KESIC: Kerberos Extensions for Smart, IoT and CPS Devices

Renascence Tarafder Prapty
University of California Irvine
rprapty@uci.edu

Sashidhar Jakkamsetti
Bosch Research
sashidhar.jakkamsetti@us.bosch.com

Gene Tsudik
University of California Irvine
gts@ics.uci.edu

***Abstract*—Secure and efficient multi-user access mechanisms are increasingly important for the growing number of Internet of Things (IoT) devices being used today.**

Kerberos is a well-known and time-tried security authentication and access control system for distributed systems wherein many users securely access various distributed services. Traditionally, these services are software applications or devices, such as printers. However, Kerberos is not directly suitable for IoT devices due to its relatively heavy-weight protocols and the resource-constrained nature of the devices.

This paper presents KESIC, a system that enables efficient and secure multi-user access for IoT devices. KESIC aims to facilitate mutual authentication of IoT devices and users via Kerberos without modifying the latter's protocols. To facilitate that, KESIC includes a special Kerberized service, called IoT Server, that manages access to IoT devices. KESIC presents two protocols for secure and comprehensive multi-user access system for two types of IoT devices: general and severely power constrained. In terms of performance, KESIC consumes $\approx$ 47 times less memory, and incurs $\approx$ 135 times lower run-time overhead than Kerberos.

I. Introduction

Internet-of-Things (IoT) and Cyber-Physical Systems (CPS) encompass embedded devices with sensors, actuators, control units, and network connectivity. They are widely adopted in both private and public spaces. By 2030, the number of IoT devices is projected to surpass 29 billion [1].

Except for personal devices, such as wearables, most IoT devices benefit from multi-user access. For instance, multiple family members can manage a smart home system comprised of smart TVs, voice assistants, and smart appliances. Similarly, in a shared office setting, many employees operate IoT devices, (e.g., smart projectors, meeting room schedulers, and interactive whiteboards). Maintenance crews or engineers often monitor IoT devices used for automation and process control in industrial settings, such as factories or warehouses. Multiple users being able to control devices provides convenience and facilitates coordination.

On another note, IoT devices usually have limited or no security features due to their resource constraint nature. This makes them vulnerable to different attacks, e.g., Mirai Botnet [2], Triton [3]. Therefore, it is crucial for users to be able to ensure the integrity of an IoT device before using it

With respect to multi-user support, one intuitive idea is to require each user to individually register with each device and establish a security context, e.g., by sharing a unique symmetric key. However, this requires linear amount of storage on the device, which is impractical due to resource constraints.

To this end, some prior work introduced the notion of communication proxies [4]. Such proxies run a lightweight protocol between themselves and devices, acting as intermediaries when devices communicate with users/clients. One drawback is the presence of an additional intermediate hop for every user request. Another prior effort [5] proposed extending Kerberos to IoT devices, which involves significant changes to the Kerberos protocols. Specifically, additional message exchanges are needed between the device and modified Kerberos servers to authenticate service tickets, thus significantly impacting device runtime performance and availability. Furthermore, aforementioned approaches do not inform the user about software integrity of IoT devices, i.e., users do not learn whether a given device is healthy or compromised.

Motivated by aforementioned issues, this work revisits using Kerberos for IoT devices to enable multi-user support. We present KESIC: Kerberos Extensions for Smart, IoT and CPS Devices – a design that requires no changes to Kerberos and includes attestation of the device's software state as a built-in service. We prefer Kerberos to other multi-user authentication schemes (e.g. OAuth, or OpenID) because it is specifically designed for accessing both hardware and software resources within a network. The original concept of Kerberos closely aligns with the idea of a network of IoT devices, such as a smart home/office or an factory, making it an efficient and easily adaptable protocol for IoT devices.

Configuring and using devices directly as Kerberos application services is impractical due to the lack of required hardware features: many (especially lower-end) IoT devices do not have real-time and/or secure clocks necessary for verifying Kerberos tickets, or sufficient memory to host the entire Kerberos library. More generally, storage, memory, runtime, and network overhead incurred by Kerberos is also significant for targeted devices (see Section VI-D).

Thus, instead of modifying Kerberos, KESIC uses an IoT Server (ISV) as a Kerberos service, that manages access to all constituent IoT devices, while relying on Kerberos for user authentication. After initial Kerberos login, a user requests a service ticket for ISV from the Kerberos Ticket Granting Service (TGS). Next, the user asks ISV to grant specific (IoT) tickets to access the desired IoT device. As part of this process, the user can request an attestation report for the IoT device in order to verify the latter's software integrity before actually using it.

KESIC partitions IoT devices into two groups: general and

power constrained (see Section III-A2). This grouping is based on the power consumption, hardware resources, and activity time of the devices.

KESIC considers three types of communications in the proposed ecosystem: user $\leftrightarrow$ ISV, ISV $\leftrightarrow$ IoT device, and user $\leftrightarrow$ IoT device. KESIC includes a protocol for each device category, covering all three interactions, resulting in two protocols. KESIC does not require real-time clocks on devices: at each boot/wake-up time, devices obtain current time from ISV. Afterward, devices equipped with a timer use it to emulate a local clock. For devices that are more power constrained and have no timers, we use a nonce-based approach.

Also, in terms of cryptography, KESIC exclusively uses keyed hash (HMAC) operations in all of its protocols, instead of encryption and decryption operations, which can be slower and incur higher storage and run-time memory footprints. Based on our proof-of-concept implementation, KESIC incurs 47 times lower memory overhead and 135 times lower run-time overhead than Kerberos.

Expected contributions of this work are:

- Design of two lightweight and secure protocols for multi-user access to IoT devices.
- Open-source implementation of KESIC [6] which includes: (i) a prototype ISV integrated with Kerberos as an application service, (ii) two prototypes for IoT devices based on ARM Cortex-M33 that uses TrustZone-M for implementing an attestation RoT, and (iii) a client application that uses ISV to obtain IoT tickets, and then requests access to IoT devices.

II. Background

This section overviews Kerberos and Root-of-Trust concepts. It can be skipped with no loss of continuity.

A. Kerberos

Kerberos is an authentication, authorization, and access control (AAA) system for distributed systems that originated at MIT Project Athina in mid-1980s. It allows users to authenticate themselves once via username/password (via Single Sign-On aka SSO or login) for all services that they are allowed to use. Once a user logs in, no further human interaction is required for the duration of the login session. All other protocols used to access services are transparent to the (human) user. There are three types of entities in Kerberos:

Client or Principal (C): This is usually represented by a software component called *kinit* which resides in the user's workstation. It manages Kerberos tickets for users.

Service Provider or Application Server (V): This software manages resources or services, e.g., graphic software or printer, and grants access to them by handling Kerberos service tickets.

Key Distribution Center (KDC): The central third party trusted by both clients and application servers. KDC maintains a database to store all user passwords and long-term keys for application servers. It also facilitates mutual authentication among clients and servers. KDC can also implement granular access control. It has two main components:

- **Authentication Server (AS)** authenticates clients and issues Ticket Granting Tickets (TGT).
- **Ticket Granting Server (TGS)** verifies TGTs and issues service ticket (T_V) to access V.

