ViotSOC: Controlling Access to Dynamically Virtualized IoT Services using Service Object Capability

Hajoon Ko
Harvard Unviersity
hrko@g.harvard.edu

Jiong Jin
Swinburne University of Technology
jiongjin@swin.edu.au

Sye Loong Keoh
University of Glasgow
SyeLoong.Keoh@gla.ac.uk

ABSTRACT

Virtualization of Internet of Things(IoT) is a concept of dynamically building customized high-level IoT services which rely on the real time data streams from low-level physical IoT sensors. Security in IoT virtualization is challenging, because with the growing number of available (building block) services, the number of personalizable virtual services grows exponentially. This paper proposes Service Object Capability(SOC) ticket system, a decentralized access control mechanism between servers and clients to efficiently authenticate and authorize each other without using public key cryptography. SOC supports decentralized partial delegation of capabilities specified in each server/client ticket. Unlike PKI certificates, SOC's authentication time and handshake packet overhead stays constant regardless of each capability's delegation hop distance from the root delegator. The paper compares SOC's security benefits with Kerberos and the experimental results show SOC's authentication incurs significantly less time packet overhead compared against those from other mechanisms based on RSA-PKI and ECC-PKI algorithms. SOC is as secure as, and more efficient and suitable for IoT environments, than existing PKIs and Kerberos.

CCS Concepts

•Security and privacy → Key management; Authentication; Access control; *Authorization;* •Networks → Network security; Security protocols;

Keywords

Key Management; Authentication; Access Control; Internet-of-Things; Virtualized Services; Access Control

1. INTRODUCTION

As the number of IoT devices increases, multiple IoT devices are likely to collaborate to address high-level issues such as comprehensive data aggregation, analysis and processing. As a consequence, IoT devices providing services can be dynamically composed or *virtualized* [1–3], hence facilitating re-use of IoT services. IoT virtualization is a concept of dynamically building customized high-level IoT services which rely on the real-time I/O data streams from low-level IoT sensors/actuators. An IoT device having access to the Internet can potentially interact with any other IoT devices in the world, exchange data with them, and thereby creating a customized virtual service for its own clients. Virtualized IoT services provide scalability to build large-scale pervasive systems, and they can be dynamically composed and disbanded according to the requirements and context.

Meanwhile, security becomes complex because a virtualized IoT service is a composition of many IoT devices. Virtualized IoT services may have complicated service dependency, and virtualization may occur not only within a closed local network but through external open networks governed by mutually untrusted administrators or organizations [4]. Furthermore, virtualized IoT services may be re-virtualized in a recursive manner. A client who wishes to use a virtual IoT service needs to verify if it indeed has the capability of providing the claimed virtual service. Likewise, the server needs to verify if the client is authorized to use the requested virtual service. Even if IoT services become virtual in a highly recursive manner, the mutual security verification process between the client and server has to remain light-weighted; otherwise it becomes difficult to support resource-constrained IoT devices.

As a solution for the aforementioned security problems, this paper proposes a Service Object Capability (SOC) ticketing system that allows each server and client to efficiently authenticate and authorize each other in a dynamic environment where virtual services can be created in highly recursive and decentralized manner. SOC uses dynamically created secret *passcodes*, generated by a security ticket issuer, as a secret to enable each server and client to mutually authenticate each other. Each trusted ticket issuer creates two types of *passcodes*: S (Server) passcode and C (Client) passcode. The ticket issuer gives out these passcodes to its approved clients or servers, and the holder of passcode(s) can delegate them to other entities. For each service, server passcodes are generated as a one-way hash chain [5] by recursively applying an one-way hash function to the issuer-created server seed, and such created hash values are sequentially one-to-one mapped to distinct timeslots in a reverse time order. Thus, the holder of a particular passcode can compute any other passcodes mapped to past

timeslots by using the same one-way hash function, while it is not possible to compute any other passcodes mapped to future timeslots. This non-invertibility of passcodes securely enforces an expiry on each passcode, because in SOC, each passcode is valid and usable only during its assigned timeslot. One the other hand, each service's client seed and client passcodes are different and randomly chosen per client by the trusted ticket issuer. In addition, each client passcode comes with a '*hint*', an encryption of the client passcode by its corresponding server passcode (where both C/S passcodes are mapped to the same timeslot). The client authenticates the server by checking if it it can decrypt the *hint* with the server passcode, while the server authenticates the client by checking if it knows the decrypted value of the *hint*, which is the client passcode. This way, they mutually authenticate without revealing the server's secret passcode to the client. At the end of SOC authentication, both the client and server establish a shared session key. SOC's computational overhead is lower than any other PKI-dependent IoT security framework, because its authentication only depends on symmetric key cryptography and one-way hash function.

This paper's contributions are as follows:

- Proposing SOC ticket solution as a lightweight and efficient mechanism for securing personalizable IoT service virtualization. SOC's decentralized access control is computationally more efficient and suitable for IoT environments.

- Proposing a lightweight security solution allowing each server to delegate its service capabilities to facilitate service virtualization. The delegator can optionally reduce the service capability to be delegated The holder of server and client tickets mutually authenticate each other's capability without using public key cryptography.

- Keeping the authentication time and handshake packet overhead constant for a server and client's mutual authentication and authorisation regardless of the service virtualization level (i.e. the service delegation distance from their root delegators)

This paper is organized as follows: In Section II presents virtualized IoT use cases and a set of security requirements. Section III introduces the Service Object Capablity (SOC) architecture, describing the ticket issuance, security handshake, authentication and authorization, as well as delegation and revocation. In Section IV, we presents the implementation of SOC architecture and its experimental results, while Section V provides some discussion. Section VI presents related work and we conclude the paper with future work in Section VII.

2. SMART LIVING USE CASES AND SECURITY REQUIREMENTS

This section describes a smart IoT service network as a comprehensive use case to derive the operation and security requirements.

Figure 1 describes an exemplary smart IoT network, which will be used throughout the whole section. It shows a scenario in which different kinds of locks, i.e., home lock, work (office) lock, and car lock with computational capabilities can be locally and remotely managed by using a smartphone or a digital key. By using a smartphone, the user
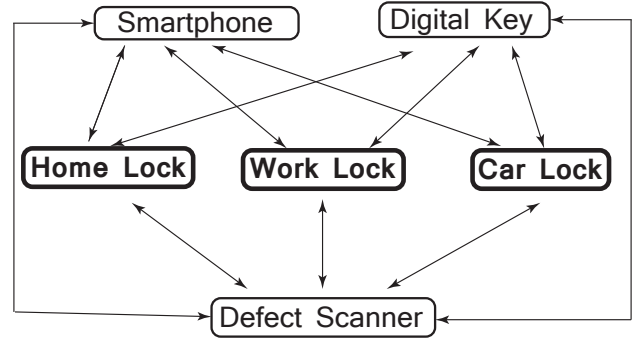

Figure 1: A smart IoT service network

is able to *lock, unlock* and *log* the door, assuming that the home lock offers its service object, "SmartHomeLock". The *Lock/Unlock* commands correspond to executing a command to physically lock/unlock the door, while *log* command corresponds to querying the lock/unlock history of the home lock. The same capabilities and services are provided by the work lock and the car lock in this case. Access capability can also be granted to the digital key to *lock/unlock/log* any of the locks in the system. From this scenario, we derive the basic security requirements as follows:

- *HomeLock* should be able to enforce independent access control on each of its three different service methods, i.e., *lock/unlock/log*. Specifically, home lock should allow smartphone to query logs and lock the door, but shouldn't allow it to unlock the door. The rationale for this is that if the user's smartphone is stolen, the attacker will not be able to gain access to the user's home. However, the home lock will allow the digital key to both lock/unlock the door, but it can't query about logs, because the key does not physically have an LCD screen to display the received log file to the user.

- *Defect Scanner* is a service to perform system check on the locks. In this case, it should be able to query all the locks' logs to analyse their system defect and perform lock/unlock operation to test their functionality. Once the lock passes all security checks, the defect scanner approves their corresponding service capability, certifying that the locks are genuine and authentic.

