implementation

The PSI is used to directly execute instructions written in a policy script language. For the above-mentioned policy script, the process of script execution is described as *evalScript*() in Algorithm 1. According to the type of the element pointed by the top-of-stack pointer (called script pointer), the execution of the script has two different cases.

1) When the script pointer points to an OPERAND, it is pushed onto the top of the stack;
2) When the script pointer points to an OPCODE, the operation defined by the opcode is executed by the scripting interpreter and one or more top-level elements are popped up as the input of the operator if needed; then, the output of the operator is pushed onto the stack.

Algorithm 1 evalScript(*scripts*)

```
Input: Script[] scripts;
Output: Boolean result;
Stack s ← Initialize Stack;
for all item in scripts do
    if item is instance of OPERAND then
        s.push(item);
    else if item is instance of OPCODE then
        length ← get number of inputs of the OPCODE item;
        inputs[];
        for i = 0 to length − 1 do
            inputs.add(s.pop());
        end for
        output ← execute OPCODE item with inputs;
        if output != null then
            s.push(output);
        end if
    end if
end for
return s.Top;
```

In Algorithm 1, the script interpreter executes the script from left to right, and it only needs to look forward to a symbol (expressed by "item") to determine the parsing action at each step. This ensures the execution of the script is sequential and unambiguous. Therefore, the expression and execution of script can provide effective support for access control.

For a given policy script with attribute acquisition opcodes, we here take OP_SUBATTR as an example to illustrate the execution process of the script, as follows.

1) Check if the stack height is greater than 1, and throw an exception if it is not.
2) Pop up the top element of the stack as the attribute name and request the PIP to get the subject's attribute value from the context of the PIP or the access request.
3) If the returned value is not empty, push it onto the stack; otherwise, an exception is thrown.

Note that, the exception usually causes the termination of script execution and the invalidation of the script decision.

V. REPRESENTATION OF POLICIES IN POLICYCHAIN

One of our core techniques can translate access policies into the format of a blockchain transaction, which is called transaction-derived policies. This new format will help us easily share and propagate the policies among all nodes in the whole blockchain, as well as to help the PDP make a policy decision for access requests. We below focus on the expression and storage of ABAC/XACML policies in blockchain. We will present a formal model for access policies applied to our framework, and further propose a new expression syntax of policies to adhere to blockchain's transaction format.

A. Script-Driven Policy Model

We here propose a formal model to represent the access policy in the blockchain. We state that an XML-based syntax of policy expression in the XACML model is too cumbersome and redundant to be directly used into blockchain transactions [22]. Therefore, we need to define a more concise syntax

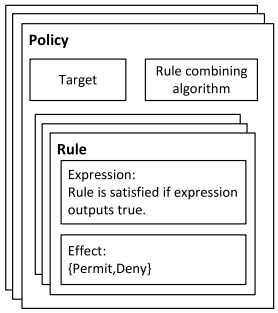


Fig. 6. Structure of the script-driven policy model.

that can be directly applied to blockchain. An entity is represented by a set of attributes. The symbols, S, O, A, and E, represent four types of entities (including subject, object, action, and environment) in the ABAC model, respectively. Denotes $X = S \cup A \cup O \cup E$ as a set of all entities.

A function $\mathrm{ATTR}(x) : X \to V$ is called attribute acquisition function on the attribute value set V, if for any entity $x \in X$, $\mathrm{ATTR}(x) = v$ means that x on the attribute ATTR has a specified value $v \in V$. For example, given a function Role(s) and the current requester "Alice," the PIP will retrieve and return doctor as the Role of subject $s = Alice$, i.e., Role(s) = doctor.

Let $\wedge$, $\vee$, and $\neg$ denote the typical boolean operators (AND, OR, and NOT) and $\triangleright$ denote a binary predicate (e.g., $<$, $\leq$, $=$). For the policy space *Expr*, a policy expression $e \in Expr$ is of the form

$$e \coloneqq \mathrm{ATTR}_1(x_1) \triangleright v | \mathrm{ATTR}_1(x_1) \triangleright \mathrm{ATTR}_2(x_2) | \\ e_1 \wedge e_2 | e_1 \vee e_2 | \neg e_1 | \phi \tag{1}$$

where $x_1, x_2 \in X$, $v \in V$ is a concrete value, e_1 and e_2 are other expressions, and ϕ is an empty expression. For instance, the expression $\mathrm{expr}_1 \coloneqq (\mathrm{Level}(s_1) < \mathrm{Level}(o_1)) \wedge (\mathrm{Role}(s_1) =$ "doctor") is to determine if the level of the subject s_1 is higher than that of the object o_1 and the role of s_1 is doctor.

Let R and P be rule set and policy set, respectively. Define $\mathcal{D} = \{Permit, Deny\}$ be a decision set. The rule and the policy are defined, respectively, as follows.

1) A rule $r \in R$ is a pair (expr, eff) for an expression expr $\in$ *Expr* and a specified effect eff $\in \mathcal{D}$;
2) A policy $p \in P$ is a triple (t, r_c, comb) for a target expression $t \in Expr$, an ordered rule set $r_c = \{r_1, \ldots, r_m\}$ and a specified combining algorithm comb, where $r_i \in R$ for all $i \in [1, m]$.

For convenience, r.expr indicates expr is in rule r. The effect eff (*Permit* or *Deny*) denotes the expected decision to allow or deny access to resource, i.e., the output of r is equal to eff if expr in r is true. For example, $\mathrm{rule}_1 \coloneqq (\mathrm{expr}_1, Deny)$ means that the output of rule_1 is *Deny* if expr_1 is true. Moreover, we abandon the "indeterminate" policy decisions defined in XACML 3.0, so that the relation $\neg Permit = Deny$ holds. This ensures that any exception or mis-matching expression in the decision process will directly lead to the policy decision failure.

As shown in Fig. 6, the access policy is composed of one target and many rules. In a policy $p \in P$, the target t can be considered as a simple rule. It is used to quickly match the access policy according to the access request, but not participate in the policy decision-making. To evaluate a policy, the rule-combining algorithm $\mathrm{comb} : \mathcal{D}^m \to \mathcal{D}$ is presented to combine many rule's decision results into one final policy's result. To this end, we define the functions *evalRule* and *evalPolicy* as follows.

Definition 1 (Rule Evaluation Function): The rule evaluation function $evalRule : R \times 2^X \to \mathcal{D}$ is used to evaluate any rule $r \in R$ for a given entity set $X_c \subseteq X$, i.e., $evalRule(r, X_c) \in \mathcal{D}$.

For example, $evalRule(\mathrm{rule}_1, \{Alice, \mathrm{MedicalRecord}\}) = Deny$ indicates that the output of rule_1 is *Deny* if the expression $(\mathrm{Level}(Alice) < \mathrm{Level}(\mathrm{MedicalRecord})) \wedge (\mathrm{Role}(Alice) =$ "doctor") in rule_1 is true (in this case, eff = *Deny*) for a subject *Alice* and an object MedicalRecord, as described above.

Definition 2 (Policy Evaluation Function): The policy evaluation function $evalPolicy : P \times 2^X \to \mathcal{D}$ takes an access policy p and an entity set $X_c \subseteq X$ as inputs and a decision $d \in \mathcal{D}$ as output, i.e., $evalPolicy(p, X_c) = d$. The decision-making process can be divided into two steps.

1) For each rule $r_i \in p.r_c$ in policy p, it executes the function $evalRule(r_i, X_c) = d_i$ for $i \in [1, m]$ to produce the result set $D = \{d_1, d_2, \ldots, d_m\}$ of $p.r_c$.
2) Using the rule-combining algorithm $p.\mathrm{comb}(D)$ defined in Policy p to combine all rule's evaluation results into a final policy's evaluation result.

We now illustrate three typical modes of the rule-combining algorithm used in our IoT-ICS, as follows.