The database in KDC contains a long-term key for TGS, which is also available to AS.

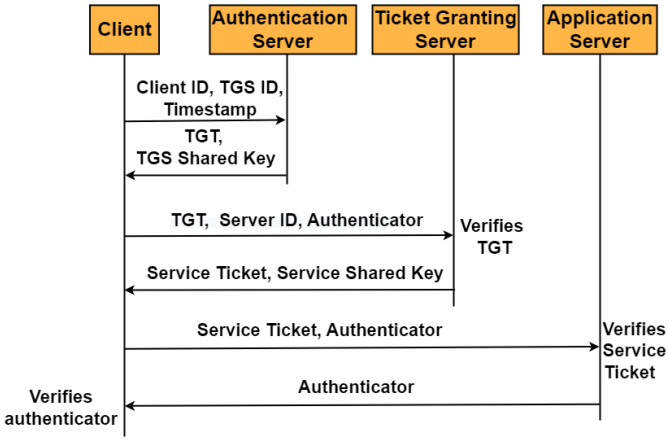

Fig. 1: Kerberos Steps.

Figure 1 shows an overview of Kerberos functionality.

At the beginning of the Kerberos authentication process, C is authenticated by AS and provided a TGT for the next steps. After receiving TGT, C can call TGS using the TGT. The purpose of this call to TGS is to request Kerberos tickets for different services. Afterward, when a user requires a service, C calls the application server V with the corresponding Kerberos ticket. We do not provide the full details of the verification processes for tickets and authenticators in different steps due to space constraints. If interested, the details of the protocol are widely available in various textbooks such as [7].

B. Root-of-Trust (RoT)

In KESIC, a RoT on an IoT device is needed for secure storage and secure computation. We need to ensure that shared long-term secret keys are not revealed and other authentication related metadata (such as current timestamp obtained from ISV) are not modified by potentially present malware. During attestation, we also need to securely compute HMAC of specified memory region without interference from any possibly present malware.

KESIC can be applied to three types of devices:

- Devices equipped with verified, secure, hybrid (SW/HW) RoTs, such as SANCUS [8], PISTIS [9], VRASED [10], [11], or RATA [12].
- Off-the-shelf devices with hardware RoTs, such as ARM TrustZone[13], Intel SGX [14], or AMD SEV [15].
- Legacy devices without any hardware RoT. In this case, there are two options: (i) rely on verified RoTs [16] based on trustworthy microkernels [17], or (ii) however aspirational this might be, consider the OS to be trusted.

It is to be noted that having hybrid or hardware RoT is not a prerequisite for using KESIC in IoT devices. Even though our proof of concept implementation uses TrustZone-M as RoT, KESIC is equally applicable to IoT devices where OS is trusted.

Remote Attestation ($\mathcal{RA}$) is a security service that allows a trusted client (aka, verifier or $\mathcal{V}$rf) to measure software integrity on a remote device (aka, prover or $\mathcal{P}$rv). $\mathcal{RA}$ is a challenge-response protocol, usually realized as follows:

- $\mathcal{V}$rf sends an $\mathcal{RA}$ request with a challenge ($\mathcal{C}$hal) to $\mathcal{P}$rv.
- $\mathcal{P}$rv receives the request, computes an authenticated integrity check over its program memory region and $\mathcal{C}$hal, and returns the result to $\mathcal{V}$rf.
- $\mathcal{V}$rf checks the result and determines whether $\mathcal{P}$rv is compromised.

The integrity check is computed via either a Message Authentication Code (e.g., HMAC) or a digital signature (e.g., ECDSA) over $\mathcal{P}$rv program memory. The former requires a long-term symmetric key shared between $\mathcal{P}$rv and $\mathcal{V}$rf. For the latter, $\mathcal{P}$rv must have a private key that corresponds to a public key known to $\mathcal{V}$rf. Both approaches require secure key storage on $\mathcal{P}$rv.

III. Design Overview

As mentioned earlier, Kerberos in its regular incarnation is unsuitable for low-end IoT devices for several reasons: (1) most IoT devices do not have real-time clocks and cannot verify timestamps on tickets, (2) storage and memory of IoT devices are limited and can not accommodate the Kerberos library (see Section VI-D for details), (3) since Kerberos tickets are encrypted, an expired ticket can not be detected until it is decrypted, which is time and resource-consuming, especially, for mission-critical devices. In fact, this could be abused as a means of DoS attacks.

Therefore, we opt to extend Kerberos – without modifying it – to support low-end IoT devices. Additionally, since IoT devices are increasingly subject to malware attacks, we want to provide attestation of device software to assure the user that the device is not compromised prior to its use.

A. System Model

Figure 2 overviews KESIC system model. As described before in Section II-A, Client (C) is a software component running in the human user's device (i.e. workstation or cellphone). We use the terms user and client interchangeably to refer to the same entity.

1) IoT Server (ISV): We introduce a special Kerberos application service called IoT Server – ISV. After initial log-in, C obtains a regular Kerberos ticket for ISV from TGS. ISV is responsible for granting users access to IoT devices and managing authenticated communication between users and IoT devices. ISV has two main components:

Ticket Manager (TM) is responsible for authenticating users (clients) and issuing IoT tickets to access devices. Before granting an IoT ticket to a user, TM uses access control policies set up by the device owner to ensure that the user is permitted to access that device. While important, access control policies are out of the scope of this paper and are not discussed further. KESIC uses HMACs instead of encryption to generate tickets and authenticators. TM shares two long-term secret keys with each IoT device: one to generate IoT tickets for that device, and the other – to generate the corresponding session key.

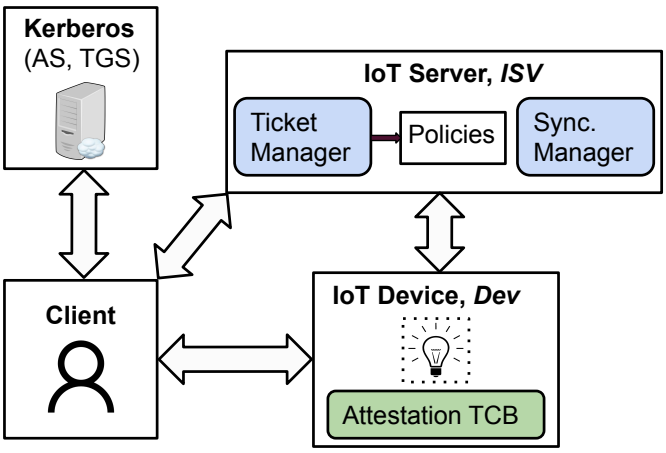

Fig. 2: System Model.

Synchronization Manager (SM) is responsible for time synchronization between an IoT device and ISV, which determines the validity (i.e., freshness) of IoT tickets. SM maintains a distinct long-term key with each IoT device in order to secure the synchronization process. Synchronization details vary depending on the type of the device; this is discussed in Section IV. The Synchronization phase is crucial since the device cannot enter the service phase without successfully completing it.

