Chen, Fei; Xiao, Zixing; Xiang, Tao; Fan, Junfeng; Truong, Linh

A Full Lifecycle Authentication Scheme for Large-scale Smart IoT Applications

Published in:
IEEE Transactions on Dependable and Secure Computing

DOI:
10.1109/TDSC.2022.3178115

Published: 26/05/2022

Document Version
Peer reviewed version

Please cite the original version:
A Full Lifecycle Authentication Scheme for Large-scale Smart IoT Applications

Fei Chen, Zixing Xiao, Tao Xiang, Junfeng Fan, and Hong-Linh Truong

Abstract—The rapid development of IoT (Internet of Things) brings great convenience to people through the utilization of IoT applications, but also brings huge security challenges. Existing IoT security breaches show that many IoT devices have authentication flaws. Although many IoT authentication schemes were proposed, they are not fit for recent smart IoT applications covering IoT device, back-end server, and user-end mobile applications. To build the first line of defense for smart IoT systems, this paper proposes a new authentication scheme. The proposed scheme first models the entire lifecycle of the IoT device authentication for real-world scenarios of smart IoT systems that contains factory manufacturing, daily usage, and system resetting. For each stage in the lifecycle, the proposed scheme employs efficient symmetric key mechanisms to achieve the authentication between IoT device, back-end server, and mobile application. The proposed scheme supports both server-free local area network communication and sever-involved remote public area communication. Formal security verification shows that the proposed scheme resists existing attacks. The open-source experimental evaluations also show that the proposed scheme is efficient and promising for practical usage.

Index Terms—smart IoT application, authentication, lifecycle management, scalability, formal verification

1 INTRODUCTION

Internet of Things (IoT) applications, integrated with cloud computing and mobile computing, have been shaping a new computing paradigm for modern society as demonstrated through the usefulness of various smart IoT applications, e.g., smart camera, smart bulb, smart logistics, and smart manufacturing. Because of such usefulness, smart IoT applications have obtained a remarkable success. It is projected that the number of IoT devices will reach 30.9 billion units by 2025 [1]. They have also attracted various research efforts [2]–[4].

Typically, shown in Fig. 1, an architecture of a smart IoT system mainly includes three entities, i.e., IoT device, user mobile application, and back-end server. Consider the smart camera as a use case. A vendor produces millions of smart cameras that are later used by millions of users around the world for smart home security surveillance. The vendor also provides a mobile application for the user to interact with the smart camera both locally and remotely through a back-end server that is run by the vendor. When the user and the IoT device are located in the same local area network, the user can directly issue commands to control the device using a mobile application. Similarly, the device can directly report data to the user. Besides local and direct communication, the user can communicate with the IoT device remotely by employing the back-end server to relay the communication, i.e., the user sends a command to the server and then the server forwards it to the device. The convenient access and control of an IoT device makes smart IoT applications very attractive for users. These applications are considered smart IoT applications in this paper. They and their deployment models continuously attract large-scale number of users in many application domains. They are also the focus of this work. We note that the IoT applications that do not fit the architecture of Fig. 1 are out of the scope of the paper.

While such smart IoT applications bring considerable convenience to end users, they also incur a lot of security problems. Researchers have found various security risks in existing smart IoT applications. For example, Obermaier and Hutle found that the video stream captured by a smart camera could be leaked, deleted, or modified due to insecure authentication [5]. In this case, the video camera used the MAC address (or some computation of MAC) for remote authentication. However, an attacker is able to predict the MAC addresses from the same manufacturer, which results in impersonation attacks. Ling et al. found that attackers could impersonate a user to communicate with the back-end server, making the real user’s smart plug not working due to the impersonation [6]. This was also caused by insecure authentication. This scheme used some variant of the MAC.
address to conduct authentication remotely. Fernandes et al. found that applications in Samsung SmartThings could leak user privacy due to insecure authorization [7]. Similar security issues are common and are being reported in researches, even for famous products from influential companies, e.g., Google Home [8].

These reported security issues are caused by the various vulnerabilities of the smart IoT devices. Among the vulnerabilities, insecure authentication is a very critical security flaw, which includes authentications between the device, the back-end server, and the mobile application. This is because an attacker could impersonate a cloud server or a device in the IoT system architecture that the attacker does not possess. Such impersonation attack causes the normal service of other victims’ smart IoT devices failing to work (e.g., [5], [6]). Thus, the attack brings loss to the vendor and incurs bad experiences to the smart IoT application users. Moreover, insecure authentication may also cause data security problems that depend on the underlying design of the smart IoT application. Therefore, to improve smart IoT application security, this paper studies the critical authentication problem for the smart IoT systems.

State-of-the-art works and limitations. Researchers have studied authentication in traditional IoT systems intensively. Many proposed IoT authentication schemes (e.g., cable TV system, 4G mobile phone network) are based on the security factor of smart cards [9], [10]. In such a scheme, the user employs a smart card and/or a password to authenticate itself to a service. Some schemes (e.g., smart home gateway) employ a separate, centralized authentication gateway to authenticate the associated IoT devices [4], [11]–[13]. The user communicates with the IoT devices through the centralized gateway. Recent researches are also trying to leverage emerging technologies (such as edge computing, blockchain) to enable IoT authentication [14]–[16].

While these authentication schemes work for traditional IoT systems, they are not a reasonable fit for large-scale smart IoT systems. First, existing authentication schemes only support limited communications. The smart IoT systems however require more flexible user access and control, either locally or remotely using the end-user’s mobile applications. The communication is more complicated because of more system participants and more interactions. Second, existing schemes face usability issues. In case of a back-end server breakdown, traditional IoT authentication schemes fail to work. This is not a desired property for a smart IoT application. Third, when the number of IoT devices scales up, the performance of traditional authentication schemes is also an issue. When more IoT devices are deployed, the pressure of the authentication server grows considerably.