1) *Permit Overrides:* It indicates that as long as there is a rule whose evaluation result is *Permit*, the decision of the policy is output as *Permit*, i.e., $evalPolicy(p, X_c) = Permit \leftrightarrow \exists r \in p.r_c \wedge evalRule(r, X_c) = Permit$.
2) *Deny Overrides:* It indicates that as long as there is a rule whose evaluation result is *Deny*, the decision of the policy is output as *Deny*, i.e., $evalPolicy(p, X_c) = Deny \leftrightarrow \exists r \in p.r_c \wedge evalRule(r, X_c) = Deny$.
3) *First Applicable:* It takes the decision result of the first rule in the policy as the decision result of the policy, i.e., $evalPolicy(p, X_c) = evalRule(First(p.r_c), X_c)$, where $First(r_c)$ returns the first rule in the rule set r_c.

B. Policy Components in Transaction

Most existing blockchains, such as Bitcoin and Ethereum, use JSON as the transaction description language. This is because JSON is a lightweight and ideal data-interchange format for machines generation and parsing. Therefore, we now describe an effective approach to convert the aforementioned policy model into a JSON-formatted policy representation. The core technology behind this conversion approach is to adopt the script of blockchain to explain the expression of policies, so that the policies can be executed efficiently by the blockchain's scripting interpreter.

We first introduce the syntax structure of JSON. A JSON message is usually composed of JSON objects and arrays defined as follows.

1) JSON object is a set of attribute-value pairs enclosed within curly braces, i.e., $\{name_1 : \text{value}_1, name_2 : \text{value}_2, \ldots\}$. For example, {"Role" : "doctor"}.
2) JSON array is an ordered list of values, whose type may be string, number, object, boolean, array and null, enclosed within square brackets, i.e., $[\text{value}_1, \text{value}_2, \ldots]$.

We now introduce the JSON-formatted script language to express each component in the policy, as follows.

Definition 3 (Attribute Script): An attribute script represents that the attribute acquisition function ATTR(x) obtains the attribute value of entity x. The format of attribute script is defined as "⟨attr⟩ OP_CODE," where OP_CODE is the opcode defined in Table II.

For example, the script "⟨*Age*⟩ OP_SUBATTR" is equivalent to the function $Age(s)$ for getting the age of subject.

Definition 4 (Expression Script): The JSON format of an expression script corresponding to (1) is defined as

$$\{\text{id} : \langle\text{ExprID}\rangle, \text{expr} : \langle\text{Script}\rangle\}$$

where id is used to identify the expression, expr is a script which is used to describe the expression.

Scripts in expression can be divided into three cases.

Case 1: The expression $\text{ATTR}_1(x_1) \rhd v$ can be translated into the JSON-formatted representation which contains ONE single attribute script and a comparison opcode, such as

{id : "expr1", expr : "⟨Role⟩ OP_SUBATTR ⟨doctor⟩ OP_EQUAL"}
{id : "expr2", expr : "⟨ID⟩ OP_OBJATTR ⟨MedicalRecord⟩ OP_EQUAL"}.

This indicates "expr1" is Role(s) = doctor and "expr2" is ID(o) = MedicalRecord, respectively.

Case 2: The expression $\text{ATTR}_1(x_1) \rhd \text{ATTR}_2(x_2)$ can be translated into the JSON-formatted representation which contains TWO attribute scripts and a comparison opcode, e.g.,

{id : "expr3", expr : "⟨Level⟩ OP_SUBATTR ⟨Level⟩ OP_OBJATTR OP_LESSTHAN"}.

This indicates "expr3" is Level(s) < Level(o).

Case 3: The expression $e_1|e_1 \wedge e_2|e_1 \vee e_2|\neg e_1|\phi(e_1, e_2 \in Expr)$ can be translated into the JSON-formatted representation which consists of boolean logic (expressed by the opcode in {OP_BOOLAND, OP_BOOLOR, OP_NOT}) over the expressions cited by ⟨ExprID⟩. For example

{id : "expr4", expr : "⟨expr1⟩⟨expr2⟩ OP_BOOLAND"}

indicates "expr4" is (Role(s) = doctor) $\wedge$ (ID(o) = MedicalRecord). In special cases, expr can be empty or only a reference to the condition on case 1 or 2.

Algorithm 2 evalRule(*rule*)

Input: Rule *rule*;
Output: Effect *result*;
1: Script[] *script*=[];
2: **for all** *item* **in** *rule.expr* **do**
3: **if** ! *item* is instance of OPCODE **then**
4: Expression *temp* = getExpressionById(*item*);
5: **if** *item*==null **then**
6: **return** "*Deny*";
7: **end if**
8: *script*.add(evalScript(*temp.expr*));
9: **else**
10: *script*.add(*item*);
11: **end if**
12: **end for**
13: **if** evalScript(*script*) **then**
14: **return** *rule.effect*;
15: **else**
16: **if** *r.effect* == "*Permit*" **then**
17: **return** "*Deny*";
18: **else**
19: **return** "*Permit*";
20: **end if**
21: **end if**

The expression abiding by case 1 or 2 is often called as "condition" to distinguish them from boolean logic.

Definition 5 (Rule Script): For a given rule expressed by the pair (expr, eff), its JSON-formatted representation is defined as

$$\{\text{id} : \langle RuleID\rangle,\ \text{effect} : \langle\text{eff}\rangle, \text{expr} : \langle\text{Expression Script}\rangle\}$$

where id is used to identify the rule, effect is the value of eff, and expr is the expression on case 3.

Here, we take a rule example including the aforementioned examples

{id : "rule1", effect : "Deny", expr : "⟨expr1⟩ ⟨expr2⟩ OP_BOOLAND ⟨expr3⟩ OP_BOOLOR"}

which declares that the rule "rule1" outputs "Deny" if the expression ((Role(s) = doctor) $\wedge$ (ID(o) = MedicalRecord)) $\vee$ (Level(s) < Level(o)) is true. Note that, an empty rule is always satisfied and the predefined effect is returned as output.

For a rule script, we describe its evaluation process *evalRule*() in Algorithm 2. This process can be accomplished by evaluating the expressions of rule in a nested manner, where the expression evaluation is done by invoking *evalScript*() as described in Algorithm 1. Moreover, getExpressionById(*item*) is used to return an expression object whose id property can match the specified string *item*.

In particular, there exist two cases in the evaluation process of rule: 1) the iterator *item* points to an opcode, where *item* is added to *script* and 2) *item* points to an expression id, where the expression is retrieved according to id [using getExpressionById(*item*)] and its evaluation result [using *evalScript*()] is added to *script*. Finally, the desired effect associated with the rule is returned only if the final result of script evaluation is true. Note that if the expression associated with id does not exist, the rule evaluation returns "Deny."

C. Target and Policy

We introduce the JSON-formatted representation of the target, and then comprise the aforementioned policy components into a complete policy. A target is basically a collection of simple expressions (refer to case 1 for a static value) of three "matching" entities: 1) *subject*; 2) *object*; and 3) *action*. In fact, the target specifies "matching regulations" to determine whether the policy is applied to an incoming request. Thus, the target is used to quickly find the policy applied to a given request.

Definition 6 (Script-Driven Target): For a given target composed of a set of expressions {ATTR(Entity) = attrvalue}, the JSON-formatted representation of this target is defined as

$$\left[\{\text{attr} : \langle \text{ATTR\#Entity} \rangle,\ \text{value} : \langle \text{attrvalue} \rangle\}, \ldots\right]$$

where attr indicates an attribute of entity with the attribute ATTR, "#" is a connector, the entity type Entity $\in$ {Sub, Obj, Act}, and value is a string of the corresponding attribute value.

While evaluating the request, the PDP is responsible to search policy whose target matches the request. The matching regulations used to evaluate the target are outlined as follows.