- *Smartphone* and *Digital Key* should be granted permission to access the lock services provided by the home/work/car locks. At the same time, they should also obtain a certification from the defect scanner to ensure that the lock services are genuine.

The use case can be extended further to dynamically create a virtualized IoT service called "SmartAllLock" as shown in Figure 2. In many cases, permission and service provisioning can be delegated to authorized parties, and they may optionally be weaken to be delegated, such as shortening the permitted time to access a service, removing some service methods, or changing the permission types.

As shown in Figure 2, the *Smart Security Server* has been delegated by *Home Lock*, *Work Lock* and *Car Lock* to provide their respective services. This also implies that the
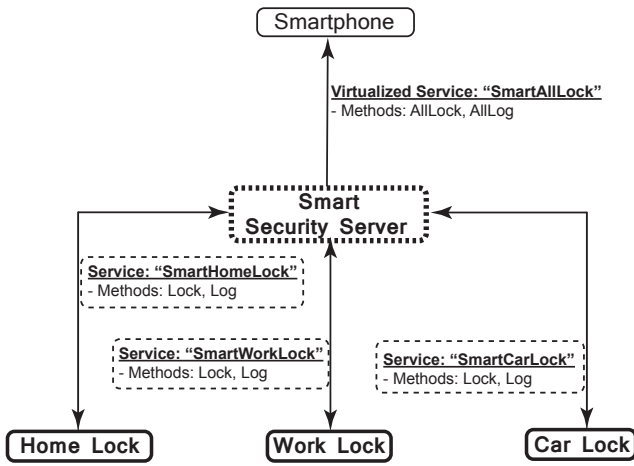
Figure 2: A virtualized service called "SmartAllLock", which depends on three lower-level service objects delegated from three locks.

Smart Security Server is capable of inventing its own service object based on its *delegated* service objects. The object invented in such a way is called a *virtualized* object [6], [3], whose purpose is to provide a high-level service based on integrating data, resource or services offered from low-level service objects. The first usefulness of the service delegation is, when the locks have poor internet connection while the user is far away, it may instead connect to the fast home security server having both good internet connection and good local network connection with local locks. Thus, the user may send a service request to the *Smart Security Server*, instead of the individual lock. The second usefulness of the service delegation is, the *Smart Security Server* can invent a virtualized service object that gives a convenient high-level interface of controlling all three locks simultaneously in an easier manner instead of contacting each lock individually at a time. In this case, the role of the*Smart Security Server* is different from a local firewall for locks, because these locks (i.e. home, work, car) may not in the same local network.

The *Smart Security Server* offers its users the following three high-level operations: First, to view the system logs and lock history of the home lock, work lock and car lock all at once. Second, to execute a lock command on all three locks at once. These new operations are provided as the *Smart Security Server*'s two virtualized service object methods: ["S-Server:/AllLock"] and ["S-Server:AllLog"]. In this model, the *Smart Security Server* is more than a proxy between the client and each smart lock, because it actively makes its own service available to the smartphone, rather than simply forwarding data packets of lower-level servers to the client from the middle. To this end, we further add the following security requirements:

- *Home/Work/Car Lock* should be able to delegate its service provisioning to a trusted entity such as the *Smart Security Server* to handle service requests from clients.

- *Smart Security Server* which has been delegated the corresponding services by *Home/Work/Car Lock* should be capable of handling the service requests on behalf of the low-level service objects.

- *Smart Security Server* which is a virtualized service can grant capability to *smartphone* or *digital key* to access its new virtual service. Similarly, it also has to prove to its client that it is capable of providing the new virtualized service.

3. SERVICE OBJECT CAPABILITY (SOC) ARCHIECTURE

3.1 Overview

This subsection defines key terms and functions which are necessary to describe SOC.

Service Object: It can refer to a variety of things: a single file, a bundle of files, a system resource, a network service, a service port, etc. Each service object's name has to be uniquely identifiable within a network, just like URIs. One way of enforcing it is to represent each name as a combination of a server's IP address and its unique service name. In SOC model, we assume that each service object's name is publicly known- or publicly reachable- by any entities in the network, just as every website visitor already knows or can learn the website's URL.

SOC tickets: They are used by two communicating entities to prove one's own capability and to validate the opponent's capability. There are two ticket types: C (Client) ticket and S (Server) ticket. A client ticket is required for proving one's eligibility to use a particular service object, while a server ticket is used to prove one's eligibility to provide a particular service.

Ticket Issuer: An entity who issues SOC tickets. The ticket issuer is the root of trust in our scheme, analogous to a certificate authority (CA) in PKI. An issuer defines each of its ticket's capability type (C/S), targeting service object, the list of approved sub-services for that service object, the ticket's expiry, and secret *passcodes* as the ticket's secret credential to prevent illegal capability forging. The detailed passcode mechanism is illustrated in Section 3.2.

Any trusted entity can become a ticket issuer: a human user, an OS kernel, a device, an application process, a server, a client, or a third party process, etc. Each ticket issuer has to be uniquely identifiable within a network, in order to differentiate tickets issued from different issuers. As such, each issuer puts its unique fingerprint on every ticket it issues, which is an integer large enough (e.g. 32 bytes) not to coincidentally overlap with other issuer's fingerprint. Fingerprints exist only to help ticket holders sort their tickets by issuer. Each ticket holder verifies its ticket in the sense that its secret passcodes were confidentially transmitted by its ticket issuer (e.g. via secure communication channel or NFC).

Our framework assumes ticket holders know and trust their ticket issuers. Tickets need not be signed by the issuer, because ticket holders do not need to use public keys to verify each other's ticket, but with their tickets' secret passcodes confidentially given by their issuers. Issuers grants tickets with their associated passcodes.

The role of a ticket issuer is to approve whether a particular entity may have server/client capabilities for some target [service, method], and also to provide a mechanism for the

holders of C/S tickets targeting the same [Service:Method] to authenticate each other's client/server capability for the target. This means, the server ticket holder validates if the client ticket holder is eligible to request for a particular service or sub-service (i.e. [service, method]), while the client ticket holder validates if the server ticket holder is a trustworthy source of that service. One ticket is a *counterpart* of another ticket if both tickets have the the common [service, method] target parameter but their capability types are opposite: One is a client type and the other a service type. The way the *counterpart* C/S tickets validate the genuinness of each other is done by matching their secret passcodes, which is discussed in depth in Section 3.2.

C⟨issuer$_a$, *ServiceObj*, MethodList, IsValidator, Expiry⟩

This represents a client capability ticket for *ServiceObj*'s method list, *MethodList*, which is issued by *issuer$_a$*. An entity who wants to use *ServiceObj*'s particular method that is part of *MethodList* can use this ticket to the server servicing *ServiceObject* to prove its eligibility and authorization. This ticket is a capability proof that its holder is approved by *issuer$_a$* to use a particular service method belonging to *MethodList* in *ServiceObj*. This ticket expires at timeslot *Expiry*. This ticket can also be used as a validator to check if the server is approved by *issuer$_a$* to service *ServiceObject*'s *MethodList*.

S⟨issuer$_a$, *ServiceObj*, MethodList, IsValidator, Expiry⟩

This represents a server capability ticket for *ServiceObj*'s method list, *MethodList*, which is issued by *issuer$_a$*. This ticket expires at timeslot *Expiry*. A server servicing *ServiceObj*'s *MethodList* requires this ticket to be presented to its clients to prove its server capability in the similar manner as aforementioned.