Our Work. This paper proposes a secure, flexible, and efficient authentication scheme for the typical smart IoT system as in Fig. 1. The proposed scheme supports mutual authentication between the IoT device, the server, and the user (i.e., the mobile application). Corresponding to local communication, the proposed scheme allows the IoT device and the user to authenticate each other. Corresponding to remote communication, the scheme supports two folds, i.e., the server and the IoT device authenticates each other first, and then the server and the user authenticates each other subsequently to establish a relay channel. Besides the basic authentication requirement, the proposed scheme also covers the whole lifecycle of a smart IoT device, from manufacturing to off-shell use, and to ownership change. To make the proposed scheme efficient, we employ efficient cryptographic mechanisms (e.g., symmetric encryption, message authentication code, etc.) in the design.

Specifically, the proposed scheme works as follows. First, at the production of the IoT device, the smart IoT system producer/manager/service provider authenticates a device uniquely. It assigns a unique ID (i.e., identity) and embeds a unique secret key for each device. The ID and the secret key are used to authenticate the device to the back-end server. Second, after the user buys a smart IoT device and performs the setup, the user, the device, and the server communicate all together to securely bind the user to the device. Through binding, this step prevents attackers getting unauthorized access of the server and the IoT device. Third, after setup, the user could access and control the device locally and remotely in normal daily uses. During access and control, the user needs to authenticate to the IoT device and the back-end server using the detailed authentication algorithms in the proposed scheme. After successful authentication, the user can get data from the device and send commands to the device. Fourth, a user is also able to make updates to the device in later usage. The updates could be changing credential and restoring factory settings (which is useful for transferring the device to another user).

We further analyze the security of the proposed scheme using both heuristic approaches and formal approaches. We list common attacks and explain why the proposed scheme can resist these attacks. This enables an intuitive understanding of the proposed scheme. We also conduct a formal security analysis based on the AVISPA tool [17]. It exhausts and analyses the flow of the proposed scheme using formal logic. Besides, we also measure the performance of the proposed scheme experimentally.

Results. We conducted formal security verification using the AVISPA tool [17]. The results show that the proposed scheme achieves sound authentication. It resists impersonation attacks, man-in-the-middle attacks, and replay attacks. We further prototyped the proposed scheme using Python by simulating IoT device, authentication server, relay server, and user’s mobile application. Experimental results show that the proposed scheme can correctly complete all communication processes and that the proposed scheme is efficient during frequent online authentication processes. Specifically, the storage overhead of the authentication server and relay server are 124 bytes and 98 bytes per device. The maximal communication cost is around 1.2 kilobytes, which is used to set up one-round authenticated remote communication. The maximal computation cost is about 2.7 milliseconds, which is also used in the authenticated remote communication setup.

Contribution. In summary, this paper makes the following contribution:

- Conceptual construction. We proposed a specific authentication scheme for smart IoT applications. The proposed scheme enables secure authentication between the IoT device, the back-end server, and the
mobile application. The proposed scheme enables full-lifecycle authentication in the sense that it covers device manufacturing, device operation, and device ownership transfer, i.e., the while lifespan of the device.

- **Solid analysis.** Formal analysis using the AVISPA tool verified the security of the proposed authentication scheme. We also prototyped and opensourced the proposed authentication scheme. Experimental results also validated the efficiency of the proposed scheme.

2 RELATED WORK

IoT systems are evolving in the past years. Researchers have uncovered various vulnerabilities of the IoT devices on the market. We first present a sample review of real smart IoT application security cases caused by the insecure authentication vulnerability in recent five years. This case review also shows the need of a new secure authentication design. To enable IoT security, authentication has been studied from various perspectives. We classify existing authentication schemes into three categories, i.e., authentication between device and server, between device and user, and between server and user. Then for each category, we review related research works. Compared with recent surveys in the IoT authentication area (e.g., [18]–[20]), our review here focuses more on the interactions between the IoT device, the server, and the user. The consideration is that it intuitively shows the way how the IoT device is used in practice and that it is more close to our work in this paper.

2.1 Insecure IoT Authentication Case Review

In 2017, Ling et al. found that a smart plug had serious security risks due to insecure authentication [6]. In this application, the smart plug used the MAC address to authenticate the device. An attacker could write a script to impersonate a victim device. In this case, the victim device is forced to disconnect to remote cloud; thus, its owner cannot operate on the device remotely.

In 2018, Wang et al. conducted a security research on password reset attack for IoT applications [21]. They found that a brute force attack against Short Message Service (SMS)-based authentication could be performed without firmware analysis. When the user forgets the password, the user can use the SMS verification code sent to the mobile phone to make the user authenticated. An adversary can automatically conduct brute-force attacks on SMS verification codes to steal accounts and gain control of the device. With this discovery, they implemented a prototype tool called SACIntruder and discovered 12 0-day vulnerabilities in devices including smart locks, shared cars, smartwatches, and more.

In 2019, Junior et al. analyzed 32 companion smartphone apps matched for the best-selling 96 Wi-Fi IoT devices on Amazon [22]. They found that TP-Link’s smart sockets shared the same hard-coded key for all devices in one product line. The initial configuration of the devices was just through the app without any authentication. Based on this information, they demonstrated how to impersonate applications via spoofing attacks. To avoid these vulnerabilities, they also discussed two strategies for securing communications for IoT applications.