1) An empty target value can match any request, but the request must contain this target attribute.
2) The evaluation result will be "true" if the request can match all of the attr/value pairs with the different attr in the target, or "false" otherwise.
3) The evaluation for multiple attr/value pairs with the same attr is considered to be successful if the request can match at least one of them.

For example, a target which is defined as

$$\begin{aligned}[\{&\text{attr} : \text{"Role\#Sub"}, \text{value} : \text{"doctor"}\}\\ &\{\text{attr} : \text{"ID\#Act"}, \text{value} : \text{"read"}\}\\ &\{\text{attr} : \text{"ID\#Act"}, \text{value} : \text{"write"}\}]\end{aligned}$$

will apply to "either read or write access required by a doctor," that is, $\text{Role}(s) = \text{doctor} \wedge \text{ID}(a) \in \{\text{read},\ \text{write}\}$. The JSON-formatted policy is defined as:

Definition 7 (Script-Driven Policy): For a given policy corresponding to the triple (target, rule, comb), its JSON-formatted representation is defined as

$$\begin{aligned}\{&\text{id} : \langle \text{PolicyID} \rangle,\ \text{condition} : \left[\langle \text{Expression Script} \rangle, \ldots\right]\\ &\text{rule} : \left[\langle \text{RuleScript} \rangle, \ldots, \right], \text{target} : \langle \text{Target} \rangle\\ &\text{ruleCombiningMethod} : \langle \text{comb} \rangle\}\end{aligned}$$

where id is used to identify the policy, condition is the expression abiding by case 1 or 2, and ruleCombiningMethod specifies how to combine multiple rules into the final evaluation result.

The decision-making process of access policy can be divided into two steps: 1) the PDP finds out the policy whose target matches the access request and 2) then executes the rule scripts in policy and outputs their combination results. For a given query policy request, there are two typical approaches: 1) the PDP maintains an indexing table of policy's targets that allow queries to efficiently retrieve policies from Policychain and 2) the PDP loads all available policies and matches their target elements with the context of a particular request to identify one or more policies applied to the request. For a retrieved policy, *evalPolicy*() describes the evaluation process of a JSON-formatted policy in Algorithm 3. In this process, each rule in policy will be evaluated by invoking *evalRule*(). Further, three common rule-combining methods are used to evaluate the final output of all rules in policy. In addition, new rule-combining methods could be appended into *evalPolicy*() according to practical requirements on IoT-ICS.

Algorithm 3 evalPolicy(*policy*)

```
Input: Policy policy;
Output: Effect result;
switch policy.ruleCombiningMethod
    case Permit-overrides:
        for all rule in policy.rule do
            if evalRule(rule)=="Permit" then
                return "Permit";
            end if
        end for
        return "Deny";
    end case
    case Deny-overrides:
        for all rule in policy.rule do
            if evalRule(rule)=="Deny" then
                return "Deny";
            end if
        end for
        return "Permit";
    end case
    case First-applicable:
        return evalRule(First(policy.rule));
    end case
end switch
return "Deny";
```

VI. Policychain Management and Security Evaluation

On the basis of the JSON-formatted policy expression and script-driven policy execution, we develop an effective mechanism for access policy encapsulation and distribution in this section. This mechanism will conduce to policy lifecycle management from creation through use, renewal, and finally to revocation. Moreover, cryptography-based authentication script is used to enhance the Policychain management against policy tampering, forgery and misuse. Finally, we analyze the security of the Policychain framework designed for an IoT-ICS.

A. Policy Issuing Transaction

The PIT, as the fundamental recording unit in blockchain, is designed to record access policy in our framework. To have an intuitive and clear understanding, we will describe the PIT at two aspects: one is the structure of the PIT, and the other is the approach to manage policies at the transaction level.

As shown in Table III, the PIT consists of the encapsulation of aforementioned policy components and some items required for access control. First, all of the highlighted items, including target, condition, rule, and ruleCombiningMethod, are grouped together to represent the access policy stored in the PIT. Second, three additional items, *state, URL,* and *TxID*,

TABLE III
PIT DATA STRUCTURE

Filed		Type	Description
ver		Int	Transaction version.
URL		Char	The URL of resource related to this policy.
ruleCombiningMethod		Char	Combination method of rule evaluation results.
target[]	attr	Char	Attribute name and entity type.
	value	Char	attribute value.
condition[]	id	Char	Condition identifier.
	expr	Script	An expression in the form of script that represents attribute predicate.
rule[]	id	Char	Rule identifier.
	effect	Char	The intended consequence of a satisfied rule. (either "Permit" or "Deny")
	expr	Script	An expression in the form of script, representing the logical relationship among conditions.
state		Uint32	State of the policy.
vin[]	TxID	Uint256	The identifier (Hash) of the quoted transaction, if it is a initial transaction, TxID is all zeros.
	scriptSig	Script	Signature scripts provided by the user who owns the quoted transaction.
vout[]	scriptPubKey	Script	Pubkey script used to declare the recipient of the transaction.
TxID		Uint256	The identifier (Hash) of the transaction.

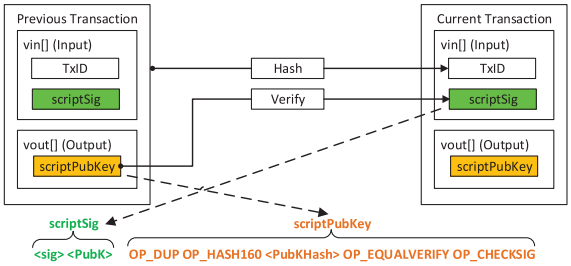


Fig. 7. Authentication procedure for scriptPubKey and scriptSig.

are designed to assist the policy execution, where *state* indicates that the policy is valid (1) or invalid (0), *URL* points to the resource managed by the policy, and *TxID* is the transaction hash for identifying the policy in the blockchain. Finally, two fields, *vin* and *vout*, have the same structure with those of Bitcoin transactions, but they have new meanings for policy management, where *vout* (as the output of transaction) is used to declare the policy's ownership, and *vin* (as the input of transaction) is used to verify the policy's ownership.

In the PIT structure, the cryptography-based authentication script, including *scriptPubKey* and *scriptSig*, is used to confirm the ownership of policy. This script is in essence a cryptographic signature, which describes how to verify the consumer of the current transaction as the asset holder declared in the previous transaction indexed by "TxID" (transaction hash).

As shown in Fig. 7, we illustrate the authentication procedure for *scriptPubKey* and *scriptSig*, where the *scriptPubKey* is placed on the output of the previous transaction to declare the resource's ownership, and the *scriptSig* is on the input of the current transaction to verify the resource's ownership.

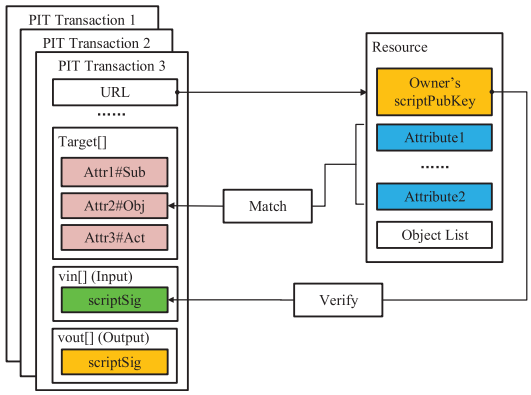


Fig. 8. Relationship between PITs and resources.

The *scriptSig* contains both consumer's signature and a public key, which are first pushed onto the stack. To verify these two items, the corresponding *scriptPubKey*, consisted of a hash of the consumer's public key and four opcodes, OP_DUP, OP_HASH160, OP_EQUALVERIFY, and OP_CHECHSIG, are sequentially pushed and executed onto the stack. Here, the first three opcodes make sure that the hash of ⟨PubK⟩ in *scriptSig* is equal to the ⟨PubKHash⟩, and then the signature ⟨sig⟩ is verified by the checked ⟨PubK⟩.