Indeed, ISV represents a single point of failure: if it is down, devices become non-operational. This risk can be mitigated by deploying multiple ISV instances.

2) Device Types: KESIC supports two types of IoT devices:

General Devices (Dev_g) are always awake; they have a direct power source or a long-lasting battery. They can receive requests over the network at any time. Examples of such devices are Blink Security Camera [18], Google Nest Thermostat [19], and Lumiman Smart Bulb [20].

Power Constrained Devices (Dev_{pc}) spend most of their time in a low-power sleep state due to stringent energy constraints. Periodically, they wake up, perform brief tasks, and return to sleep. Examples of such devices are: ThermoPro TP357 Digital Hygrometer Indoor Thermometer [21], Netatmo Weather Station [22], and Nordic nRF9160 system-in-package (SiP) [23].

Both Dev_g and Dev_{pc} devices are assumed to be equipped with either a hardware RoT or a trusted OS (in case of legacy devices). For the sake of simplicity, from now on, we refer to both of them as RoT.

3) Protocol Overview: KESIC has three run-time phases:

In **Ticket Issuing Phase**, C obtains a service ticket (T_{ISV}) for ISV after AS & TGS log-in. Next, C obtains IoT tickets

TABLE I: Notation Summary.

Notation	Description
ID_x	Identity of entity x
AD_c	Identity of the Client Interface
Dev_g	General Device
Dev_{pc}	Power Constrained Device
$TS_{x \to y}$	Timestamp sent from x to y
L_n	Timestamp when the ticket will expire
$Req_{x \to y}$	Request from x to y
$Res_{x \to y}$	Response from x to y
$K_{x \leftrightarrow y}$	Long-term key between x and y
$k_{x \leftrightarrow y}$	Session key between x and y
T_x	Ticket for entity x
$A_{x \to y}$	Authenticator sent from x to y
$A^{attest}_{x \to y}$	Authenticator sent from ISV to Dev_{pc} and vice versa as part of attestation request/response
$Req^{attest}_{ISV \to Dev_{pc}}$	Attestation Request from ISV to Dev_{pc}
$Res^{attest}_{Dev_{pc} \to ISV}$	Attestation Response from ISV to Dev_{pc}
$Co_{Dev_{pc}}$	Counter maintained by ISV for Dev_{pc}, used to issue tickets
Co_{sync}	Synchronized counter between ISV and Dev_g/Dev_{pc}, stores the number of synchronization requests

from ISV. IoT tickets issued for Dev_g and Dev_{pc} have different formats.

In **Synchronization Phase**, which happens upon each boot, a device communicates with ISV to obtain the latter's current timestamp, which serves as the synchronization value for the current session. After that, a device uses a local timer (Dev_g) or a counter array (Dev_{pc}) to emulate a clock.

In **Service Phase**, an IoT device accepts IoT tickets and service requests from clients. Dev_g uses the synchronized local clock and Dev_{pc} uses its synchronized local counter array to determine ticket validity.

Section IV presents protocols covering each phase for both device classes.

B. Adversary model

KESIC comprises multiple clients, application services, and IoT devices. It also includes trusted third parties: Authentication Server (AS), Ticket Granting Server (TGS), and IoT Server (ISV). We assume an adversary $\mathcal{A}$dv that can remotely attack an IoT device and compromise its software. However, $\mathcal{A}$dv cannot attack any software and data inside the device RoT. Furthermore, $\mathcal{A}$dv can compromise clients with the intent of:

- **Impersonate Clients** in order to circumvent access policy regarding a service/IoT device.
- **Eavesdrop** on other clients' message exchanges to obtain confidential information, such as passwords and keys.
- **Tamper** with other clients' message exchanges to gain unauthorized access.
- **Replay** other clients' message exchanges to gain unauthorized access.

We do not consider denial-of-service (DoS) attacks whereby $\mathcal{A}$dv floods the device with fake requests (tickets). Techniques such as [24], [25], [26] can mitigate these attacks. We also do not consider physical attacks whereby $\mathcal{A}$dv physically tampers with devices via inducing hardware faults or modifying code in RoT. We refer to [27], [28] for an overview of countermeasures against such attacks. Finally, we also do not consider side-channel attacks, similar to the Kerberos threat model.

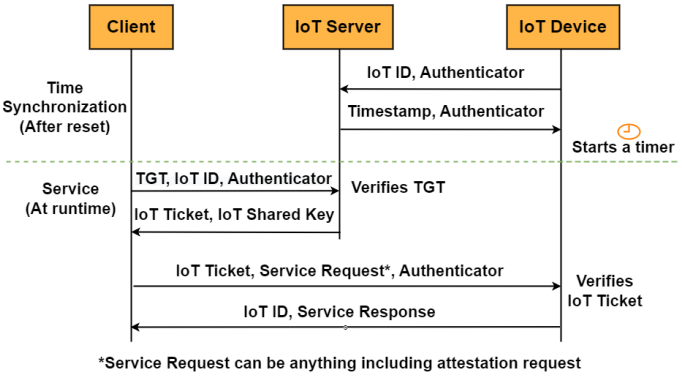

Fig. 3: $\boldsymbol{Dev_g}$ Protocol Overview.

IV. KESIC Protocols

Recall that ISV is treated as any other application server. The steps to obtain a ticket for ISV are the same as obtaining a Kerberos ticket. Hence, we focus on the steps starting from ISV issuing a ticket to a client for IoT device access.

ISV maintains a database with each device's ID, device type, long-term secret key(s), access control policy, and current synchronization value (in case of Dev_{pc}).

Protocol notation is summarized in Table I. One term that needs further clarification is Co_{sync}: it is the counter synchronized between ISV and the device, which keeps track of the number of synchronization requests sent by the device so far. It is used by ISV to authenticate the device during the synchronization phase. Therefore, the device needs to store Co_{sync} in non-volatile (persistent) memory. To simplify protocol description, we use 'K' to denote all long-term secret keys. However, in reality, three distinct long-term keys are shared between ISV and each device. One key is used for ticket generation and verification, another – for session key generation, and the third – for synchronization.

A. General Device Protocol

Figure 3 provides an overview of KESIC protocol for Dev_g, while detailed description is presented in Figure 4.

Time Synchronization Phase: Time Synchronization Phase of Protocol 1 approximates wall-clock time by using a timer. After reboot, Dev_g obtains the current timestamp from ISV, stores it as its start_time, and starts a timer.

Upon receiving a synchronization request $Req_{Dev_g \to ISV}$, ISV verifies it. In step 3, ISV verifies Co_{sync} by checking that it either equals local Co_{sync} for Dev_g or exceeds it by 1. Co_{sync} is included in the authenticator $A_{Dev_g \to ISV}$ in order to prevent replayed synchronization requests. Ideally, received Co_{sync} should be greater than the local version by 1. However, it might be equal to the local version due to lost response and subsequent re-transmission. ISV also verifies authenticator $A_{Dev_g \to ISV}$. If both Co_{sync} and $A_{Dev_g \to ISV}$ are valid, then Dev_g is considered authenticated and ISV replies with a synchronization response – $Res_{ISV \to Dev_g}$.