In 2020, Janes et al. explored the challenges of propagating credential revocation and access control list modification in a shared IoT ecosystem [23]. They found that a smart wireless doorbell failed to implement a proper authentication scheme for multiple users sharing a single account. The doorbell didn’t immediately force other users to re-authenticate when the device owner changed the password. Conversely, users who are currently in a session can stay connected without interruption. The vendor modified that authentication model after the vulnerability was disclosed. The authors also proposed an attack to prevent identity invalidation after revocation of authorization and discovered vulnerabilities in 16 other devices.

In 2021, Santos et al. proposed a malware targeting a building automation system (BAS) [24]. They pointed out that in smart lighting systems, Philips Hue supported an API that allows users to interact with bridges and light bulbs using HTTP requests. Authentication in this API was achieved by carrying a token generated when the user registered with the bridge in the request. However, API requests that carry this token were sent in clear text. Therefore, attackers can sniff traffic and capture the token of existing/new registered users to gain malicious access. When the bridge authorized a new application or user, it remained white-listed until a factory reset was performed on the device. Additionally, the network configuration of the bridge could be changed by sending an HTTP request. Combined with these vulnerabilities, the authors could set the device to a public IP, enabling remote access over the Internet. This allowed the bridge to be used as an entry point or hub to attack the BAS network.

2.2 Device and Server Authentication

This category includes researches for authentication between the device and the server. The device needs to authenticate the server to avoid leaking sensitive data to opponents or being illegally controlled. The server needs to authenticate the device to avoid exposing control commands and user privacy to opponents.

**Network protocol based schemes.** This group of research proposed to add authentication mechanisms in existing network communication protocols to support IoT authentication. Jan et al. [25] proposed a lightweight authentication scheme. This scheme is based on a pre-shared key and implements device-server authentication by adding a four-way handshake mechanism to the Constrained Application Protocol (CoAP) [26]. Roughly, it works as follows. An IoT device first sends a request with its ID to the server. The server then searches for the corresponding pre-shared key according to the device ID, generates a session key and a random number (encrypted), and returns them to the device. Later, the device decrypts the message, responds to the server’s challenge by returning the random number, and issues a new challenge. The server checks the device’s response and responds to the client’s challenge. Through the four-way handshake mechanism, the device and the server complete mutual authentication.
Khan et al. [27] proposed a lightweight handshake algorithm for authentication. The scheme is based on the device-server interaction model using CoAP. After passing the authentication, each device registers itself with the server to obtain resources. Compared with [25], this scheme can prevent malicious devices from viewing resources and establishing connections with the server. Kothmayr et al. [28] proposed a mutual authentication scheme based on the Datagram Transport Layer Security (DTLS) protocol [29]. It works on a standard communication protocol stack, which provides UDP/IPv6 networks for low-power wireless personal area networks. The authentication is performed during the DTLS handshake, which is based on the exchange of X.509 certificates containing public key cryptographic keys. This scheme also provides message integrity and confidentiality. Angelo [30] integrated the DTLS protocol into CoAP. This scheme uses elliptic curve cryptography to optimize the use of device memory.

**PUF based schemes.** This group of research proposed to employ unique physical characteristics of an IoT device for authentication. The physical unclonable function (PUF) module uses the inherent physical structure to uniquely identify the silicon chip of a device. Given any stimulus input, PUF outputs a unique and unpredictable response. Thus, this feature can be used to authenticate the device. Muhammad [31] proposed a PUF-based identity authentication mechanism, which has very low communication / computational overhead and low memory requirements. The scheme includes a key establishment process without additional overhead. Because there is no key stored in the device, this scheme is immune to physical and side channel attacks. Researchers also proposed other PUF based authentication schemes, e.g., [32], [33]. However, PUF-based authentication was attacked successfully using machine learning approaches [34].

### 2.3 Device and User Authentication

In the process of establishing communication between the device and the user, one party needs to authenticate the other party. The device needs to authenticate the user to avoid leaking sensitive data to opponents or being illegally controlled. To prevent opponents from impersonating a device, users need to authenticate the device to avoid revealing their user credentials and privacy. We review related research as follows.

**Proximity based schemes.** The proximity-based authentication schemes require that the user’s mobile phone and the IoT device are geographically close (e.g., within 1 meter). Specifically, it determines whether the user is authenticated using physically sensible information. Zhang et al. [35] proposed a novel IoT device authentication mechanism called Move2Auth, which aims to enhance the security of IoT devices. In Move2Auth, the user needs to hold the smart phone and perform gestures, e.g., moving backward or rotating in front of the IoT device. By combining changes in RSS (Received Signal Strength) and matching between RSS trajectories and smartphone sensor trajectories, Move2Auth can reliably detect distances and authenticate IoT devices accordingly.

Along this research line, Neil et al. [36] also proposed a proximity based user authentication scheme for access control of voice devices. The proposed scheme estimates the distance between two devices by playing and detecting certain sound signals. If the estimated distance is not greater than the threshold selected by the user, access right is granted.

**Gateway based schemes.** This line of research requires off-line registration of the device and the gateway at a registration agency. After successful registration, the device and the gateway save each other’s relevant information. Dammak et al. [13] proposed a new token-based lightweight user authentication protocol. The registration agency first stores information in the gateway and the devices in an off-line manner. In order to obtain the service of the device, the user needs to register on the registration agency in advance. After that, the gateway periodically issues the user an access token to a group of devices designated by the user.

Along this line, Wu et al. [37] pointed out that using multi-gateway can help users access data from different sensor areas. Based on this, the authors proposed a new authentication scheme for multi-gateway wireless sensor network, which is suitable for resource-constrained IoT devices. Cirani et al. [38] proposed an authorization framework for HTTP/CoAP services. The proposed framework used an external, off-the-shell, OAuth-based authorization service.