3.1.1 One-way Authentication

There are cases where mutual authentication is unnecessary, but only one-way authentication is required. For instance, suppose a system defect scanner scans various service devices to check their system integrity, and after passing all checks the scanner issues them S capability tickets for servicing their services. Then, the scanner issues the counterpart C tickets to external clients who is willing to use those service devices. The clients' C tickets are used to validate if the server device(s) they communicate with has the valid S ticket issued by the defect scanner. This scenario's security enforces that if the client holds an C ticket issued by the defect scanner but a particular service device doesn't have the counterpart S ticket, the device is not validated by client and thus the client rejects the service; but if the client doesn't hold defect scanner's C ticket while the server device has the S ticket, server device should not deny offering its service to the client even if the client doesn't have the counterpart C ticket. This is because the role of the S ticket issued by the defect scanner is to prove its holder's(server's) server capability to its counterpart C ticket holder(client), but the client's C ticket is not designed as a proof for the client capability to use the service, because the defect scanner only deals with service device integrity in this scenario. In other words, the defect scanner designed and created its C tick-

ets as validators for clients to validate the server's S tickets. Thus, we introduce an extra flag in each (S and C) ticket denoting if the ticket issuer designed the ticket for validating its counterpart ticket or not. To this end, there is "*IsValidator*" flag field in each SOC ticket. For the defect scanner's C tickets, this field's value is marked as "*V(Validator)*", but for the defect scanner's S tickets, this field's value is "*N(Non-Validator)*". In the previous file sharing example, this field's value is "V" for both S and C tickets, because those tickets are designed for mutual capability validation between the server and client. It is the ticket issuer who is responsible for properly marking *isValidator* field, as it defines the usage of each ticket.

3.1.2 Multi-Issuer Authentication

Authorising clients can be done by another independent issuer who is specialized in approving human clients and their personal devices. Such scenario ends up involving more than a ticket issuer. For instance, if the maintenance of a security door lock sensor belongs to both the laboratory as well as to the building management department, the security door lock may need to check if the lock user is approved by both the laboratory leader and the building manager to send lock/unlock commands on it. To support this requirement, SOC allows checking tickets issued from multiple issuers all at once. In this case, the door requires its clients to hold two C tickets: one issued by the lab manager and the other issued by the building manager, whereas the client requires the door lock to submit an S ticket issued by the defect scanner. In total, the client and door lock hold three tickets in each: client's two tickets are non-validator and one ticket is a validator, while the door lock's two tickets are validators and one ticket s a non-validator. In fact, depending on the ticket issuer's design decision, the client's two C tickets from the building manager and laboratory header could be designed as validators just like their counterpart S tickets. For example, the C ticket issued by the building manager can be designed as a validator, if it is also in charge of approving the service integrity of the door lock. The C ticket issued by the laboratory leader would be designed as a non-validator, if it is not in charge of approving the door lock's service integrity.

3.1.3 Fine-granularity Access Control

For the same service object, SOC can grant a capability for each method independently. For example, a smart door lock service may consist of three sub-services: querying status history (e.g. the latest time the door was unlocked), executing locking operation, and executing unlocking operation. We may want to give our smartphone the capability to lock the door, so that it can remotely lock the door if we suddenly realize having forgotten to lock the door when leaving home. However, we may want to give the capability for unlock method exclusively to our digital key that supports only NFC. The reason behind this is, smartphones are generally prone to remote cyber attacks via Wifi or 3G, and if they ever get compromised, an attacker can unlock our home door and intrude, which is a lethal security threat. On the other hand, a digital key supporting only NFC is safer from remote cyber attacks because it does not support remotely accessible network backdoors. To this end, the issuer can grant a specialized digital key device a C ticket approving the capability for both [doorlock, lock] and [door-

lock, unlock] sub-services, while our smartphone another C ticket approving the capability for only [doorlock, lock] sub-service.

$$S_1\langle issuer_a,\ Svc_n,\ MList_m\ isV_i,\ Exp_1\rangle \preceq S_2\langle issuer_a,\ Svc_n,\ MList_e,$$
$$isV_i\ ,\ Exp_2\rangle \iff (MList_m \subseteq MList_e) \wedge (Exp_1 \leq Exp_2) \tag{1}$$

Given Equation 1, a particular SOC ticket can be converted into any other *reduced* capability tickets whose target sub-services, [service, methodList], are a subset of the original ticket's [service, methodList]. Furthermore, the delegated ticket's expiry can be shortened by the delegator. One SOC ticket authenticates its client/server capability to the holder of its counterpart ticket holder, and then in turn authenticates the counterpart ticket.

Issuers(A, C, *ServiceObj*, Method, isV_i): This function outputs the set of issuers, based on all tickets A holds, who have approved entity A the S(server) capability for servicing [Service, Method]. isV_i sets the requirement type of tickets to be investigated, whose type can be either *V(validator)* or *N(non-validator)*.

Issuers(A, C, Service_n, Method_m, isV_i)
$= \{issuer_a \mid C\langle issuer_a,\ Service_n,\ Method_m,\ isV_i\rangle \in Tickets(A)\}$
Issuers(A, S, Service_n, Method_m, isV_i)
$= \{issuer_a \mid S\langle issuer_a,\ Service_n,\ Method_m,\ isV_i\rangle \in Tickets(A)\}$
$$\tag{2}$$

Equation 2 describes the function *Issuer*, which extracts the list of all issuers who issued A tickets whose capability target is [Service_n, Method_m], capability type is either client (first equation) or server (second equation), and validator type is isV_i.

$A \overset{[Svc_n,\ M_m]}{\longleftarrow} B$, **iff**

Issuers(A, C, Svc_n, M_m, V/N) $\supseteq$ Issuers(B, S, Svc_n, M_m, V)
AND
Issuers(A, C, Svc_n, M_m, V) $\subseteq$ Issuers(B, S, Svc_n, M_m, V/N)
$$\tag{3}$$

Equation 3 describes the finalized SOC mutual authentication check that takes place whenever a client attempts to use a particular service object offered by a server. In the equation, A is a client and B a server. A is allowed to use [Service_n, Method_M] serviced by B if and only if the following two conditions hold. First, B's every trusted ticket issuer who instructed B to authenticate the client's client capability for [Service_n, Method_M] (by having issued B a *validator*-marked S ticket) has to also have approved A of using [Service_n, Method_m] (by having issued A the counterpart C capability ticket). Likewise, A's every trusted ticket issuer who instructed A to authenticate the server's server capability for [Service_n, Method_M] (by having issued A a *validator*-marked C ticket) has to also have approved B of the server capability on [Service_n, Method_M] (by having issued B the counterpart S capability ticket).

SOC allows issuers of physical (lower-level) services to selectively delegate their ticket issuance role of particular [Service_n, Method_m] to a ticket issuer of a virtual (higher-level) service. Therefore, clients of the virtual service only need to contact the ticket issuer of the virtual service in order to retrieve all necessary tickets to verify an entity offering this virtual service.

3.2 Ticket Issuance and Generation

This section discusses the structure of SOC tickets and how the ticket validation mechanism works between ticket holders.

Each ticket is locally generated by a ticket issuer's device. The issuer device may be the service provider itself, or it can be a third party device who does not provide the service. The genuineness of a particular C ticket can be verified only by using its genuine counterpart S ticket, and vice versa. Each S ticket comes with a set of secret *passcodes* generated from its issuer's secret seed.

In a ticket, each [service, method] target comes with a *passcode* as a capability credential. Every passcode has a one-to-one mapping to a distinct timeslot as its expiry (see Table 1), and for each service, passcode chains are generated by applying a crypto-hash function in a reversed order of timeslots on the ticket issuer's initial seed. Therefore, an entity having a certain passcode mapped to a certain timeslot is capable of recursively generating any other passcodes whose targeting [service, method] are the same but mapped to past timeslots, by using the same crypt-hash function used by the ticket issuer to generate passcode chains. However, it cannot generate any passcodes mapped to future timeslots because that's equivalent to an inverse of the crypto-hash function, which is computationally infeasible. As such, the use of each passcode is bound to its mapped timeslot, and it becomes unusable as the current time passes its mapped time. This way, each passcode has its own expiry.