Protocol 1: General Device Protocol

Time Synchronization Phase ($Dev_g \leftrightarrow$ ISV)

1) After booting up Dev_g increases persistent sync counter Co_{sync} value by 1 and computes authenticator $A_{Dev_g \rightarrow ISV}$:

$$HMAC(K_{ISV \leftrightarrow Dev_g}, [ID_{Dev_g} || Co_{sync}]) \quad (1)$$

2) Then Dev_g builds $Req_{Dev_g \rightarrow ISV}$ and sends it to ISV: $Req_{Dev_g \rightarrow ISV} = ID_{Dev_g} || Co_{sync} || A_{Dev_g \rightarrow ISV}$
3) ISV verifies Co_{sync}. It also verifies $A_{Dev_g \rightarrow ISV}$ using Equation 1. Upon successful verification, ISV updates local Co_{sync} with received Co_{sync}.
4) Then ISV computes authenticator $A_{ISV \rightarrow Dev_g}$:

$$HMAC(K_{ISV \leftrightarrow Dev_g}, [ID_{ISV} || Co_{sync} || TS_{ISV \rightarrow Dev_g}]) \quad (2)$$

5) Finally ISV builds $Res_{ISV \rightarrow Dev_g}$ and sends it to Dev_g: $ID_{ISV} || Co_{sync} || TS_{ISV \rightarrow Dev_g} || A_{ISV \rightarrow Dev_g}$
6) Dev_g verifies $A_{ISV \rightarrow Dev_g}$ using Equation 2. Then it updates start_time with received $TS_{ISV \rightarrow Dev_g}$ and starts a timer.

Ticket Issuing Phase (C $\leftrightarrow$ ISV)

1) C performs the steps of Kerberos protocol to obtain Kerberos service ticket, T_{ISV} and shared key $k_{C \leftrightarrow ISV}$ for ISV.
2) Computes $A_{C \rightarrow ISV} = E(k_{C \leftrightarrow ISV}, [ID_C || AD_c || TS_{C \rightarrow ISV}])$
3) Builds ticket request $Req_{C \rightarrow ISV}$ and sends it to ISV: $Req_{C \rightarrow ISV} = ID_{Dev_g} || T_{ISV} || A_{C \rightarrow ISV}$
4) ISV decrypts T_{ISV} by using $K_{ISV \leftrightarrow TGS}$ and verifies it. Then ISV issues IoT ticket T_{Dev_g} and session key $k_{C \leftrightarrow Dev_g}$ using different long term keys:

$$HMAC(K_{Dev_g \leftrightarrow ISV}, [ID_C || AD_c || L_6 || ID_{Dev_g}]) \quad (3)$$

5) Finally ISV builds response $Res_{ISV \rightarrow C}$ and sends it to C:

$$E(k_{C \leftrightarrow ISV}, [ID_{Dev_g} || k_{C \leftrightarrow Dev_g} || TS_{ISV \rightarrow C} || L_6 || T_{Dev_g}])$$

6) C decrypts $Res_{ISV \rightarrow C}$ with $k_{C \leftrightarrow ISV}$, retrieves and caches $k_{C \leftrightarrow Dev_g}$, T_{Dev_g}.

Service Phase (C $\leftrightarrow$ Dev_g)

1) C computes authenticator $A_{C \rightarrow Dev_g}$:

$$A_{C \rightarrow Dev_g} = HMAC(k_{C \leftrightarrow Dev_g}, [TS_{C \rightarrow Dev_g}]) \quad (4)$$

2) Then C builds $Req_{C \rightarrow Dev_g}$ and sends it to Dev_g:

$$serv\_req || ID_C || AD_c || L_6 || T_{Dev_g} || TS_{C \rightarrow Dev_g} || A_{C \rightarrow Dev_g}$$

3) Dev_g checks plaintext $TS_{C \rightarrow Dev_g}$ and L_6 values, generates $k_{C \leftrightarrow Dev_g}$ locally and verifies T_{Dev_g} using Equation 3, verifies $A_{C \rightarrow Dev_g}$ using Equation 4.
4) Finally Dev_g performs the requested service and sends back either plaintext service_response (non-sensitive information) or service_response encrypted with $k_{C \leftrightarrow Dev_g}$ (sensitive information).
5) C receives service_response and decrypts with $k_{C \leftrightarrow Dev_g}$ if necessary.

Fig. 4: Dev_g Protocol

After receiving $Res_{ISV \rightarrow Dev_g}$, Dev_g authenticates ISV in step 6 by verifying $A_{ISV \rightarrow Dev_g}$. Since Co_{sync} grows monotonically, it also acts as a nonce in computing $A_{ISV \rightarrow Dev_g}$. This prevents replays of old synchronization responses. If verification succeeds, Dev_g stores $TS_{ISV \rightarrow Dev_g}$ as its start_time. After synchronization of start_time, local time is set to: start_time plus current timer value. If Dev_g's timer drifts drastically, the calculated timestamp would not be reliable. In such cases, it is recommended that this protocol is executed not only at boot time but also at regular (long-term) intervals.

Ticket Issuing Phase: A user wishing to request a service from Dev_g must obtain an IoT ticket from ISV following the steps described in the Service phase of Protocol 1. This protocol includes ticket lifetime L_6 in IoT tickets to maintain ticket validity periods. This enables ISV to grant tickets to multiple clients for the same time period. Also, each ticket can be used multiple times before its expiration time. IoT ticket for Dev_g is generated by computing an HMAC over (ID_C, AD_c, L_6, ID_{Dev_g}) with a long-term key. The corresponding session key is generated by computing HMAC over the same values with a *different* long-term key. The IoT ticket generation process is different from that in Kerberos. The latter is secured using encryption, while IoT tickets are secured via HMAC.

Service Phase: This part of the protocol is used by users to request a service from a Dev_g. Since IoT tickets are multi-use, replay attacks pose a problem. To mitigate them, in step 1 of the Service Phase, C calculates an authenticator $A_{C \rightarrow Dev_g}$ by computing HMAC over its current timestamp $TS_{C \rightarrow Dev_g}$ with a session key $k_{C \leftrightarrow Dev_g}$. Then, C includes both $TS_{C \rightarrow Dev_g}$ and $A_{C \rightarrow Dev_g}$ in its service request, $Req_{C \rightarrow Dev_g}$. Together with $TS_{C \rightarrow Dev_g}$, $A_{C \rightarrow Dev_g}$ mitigates replays.