B. Access Policy Validation

In our framework, the access policy validation is an important mechanism to prevent incorrectly submitted policies from interfering with the other protected resources. In other words, it is used to avoid a policy-maker or an attacker submitting resource policies beyond their scope of management.

To implement this mechanism, each node must take strict approaches to verify the validity of PIT during the process of submitting PIT to the blockchain. According to the ownership between access policy and the corresponding protected resources, the *validation principles* are defined as follows.

1) *Policy's Excludability:* A policy can only be bound to one specified resource that could be a file, folder, or device.
2) *Target's Specificity:* The target in the PIT must be able to express the specified resource in the right way.
3) *Ownership's Verifiability:* The creator of the PIT must be the owner of the resource managed by the PIT's policy, i.e., the owner's signature in the PIT must be verified by the resource owner's public key.

Note that, the validation principles of access policies can be loosely or tightly restricted as needed.

In order to achieve the above principles, we introduce a new resource representation structure to describe the protected resource. This structure contains the necessary resource information, including the owner's public key, all objects, and attributes, for supporting the ABAC model. Fig. 8 demonstrates how to implement these three principles by establishing the relationship between PIT and resource representation structure (in short *resource*).

1) Abiding by Principle 1), we make use of the URL to point at the resource protected by the PIT's policy.

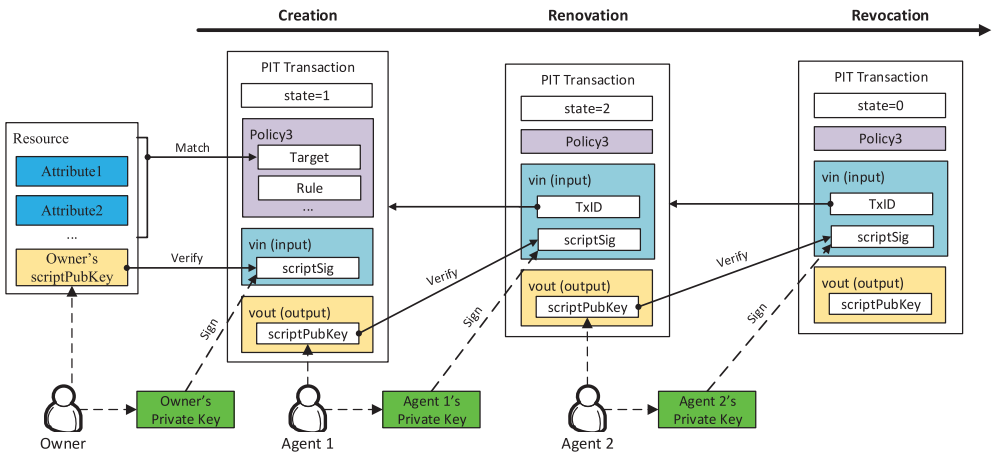


Fig. 9. Example of lifecycle of PIT from creation, through renovation to revocation.

2) According to Principle 2), the attributes of the resource must in turn match the attr/value pairs of the PIT's target in terms of the object's "matching regulations."
3) Principle 3) may be satisfied by using the owner's public key stored in the resource to verify the signature in the input-field *vin* of the PIT because only the owner's public key is able to validate the signature.

The third item is implemented by the authentication script in Fig. 7. The aforementioned relationship supports many-to-one mapping from policies to resources. For example, we can establish various access policies for different subjects and actions aiming at the same resource. Moreover, the access policy validation is one of the PAP's responsibilities, but it will be partly replaced by *consensus protocol* in our framework. That is, after receiving PIT, each node in the IoT-ICS verifies it according to three validation principles, and then the PIT is finally appended into the blockchain if the validation result can be consistent across all nodes through consensus protocol.

C. Access Policy Renovation and Revocation

The renovation and revocation mechanisms are the important parts of access policy maintenance and management after the policy is published into the blockchain. They are also very significant to maintain the sustainability of the system.

Due to the immutability of data stored on blockchains, it is not feasible to directly modify the access policies. This leads us to adopt a transaction-based update mechanism for policy modification. That is, new PIT transactions are submitted to replace their previous records in blockchain. The advantage of this update mechanism is that all the records will be preserved, so that we can achieve tracking and auditing for access policies. Additionally, strict authentication must be performed to ensure the security of the updating policy.

In Fig. 9, we show a simple example of the PIT's lifecycle, which includes three main stages: 1) policy creation; 2) renovation; and 3) revocation. To accomplish this, we keep the format of PIT transaction unchanged, but change the *state* value to indicate the stage of policy (create: 1, update: 2, and revoke: 0). Besides, we make use of the PIT's input-field *vin* and the output-field *vout* to establish strict authentication for context dependence between pre-and post-transactions. The three stages of policy are described as follows.

1) In the stage of policy creation, the PIT must enforce the ownership verification that checks whether the signature script (*scriptSig*) in *vin* matches with the owner's public-key script (*scriptPubKey*) in *resource* as described in Fig. 8. Moreover, the owner should reserve one public key of either himself or the agent in the *vout* field for future policy management.
2) Then, in the stage of policy renovation, either the owner or the designated agent is allowed to update the policy by submitting a new PIT transaction with his signature, which should be verified by the reserved public key in the previous PIT (see Fig. 7). Furthermore, the power of policy renovation could be transferred to another agent by setting his public key into the *vout* field, so that the agents may be able to continue this processing.
3) Finally, in the stage of policy revocation, all previous policies pointed by a new submitted PIT would be no longer used only if the verifiable PIT's author may set the "state" to 0 in PIT.

In the above policy management, we adopt a more rigorous authentication strategy for the resource's owner and agent that will significantly strengthen protection against policy forgery, falsifying or tampering from identity thieves. Moreover, whenever the policy is updated, all nodes in the whole blockchain need to verify the PIT's state and the submitter's signature.

D. Security Evaluation on Policychain

We now turn our attention to the security of policy storage, evaluation, and management on the Policychain platform. Without loss of generality, we assume that the protected resources, called off-chain resources, are stored in client-side servers or trusted cloud platforms, which are secure through some elegant technologies. And, these servers or platforms are trustworthy to fulfill the user-specified access requests complying with the on-chain policy decision-making [23]. Moreover, the PIP could act as a trusted source of attribute values. In addition, in our previous researches [24], [25], the Policychain platform is applicable to the cryptographic ABAC mechanism, which is called Crypto-ABAC.

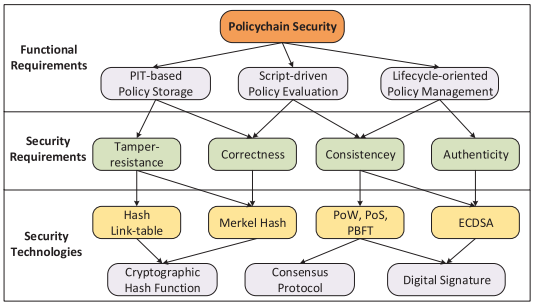


Fig. 10. Relationship among policy's functional requirements, security assurances, and security technologies.

We require that the security of our Policychain platform must meet three typical functional requirements: 1) policy storage; 2) policy evaluation; and 3) policy management. As shown in Fig. 10, these three functional requirements can be guaranteed from four basic security assurances, including tamper resistance, correctness, consistency, and authenticity. Here, correctness means that the behavior of policy storage and evaluation conforms to the expected results, and consistency indicates that all nodes joining the collaborative evaluation and management policies could make the same response to the same behaviors. Furthermore, these four security assurances can be guaranteed by blockchain-related security technologies [26]. We illustrate the details as follows.