The middle box in Table 1 describes an example where the issuer issues to $Server_A$ a server passcode expiring at timeslot 80, and issues to $Client_A$ two client passcodes and hints valid for timeslot 10 and 11, respectively. The bottom box in Table 1 describes an example of mutual authentication between $Server_A$ and $Client_A$ at timeslot 10. $Client_A$ sends to $Server_A$ its hint for timeslot 10 (as well as its C ticket), then $Server_A$ decrypts it and gets the client passcode. Finally, $Server_A$ sends the client a random session key encrypted with $Client_A$'s passcode. If the client holds client passcodes for multiple services/methods, s/he sends a hint corresponding each service/method, and the server decrypts each hint with the corresponding server passcode. SOC scheme requires that IoT devices using SOC tickets have the ability to keep time and adjust it once every while by querying their local or global clock server(s). Ticket holders are allowed to have time drift as big as the window size of their ticket's timeslot.

For simplicity, all hexadecimal numbers in the Figure 3 are represented as 10-byte-long numbers. The actual SOC library implementation uses security enhanced 32-byte SHA256 hash values. Figure 3 shows an example of **S**⟨HomeLock, *"SmartHomeLock"*, Lock & UnLock, Validator⟩ ticket, which consists of its issuer's unique fingerprint, service name or method, ticket type (i.e. S), an expiry, and *validator* flag. The Issuer's Seed Era represents the seed version used for generating this ticket's passcodes. Unit Timeslot Length is the atomic length of each timeslot used for this type of ticket by its issuer, which are crucial information for two communicating ticket holders when they negotiate their common

An example of an issuer's C/U Ticket Passcode Generation Table

Target (service, method) = *("SmartHomeLock", "Lock")*

	Server_A Passcode	Client_A Passcode	Client_A Hint
Time=0	$h^{180}(seed)$	$rand_{(A,0)}$	$Encrypt\{rand_{(A,0)}\}_{h^{180}(seed)}$
Time=1	$h^{179}(seed)$	$rand_{(A,1)}$	$Encrypt\{rand_{(A,1)}\}_{h^{179}(seed)}$
......	......	......	......
Time=80	$h^{100}(seed)$	$rand_{(A,80)}$	$Encrypt\{rand_{(A,80)}\}_{h^{100}(seed)}$
......	......	......	......
Time=178	$h^{2}(seed)$	$rand_{(A,178)}$	$Encrypt\{rand_{(A,178)}\}_{h^{2}(seed)}$
Time=179	$h^{1}(seed)$	$rand_{(A,179)}$	$Encrypt\{rand_{(A,179)}\}_{h^{1}(seed)}$

[Issuer's Ticket (Passcode) Assignment to Server_A, Client_A and Client_B]

$\text{Server}_A \leftarrow$ Issuer: $h^{100}(seed)$ *(⇒this server passcode is usable until Timeslot=80)*

$\text{Client}_A \leftarrow$ Issuer: $rand_{(A,10)}, rand_{(A,11)}$, $\text{Hint}_{(A,10)}, \text{Hint}_{(A,11)}$ *(⇒these client_A passcodes are usable for Timeslot=10, 11)*

[Mutual Authentication Steps between $\text{Client}_A \leftrightarrow \text{Server}_A$ at current Time=10]

1. $\text{Client}_A \rightarrow \text{Server}_A$: $\text{Hint}_{(A,10)}$ *(⇒this hint is valid for Timeslot=10)*
   Server_A derives $h^{170}(seed)$, decrypts $\text{Hint}_{(A,10)}$ with it, and gets $rand_{(A,10)}$.
2. $\text{Client}_A \leftarrow \text{Server}_A$: $Encrypt\{SessionKey\}_{rand_{(A,10)}}$
   Client_A derives (decrypts) the server's response with $rand_{(A,10)}$ and get $SessionKey$.

Table 1: A ticket issuer's SOC ticket generation table for server passcode, client passcode and client hint. h is a SHA256 crypto-hash function and *seed* is the issuer's initial seed commonly used for generating common passcodes for all servers for the same service, whereas *rand* is a randomly chosen client seed for generating unique client passcodes for each new client.

passcodes to verify each other's capability. As described in Table 1, each (server and client) passcode is mapped to its unique expiry timeslot, and the ticket issuer generates a chain of passcodes for S tickets by recursively applying each of its secret seed on a crypto-hash function. Each of the generated hash chain is sequentially mapped to expiry timeslots in reversed time order, as shown in Table 1. This table's example simplifies the passcode management to be indended for a single [service, method], and the issuer manages passcodes for 180 timeslots. Whenever the issuer issues a server ticket, it picks (or computes) the particular passcode corresponding to the desired expiry and gives it with the ticket. When the issuer issues a client ticket, the issuer issues the client passcode only for the current timeslot, thus a client ticket is valid only for a single timeslot. While the server's passcode concerns hash chain mechanism, the client's passcode does not concern hash chaining, and it is simply a random number credential valid only for a single timeslot. In addition, the issuer computes client passcode hints (by encrypting the client passcode with its counterpart server passcode corresponding to the current timeslot). The duration of each timeslot is designed by the issuer at setting. If each timeslot is long (e.g. 1 day), the issuer needs not be online to frequently renew client passcodes. If each timeslot is designed to be short (e.g. 1 hour), it's easy to revoke clients responsively, while the issuer can issue the client several future passcodes and hints at once, so that it does not have to renew the ticket every hour.

3.2.1 Server(S) Ticket Structure

Figure 3 and 4 are examples of the data structure of SOC tickets for S type and C type, respectively.

Suppose Table 1's unit timeslot is 1 day. Thus, each client passcode is valid for 1 day, and each server's passcode expiry granularity is 1 day. The mutual authentication steps in Table 1 is similar to how Kerberos client and server authenticates each other. Our approach extends it by adding a dimensionality to the client and server secrets based on one-way hash chain relationships. Even if an attacker hijacks

$\mathbf{S}\langle$HomeLock, *"SmartHomeLock"*, Lock & UnLock, Validator, $\text{Exp}_1\rangle$
{ **Secret Ticket ID:** (32-byte long value)
  Issuer (Fingerprint): HomeLock
  Issuer Seed Era: 15
  Service Name: *"SmartHomeLock"*
  Methods: Lock, Unlock
  Ticket Type: S (Service)
  IsValidator: V (Yes)
  Expiry: 2015.09.15.23:30
  Unit Timeslot Length: 1 minute
}
Passcode {
  Lock   0xc6s27d7b09b89a571bd7
  Unlock   0xbc6s27d9a57bd77b0981
}

Figure 3: An example of a S (server) ticket and its corresponding S passcodes

packets in step 1 to get the client's hint, it is impossible to do a replay attack, unless the attacker holds the server's passcode to decrypt the hint to get the client's passcode. Then, the server sends the client a random session key encrypted by the client's passcode, as a proof that the server succeeded in decrypting the client's hint. Two parties end up with the same session key only if both of their passcodes are authentic. This mechanism allows the server and the client to mutually authenticate, while keeping the server's passcode secret from the client.

3.2.2 Client(C) Ticket Structure

Client tickets have two differences from [Server] tickets. In case of S tickets, if two tickets' (service, method, expiry) parameters exactly match, their passcodes are *partially* shared (see subsection Delegation). In other words, the servers whose S ticket' target [service, method] match and valid for the same (or overlapping) time period share the common secret. On the other hand, the passcode for C tickets is always randomly generated by the issuer even

if the two tickets' (service, method, expiry) parameters are the same.

As described in Table 1, when the issuer creates a C ticket, it randomly chooses a C passcode for the user's each capability target [service, method]. Then, for each passcode, it encrypts each C passcode with its counterpart S (service capability) passcode whose timeslot corresponds to the current time. This encrypted value is the hint for its corresponding C passcode.