### 2.4 Server and User Authentication

The communication of the server and the user also requires authentication. The server needs to authenticate the user to avoid being controlled illegally and leaking sensitive data of the device. To prevent opponents from impersonating a server, users also need to authenticate the server to avoid revealing their user credentials and privacy. We review related research as follows.

**Multi-factor based schemes.** This line of research uses more factors (besides username and password) to authenticate the user. For example, one could use smart cards, biometrics, etc. Wazid et al. [4] proposed a multi-factor-based user authentication scheme suitable for smart home environments. The authentication factors used in this scheme include user biometrics, user passwords, and information stored in mobile phones.

Sharma [9] also proposed a lightweight user authentication protocol for IoT applications. The scheme is two-factor based, which employs a user credential and a smart card. The user first registers with the server in a secure channel. The server stores the relevant information in the smart card and sends it to the user. During authentication, the user inserts the smart card into the terminal, enters the user credentials, and then performs mutual authentication with the server.

**Context based schemes.** Some researchers proposed to use context information, such as user preferences, actual environment, etc., to support secure authentication. Ashibani et al. [39] introduced a context-aware authentication framework for smart homes. The framework uses contextual information such as the user’s location, request time, and access behavior patterns to enable the access to home devices. After this work, Ashibani et al. [40] proposed to employ network traffic patterns to authenticate a user. The proposed scheme is able to authenticate users with a minimum of 95% accuracy.
2.5 Comparison with Related Work

Compared with traditional IoT authentication research, this research has two main differences. One lies in the studied subject/application scenario. Traditional IoT architecture mainly concerns client-server architecture. This paper, however, focuses on more recent IoT system architecture as in Fig. 1. It involves IoT device, server, mobile app, and various interactions between them. In such an architecture, the number of IoT devices and corresponding users is much larger. The other difference lies full lifecycle management. This paper designs an authentication scheme that covers the manufacturing phase, normal usage phase, and status update phase (e.g., resetting, ownership change, etc.). In contrast, most of traditional schemes only focused on the normal usage phase.

Besides research works, the smart IoT application industry is also using authentication in their products. We tried to survey related works on schemes used by the industry; however, little information is public. Indeed, authentication is critical and it is reasonable to keep it secret to further protect system security. As far as we surveyed, current industry practice could be divided into two groups. One is to design the authentication scheme on its own. If not well-designed, security flaws could exist, as reported by recent research works, e.g., [5], [7]. The other is to use a third-party product (e.g., [41]). The drawback is on privacy and system complexity, i.e., the smart application developer needs to trust the solution supplier and the corresponding system gets more complex. The work here is useful for the first group that wants to design dependable authentication by its own.

3 Smart IoT Application Authentication Model

In this section, we formulate the authentication problem for the smart IoT applications by abstracting current practices. We first show a running example and explain a basic model and common processes in the authentication lifecycle that we follow. We then discuss its network communication model, threat model, and the design goals for a practical authentication scheme.

3.1 A Running Example

We first use the smart home monitoring application as a running example to explain a modern smart IoT system. In this application scenario, a vendor develops a smart camera product and sells it to millions of users. A user buys a smart camera from the market to enhance the user’s home security. The smart camera supports the user to check and process the captured image/video both at home and outside home through a vendor developed mobile app. In the former case, the user accesses and controls the smart camera through a local area network which promotes system performance due to fast, stable local communication. In the latter case, the user employs the public network infrastructure to send access and control commands. To run the mobile app, the vendor maintains some servers either in public cloud infrastructures or in its own data center. Thus, millions of users interact with their smart cameras, vendor servers, and the mobile app. During normal usage, the smart application also conveniently supports one user to reset the device or sell the device to another user.

3.2 Authentication System Architecture

Figure 2 shows the authentication architecture of the smart IoT applications in practices that we will consider in this paper. Abstracting from real-world complex systems, the architecture consists of four entities: IoT device, back-end authentication server, back-end relay server, and a mobile application. The IoT device is a typical smart device, which is widely available in the market, e.g., smart camera, Google Home, etc. The authentication server and the relay server are two components in the back-end, which are maintained by the smart IoT application service provider. The authentication server is responsible for authenticating the device and the user. The relay server is responsible for forwarding remote communications between the device and the user. Separating the two servers are useful for scaling the authentication system. Because most of the remote communication is on relaying user’s access and control commands, the service provider can scale up the system by deploying more relay servers. The mobile application runs on the user’s mobile phone. Through it, the user is able to communicate with the IoT device either locally or remotely.

3.3 Network Model, Threat Model, Design Goals

Figure 3 shows the network model for a smart IoT system. IoT devices are connected to an access point (AP) or a gateway. They can communicate with users and remote system servers using the gateway. Users can communicate directly with IoT devices if they locate in the same local area network, which could require no interaction with the server. When not in the same local area network, users can communicate with the server to control the IoT devices. We employ the widely used Dolev-Yao threat model [42]. In this model, the involved parties communicate in insecure channels and do not trust each other. An adversary can eavesdrop and tamper with communication messages. However, the denial-of-service attack (e.g., dropping every network data packet) on the communication network level is out of the scope of this work; this line of research is
parallel to the work here. An adversary is able to buy a smart IoT device; thus, the adversary is able to experiment on the device and the back-end server. We assume that the adversary will not tamper the user during the device setup process. This is because the setup process is run very rarely. We also assume that the adversary is not able to crack the back-end server. If the back-end server is compromised, authentication has no sense then. As a result, this assumption is reasonable.

The proposed authentication scheme should achieve the following design goals.