1) *Security of Policy Storage:* The security requirement of policy storage is to ensure that the policy transaction stored in blockchain will not be tampered by the adversary. This requirement is satisfied by the transaction correctness and the storage reliability (fault tolerance) based on the above security technologies. Specifically, the blockchain adopts the hash functions (e.g., SHA-256 and RIPMD-180) to construct a Merkel hash tree of transactions and a singly hash link-table of blocks. This can also ensure the integrity of PIT transactions in the Policychain. Furthermore, the Crypto-ABAC scheme is able to generate a secure representation of policy, called cryptographic policy, against forgery, tampering, and replaying attacks.
2) *Security of Policy Evaluation:* The security requirement of policy decision-making is to ensure the consistency and correctness of the decision results. That is, consistency means the results of all honest nodes are the same (but it is uncertain whether they are correct), and correctness means the final decision result of the attacked platform is the same as that of the trusted third party without attack. In our platform, we utilize a cryptographically verifiable method of the Crypto-ABAC scheme to guarantee the validity of policy decision making, and introduce the consensus protocol into the policy decision-making process to provide fault tolerance for the final decision result. Since the consensus protocols, such as PoW, PoS or PBFT, provide some security properties [27], including agreement, termination, and validity, the correctness and consistency of policy evaluation can be guaranteed.
3) *Security of Policy Management:* The security requirement of policy management heavily focuses on the authenticity of the policy that is considered as the foundation of forgery resistance in the lifecycle from creation, renovation to revocation. To meet this requirement, we use Bitcoin's elliptic curve digital signature algorithm (ECDSA) based on the secp256k1 curve to sign the PIT transactions. This signature offers the feature of authentication and nonrepudiation for the owner or agent at the different PIT's lifecycle stages. For avoiding single node failure, we use the consensus protocol along with the signature's verification to ensure fault-tolerance in the process of appending PIT onto the chain. Therefore, the authenticity requirement can be guaranteed with two implementation techniques mentioned above.

In Fig. 10, we offer a brief description of the relationship among policy's functional requirements, security assurances, and security technologies. As shown, the security foundations of our platform are mainly constructed on three primitives.

1) *Cryptographic techniques,* which involve the Crypto-ABAC scheme with escrowed object encryption, secure decision-making of dynamic policies, real-time attribute tokens, and collision-resistant hash function (CRHF) [26].
2) *Cryptographic digital signature,* e.g., ECDSA and interactive incontestable signature (IIS) [28], which is existential unforgeable under chosen message attacks (EUF-CMAs).
3) *Consensus protocol,* which has the properties of fault- and attack-tolerance that are implemented with *N*-modular redundancy (NMR) and PBFT [20], [27].

In the above analyses, the attacker has the ability to completely control a single node and monitor the whole network, and we do not impose any security obligation or limitation on each node in the Policychain. However, we require that the majority (e.g., more than 2/3) of nodes are honest according to the security requirements of the consensus protocol. This means that the adversary can only control the minority (e.g., at most 1/3) of the nodes under collusion attack, and cannot manipulate other nodes' communication except the nodes he/she controls. In short, the above analyses show that our framework has the ability to improve the protection of policy storage, evaluation, and management by using blockchain and cryptographic techniques in an IoT-ICS setting.

VII. Performance Evaluation for Policychain

In this section, we evaluate the validity of the solution under the experimental environment with a dozen nodes. Moreover, we develop a prototype blockchain system built on the open-source Bitcoin Core. Based on this, we run various experiments to measure the performance on the Policychain framework.

A. ICS-IoT Experimental Environment

We design a small experimental environment of ICS based on IoT. As shown in Fig. 11, this ICS-IoT environment consists

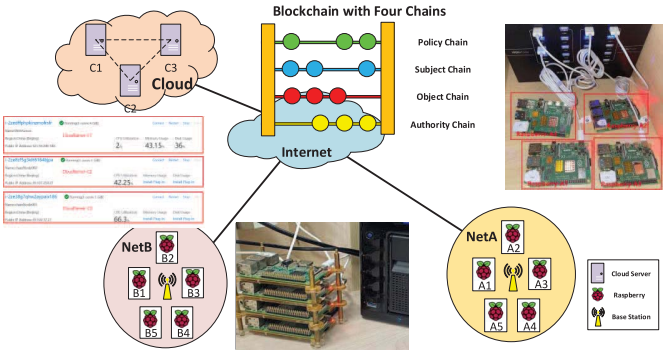


Fig. 11. Network topology of our Policychain system.

TABLE IV
RELATED PARAMETERS AMONG VARIOUS DEVICES IN OUR EXPERIMENTS

	NetA	NetB	Core Network
OS version	Ubuntu 18.04/Mate	Raspbian 10	Ubuntu 16.04/18.04
Hardware	BCM	BCM	Virtual Machine
Hardware Number	2837/2711	2837/2711	-
Processer	ARM Cortex	ARM Cortex	Intel Xeon Platinum
Model Name	A53/A72	A53/A72	8269CY
CPU Core	4	4	1
Architecture	ARMv7l/aarch	ARMv7l	X86_64
Frequency (GHz)	1.4/1.5	1.2/1.5	2.5
BogoMIPS	38.04/108.00/207.00	108.00/207.00	5000.00
Kernel (Linux)	4.15.0/5.4.0	4.19.75/5.10.17	4.4.0
RAM (MiB)	923/3791/3822	924/3906	942/3808
Shell Version (Bash)	5.0.3	5.0.3	4.3.48/4.3.20

of a core network and two sensor networks (called NetA and NetB) located in three different geographical areas, as follows.

1) *Core network* consists of three full nodes, C1-3, which are deployed in the cloud environment.
2) *Sensor network NetA* consists of five lightweight nodes, A1-5, under the Ubuntu systems.
3) *Sensor network NetB* consists of five lightweight nodes, B1-5, under the Raspbian systems.

In Fig. 11, we show the network topology of the blockchain system deployed on ten Raspberry Pi nodes in two different areas, i.e., NetA and NetB. By using the ZeroTier software, all of the Raspberry Pi nodes are connected and integrated into a unified virtual LAN to realize the interconnection between the nodes.

In the experimental environment, the blockchain system is deployed into 13 nodes. Among them, ten nodes in both the NetA and the NetB are running on the Raspberry Pi-based sensors. Exactly, we uses two kinds of Raspberry Pis: one is the Pi-3 module with 4x ARM Cortex-A53 CPU, BCM-2387 chipset, and 1.2 or 1.4 GHz, the other is the Pi-4 module with 4x ARM Cortex-A72 CPU, BCM-2711 chipset, and 1.5 GHz. In addition, three full nodes in the blockchain core network are built on cloud server nodes with single 2.5-GHz Intel Xeon Platinum 8269CY CPU. As shown in Table IV, we compare some parameters of these three categories, e.g., hardware type, processor, BogoMIPS, and architecture.

All nodes of the Policychain are built on Linux. The operating systems of two nodes (C1 and C2) in cloud are Ubuntu 16.04 and one node (C3) is Ubuntu 18.04. The Ubuntu kernel of nodes in the NetA is Linux 4.15.0 or 5.4.0, and the Raspbian kernel of nodes in the NetB is Linux 4.19.75 or 5.10.17. The reason for adopting different operating systems is that we expect to test the platform independence in our Policychain.

B. Blockchain Testbed

Our Policychain is in essence a permission consortium blockchain, which maintains an access control layer to allow certain actions to be performed only by certain identifiable participants. Moreover, the Policychain supports multiple chain structure (similar to multichain[3]) and cross-chain operations to enhance the interoperability between two relatively independent blockchains. On this basis, our experimental system deploys four following chains.

1) *Subject chain* which registers the subject's identity attributes through subject registration transaction (SRT).
2) *Object chain* which establishes the object trusteeship and realizes the registration of object's attributes through object escrow transaction (OET).
3) *Policy chain* which is responsible to policy management and evaluation through PIT transaction described above.
4) *Authorization chain* which encapsulates the whole process of decision making for access request through access granted transaction (AGT).