Each C ticket contains a client passcode and a hint for each of its target [service, method]. A particular hint can be decrypted only by the server ticket holder who can derive the passcode with the same (service, method, expiry) parameter as the C ticket's hint. This is in resemblance to Kerberos' client ticket verification [7], where each client ticket's secret is encrypted by its target server's secret key. In SOC, a client holding a C ticket knows its ticket's secret, which is its C passcode, and its corresponding hint is an encryption of the C passcode by the server's S passcode. So, the client can challenge the server with its hint to test if the server can decrypt and get its client passcode. In turn, holder of an S ticket can authenticate the holder of the C ticket by verifying if the C ticket holder knows the decrypted value of the provided hint, because the holder of the C ticket cannot know the encrypted value inside the hint unless this information has been provided by the ticket issuer who created it. And the issuer will give this information only to an authorised service user by issuing him a C ticket.

In essence, the SOC's C ticket validation mechanism is an extension of Kerberos' client ticket validation mechanism, because SOC leverages on expiry-timeslot-specific passcodes for S and C tickets for their mutual ticket verification.

C⟨HomeLock, "SmartHomeLock", Lock & Unlock, Validator, Exp_2⟩
{ **Secret Ticket ID:** (32-byte long value)
  Issuer: HomeLock
  Issuer Seed Era: 15
  ServiceName: *"SmartDoorLock"*
  Methods: *Lock, Unlock*
  Ticket Type: C (Client)
  IsValidator: V (Yes)
  Expiry: 2015.05.14.10:00
  Unit Timeslot Length: 1 day
  Hint { *// suppose current time is 2015.09.14.11:10*
    Lock 0x6b85710c2af79bd77d9b
        0x128469cb1adb69af6b3c
    Unlock: 0xfd10c276b859bd79b77a
        0x2403bcb920306340c263
}}
Passcode { *// suppose current time is 2015.09.14.11:10*
    Lock: 0xd77d9b2a571b879b0c63
        0x23bda9b5dacb9a56cb95
    Unlock: 0xfd10c276b859bd79b77a
        0x6abf58af4fabc7a7f6b9
}

Figure 4: An example of a *C* (client) ticket and its corresponding *S* passcodes

Figure 4 is an example of a particular C ticket's data structure. Its format appears almost the same as an S ticket, except that it contains a hint for each passcode. In this example, it has two passcodes, each of which is a C capability proof for [*HomeLock, lock*] and [*HomeLock, unlock*]. Each passcode is accompanied by a hint chain whose length is 2, corresponding to two expiry timeslots

between the ticket's issuing time (2015.05.14.11:10) and its expiry(2015.05.15.23:30). When the holder of this C ticket performs a mutual validation with the holder of its counterpart S ticket, it selects one of two hints from its C ticket, which should correspond to the earlier expiry timeslot between the C ticket and S ticket, and then sends this hint to the S ticket holder to challenge if it can be decrypted using the corresponding S passcode. The decrypted C passcode will be used as part of their session encryption key.

Algorithm 1 describes how a ticket issuer generates S and C passcodes. In the algorithm, $passcode_s$ denotes an S passcode and $passcode_c$ a C passcode.

Algorithm 1 Issuer's S & C Ticket Passcode Generation

1: $SEED$;   ▷ pre-defined secret seed for passcodes
2: $MAX\_TIMESLOT$;   ▷ passcode table's width
3: **function** GET PASSCODE([$service, method$], $expiry$, $ticketType$)
4:    $passcodeList :=$ **new** $ArrayList()$
5:    $n = MAX\_TIMESLOT - Timeslot(expiry)$
6:    $rowIndex := GetTableRow([serviceName, method])$
7:    $passcode_s := h^n(SEED + rowIndex)$
8:    **if** ($ticketType == SERVICE$)
        return $passcode_s$
9:    **else if** ($ticketType == CLIENT$)
10:     $passcode_c = randomNumber()$
11:     $hint = Encrypt(passcode_c)_{passcode_s}$
        return $[passcode_c, hint]$
12: **end function**

3.3 SOC Authentication and Authorization

This subsection extends the simplified server-client handshake as described in Table 1 and gives a comprehensive description of the SOC handshake. A ticket issuer manages a table like Table 1 for all the [service, method] targets, and use them to compose tickets for various [service, method] targets. When multiple capability targets are used for verification between a client and server, they first agree on the identical client passcode for each capability target based on their mutual ticket header declaration, *XOR* all their derived client passcode, and then the server uses it as an encryption key to send a randomly chosen final session key. The purpose of the *XOR* operation is to ensure all passcodes are correct. That is, if ever one of their passcodes is wrong (or randomly guessed), the final *XOR*ed value will not match between the server and client, thus they cannot exchange the same session key. The detail of the protocol is described below.

1. $IoT_{Client} \rightarrow IoT_{Server}$: *"Client_Hello"*

2. $IoT_{Client} \leftarrow IoT_{Server}$: *"Server_Hello"*
                "Current_Time"
                "SOC_Server_Validator_Ticket_Declare"

3. $IoT_{Client} \rightarrow IoT_{Server}$: *"SOC_Client_Validator_Ticket_Declare"*
                "Client_Passcode_Hints"

4. $IoT_{Client} \leftarrow IoT_{Server}$: *$Encrypt_{C\_Passcodes}\{Rand\_Session\_Key\}$*

The above four steps describe the mutual SOC authentication between an (IoT) server and client during their initial connection. The protocol assumes that the server can be a virtual server that has been delegated service capability for
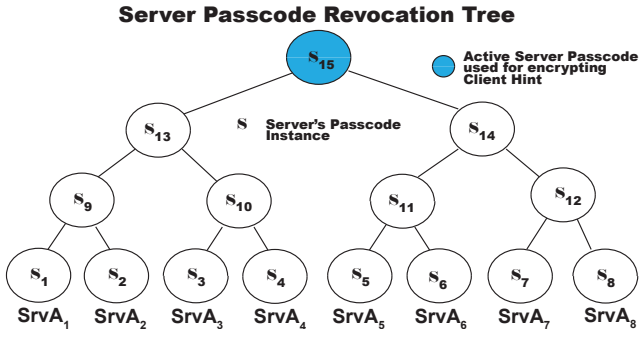
Server Passcode Revocation Tree

Figure 5: An example of S passcode revocation tree for a particular [service, method] target. There is no revoked node, so the issuer encrypts client hints with S_{15} server passcode.

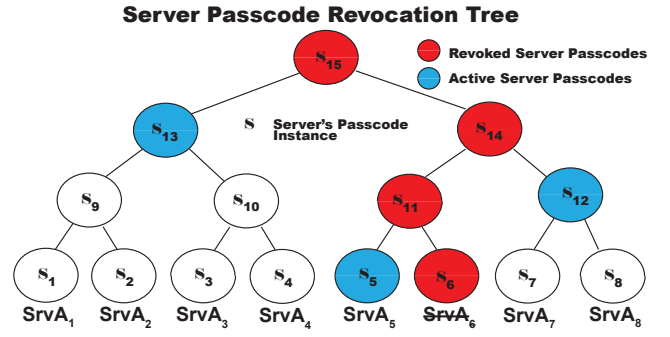**Server Passcode Revocation Tree**

Figure 6: The clone server $SrvA_6$ has been revoked, so all nodes comprising its path to the root are revoked. Accordingly, the issuer encrypts future client hints with S_5, S_{12}, S_{13} server passcodes.

multiple services, thus it holds multiple server tickets. Also the client can hold more than one server ticket.

In step 1, the client sends a *Client Hello* to the server to initiate a connection. In step 2, the server declares the current timeslot to synchronize time between the client and server. We advocate that the issuer should design each timeslot to be large (e.g. 10 hours or 1 day), so that minor time skews between the server and client are not a critical issue. Should their times not agree with each other more than a pre-set threshold, they use the time provided by a publicly well-known clock server as their temporary session time. Then, the server declares the header of its validator-type server tickets for its tickets issued by one or more issuers. There will be multiple services if the service in concern is a virtual service comprising multiple low-level services. The purpose of this header declaration is to require the client to send corresponding hints in order for the server to verify the client's expected capability.