Upon receiving $Req_{C \rightarrow Dev_g}$, Dev_g computes the local timestamp by adding the current timer value to start_time. Then, in step 3, Dev_g checks if the difference between plaintext $TS_{C \rightarrow Dev_g}$ and the local timestamp is within a predefined short range. If not, it discards the request and sends back an "Invalid Request" response. Dev_g also checks if plaintext ticket lifetime L_6 is later than the current local time. If not, it discards the request and sends back a "Ticket Expired" response. Next, Dev_g verifies $A_{C \rightarrow Dev_g}$. Finally, Dev_g verifies the integrity of the IoT ticket by computing an HMAC over plaintext (ID_C, AD_c, L_6, ID_{Dev_g}) with its long-term key and comparing it with the corresponding value in the received ticket. This integrity check prevents the adversary from modifying the expiration time with the purpose of extending the ticket lifetime.

Attestation as a Service: Along with standard IoT device functionalities, Dev_g makes its own attestation available as a service. Any user with a valid IoT ticket can act as a verifier and request Dev_g to attest itself. Upon receiving an attestation request and validating the IoT ticket, RoT inside Dev_g calculates an HMAC over the program memory using the session key $k_{C \leftrightarrow Dev_g}$. Dev_g sends back the computed attestation result (HMAC) to the user. Acting as a verifier, the user knows the expected HMAC value for the benign (expected) software state of Dev_g and thus can verify the response to determine Dev_g's current software state.

B. Protocol for Power Constrained Devices

Figure 5 provides an overview of KESIC protocol for Dev_{pc}, while detailed description is presented in Figure 6.

Counter Synchronization Phase: ISV maintains a separate synchronization value for each Dev_{pc}, and this value is reinitialized with the current timestamp from the local clock of ISV every time Dev_{pc} performs synchronization.

When Dev_{pc} boots up, it follows the steps outlined in Counter Synchronization Phase of Protocol 2 to obtain the

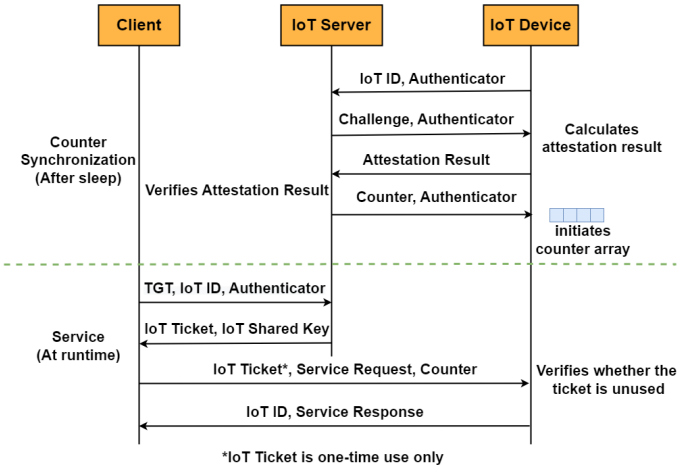

Fig. 5: $\boldsymbol{Dev_{pc}}$ Protocol Overview.

synchronization value from ISV. ISV also requests an attestation report from Dev_{pc} during this process. Only if that report is valid, i.e., if Dev_{pc} is healthy, ISV grants tickets to clients to access Dev_{pc}. This eliminates the need to expose attestation as a service for Dev_{pc}, given that Dev_{pc} is low in runtime and power budget.

Ticket Issuing Phase: This part of Protocol 2 describes how clients obtain IoT Tickets from ISV, prior to to requesting service from Dev_{pc}. These tickets are single-use only. The rationale is that synchronizing and running a timer in Dev_{pc} for a short period of activity is expensive and offers low utility. As a result, tickets can not include a timestamp to indicate their validity period. Instead, ISV maintains a separate counter initialized with the synchronization value for each Dev_{pc}. Recall that the synchronization value is the timestamp sent to Dev_{pc} during Counter Synchronization Phase. Each time ISV receives a ticket request from a client, it increases the counter value by 1 and uses the counter value as the nonce in the ticket, ensuring the ticket is fresh and unique for a single use.

Service Phase: In this phase, clients use Protocol 2 to request service from Dev_{pc}. However, nonce-based Dev_{pc} tickets are prone to race conditions. For example, suppose that Client-A and Client-B both obtain IoT tickets from ISV. However, Client-A obtains its ticket before Client-B, resulting in the former's ticket having a lower-numbered nonce than the ticket of the latter. (Recall that we treat nonces as monotonic counters.) If Client-B presents its ticket to the device first, Dev_{pc} updates its local counter value with the nonce of Client-B's ticket. In that case, Client-A becomes unable to use its ticket since the nonce in Client-A's ticket is lower than the current counter value of Dev_{pc}.

To avoid such anomalies, Dev_{pc} maintains a *counter buffer* of size *n*, where *n* is the maximum number of clients that can obtain tickets for Dev_{pc} during each of its liveness (awake) period. In step 2 of the Service Phase of Protocol 2, when Dev_{pc} receives a ticket, it checks whether its nonce is within the counter buffer and is still unused. If this check fails, Dev_{pc} discards the request and returns an "Invalid Counter" response.

Protocol 2: Power constrained Device Protocol

Counter Synchronization Phase ($\boldsymbol{Dev_{pc} \leftrightarrow}$ ISV)

1) Upon Dev_{pc} waking up, Dev_{pc} and ISV perform the same steps as steps 1 to 3 from the Time Synchronization Phase of Protocol 1 to build and verify $Req_{Dev_{pc} \rightarrow ISV}$ respectively.
2) Then ISV generates a random number as challenge and computes authenticator $A^{attest}_{ISV \rightarrow Dev_{pc}}$:
$$HMAC(K_{Dev_{pc} \leftrightarrow ISV}, [ID_{ISV} || challenge]) \quad (5)$$
3) ISV builds attestation request, $Req^{attest}_{ISV \rightarrow Dev_{pc}}$ and sends it to Dev_{pc}: $ID_{ISV} || challenge || A^{attest}_{ISV \rightarrow Dev_{pc}}$
4) Dev_{pc} verifies $A^{attest}_{ISV \rightarrow Dev_{pc}}$ using Equation 5 and generates Attestation Key, $k_{Dev_{pc} \leftrightarrow ISV}$:
$$k_{Dev_{pc} \leftrightarrow ISV} = HMAC(K_{Dev_{pc} \leftrightarrow ISV}, [challenge]) \quad (6)$$
5) Dev_{pc} calculates HMAC of memory region and sends it to ISV:
$$Attst_{hmac} = HMAC(k_{Dev_{pc} \leftrightarrow ISV}, [Memory]) \quad (7)$$
6) Upon receiving $Attst_{hmac}$, ISV generates $k_{Dev_{pc} \leftrightarrow ISV}$ using Equation 6 and verifies $Attst_{hmac}$ using Equation 7. It also assigns current timestamp value $TS_{ISV \rightarrow Dev_{pc}}$ to $Co_{Dev_{pc}}$.
7) Then ISV and Dev_{pc} perform the same steps as steps 4 to 6 from the Time Synchronization Phase of Protocol 1 to build and verify $Res_{ISV \rightarrow Dev_{pc}}$ respectively.
8) Dev_{pc} initializes local counter array using $TS_{ISV \rightarrow Dev_{pc}}$.