The technical details among the above SRT, OET, and AGT can be found in our previous works [24] and [25].

Our Policychain is developed and implemented on the multichain platform that uses Bitcoin as the underlying layer. However, it adopts a customizable Round-Robin consensus scheme, rather than PoW as in Bitcoin. On this basis, we additionally introduce a script engine to explain and execute the specified syntax as mentioned above. In addition, the cross-chain access adopts remote procedure call (RPC) for sending data requests to different chains, and then the script engine realizes the final logical decision of policies according to the results returned by the RPC.

We use Apache JMeter that is a powerful automated testing tool to evaluate the performance of our blockchain testbed. In our experiments, the JMeter is developed to simulate multiuser concurrent requests on the Raspberry Pi nodes distributed in two different areas. In Table V, we show the experimental results of subject, object, and policy chains through the tests of 60 concurrent transactions per second. The average response time of the whole system is about 400 ms, and the subject chain is the slowest chain due to the fact that it consumes a lot of time for cryptographic authentication and key escrow. In blockchain, the number of transactions executed per second (called throughput or TPS) is a vastly important indicator to represent the performance of the blockchain system. The total throughput of our blockchain testbed is about 87.01 TPS and the average is about 29 TPS for each chain. The reason for low throughput is that the network bandwidth of the experimental environment is very low (the average bandwidth of data received and sent is about 23 and 66 kB/s, respectively). However, it is enough to support general applications and our system is stable enough to be tested by IoT devices.

[3]https://www.multichain.com

TABLE V
PERFORMANCE TEST RESULTS OF THE BLOCKCHAIN TESTBED FOR SUBJECT, OBJECT, AND POLICY CHAINS

Requests	Executions		Response Times (ms)						Throughput	Network (KB/sec)	
Label	Samples	Error%	Average	Min	Max	90%	95%	99%	Transactions/s	Received	Sent
Subject Chain	60	0.00%	847.18	299	1350	1243.70	1325.65	1350.00	27.32	7.20	18.14
Object Chain	60	0.00%	186.00	72	363	291.50	328.45	363.00	28.75	7.58	20.92
Policy Chain	60	0.00%	166.02	57	280	232.40	273.90	280.00	30.94	8.16	27.11
	180	0.00%	399.73	57	1350	589.20	642.67	664.33	87.01	22.94	66.17

C. Policychain's Transaction and Overheads

To evaluate the performance of our solution, we develop a prototype system to implement our Policychain framework. This system is constructed on the source code of XACML and Bitcoin. Based on this prototype, we can evaluate the performance of our framework from three aspects: 1) PSI; 2) execution for batch rules; and 3) the whole system testing. Our developed prototype system consists of three parts.

1) The first part is the blockchain program, running on each IoT node, that is extended from the source code of Bitcoin Core. This program is implemented in C++ and has approximately 3500 more lines than the original code.
2) The second part is the RS which is built on a file server to simulate resource sharing services on the cloud. It also plays the role of PEP for providing an interface to format and enforce access requests.
3) The third part is the PIP server that is built on the relational database (MySQL) to store attributes and simulate the behaviors defined in the XACML.

We send policy JSON request packets to the blockchain system through the clients simulated by JMeter. As described in Section VI-A, the policy stored in the PIT is represented by the combination of target, condition, and rule. Commonly, the target describes object information, condition indicates condition identifier and rule represents rule identifier. Sending different policies will lead to differences in the size of transactions generated in the blockchain system.

We also design many experiments to check the size of the PIT transaction generated in our ICS-IoT by sending policy data packets with different number of conditions and rules. Specifically, we fix the number of rules to 1, 3, and 5 in the policy data packets, and then we investigate the relationship between the number of conditions and the size of transactions in the Policychain, respectively. As shown in Fig. 12, it is not difficult to find that the number of conditions has a linear relationship with the size of transactions under the same number of rules. Moreover, under the same number of conditions, the larger the number of rules, the larger the size of transactions.

In Fig. 13, we show an PIT instance used in this prototype. This transaction specifies a policy of the resource whose URL is "medical01/server.store.org." The policy can match the "read/write" request that is enforced on the service interface *org.apache.pdfbox.pdfctrl*. Also, the request should satisfy the rule with three conditions, i.e., $(object-\text{id} = medical-record-001.pdf \wedge action-\text{id} = \text{read}) \vee (faculty =$

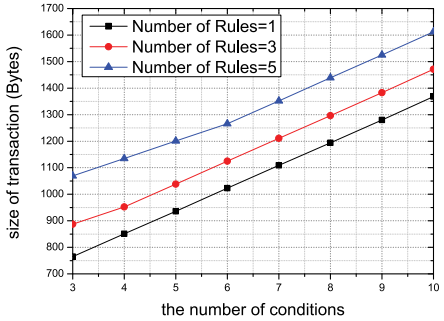


Fig. 12. Relationship between the size of transactions and the number of conditions and rules.


```
{
    ver : "PIT",
    URL : "medical01/server.store.org",
    ruleCombingMethod : Deny-overrides,
    target : [{
          attr : "aotion-id#Act", value : "org.apache.pdfbox.pdfctrl#read"
        },{
          attr : "action-id#Act", value : "org.apache.pdfbox.pdfctrl#write"
    }],
    condition : [{
          id : "expr1",
          expr : "<object-id> OP_OBJATTR <medical-record-001.pdf> OP_EQUAL"
        },{
          id : "expr2",
          expr : "<faculty> OP_SUBATTR <doctor> OP_EQUAL"
        },{
          id : "expr3",
          expr : "<action-id> OP_ACTATTR <org.apache.pdfbox.pdfctrl#read> OP_EQUAL"
    }],
    rule : [{
          id : "rule1",
          effect : "PERMIT",
          expr : "<expr1> <expr3> OP_BOOLAND <expr2> OP_BOOLOR"
    }],
    input : [{
          TxID : 04882c82ed0a4abc7444650d2612e963e14e78e30552baff1033e0f5ded8fea17c,
          scriptSig : "3273c1b4a0f...0cc35fd2325c66fa 037b165c53...217d411c9ef"
    }],
    output : [{
          scriptPubKey : "OP_DUP OP_HASH160 l6b023040bc58fldc63
              ea4b22e033ed206d925d8 OP_EQUALVERIFY OP_CHECKSIG"
    }],
    state : 1,
    TxID : 049f816b8699f9318f6d95adddda91c99ca459cc3cb798eafba653f0a34b78556e,
    time : 1515647112
}
```


Fig. 13. Example of the PIT.

doctor). Moreover, the *input* item of the transaction includes the signature from the policy issuer and the *output* item declares the owner or agent of the policy, as mentioned in Fig. 9 of Section VI-C. The policy set used in the test is from an open-source implementation of the XACML specification, called Balana.[4] We have converted 200 typical XML-formatted policies into the JSON-formatted PIT transactions, in which each policy has around two to five rules and each rule contains at most ten attribute scripts.

D. Performance of Policy Script Interpreter

To evaluate the performance of the PSI, we randomly chose 50 policies with two rules from the above policy set and repeated the decision-making process 100 times on every policy. In this case, the time overheads of two algorithms, *evalRule*() and *evalPolicy*(), are shown in Fig. 14 under two

[4]https://github.com/wso2/balana/tree/master/modules/balana-core

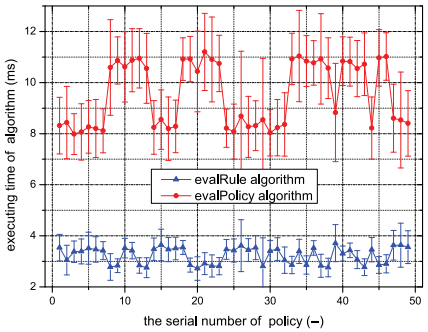


Fig. 14. Performance of script algorithms under different policies and rules.