In step 3, for the server's each (serviceName, issuer) header declaration, the client should hold a counterpart client ticket whose service name and issuer matches; Otherwise, the client is not eligible and not authorized to use the service. This client authentication corresponds to the first equation in Equation 3. For all such client tickets, the client picks up and sends the client passcode hints corresponding to the current timeslot. In addition, the client sends the header of all the validator-typed client tickets whose service name exists in the ticket's header declaration, which are for verifying the server's expected service capability. This corresponds to the second equation in Equation 3. Then the client picks up and sends the corresponding client passcode hints corresponding to the current timeslot chosen by the server.

In step 4, if the server has all counterpart server tickets corresponding to what the client has, it should be able to decrypt all hints sent by the client in step 3 and derive the client's passcode in each of them. The server *XOR*s all client passcodes and use it as an encryption key to encrypt a random session key chosen by the server. Then, the server sends it to the client. In order for the client to decrypt it, it also *XOR*s all the client passcodes whose encryptions were sent to the server, use it as a decryption key for the server's reply, and derive the server's chosen random session key. The fact that both parties reach the same session key in the end proves that their expected capabilities are mutually fully authenticated.

3.4 Ticket Delegation

A server ticket holder may delegate its server ticket to another party. When delegating, the holder can optionally remove some of its original ticket's [service, method] capability targets (as well as their corresponding passcodes) in order to weaken the delegated ticket's capability. As previously described in Equation 1, the S ticket delegator can reduce the expiry of the ticket by recursively computing the passcode hashes as many times as needed to move their expiry timeslots to earlier positions. It can also eliminate some service methods (and their corresponding passcodes) in the original ticket to reduce the number of sub-services (i.e. [service, method] targets).

3.5 Ticket Revocation

Although SOC tickets become automatically unusable after their defined expiry, SOC also provides a ticket revocation mechanism which can revoke tickets before their expiry, in case of system compromise. The ticket revocation mechanism is different for the client and server ticket. For client tickets, revocation is straight-forward because each client ticket contains its unique client passcodes. Thus, when the issuer needs to revoke a particular client ticket, it sends the target C ticket's passcodes to its servers offering services concerning the C ticket. Provided that the servers are always running, they can immediately receive client revocation messages from issuer(s) and reflect them, who in turn pass it to their delegate servers. Alternatively, the issuer can store revoked C passcodes in publicly accessible location(s), like managing CRLs. The difference from CRL is that the revoked passcodes need not be signed by the issuer, because they were known only to the issuer and server(s) who encountered them via client hints. Once a C passcode is exposed to the public, it loses its validity, anyway. If the ticket issuer itself is the server, it needs not contact others.

Revoking server tickets is slightly different, because server passcodes are shared among the original server and its delegates. To this end, we leverage on a revocation mechanism that is based on binary tree [8]. Figure 5 is an example of a particular server passcode's revocation tree. The passcode tree is for a particular [service, method] target, and it is initially created by the ticket issuer before issuing any server passcodes to others. The number of leaves is the maximum number of allowed entities who can serve this same [service, method]. When the initial server gets an S ticket from the

issuer, it receives this entire tree. The server assigns itself to one of the leaf nodes, and whenever it delegates its ticket to another entity, it maps the entity to an empty leaf node, and in the ticket to be delegated it inserts a set of S passcode instances occurring along the path from the new leaf to the root. The binary tree in Figure 5 contains exactly four S passcode instances for every path from the leaf to the root. S_0 to S_{15} are randomly generated by the issuer. Each entity who got delegated S tickets will need to try each of these four S passcodes to decrypt client hints.

Meanwhile, the issuer also manages this binary tree just like the initial server. The issuer revokes a particular (delegate) server by revoking all nodes comprising the path from its leaf to the root, like in Figure 6. A tree's *active nodes* are those which are direct children nodes of every revoked node in a tree. In Figure 6, as the issuer revokes Server-A_6, revoked nodes are marked as red and effective active nodes after this revocation are updated and marked as blue. Active nodes get updated after the issuer's each revocation. If no one is revoked, the active node is the root node alone by default. When the issuer issues C tickets, for each C passcode it creates as many hints as the number of active nodes by encrypting C passcode with each of their S passcode instance. In Figure 5, the issuer creates each client's passcode hint with the S passcode instance in the root node (S_{15}). In case of Figure 6, the issuer creates hints with S_5, S_{12}, S_{13}; thus each C passcode comes with three hints. When clients and servers verify each others, the client submits all the three hints, and if the server can decrypt any of the hints, that implies that server has not been revoked. An unrevoked server will be able to decrypt exactly one of three hints and get the C passcode. However, a revoked server won't be able to decrypt any of them, thus the client can safely reject the server.

Delegate servers are to be revoked by the delegator server or issuer. They revoke tree nodes (and their corresponding S passcode instances) the same manner as revoking C passcodes, by storing them in publicly accessible location(s). However, the revoked C passcode is encrypted with the server passcode, so that an attacker cannot know about the revoked C passcode to impersonate the revoked client and use it to to a server who has not updated this C passcode revocation information. Revoked S passcode instances need not be signed by the issuer, provided only the issuer and servers knew these secret passcodes. The randomness of S passcodes provide the equivalent security to the unforgeability of digital signatures.

When the number of IoT service devices dynamically increases, the issuer can increase the height of revocation tree by dynamically adding a new root node. The issuer can encrypt the new root node's server passcode with existing unrevoked server passcodes, and store them in the public repository where revoked tree nodes are stored.

In case some server gets revoked after clients receive their hints, they can determine which of their hints are to be thrown away based on the information of revoked S passcode instances: if any of them successfully decrypts a particular client hint, the client should throw away that hint.

3.6 Revoking Recursively Delegated Tickets

When the initial server delegates his S ticket to others, it can grant the delegate server more than one path in the tree, by granting multiple S passcode instances covering multiple paths, in order to allow further recursive delegation by the delegatee. If the delegator server is to revoke its particular delegatee, it has to revoke all paths given to the delegatee. Likewise, if the issuer is to revoke the initial server, it should revoke the entire revocation tree, because the whole tree was granted to the initial server. This cascading revocation mechanism is similar to cutting the trust chain in X509 certificates.

## 4. EXPERIMENT & SECURITY ANALYSIS

4.1 Implementation and Evaluation

We implemented SOC as a C library of an OpenSSL wrapper. An existing application, regardless of whether it uses a secure connection protocol (e.g. SSL, TLS, DTLS) or not, can transparently adopt SOC passcode security framework and use SOC library by adding one line of function call in their original source code. Calling this function reads in passcode files, generates passcode database in the memory, performs passcode negotiation and ticket validation, and computes the server and client's final secure session key. For those applications using SSL, TLS or DTLS, the SOC's final session key is arithmetically added to their original session key in a transparent manner. As for other applications not using OpenSSL's SSL/TLS/DTLS protocols, they need to call an additional function call around each system call sending/receiving packets, but such source code modification can be automatically patched given its original source code. In this case, data buffer is encrypted or decrypted with the final passcode before being sent to or received from the network layer.

We tested the validation time and its total handshake packet overhead, and compared them against the performance of other security mechanisms based on ECC-PKI and RSA-PKI which conduct the same security validation between the server and client. The experiment was conducted between two servers, physically located in Singapore and San Francisco, respectively. Their CPUs were 32-bit Dell 2.0GHz single core, and memory size was and 512MB. We ran the experiment in two phases. First, we measured the validation time and handshake packet overhead of three cryptographic algorithms for different numbers of co-existing dynamic services. Second, we measured their performance by varying the average number of S ticket delegations, which is equivalent to the average service virtualization level.