- Secure: An adversary without the user’s credential should not able to access and control a device through successful authentication.
- Scalable: A typical smart IoT system has more than thousands or even millions of devices and users. The proposed scheme should still work in such scale.
- Robust: When the back-end server is down, local communication should still work. When a device is tampered by an adversary, security of other devices should not be influenced.

In this work, we mainly consider symmetric key based authentication design. The main reason is twofold. First, the number of IoT devices in a smart IoT application could achieve the scale of millions, or even more. If employing a public key based scheme, the back-end server encounters more intensive computation burden than a symmetric one. Second, the current public key infrastructure (PKI) in the global scale involves many parties. To enable robustness in case of unpredictable PKI server failures, an IoT application needs to cost more system complexity in its design if employing PKI. Therefore, we made such design choice in this work.

4 MAIN IDEA

We first identify challenges and show the high-level solution idea during the design of a secure authentication scheme for smart IoT applications. Then, we show how to implement the idea in different authentication processes corresponding to different lifecycle processes.

4.1 Challenges

The proposed scheme aims to solve three challenges for smart IoT system authentication. First, the authentication scheme should be scalable. When the number of the IoT devices scales up, the authentication scheme should still be efficient. Second, the scheme should be robust. When the back-end server is down, users and devices should still be able to communicate locally. Third, the scheme should be resilient. Even if a device or application is damaged, the security of other devices or applications should not be affected.

To address the first challenge, the proposed scheme uses two methods to boost system scalability. The proposed scheme divides the communication among the IoT device, server, and the user into two types. One is local communication when the IoT device and the user lies in the same local area network; the other is remote communication. Only remote communication requires the authentication server’s participation. Besides, the proposed scheme divides the server into two logical components, i.e., authentication server and relay server. When a user accesses the IoT device remotely, most of the traffic goes to the relay server. The system also only employs efficient symmetric cryptographic operations in these communications. The two methods combined reduce the burden of the authentication server, which boosts scalability.

To deal with the second challenge, the proposed scheme uses local communication when the back-end server is down, which enables robustness. Specifically, when the user first sets up the device using a mobile application, the user and the IoT device set up a shared random secret key for local communication. The secret key could be used for a fixed time to authenticate the IoT device and the user locally, which enables secure local communication. No server participation is needed.

To tackle the third challenge, the proposed scheme assigns an independent, unique secret device key for each IoT device to ensure resilience. The user registers for one specific device on the server in the binding process. A local shared secret key for the user and the device is also generated in the binding process. This implies that each device has a unique, independent key for authentication and communication, either locally or remotely. Once an IoT device is damaged or leaks its secret device key, it does not leak any useful information about the keys of other devices. Even in the case an attacker obtains the device key of a broken IoT device, the attacker can only impersonate the user of this specific IoT device, but not other devices. This is because the server binds each user with a specific device using device-dependent secret keys.

4.2 Authentication Processes

We follow current smart IoT application authentication practices dividing the lifecycle of a smart IoT application into four processes, namely device registration, user binding, local and remote communication, and status update [20]. For each process, we outline how to embed authentication.

Device registration. In this process, we embed a secret device key in the device. It is carried out by the smart IoT application provider. It is executed before the device leaves the factory to complete the registration of the device on the back-end server. Each device will have a unique secret key. It thus enables future mutual authentication between the device and the server uniquely.
5 Proposed Authentication Scheme

This section presents all the details of the proposed authentication scheme for smart IoT applications. Corresponding to the discussed lifecycle as in Section 4, we divide the authentication into four parts and show each part in the following. For convenient reference, Table 1 lists all the notations we used. For easy remembering the meanings of the keys listed in the table, we name them regularly. We use two letters in the subscript to denote the two roles that use the key. Specifically, ‘d’ denotes device; ‘a’ denotes authentication server; ‘r’ denotes relay server; ‘p’ denotes mobile phone application. For example, $K_{da}$ is shared by the device and the authentication server.

### Table 1: Notation Summary

<table>
<thead>
<tr>
<th>Notations</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProductName, PN</td>
<td>product name</td>
</tr>
<tr>
<td>ProductSecret, PS</td>
<td>product secret key (short-term)</td>
</tr>
<tr>
<td>DeviceName, DN</td>
<td>device name</td>
</tr>
<tr>
<td>DeviceSecret, DS</td>
<td>device secret key (long-term)</td>
</tr>
<tr>
<td>SD</td>
<td>smart device unique identity</td>
</tr>
<tr>
<td>MA</td>
<td>mobile application identity</td>
</tr>
<tr>
<td>$R$</td>
<td>random number</td>
</tr>
<tr>
<td>Enc($K, M$)</td>
<td>encryption of $M$ using secret key $K$</td>
</tr>
<tr>
<td>Dec($K, C$)</td>
<td>decryption of $C$ using secret key $K$</td>
</tr>
<tr>
<td>$H(\cdot)$</td>
<td>cryptographic hash function</td>
</tr>
<tr>
<td>HMAC($K, M$)</td>
<td>hash based message authentication code</td>
</tr>
<tr>
<td>$(M_1, \ldots, M_n)$</td>
<td>a message where $(M_1, \ldots, M_n)$ is authenticated by HMAC</td>
</tr>
<tr>
<td>$K_{da}$</td>
<td>device and the authentication server session key (short-term)</td>
</tr>
<tr>
<td>$K_{dp}$</td>
<td>device and app session key (short-term)</td>
</tr>
<tr>
<td>$K_{ap}$</td>
<td>authentication server and app session key (short-term)</td>
</tr>
<tr>
<td>$K_{rs}$</td>
<td>device and relay server session key (short-term)</td>
</tr>
<tr>
<td>$K_{rdr}$</td>
<td>relay server and app session key (short-term)</td>
</tr>
<tr>
<td>$K_{a}$</td>
<td>authentication server session key (short-term)</td>
</tr>
<tr>
<td>$K_{l}$</td>
<td>device and app local communication key (long-term)</td>
</tr>
<tr>
<td>$user_{name}$</td>
<td>user name</td>
</tr>
<tr>
<td>$password$</td>
<td>password</td>
</tr>
<tr>
<td>$salt$</td>
<td>a random number</td>
</tr>
<tr>
<td>$sp_{d}$</td>
<td>salted password</td>
</tr>
<tr>
<td>$new_{salt}$</td>
<td>new salt</td>
</tr>
<tr>
<td>$new_username$</td>
<td>new user name</td>
</tr>
<tr>
<td>$new_{psd}$</td>
<td>new user password</td>
</tr>
<tr>
<td>RS_{addr}</td>
<td>relay server address</td>
</tr>
<tr>
<td>RS_{a}</td>
<td>relay session identifier</td>
</tr>
<tr>
<td>$M$</td>
<td>a message</td>
</tr>
<tr>
<td>$T$</td>
<td>a timestamp</td>
</tr>
</tbody>
</table>