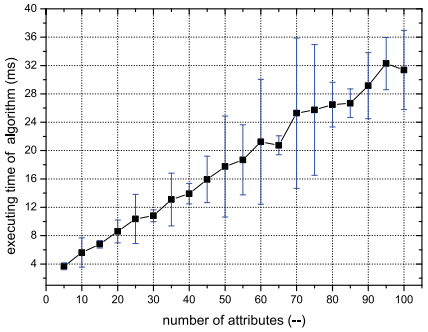


Fig. 15. Performance of batch attribute-script execution.

groups of different rules. Moreover, the error bars are used to express the standard error for 100 tests.

As shown in Fig. 14, the mean executing time of algorithm *evalRule*() is between 3 and 3.5 ms with a relatively small fluctuation. In contrast, the meantime of algorithm *evalPolicy*() is between 8 and 11ms, but it can be seen that the tested policies can be divided into two groups: one group has only one rule that needs to be executed and its average evaluation time is 8 ms, and the other has two rules and its average evaluation time is 11 ms. The reason is that, for the different rule-combining methods, e.g., "Permit-overrides" and "Deny-overrides," the number of the executed rules is uncertain during the policy decision-making process, so that such two groups have a striking difference (around 3 ms) on their mean evaluation time which is approximately equal to that of one rule.

E. Performance of Batch Attribute-Script Execution

Unlike the above testings with only two rules, we further measure the performance of attribute-script execution. In this test, the script algorithm *evalRule*() is evaluated by executing rules with different number (from 1 to 100) of attribute scripts [invoking *evalScript*()]. Generally, there are no more than 20 attribute scripts for a single rule, and accordingly 100 attribute scripts in this test can cover most practical situations, and exceed the general scale by three or four times. In addition, we repeat the above execution 100 times on every rule to reduce the test errors.

Fig. 15 shows the mean and variance of the test results for the above batch attribute-script execution. In view of the trends of the results, it is easy to find that the executing time of the rule is directly proportional to the number of attributes. This

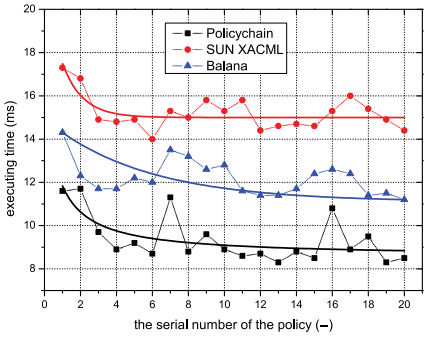


Fig. 16. Performance comparison of whole policy evaluation between Policychain and other XACML implements.

is consistent with the description of the algorithm *evalRule*, in which each of the attribute scripts should be executed once. To improve the execution efficiency, we make a one-time process to obtain all attribute values required by the rule from PIP, and then, cache these values in the program context before rule evaluation. This approach gets rid of the performance degradation caused by frequent requests for attribute values.

F. Performance Comparison

We finally focus on the performance comparison of whole policy evaluation among three different platforms, including Balana, SUN XACML,[5] and our Policychain. We randomly pick up 20 policies from the above policy set, and then perform the decision-making process on each policy 50 times. The test results and their corresponding trend lines are shown in Fig. 16, where the horizontal ordinate is only the serial number of the test policies, and there is no direct correlation between each policy. The fitting line is just to highlight the differences in the performance of those platforms.

As shown in Fig. 16, it can be seen that the execution time (around 9 ms) of policy evaluation under the Policychain platform is less than that of two other platforms (around 11 ms for Balana and 15 ms for SUN XACML). We consider the possible reasons as follows.

1) The parsing process of JSON-formatted policies takes less memory and time than that of XML-formatted policies in the other two platforms;
2) The policies expressed by scripting language can be directly executed by the script interpreter, so that our platform gets rid of the process of constructing a policy decision tree which is needed in the other two platforms.

Note that, the above testings only take place on a single node and the whole decision-making process should contain the consensus verification of the final result on all nodes in the network. Thus, the time cost is associated with the consensus protocol in the system.

In summary, the testing results through three aspects indicate that the policy evaluation on our prototype system has better performance than the common XACML systems. Thus, the Policychain can be seen as a more efficient and easy-to-develop platform for the ABAC model in an IoT-ICS.

[5]http://sunxacml.sourceforge.net/index.html

VIII. Conclusion

To reduce the risks of sensitive data leakage in IoT-ICS, we use the blockchain inherent scripting language to express the complex logic of ABAC/XACML policies and apply the script interpreter to make a quick execution of the access control policy. This script-based approach makes possible for highly available, autonomous, traceable authentication services. However, there are still some aspects to be improved. For example, the environmental attributes should be acquired and verified in an automatic and dynamic way, so as to enhance the flexibility of authentication. Moreover, a new authorization service on smart contracts should be further studied because this smart contract platform (Docker or virtual machine) has stronger computing power than script engine.

References

[1] S. Grüner, J. Pfrommer, and F. Palm, "RESTful industrial communication with OPC UA," *IEEE Trans. Ind. Informat.*, vol. 12, no. 5, pp. 1832–1841, Oct. 2016.

[2] E. Yuan and J. Tong, "Attributed based access control (ABAC) for Web services," in *Proc. IEEE Int. Conf. Web Services (ICWS)*, 2005, pp. 561–569.

[3] W. Tolone, G.-J. Ahn, T. Pai, and S.-P. Hong, "Access control in collaborative systems," *ACM Comput. Surveys*, vol. 37, no. 1, pp. 29–41, 2005.

[4] G. Demesure, M. Defoort, A. Bekrar, D. Trentesaux, and M. Djemai, "Decentralized motion planning and scheduling of AGVS in an FMS," *IEEE Trans. Ind. Informat.*, vol. 14, no. 4, pp. 1744–1752, Apr. 2018.

[5] M. Vallee, M. Merdan, W. Lepuschitz, and G. Koppensteiner, "Decentralized reconfiguration of a flexible transportation system," *IEEE Trans. Ind. Informat.*, vol. 7, no. 3, pp. 505–516, Aug. 2011.

[6] V. C. Hu, D. F. Ferraiolo, and D. R. Kuhn, "Assessment of access control systems," U.S. Dept. Commerce, Nat. Inst. Stand. Technol., Rep. NISTIR 7316, 2006.

[7] D. D. F. Maesa, P. Mori, and L. Ricci, "Blockchain based access control," in *Proc. IFIP Int. Conf. Distrib. Appl. Interoperable Syst.*, 2017, pp. 206–220.

[8] G. Kappes, A. Hatzieleftheriou, and S. V. Anastasiadis, "Multitenant access control for cloud-aware distributed filesystems," *IEEE Trans. Dependable Secure Comput.*, vol. 16, no. 6, pp. 1070–1085, Nov./Dec. 2019.

[9] F. Kelbert and A. Pretschner, "Data usage control for distributed systems," *ACM Trans. Privacy Security*, vol. 21, no. 3, pp. 1–32, 2018.

[10] Y. Xue, K. Xue, N. Gai, J. Hong, D. S. L. Wei, and P. Hong, "An attribute-based controlled collaborative access control scheme for public cloud storage," *IEEE Trans. Inf. Forensics Security*, vol. 14, pp. 2927–2942, 2019.

[11] A. Anderson *et al.*, *eXtensible Access Control Markup Language (XACML) Version 1.0*, OASIS, Burlington, MA, USA, 2003.

[12] M. Lischka, Y. Endo, and M. Sánchez Cuenca, "Deductive policies with XACML," in *Proc. ACM Workshop Secure Web Services*, 2009, pp. 37–44.