When we varied the number of co-existing dynamic services with service ticket delegation (i.e. service virtualization) being disallowed, three cryptographic validation mechanisms showed linear increases in their validation time. As shown in Figure 7a, SOC produces a shorter validation time than RSA-PKI or ECC-PKI algorithms because its cryptographic computation does not use public key encryption. As illustrated in Figure 7d, while their total packet size shows a negligible difference, SOC always shows a slightly larger packet size than RSA-PKI or ECC-PKI, because SOC requires exchanging hints during its handshake, which can be several in their numbers per ticket issued by the same issuer, while RSA-PKI and ECC-PKI requires only exchanging a single digital signature per certificate issued by the same issuer. Yet, the size of a signature or a hint is very small (32 bytes for SHA256), thus the packet size difference between them is hardly distinguishable.
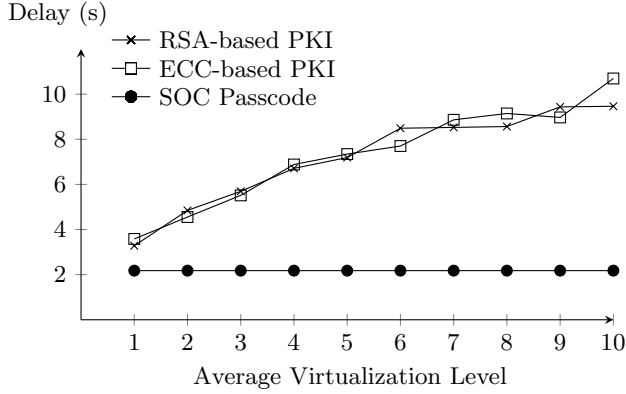
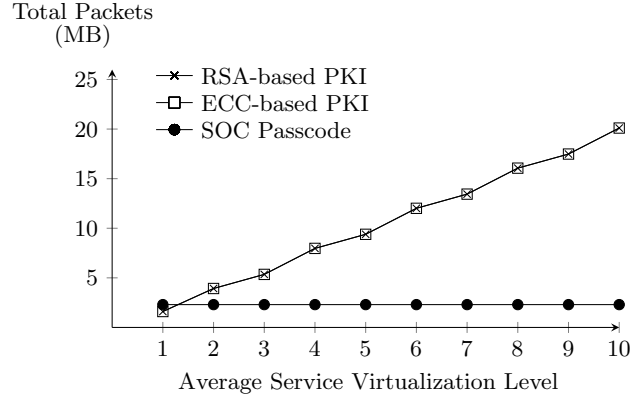SOC's major performance boost occurs when the recursive
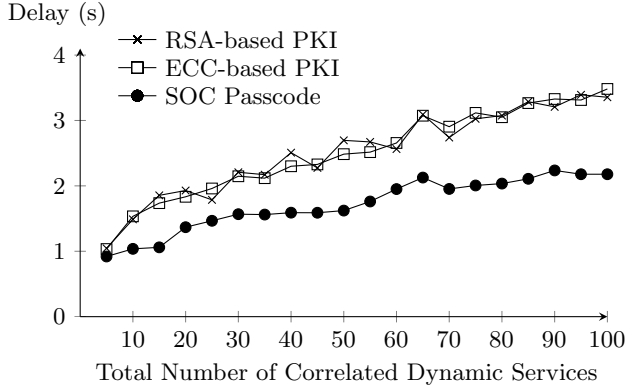
(a) Validation time comparison by varying the number of co-existing dynamic services.)
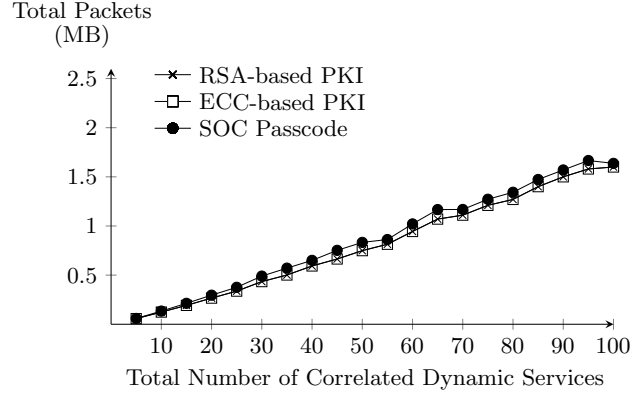
(b) Handshake packet overhead comparison by varying the number of co-existing dynamic services.)

(c) Validation time comparison for validating recursively virtualized services across multiple servers.)

(d) Handshake packet overhead comparison for validating recursively virtualized services across multiple servers.)

service virtualization is enabled. We fixed the number of co-existing (i.e. correlated) dynamic IoT services to 100, and gave variation on the average number of recursive S ticket delegations (i.e. average service virtualization level). For each service ticket delegation, ECC-PKI or RSA-PKI has to create a new certificate and add it to the certificate chain, which is imperative because of our premise that a ticket delegator should be capable of optionally weakening its service capability to be delegated. Meanwhile, SOC passcode scheme's validation time delay stays the same regardless of the service virtualization level, because its ticket's security does not depend on digital signatures, but depends on the secrecy of C/S passcodes in each ticket. Therefore, SOC's each service capability delegation is equivalent to simply handing over the ticket's secret passcodes to the delegatees. Thus, the size of each delegated ticket either stays the same or rather decreases, because reducing a ticket's capability is equivalent to removing some of the existing passcodes or hint buckets in the original ticket. As Figure 7c and 7d show, SOC passcode's validation time and handshake packet overhead stays constant, while RSA-PKI or ECC-PKI algorithms incur linearly increasing validation time and packet overhead.

4.2 Security Analysis

Table 2 compares the SOC's security features with other equivalent mechanisms, Kerberos and X509 PKI, in the context of a server's service capability delegation (i.e. service

Security Features	Kerberos	X509	SOC
Service Capability Delegation	x	o	**O**
Multiple Independent Ticket Issuers	x	o	**O**
Client&Server Revocation	o	o	**O**
Constant Authentication Overhead Regardless of Virtualization Level	-	x	**O**
Control on Further Virtualization	-	o	**O**
Decentralized Ticket Management	x	o	**O**

Table 2: Security feature comparison of SOC with Kerberos and X509 in regards to service virtualization

virtualization). Kerberos does not support service capability delegation, while SOC supports it by service passcode trees and X509 PKI possibly by certificate chaining. X509 and SOC allow tickets (or certificates) to be issued and managed by independent issuers in a decentralized manner. On the other hand, Kerberos relies on a centralized ticket server for managing all tickets, and it does not support multi-perspective authentication requiring approvals from multiple independent authorities (i.e. issuers). As for authentication overhead, X509 PKI incurs extra length in its certificate chain for every subsequent service delegation (i.e. service virtualization), so its authentication overhead between the server and client also grows linearly with the increasing number of service virtualization level- this is because the client has to verify the server's extra certificate chains. On the other hand, SOC's authentication overhead stays constant (see Subsection 4.1) irrespective of service virtualization lev-

els. Both SOC and X509 allows servers to restrict on how many more sub-delegations their delegatees can perform, in order to hold service virtualization under control.

5. RELATED WORK

While previous work [9] [10] [11] have focused on incorporating IoTs into clouds, we focus on decentralized IoT security due to its unique advantages: decentralized IoTs are managed in a decentralized manner, so IoT device owners can have less worry about their data leakage, because they manage their own data within their domain, and can have more freedom and flexibility in their IoT service capability. If our data get stored in the cloud, we never know what the cloud providers will do with our data. Furthermore, data traffic being exchanged between the clients and IoTs can be possibly monitored by the cloud, and can be filtered or altered by its implicit policies without the user's notice.

Birgisson et al [12] proposes capability cookies, called Macaroon, that can be delegated on the internet, and whose condition can be set as stricter by the delegator. While Macaroons are strictly for server use, SOC provides separate tickets for both client and server, and they can mutually validate each other by using their tickets.

In [13], a smart vehicle network is secured by appointing a central security gateway, such that all incoming data packets are inspected and sanitized before being delivered to any vehicle nodes. Our solution differs in that we do not shift the whole security management to a central gateway, but allow individual IoT devices to be ticket issuers for their service objects and configure their own security level for each of their resources.