Ticket Issuing Phase (C↔ISV)

1) C performs the steps of Kerberos protocol to obtain Kerberos service ticket T_{ISV} and shared key $k_{C \leftrightarrow ISV}$ for ISV. Then C and ISV perform the same steps as steps 2 to 4 from the Ticket Issuing phase of Protocol 1 to build and verify $Req_{C \rightarrow ISV}$ respectively.
2) Then ISV issues IoT ticket $T_{Dev_{pc}}$ and generates session key $k_{C \leftrightarrow Dev_{pc}}$ using different long term keys:
$$HMAC(K_{Dev_{pc} \leftrightarrow ISV}, [ID_C || AD_c || Co_{Dev_{pc}} || ID_{Dev_{pc}}]) \quad (8)$$
3) Finally ISV builds $Res_{ISV \rightarrow C}$ and sends it to C:
$$E(k_{C \leftrightarrow ISV}, [ID_{ISV} || k_{C \leftrightarrow Dev_{pc}} || TS_{ISV \rightarrow C} || Co_{Dev_{pc}} || T_{Dev_{pc}}])$$
4) C decrypts $Res_{ISV \rightarrow C}$ with $k_{C \leftrightarrow ISV}$, retrieves and caches $k_{C \leftrightarrow Dev_{pc}}$, $T_{Dev_{pc}}$.

Service Phase (C↔$\boldsymbol{Dev_{pc}}$)

1) C builds $Req_{C \rightarrow Dev_{pc}}$ and sends it to Dev_{pc}: $service\_request || ID_C || AD_c || Co_{Dev_{pc}} || T_{Dev_{pc}}$
2) Dev_{pc} checks plaintext $Co_{Dev_{pc}}$ value, generates $k_{C \leftrightarrow Dev_{pc}}$ locally, and verifies $T_{Dev_{pc}}$ using Equation 8.
3) Finally Dev_{pc} performs the requested service and sends back either plaintext service_response (non-sensitive information) or service_response encrypted with $k_{C \leftrightarrow Dev_{pc}}$ (sensitive information).
4) C receives service_response and decrypts with $k_{C \leftrightarrow Dev_{pc}}$ if necessary.

Fig. 6: $\boldsymbol{Dev_{pc}}$ Protocol.

This prevents race conditions for single-use Dev_{pc} tickets while keeping them non-blocking, meaning that ISV does not reserve a specific time period for a client's Dev_{pc} ticket, and does not reject tickets for other clients during that period.

V. Implementation Details

This section describes the prototype implementation of ISV, Dev_g, Dev_{pc}, and client application. All source code is available at [6].

A. IoT Server

We implemented ISV in Python. It has two main components:

Ticket Manager (TM) is implemented as a web application using Flask library. It is hosted in Apache Web Server and configured for Kerberos Authentication. A client needs to obtain a valid ticket from TGS to call TM. TM follows a static policy to grant IoT tickets: it maintains a list of allowed users. A granted IoT ticket for a given device is valid for all available functionalities of that device.

Synchronization Manager (SM) is implemented in Python. Two always-listening UDP server sockets are used for communication with devices. The first is responsible for accepting synchronization requests from devices, while the second accepts attestation reports from Dev_{pc} as part of the synchronization process. Communication between SM and devices is protected by the long-term secret key, $K_{Dev_g \leftrightarrow ISV}$ or $K_{Dev_{pc} \leftrightarrow ISV}$, shared between ISV and each device.

SM runs as a separate thread from TM. ISV is hosted on a Linux laptop with Intel(R) Core(TM) i7-8550U CPU running at 1.80GHz, with 8GB RAM.

B. Dev_g and Dev_{pc}

An NXP LPCXpresso55S69 development board with TrustZone-M emulates an IoT device. The board is based on ARM Cortex-M33 MCU. It runs at 150 MHz with 640KB flash and 320KB SRAM. Wifi 10 click board is used – along with LPCXpresso55S69 board – for WiFi connectivity. Dev_g and Dev_{pc} are emulated separately. The former is a smart bulb where clients control the following features: (1) turning on LED, (2) turning off LED, and (3) performing attestation. The latter is also a smart bulb, however, clients can only turn on and turn off LED. It does not support attestation.

The program running on the emulated device is divided into two parts:

Non-Secure Part: processes user commands. It is also responsible for the synchronization process with ISV. Network communication and actuation (turning on/off LED) are handled by this part. All communication is over UDP.

Secure Part: works as an RoT. It stores secret keys and synchronization value, as well as timer/counter value. It is responsible for all cryptographic operations, i.e., HMACs, and uses the Mbed-TLS library.

These two parts are compiled into separate .axf (binary) files and also flashed to the board separately. Moreover, they are executed using separate RAM.

C. Client Application

A sample client application is written in Python. It obtains IoT tickets from ISV and requests service from Dev_g and Dev_{pc}. Before the client application calls TM to request an IoT ticket, the client (user) must obtain a Kerberos ticket for ISV. This is done by configuring a Kerberos client on the client machine and calling kinit. Then, the client application automatically includes the Kerberos ticket in the request header when it calls TM.

D. Remote Attestation Process

Both Dev_g and Dev_{pc} can perform remote attestation. Dev_{pc} performs attestation during the synchronization phase with ISV acting as the verifier. In case of Dev_g, a user (acting as a verifier) can ask Dev_g to perform attestation during the service phase. In both cases, the attestation process is the same.

The RoT in the secure part of the device computes a hash of the entire non-secure program flash memory using *mbedtls_sha256*. Then, the appropriate key is used to compute an HMAC over this hash value using *MBEDTLS_MD_SHA256*. Note that, for Dev_g, this attestation key is the service session key, $k_{C \leftrightarrow Dev_g}$, shared with client (C). For Dev_{pc}, it is a temporary attestation key, $k_{Dev_{pc} \leftrightarrow ISV}$ computed during the synchronization process.

The computed HMAC is sent to the verifier. The correct (expected) reference hash for a given device is assumed to correspond to the latest legitimate software version that the verifier expects the device to run. Since the verifier is assumed to know both the attestation key and the expected reference hash, it computes its own expected HMAC and compares it with that received from the device, which allows to determine whether the device is malware-free.

VI. Evaluation

This section presents performance analyses of KESIC. As part of performance analysis, we evaluate KESIC in terms of storage, memory, run time, and network overhead. Since ISV and client applications are expected to run on powerful devices, overheads of these components are minimal and not discussed here due to space constraints. Thus, we focus on the overhead of IoT devices.

A. Storage & Memory Overhead

We assess storage and memory KESIC overhead for Dev_g and Dev_{pc} prototype implementations. Figure 7 shows the details about storage and memory overhead. We measure storage overhead in terms of the .axf file size. Memory overhead is measured by the increase in RAM usage. As evident from Figure 7, storage and memory overheads are very low for both device types. KESIC causes only a 60KB increase overall ((considering both secure and non-secure parts) for Dev_g over .axf file sizes of 3604 KB. For Dev_{pc}, this increase is even lower: 56KB. Storage overhead is 1.66% for Dev_g and 1.55% for Dev_{pc}. Furthermore, overall memory overhead is $\sim$0.3% for both Dev_g and Dev_{pc}. This low overhead is due to the simple and lightweight implementation in KESIC.