### 5.1 Device Registration

The first process in the lifecycle of a smart IoT device is to design and produce the IoT devices. It requires to register all the devices that will be produced in the back-end server. After this process, all the devices are produced in the factory and could be dispatched/sold to end users. This process has three phases as follows.

**Pre-registration.** In this phase, a smart IoT application developer registers the IoT devices on the authentication server AS. Generally, the IoT devices produced by a vendor correspond to a certain product model. To differentiate different products, we assign a name to a product model and also assign a name to each IoT device. Figure 4 shows the pre-registration phase.

![Fig. 4: Device Pre-registration](image)

Specifically, the developer first registers a product model on the authentication server by providing the product model name $PN$ and the product model key $PS$. The developer then sends all the device names $DN$ to the server...
for future authentication of IoT devices. Then all devices could be uniquely identified using \((PN, DN)\). We use \(SD\) to denote the identity of a specific device. We use a smart camera as an example to make explanations. The variable \(PN\) could be “camera A”, “camera B”, “camera C”, etc. The variable \(DN\) could be the MAC address of a camera; it could also be a random number generated by the producer. The variable \(SD\) could be the concatenation of \(PN\) and \(DN\). This information is registered on the server to distinguish each device.

We note that the product secret key \(PS\) is the root of trust in the proposed scheme. We assume that the device vendor/designer trusts the manufacturer during the device production process. In real world, the device vendor and the manufacturer could sign a confidentiality agreement to enforce the secrecy of \(PS\). An exemplary case is the production of Apple iPhone.

**Burning the firmware.** This phase is used to produce all the IoT devices conveniently after registration. All devices belonging to the same product model have the same firmware. The same \(PN\) and \(PS\) are burned in the firmware for all devices. However, they have different device names. The device name is embedded in the device in the form of a serial number or a MAC address. We note that the variable \(PS\) is used to generate a random device secret in later phase. Figure 5 shows the state of each device after burning the firmware.

![Fig. 5: Device State After Burning Firmware](image)

**Issuing device key.** After the firmware is burned, the device is still in an inactive state and needs to obtain the device key before it becomes active. This phase assigns a different device secret key for each IoT device. It is run before an IoT device leaves the factory. The secret key is used to authenticate each IoT device uniquely.

![Fig. 6: State Update After Activation](image)

The IoT device sends a registration request to the authentication server; the server then issues the device-dependent secret key \(DS\). After that, the device is activated and has a new triplet information, i.e., \((PN, DN, DS)\). After obtaining the device secret key \(DS\), the device deletes the product secret key. Assigning different keys to different devices ensures that the compromise of a single device will not affect other devices. After completing this phase, the device can be dispatched from the factory. Figure 6 shows the status change after a device is activated.

5.2 User Binding

With \(PS\), the device applies to the server to issue the device key. Figure 7 shows the detailed device key issuing process. The basic idea is that the server and the device authenticate each other using \(PS\), fresh random numbers \(R_1\) and \(R_2\). Then the server generates a new, random secret device key. The randomness ensures unpredictability of the device secret key, which is important to ensure secure authentication in case of replay attacks. Authentication works in a standard challenge-response fashion.

Specifically, it works as follows. The device \(Device\) sends a request to obtain its secret key from the authentication server \(AS\). \(Device\) generates a random number \(R_1\), uses \(PS\) as the key, encrypts the concatenated value of its identifier \(SD\) and \(R_1\), and sends \((SD, Enc(PS, SD\|R_1))\) to \(AS\). \(R_1\) acts as a challenge from \(Device\) to \(AS\). After receiving the registration request, \(AS\) obtains the \(PN\) from the \(SD\) and searches for the corresponding \(PS\). \(AS\) generates a random number \(DS\) as the device key of \(Device\), and generates a random number \(R_2\) as a challenge. \(AS\) sends \((Enc(PS, DS\|R_1\|R_2))\) to \(Device\).

\(Device\) decrypts the message using \(PS\) and obtains the device keys \(DS\), \(R_1\), \(R_2\). Since the correct \(R_1\) was received, \(Device\) considers that \(AS\) successfully responded to the challenge. Next, \(Device\) needs to encrypt \(R_2\) with \(DS\) and send it. While responding to the challenge from \(AS\), \(Device\) also proves that itself has correctly received the key \(DS\). Also, \(Device\) will store the \(DS\) and delete the \(PS\). This is a change from product-level to one device-level, in order to prevent the damage of a single device from affecting all devices of this type of product. After decrypting and obtaining \(R_2\), \(AS\) considers that the \(Device\) can respond correctly and has received the \(DS\) correctly. After that, \(AS\) stores \(DS\) to the database. In the future, \(AS\) and \(Device\) will use \(DS\) to authenticate each other.