Virtualization of IoT nodes was introduced by Zhang et al [1]. They developed a virtual sensor editor tool which creates a virtual representation of sensors (on a PC), based on data streams imported from remote physical sensors. This research assumed only a local network, and thus security was not taken into account. We develop IoT virtualization to be security-aware by using SOC passcode mechanism.

There has been a great deal of research on access control (authentication and authorisation). [14] enforces attribute-based access control for web services, whereas [15] uses role-based access control for financial web systems. The closest work to our research is [16] which realizes security control in a decentralized network by means of a trusted entity, and client certificates issued by it, to grant access to each IoT resource. Our work extends their mechanisms because our SOC can offer the feature provided by their work, plus allows ticket delegation for both clients and servers.

[17] keeps a service device's data secret when transferred to a cloud, by encrypting them before storing in the cloud. Each data field is encrypted using a different key, which is shared only among intended clients or application servers. While this preserves the confidentiality of each data field by relying on multiple encryption keys, our security mechanism preserves the security of each service object based on multiple SOC tickets along with dynamic passcodes associated with each of them.

There is much research on securing ad hoc networks in a decentralized manner. [18] and [19] proposes revocation algorithms for bad nodes in an ad hoc network, based on votes from neighbouring nodes. The core idea is to pre-define different keys for each time-frame of future, secure, end-to-end connections between every device, and split them into mul-

tiple parts by using a multivariate, secret-sharing algorithm, keeping each part in a separate IoT device. While this revocation process is managed completely by the collaboration of individual ad hoc nodes, SOC enables revocation by using a revocation service who offers periodic heartbeat passcodes to ticket holders upon their request.

[20] introduces a security mechanism to keep a user's password the same even after being compromised. It associates the user's password with a unique public key stored in the user's particular device, and user authentication requires not only the user password, but also using the valid public key. When the password is compromised, the user only discard the public key in the compromised device. Our passcode revocation mechanism doesn't use PKI, but use symmetric and hash algorithms, thus it is computationally more efficient.

6. CONCLUSION

This paper proposes SOC scheme that allows an IoT server and client to securely and efficiently authenticate each other's capability especially for virtual IoT services. Unlike Kerberos built around dedicated central authority server (i.e. KDC), SOC facilitates decentralized access control where independent ticket issuers can participate in and the server ticket holder can delegate his ticket to facilitate service virtualization. The service ticket holders can monotonously weaken the capability to be delegated in a decentralized manner. This mechanism does not depend on public key cryptography, and the authentication time and packet overhead stays constant regardless of the number of delegation hops (i.e. service virtualization level) from root delegator. Our experimental results indicate SOC's authentication time is approximately 0.21 of RSA-PKI and ECC-PKI algorithms, and its authentication packet overhead is approximately 0.087 of RSA-PKI and ECC-PKI.

Although SOC's symmetric encryption algorithm is faster than RSA-PKI or ECC-PKI's public key encryptions, in our experiment this cryptographic computational advantage was partially overshadowed by the network traffic. We believe SOC's benefit of computational efficiency will become more conspicuous as more resource-constrained IoT devices than ones in our experiment are used in various scenarios. Disregarding this factor, SOC is still evaluated to be more efficient than PKIs for highly dynamic environments with lots of (virtual) IoT services interoperating with each other.

In our future work, we will address how to efficiently determine the capability of each IoT service node and detect failure or compromise of any particular nodes in real time while the IoT scales up.

7. REFERENCES

[1] J. Zhang, Z. Li, O. Sandoval, N. Xin, Y. Ren, R. Martin, B. Iannucci, M. Griss, S. Rosenberg, and A. Cao, J. an Rowe, "Supporting Personizable Virtual Internet of Things," in *Proc 10th International Conference on Ubiquitous Intelligence and Computing, and Autonomic and Trusted Computing (UIC/ATC)*, pp. 329–336, IEEE, 2013.

[2] S. Alam, M. Chowdhury, and J. Noll, "Senaas: An event-driven sensor virtualization approach for internet of things cloud," in *Networked Embedded Systems for Enterprise Applications (NESEA), 2010 IEEE International Conference on*, pp. 1–6, Nov 2010.

[3] A. Mahmud, R. Rahmani, and T. Kanter, "Deployment of flow-sensors in internet of things' virtualization via openflow," in *Mobile, Ubiquitous, and Intelligent Computing (MUSIC), 2012 Third FTRA International Conference on*, pp. 195–200, June 2012.

[4] H. Ning, H. Liu, and L. Yang, "Cyberentity Security in the Internet of Things," *Computer*, no. 4, pp. 46–53, 2013.

[5] Y. chun Hu, M. Jakobsson, and A. Perrig, "Efficient constructions for one-way hash chains," in *In Applied Cryptography and Network Security (ACNS*, pp. 423–441, 2003.

[6] R. Want, "Enabling ubiquitous sensing with rfid," *Computer*, vol. 37, pp. 84–86, April 2004.

[7] B. Neuman and T. Ts'o, "Kerberos: an authentication service for computer networks," *Communications Magazine, IEEE*, vol. 32, pp. 33–38, Sept 1994.

[8] W. Aiello, S. Lodha, and R. Ostrovsky, "Fast digital identity revocation (extended abstract," in *in 18th Annual International Cryptology Conference (CRYPTO98*, pp. 137–152, Springer-Verlag, 1998.

[9] M. Henze, L. Hermerschmidt, D. Kerpen, R. Haussling, B. Rumpe, and K. Wehrle, "User-Driven Privacy Enforcement for Cloud-Based Services in the Internet of Things," in *Future Internet of Things and Cloud (FiCloud), International Conference on*, pp. 191–196, Aug 2014.

[10] K. Misura and M. Zagar, "Internet of things cloud mediator platform," in *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2014 37th International Convention on*, pp. 1052–1056, May 2014.

[11] L. Kai, S. GuoQin, and H. Xiangli, "Research on internet of things' support for ipv6 addressing strategy under the platform of cloud computing," in *Computational and Information Sciences (ICCIS), 2013 Fifth International Conference on*, pp. 1361–1364, June 2013.

[12] A. Birgisson, J. G. Politz, ÃŽlfar Erlingsson, A. Taly, M. Vrable, and M. Lentczner, "Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud," in *Network and Distributed System Security Symposium*, 2014.

[13] T. Zhang, "Defending Connected Vehicles Against Malware: Challenges and a Solution Framework," *Internet of Things Journal*, vol. 1, no. 1, pp. 15–20, 2014.

[14] S. Hai-Bo, "A semantic- and attribute-based framework for web services access control," in *Intelligent Systems and Applications (ISA), 2010 2nd International Workshop on*, pp. 1–4, 2010.

[15] Z. Wen, B. Zhou, and D. Wu, "Three-Layers Role-Based Access Control Framework in Large Financial Web Systems," in *International Conference on Computational Intelligence and Software Engineering, (CiSE)*, pp. 1–4, 2009.

[16] A. Skarmeta, J. Herna?ndez-Ramos, and M. Moreno, "A decentralized approach for security and privacy challenges in the Internet of Things," in *Internet of Things (WF-IoT)*, pp. 67–72, IEEE, 2014.

[17] M. Henze, S. Bereda, R. Hummen, and K. Wehrle, "SCSlib: Transparently Accessing Protected Sensor Data in the Cloud," in *The 5th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN-2014)*, p. 370??375, ScienceDirect, 2014.

[18] T. Moore, J. Clulow, S. Nagaraja, and R. Anderson, "New strategies for revocation in ad-hoc networks," in *Proc. 4th European Conference on Security and Privacy in Ad-hoc and Sensor Networks*, ESAS, pp. 232–246, Springer, 2007.

[19] H. Chan, V. D. Gligor, A. Perrig, and G. Muralidharan, "On the distribution and revocation of cryptographic keys in sensor networks," *IEEE Trans. Dependable Secur. Comput.*, vol. 2, no. 3, pp. 233–247, 2005.

[20] A. R. B. Daniel R Thomas, "Better authentication: password revolution by evolution," in *Security Protocols XXII*, pp. 130–145, Springer, 2014.