B. Runtime Overhead

Dev_g and Dev_{pc} incur different runtime overheads in synchronization and service phases. We measure overhead in terms of cycles to complete each phase. Using the operational frequency of the board, we also compute elapsed time. Note that we measure the mean of each performance value over 10 iterations. Runtime overheads of Dev_g and Dev_{pc} are presented in Table II.

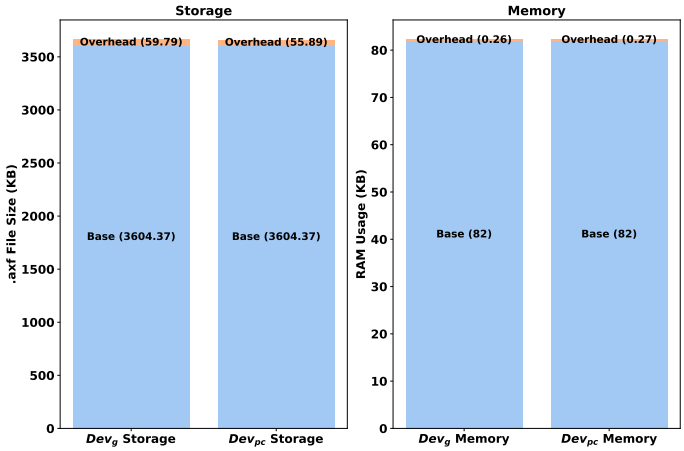

Fig. 7: Storage & Memory Overhead for $\boldsymbol{Dev_g}$ & $\boldsymbol{Dev_{pc}}$.

TABLE II: Runtime Overheads for $\boldsymbol{Dev_g}$ and $\boldsymbol{Dev_{pc}}$.

Phase	Device Type	Kilocycles	Time @ 150MHz (ms)
Synchronization	Dev_g	6624.7180	44.1648
	Dev_{pc}	16823.0482	112.1537
Service	Dev_g	1267.3084	8.4487
	Dev_{pc}	541.5319	3.6102

The synchronization time is minimal at 44.1648 ms on average for Dev_g. Recall that it only occurs at boot time. On the other hand, the average synchronization time for Dev_{pc} is 112.1537 ms. This is because the synchronization phase of Dev_{pc} involves multiple communication rounds and remote attestation. Thus, it incurs higher runtime overhead than Dev_g. However, it ensures that the device is healthy and frees the users from having to perform remote attestations separately.

During the service phase, KESIC verifies IoT tickets, which incurs runtime overhead that varies based on the service request scenario. All Values included in a ticket are checked one by one, and the ticket is rejected if any verification fails. The verification process in Dev_g takes 8.4487 ms (on average) for a valid ticket. The ticket verification process is even faster for Dev_{pc}. Validating a ticket takes only 3.6102 ms.

C. Network Overhead

We now consider network overhead on IoT devices caused by KESIC in synchronization and service phases. We consider sizes of all requests and responses exchanged by IoT devices. Network overhead is summarized in Table III.

Synchronization request and response sizes are 104 and 136 bytes, i.e., 240 bytes total. Dev_{pc} exchanges two additional messages with ISV: attestation request (104 bytes) and response (72 bytes), resulting in 416 bytes total. Similarly, network overhead during the service phase comes from service requests. It is 208 bytes for Dev_g and 112 bytes for Dev_{pc}.

TABLE III: Network Overheads for $\boldsymbol{Dev_g}$ and $\boldsymbol{Dev_{pc}}$.

Phase	Device Type	Overhead (B)
Synchronization	Dev_g	240
	Dev_{pc}	416
Service	Dev_g	208
	Dev_{pc}	112

TABLE IV: **KESIC** vs Kerberos Overheads.

Overhead	Dev_g	Dev_{pc}	Kerberos [29], [30], [31]
Storage (KB)	59.788	55.888	221.032
Memory (B)	256	272	12000
Runtime (ms)	8.4 ms (@150 MHz)	3.6 ms (@150 MHz)	7.8ms (@2.2 GHz)
Network (B)	208	112	846

D. Comparison with Kerberos

We compare the overhead of KESIC with that of Kerberos. However, as mentioned earlier in Section III, Kerberos can not be directly implemented on IoT devices. Therefore, we can not directly compare their respective overheads and instead compare KESIC with standard Kerberos for its usual setting. Table IV shows the results.

As the table shows, storage, memory, runtime, and network overheads of KESIC are significantly lower than those of Kerberos. Dev_g incurs 3.69× lower storage, 46.87× lower memory, 135.41× lower runtime, and 4.07× lower network overhead. Meanwhile, Dev_{pc} exhibits 3.95× lower storage, 44.12× lower memory, 316.88× lower runtime, and 7.55× lower network overhead.

VII. Related Work

Kerberos-Based Authentication Schemes for IoT Devices: Several efforts attempted to address the multi-user access problem for IoT devices by adopting Kerberos. There are two main directions in prior work:

The first aims to decrease the computation and communication cost of Kerberos for IoT devices [5], [32], [33], [34], [35], [36]. [5] uses a nonce-based service ticket to grant access. However, the device can not verify the ticket locally and must communicate with KDC to do so. [33] reduces the number of messages exchanged and the cost of constructing a ticket. [34] uses table representation to reduce code size, memory copies, and heap allocations. None of these results, except [5], applies to IoT devices without real-time clocks.

The second direction focuses on addressing certain use-cases, such as machine-to-machine communication (communication among IoT devices) or using a central controller to manage all IoT devices [37], [36], [38], [39]. Such use-cases are different from what KESIC targets: allowing multiple users direct and secure access to IoT devices. [37] introduces a smart central controller to implement Kerberos for smart-home systems and maintains authentication and authorization at the controller level. [36] involves a low-cost Machine-to-Machine (M2M) protocol for IoT devices to communicate with machines. [39] uses an inter-server protocol: it establishes communication between two servers and uses an improved key agreement, as compared to Kerberos.

Other Authentication Schemes for IoT Devices: Several authentication methods for IoT devices have been proposed e.g. [40], [41], [42], [43], [44], [45], based on a variety of features such as biometrics, physical unclonable functions, channel characteristics, one-time passwords (OTPs),

blockchain etc. [40] develops a deep learning based user authentication scheme that utilizes WiFi signals to capture unique human physiological and behavioral characteristics inherited from their daily activities. [42] presents a two-factor authentication protocol for IoT-enabled healthcare ecosystems using biometrics and post-quantum cryptography. [43] proposes a lightweight and secure multi-factor device authentication protocol for IoT devices using configurable PUFs and channel-based parameters. [44] introduces a multi-device user authentication mechanism for IoT devices using OTPs and a novel device usage detection mechanism. [45] proposes a system that authenticates user access to IoT devices using blockchain-enabled fog nodes. The nodes connect to Ethereum smart contracts, which issue access tokens without requiring an intermediary or trusted third party.