**Device and server session key establishment.** This phase establishes a session key between \(Device\) and \(AS\). When first run by the user, \(Device\) first communicates with
AS. Device could connect to AS in many ways. We show an example connection as follows. Commonly, Device acts as a wireless access point and is restricted to be accessed by at most one user. Then the user employs a mobile application App to connect the access point. After connecting, App could tell Device to access an available Wi-Fi using a given password. Then Device is online and could start session key establishment with AS.

Fig. 8: Device and Server Session Key Establishment

Figure 8 shows the detailed session key establishment steps. The basic idea is to use the device secret key to authenticate. Authentication works in a challenge-response fashion. This process is similar to device key issuance. After that, the two parties negotiate the session key $K_{da} = H(R_3 ||R_4)$. In subsequent communications, they use $K_{da}$ to encrypt traffic. This process is similar to the three-phase handshake during TCP connection setup. The use of $R_3$ and $R_4$ is able to prevent the denial of service attack on the server.

**User binding.** This critical phase binds a user to a specific device on the back-end server. It is run by the device, the user, and the back-end server simultaneously. With security design, this process requires a user to physically operate on the device to start binding. Only users that have bound the device could access it remotely through TCP connection setup. The basic idea is that App and Device first perform the necessary communication locally, and then each sends a binding request to AS. AS checks whether the information in the two requests is consistent, so as to determine that the device is indeed bound by the user locally. After the verification is passed, AS will store the binding relationship between them, and send the necessary information related to the key to them, so that the user can establish secure communication with the device in the future.

First, the user presses a button on the device to start the binding process. App connects to Device. We assume that Device and App negotiate a temporary session key $K_{bind}$ to protect the communication during this process. In practice, it could be a TLS session key if Device and App communicate using HTTPS. However, it could also be other secret keys depending on the underlying communication protocol. App first asks Device for its ID, which is $SD$. Later, Device sends $Enc(K_{bind}, SD)$ to App. App decrypts the message to get $SD$, and generates a random number $token$. The $token$ is used to bind Device to App on AS. App sends $Enc(K_{bind}, MA || token)$ to Device to request to bind it, where $MA$ is the identifier of App. Note that we assume that each mobile application has a unique identifier $MA$, which is used to distinguish different mobile applications.

Device decrypts the received message and obtains the $MA$ and $token$ from it. After that, Device needs to send a binding request to AS. Device and AS first establishes an authenticated connection using $K_{da}$ that is established in the previous stage. After a successful authentication, Device sends a binding message ($Enc(K_{da}, MA || token)$) to AS.

After sending the binding request to Device, App sets up a secure communication with AS for registration(e.g., using HTTPS). Assume that the session key between AS and App is $K_{ap}$. When registering on the authentication server, the user enters a username and password. After registration, App sends ($Enc(K_{ap}, SD || MA || token || username || H(password))$) to AS for binding.

After AS receives the two binding requests sent by Device and App, it checks whether the tokens match. If matched, AS considers the binding to be valid, that is, the user does bind the device locally at this time. AS stores the user’s credentials and binds this user ($MA, username$) to the device ($SD$). Specifically, AS generates a random number $salt$ and obtains $spd = H(salt||H(password))$. AS saves $MA$, $username$, $salt$, and $spd$ in the database to authenticate users for remote communication. In addition, AS generates the local communication key

$$K_{local} = H(SD || MA || username || spd) \quad \text{(1)}$$

that is shared by Device and App, and sends $Enc(K_{da}, MA || K_{local})$ to Device. At the same time, AS also uses the message $Enc(K_{ap}, SD || salt)$ to notify App that the binding is successful. The salt is used in the calculation of the local communication key $K_{local}$ in Eq. (1).

Device obtains and stores the local communication key $K_{local}$ for subsequent authentication with App. After receiving the binding success notification, App stores the $salt$ and $SD$, which are used to generate $K_{local}$ to communicate with Device in the LAN. At this point, the user has completed the binding of the device.
5.3 Local and Remote Communication

Once the user binds a device with the mobile application, the user can communicate with the device either locally or remotely. This process also runs most often during the device’s lifecycle. The authentication details are as follows.

Communication in LAN. For local communication, the user employs the local communication key that is established in the binding process for authentication. This eases the back-end server’s burden because the server is not required to participate in the local communication. The communication steps in the local area network are shown in Figure 10. The basic idea of authentication is a challenge-response mechanism. Once authenticated, the device and server communicate with each other using messages with a timestamp, and use some technique to ensure data integrity, such as HMAC.

Specific steps are as follows. In order to use a device, App first needs to send a broadcast packet to discover Device on the local area network. Device monitors broadcast packets on the network where it is located. If a matching broadcast packet is received, Device will send SD in response to App. The user enters a username and password to generate the communication key $K_{local}$ using equation (1). The authentication method is similar to that described above. Device and App use $R_5$ and $R_6$ respectively as challenges to each other. After successful authentication, both parties use the session key $K_{dp} = H(R_5||R_6)$ and an authenticated encryption mechanism to transfer the communication traffic.