[13] D. Diaz-Lopez, G. Dolera-Tormo, F. Gomez-Marmol, and G. Martinez-Perez, "Managing XACML systems in distributed environments through meta-policies," *Comput. Security*, vol. 48, pp. 92–115, Feb. 2015.

[14] Y. Demchenko, O. Koeroo, C. de Laat, and H. Sagehaug, "Extending XACML authorisation model to support policy obligations handling in distributed application," in *Proc. 6th Int. Workshop Middleware Grid Comput.*, 2008, pp. 1–6.

[15] J. Peters, R. Rieke, T. Rochaeli, B. Steinemann, and R. Wolf, "A holistic approach to security policies—Policy distribution with XACML over COPS," *Electron. Notes Theor. Comput. Sci.*, vol. 168, pp. 143–157, Feb. 2007.

[16] H. Jiang and A. Bouabdallah, "JACPoL: A simple but expressive JSON-based access control policy language," in *Proc. IFIP Conf. Inf. Security Theory Pract.*, 2017, pp. 56–72.

[17] D. D. F. Maesa, P. Mori, and L. Ricci, "A blockchain based approach for the definition of auditable access control systems," *Comput. Security*, vol. 84, pp. 93–119, Jul. 2019.

[18] Y. Zhu, Y. Qin, Z. Zhou, X. Song, G. Liu, and W. C.-C. Chu, "Digital asset management with distributed permission over blockchain and attribute-based access control," in *Proc. IEEE Int. Conf. Services Comput.*, 2018, pp. 193–200.

[19] D. Ferraiolo, R. Chandramouli, R. Kuhn, and V. Hu, "eXtensible access control markup language (XACML) and next generation access control (NGAC)," in *Proc. ACM Int. Workshop Attribute Based Access Control*, 2016, pp. 13–24.

[20] R. Zhang, R. Xue, and L. Liu, "Security and privacy on blockchain," *ACM Comput. Surveys*, vol. 52, no. 3, pp. 1–34, 2019.

[21] T. I. Kiviat, "Beyond bitcoin: Issues in regulating blockchain tranactions," *Duke Law J.*, vol. 65, no. 3, pp. 569–608, 2015.

[22] G. Wang, "Improving data transmission in Web applications via the translation between XML and JSON," in *Proc. 3rd Int. Conf. Commun. Mobile Comput.*, 2011, pp. 182–185.

[23] L. M. Vaquero, L. Rodero-Merino, and D. Morán, "Locking the sky: A survey on IaaS cloud security," *Computing*, vol. 91, no. 1, pp. 93–118, 2011.

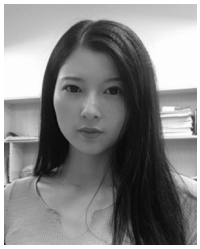
[24] Y. Zhu, Y. Qin, G. Gan, Y. Shuai, and W. C. Chu, "TBAC: Transaction-based access control on blockchain for resource sharing with cryptographically decentralized authorization," in *Proc. IEEE 42nd Annu. Comput. Software Appl. Conf. (COMPSAC)*, vol. 1, 2018, pp. 535–544.

[25] Y. Zhu, R. Yu, D. Ma, and W. C.-C. Chu, "Cryptographic attribute-based access control (ABAC) for secure decision making of dynamic policy with multiauthority attribute tokens," *IEEE Trans. Rel.*, vol. 68, no. 4, pp. 1330–1346, Dec. 2019.

[26] I. Giechaskiel, C. Cremers, and K. B. Rasmussen, "On bitcoin security in the presence of broken cryptographic primitives," in *Proc. Eur. Symp. Res. Comput. Security*, 2016, pp. 201–222.

[27] V. Gramoli, "From blockchain consensus back to Byzantine consensus," *Future Gener. Comput. Syst.*, vol. 107, pp. 760–769, Jun. 2020.

[28] Y. Zhu, K. Riad, R. Guo, G. Gan, and R. Feng, "New instant confirmation mechanism based on interactive incontestable signature in consortium blockchain," *Front. Comput. Sci.*, vol. 13, no. 6, pp. 1182–1197, 2019.

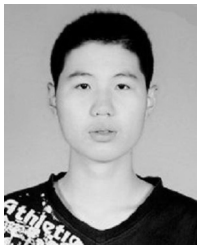


E Chen received the B.S. degree from the School of Mathematics and Physics, University of Science and Technology Beijing, Beijing, China, in 2013, where she is currently pursuing the Ph.D. degree with the Department of School of Computer and Communication Engineering.

Her research interests include attribute-based system and lattice cryptography.

Yan Zhu received the M.S. and Ph.D. degrees from Harbin Engineering University, Harbin, China, in 2002 and 2005, respectively.

He is currently a Full Professor with the School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing, China. He was an Associate Professor of Computer Science with the Institute of Computer Science and Technology, Peking University, Beijing, China, from 2007 to 2013. His research interests include cryptography, secure computation, and network security.



Zhiyuan Zhou received the B.E. and M.E. degrees from the School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing, China, in 2017 and 2020, respectively.

His research interests include attribute based access control and distributed ledger technology.

Shou-Yu Lee (Member, IEEE) received the B.S. degree in computer science from National Tsing Hua University, Hsinchu, Taiwan, in 2010, and the M.S. degree in computer science from Tunghai University, Taichung, Taiwan, in 2012. He is currently pursuing the Ph.D. degree with the University of Texas at Dallas, Richardson, TX, USA, under the supervision of Prof. W. E. Wong.

His current research interests include software fault localization, context-sensitive computing, and software risk analysis.

W. Eric Wong (Senior Member, IEEE) received the M.S. and Ph.D. degrees in computer science from Purdue University, West Lafayette, IN, USA, in 1991 and 1993, respectively.

He is a Full Professor and the Founding Director of the Advanced Research Center for Software Testing and Quality Assurance with Computer Science, University of Texas at Dallas (UTD), Richardson, TX, USA. He also has an appointment as a Guest Researcher with the National Institute of Standards and Technology, Gaithersburg, MD, USA, an agency of the U.S. Department of Commerce. Prior to joining UTD, he was with Telcordia Technologies (formerly Bellcore), Piscataway, NJ, USA, as a Senior Research Scientist and the Project Manager in charge of Dependable Telecom Software Development. He has very strong experience developing real-life industry applications of his research results. His research focuses on helping practitioners improve the quality of software while reducing the cost of production. In particular, he is working on software testing, debugging, risk analysis/metrics, safety, and reliability.

Prof. Wong is the Editor-in-Chief of IEEE TRANSACTIONS ON RELIABILITY. In 2014, he was named the IEEE Reliability Society Engineer of the Year. He is also the Founding Steering Committee Chair of the IEEE International Conference on Software Quality, Reliability, and Security and the IEEE International Workshop on Debugging and Repair.

William Cheng-Chung Chu (Senior Member, IEEE) received the M.S. and Ph.D. degrees in computer science from Northwestern University, Evanston, IL, USA, in 1987 and 1989, respectively.

He is currently a Distinguished Professor with the Department of Computer Science, Tunghai University, Taichung, Taiwan, where he had served as the Director of Software Engineering and Technologies Center from 2004 to 2016 and as the Dean of Research and Development office from 2004 to 2007. He was a Research Scientist with the Software Technology Center, Lockheed Missiles and Space Company, Inc., Sunnyvale, CA, USA. In 1992, he was also a Visiting Scholar with Stanford University, Stanford, CA, USA.

Dr. Chu was a recipient of the special contribution awards in both 1992 and 1993 and the PIP Award in 1993 at Lockheed Missiles and Space Company, Inc. He is an Associate Editor for the IEEE TRANSACTIONS ON RELIABILITY, the *Journal of Software Maintenance and Evolution*, the *International Journal of Advancements in Computing Technology*, and the *Journal of Systems and Software*.