VIII. Conclusion

This paper constructed KESIC, a secure multi-user access mechanism for a range of IoT devices. KESIC contends with hardware resource constraints of IoT devices and significant overhead associated with Kerberos. It involves a new component – an IoT Server (ISV) – a special Kerberos service responsible for managing access to IoT devices. We implemented an open-source prototype [6] of KESIC, which includes ISV, two devices based on ARM Cortex-M33 equipped with TrustZone-M, and a client application. Its evaluation shows that a general device takes only 8.45ms to verify a ticket, while a power-constrained device takes only 3.61ms.

Acknowledgements: We thank ICCCN'24 reviewers for constructive feedback. This work was supported in part by funding from NSF Award SATC-1956393, NSA Awards H98230-20-1-0345 and H98230-22-1-0308, as well as a subcontract from Peraton Labs.

References

[1] statista. "number of internet of things (iot) connected devices worldwide from 2019 to 2021, with forecasts from 2022 to 2030". https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/, 2022.

[2] Wired Magazine. The botnet that broke the internet isn't going away. https://www.wired.com/2016/12/botnet-broke-internet-isnt-going-away/, 2016.

[3] Pinto et al. Triton: The first ics cyber attack on safety instrument systems. In *Black Hat USA*, 2018.

[4] Kansal et al. Building a sensor network of mobile phones. In *IPSN*, pages 547–548. ACM, 2007.

[5] Astorga et al. Security for heterogeneous and ubiquitous environments consisting of resource-limited devices: An approach to authorization using kerberos. In *ICST*, 2009.

[6] Kesic source code. https://github.com/sprout-uci/KESIC, 2023.

[7] S. William. *Cryptography and Network Security - Principles and Practice, 7th Edition*. Pearson Education India.

[8] Noorman et al. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX Security Symposium*, 2013.

[9] Grisafi et al. PISTIS: Trusted computing architecture for low-end embedded systems. In *USENIX Security*, 2022.

[10] Nunes et al. VRASED: A verified hardware/software co-design for remote attestation. In *USENIX Security*, 2019.

[11] Nunes et al. Towards remotely verifiable software integrity in resource-constrained iot devices. *IEEE Communications Magazine*, 2024.

[12] Nunes et al. On the toctou problem in remote attestation. *CCS*, 2021.

[13] Arm Ltd. Arm TrustZone. https://www.arm.com/products/security-on-arm/trustzone, 2018.

[14] Intel. Intel Software Guard Extensions (Intel SGX). https://software.intel.com/en-us/sgx.

[15] AMD. Amd secure encrypted virtualization (amd sev). https://developer.amd.com/sev/.

[16] Nunes et al. $\mathcal{P}$ARseL: Towards a verified root-of-trust over sel4. In *ICCAD*, 2023.

[17] Gerwin et al. seL4: Formal verification of an OS kernel. In *ACM SIGOPS*, 2009.

[18] Blink mini security camera. https://www.amazon.com/dp/B07X4BCRHB?tag=meastus-20.

[19] Google nest thermostat. https://store.google.com/us/product/nest_thermostat?hl=en-US.

[20] Lumiman smart bulb. https://www.amazon.com/Lights-Multi-Colored-Google-Assistant-Required/dp/B07DLSNNDS/.

[21] Thermopro thermometer. https://www.amazon.com/ThermoPro-Bluetooth-Hygrometer-Thermometer-Greenhouse/dp/B08LKCLFR6?th=1.

[22] Netatmo weather station -weatherproof. https://www.amazon.com/Netatmo-Weather-Station-Weatherproof/dp/B0098MGWA8/.

[23] nrf9160 low power sip with integrated lte-m/nb-iot modem and gnss. https://www.nordicsemi.com/products/nrf9160.

[24] Muraleedharan et al. Jamming attack detection and countermeasures in wireless sensor network using ant system. In *Wireless Sensing and Processing*, volume 6248, pages 118–129. SPIE, 2006.

[25] Wu et al. Low-rate dos attacks, detection, defense, and challenges: A survey. *IEEE access*, 8:43920–43943, 2020.

[26] Mamdouh et al. Securing the internet of things and wireless sensor networks via machine learning: A survey. In *ICCA*. IEEE, 2018.

[27] Srivaths et al. Tamper resistance mechanisms for secure embedded systems. In *VLSI Design*, 2004.

[28] Obermaier et al. The past, present, and future of physical security enclosures: From battery-backed monitoring to puf-based inherent security and beyond. *Journal of Hardware and Systems Security*, 2018.

[29] Mit kerberos repository. https://github.com/krb5/krb5.

[30] Moralis et al. Performance comparison of web services security: Kerberos token profile against x. 509 token profile. In *ICNS*, 2007.

[31] Configuring kerberos token size using the maxtokensize parameter. https://woshub.com/kerberos-token-size-and-issues-of-its-growth/.

[32] Thomas Hardjono. Kerberos for internet-of-things. *IETF89*, 2014.

[33] AlJanah et al. A multifactor multilevel and interaction based (m2i) authentication framework for internet of things (iot) applications. *IEEE Access*, 10:47965–47996, 2022.

[34] Kazunori Miyazawa. Design and implementation of kerberos version 5 for embedded devices. In *2010 8th IEEE International Conference on Industrial Informatics*, pages 449–453. IEEE, 2010.

[35] Mohsin B Tamboli and Dayanand Dambawade. Secure and efficient coap based authentication and access control for internet of things (iot). In *RTEICT*. IEEE, 2016.

[36] Esfahani et al. A lightweight authentication mechanism for m2m communications in industrial iot environment. *IEEE Internet of Things Journal*, 2017.

[37] Gaikwad et al. 3-level secure kerberos authentication for smart home systems using iot. In *2015 1st International Conference on Next Generation Computing Technologies (NGCT)*. IEEE, 2015.

[38] Hokeun Kim. *Securing the internet of things via locally centralized, globally distributed authentication and authorization*. University of California, Berkeley, 2017.

[39] Thirumoorthy et al. Improved key agreement based kerberos protocol for m-health security. 2022.

[40] Shi et al. Smart user authentication through actuation of daily activities leveraging wifi-enabled iot. In *MOBIHOC*, pages 1–10, 2017.

[41] Dhillon et al. A lightweight biometrics based remote user authentication scheme for iot services. *Journal of Information Security and Applications*, 34:255–270, 2017.

[42] Saggaf et al. Lightweight two-factor-based user authentication protocol for iot-enabled healthcare ecosystem in quantum computing. *Arabian Journal for Science and Engineering*, 48(2):2347–2357, 2023.

[43] Mohammed Mujib Alshahrani. Secure multifactor remote access user authentication framework for iot networks. *Computers, Materials & Continua*, 2021.

[44] Eman et al. A multi-device user authentication mechanism for internet of things. *IET Networks*, 2023.

[45] Almadhoun et al. A user authentication scheme of iot devices using blockchain-enabled fog nodes. In *AICCSA*. IEEE, 2018.