To illustrate how the communication works, we show an example using HMAC. We note that other authenticated encryption could also be used depending on the specific application. Let’s assume that App wants to send a message $M_1$ to the device. App generates a timestamp $T_1$, encrypts $M_1$ with $K_{dp}$, and calculates the message authentication code with HMAC. App sends $(T_1, \text{Enc}(K_{dp}, M_1), \text{HMAC}(K_{dp}, *))$ to Device. In this message, "*" represents the data sent, namely $(T_1, \text{Enc}(K_{dp}, M_1))$. In the following HMAC, we also express the message like this. After receiving the message, Device generates a new timestamp $T_1’$ and only accepts messages with $T_1’ - T_1 \leq \Delta T$ to defend against replay attacks. The threshold $\Delta T$ can be customized. Device also use HMAC to ensure message integrity. Likewise, App performs the same checks on messages from Device.

Communication in the public network. For remote communication, the process is more complicated. The basic idea is to use the back-end server to relay the communication between a user and a device. The proposed scheme employs two kinds of back-end servers. One is authentication server $AS$; the other is relay server $RS$. The former is used to authenticate users while the latter is used to relay users’ requests to the IoT devices and data from the devices to users. This differentiation balances the back-end servers’ burden, which thus promotes scalability.

The process is as follows. App first establishes a secure connection with $AS$, e.g., using HTTPS. $AS$ authenticates the user using its database by checking $MA$, $username$, password, and $SD$. We note that it is also possible to employ other authentication mechanisms for authentication between the user and the authentication server, e.g., OpenID, cookies, etc. The only requirement is to bind the user mobile application with such information during the previous binding process. We use password authentication here to illustrate the proposed scheme. After the successful authentication, $AS$ informs $RS$, $Device$, App the information about the remote communication. Then $RS$ is ready for remote access from $Device$ and App. Later, $RS$ forwards messages for $Device$ and App.

Communication in the public network. For remote communication, the process is more complicated. The basic idea is to use the back-end server to relay the communication between a user and a device. The proposed scheme employs two kinds of back-end servers. One is authentication server $AS$; the other is relay server $RS$. The former is used to authenticate users while the latter is used to relay users’ requests to the IoT devices and data from the devices to users. This differentiation balances the back-end servers’ burden, which thus promotes scalability.

The process is as follows. App first establishes a secure connection with $AS$, e.g., using HTTPS. $AS$ authenticates the user using its database by checking $MA$, $username$, password, and $SD$. We note that it is also possible to employ other authentication mechanisms for authentication between the user and the authentication server, e.g., OpenID, cookies, etc. The only requirement is to bind the user mobile application with such information during the previous binding process. We use password authentication here to illustrate the proposed scheme. After the successful authentication, $AS$ informs $RS$, $Device$, App the information about the remote communication. Then $RS$ is ready for remote access from $Device$ and App. Later, $RS$ forwards messages for $Device$ and App.

Figure 11 shows the detailed steps. Assume that $App$ and $AS$ use a secure communication (e.g., HTTPS) with session key $K_{dp}$. Assume that $Device$ and $AS$ have authenticated each other using device secret $DS$ and their session key is $K_{ds}$. Assume that $RS$ and $AS$ established an internal session key $K_{ra}$ in the back-end.

First, $App$ generates a timestamp $T_3$ and sends
\begin{equation}
(T_3, \text{Enc}(K_{ap}, SD||MA||username||H(password)))
\end{equation}
to $AS$. After receiving the message, $AS$ searches the binding information corresponding to $SD$ and $MA$ in the database to verify whether the user credential is correct. If correct, the authentication is successful. Next, $AS$ checks if $Device$ is online. If $Device$ goes offline, $AS$ notifies the application of this information. Otherwise, a relay connection is established for $App$ and $Device$. To this end, $AS$ generates an identifier $RS_{id}$ for this relay connection. $AS$ also generates random session keys $K_{dr}$ and $K_{rp}$. The former is used for communication between $Device$ and $RS$, while the latter is used for communication between $RS$ and $App$. In addition, $AS$ also notifies $Device$ and $App$ the address $RS_{addr}$ of $RS$, including the IP address and port.
In short, AS sends information about the relay connection to RS, App, and Device. After receiving the message sent by AS, Device and the App send \((T_5, RS_{sid}, HMAC(K_{ap, *}))\) and \((T_3, RS_{did}, HMAC(K_{rp, *}))\) respectively to establish a connection with RS. If the message authentication is successful, RS will establish a relay connection for Device and App.

Similar to LAN communication, App and Device communicate via this relay connection. The last four steps in Figure 11 represent that App sends a message \(M_3\) to Device and Device returns a message \(M_4\) to App. The messages are forwarded by RS, in which the business message is encrypted by the session key. Additionally, a timestamp and HMAC are also used to enforce security.

### 5.4 Status Update and Refresh

The user could modify the username and password or restore the IoT device. In these cases, the communication keys and binding information should also be updated.

**User credential update.** Figure 12 shows the steps. Basically, the user first sends a request to the server. After the verification is passed, the server uses the new credential to generate a new local communication key. The server sends the new key to the IoT device in an authenticated manner. After receiving the confirmation from the IoT device, the server updates the user credential and notifies the user.

**Fig. 12: User Credential Update**

Specifically, it works as follows. App sends \((T_{10}, Enc(K_{ap, SD}, MA[R][\text{new_username}][\text{new_password}][\text{new_userid}][\text{new_password}], HMAC(K_{ap, *}))\) to AS for updating his credential. After verifying the message, AS calculates the new local communication key \(temp\) for Device and App according to the equation (1). In order to update the user key \(K_{local}\) stored in Device, AS first ensures that Device is online. AS sends the relevant information to Device to make it update the local key \(K_{local} = temp\). After receiving confirmation from Device (using \(R_7\), AS sends \((Enc(K_{ap, R_7}, Enc(K_{ap, SD}, MA[R][\text{new_username}][\text{new_password}][\text{new_userid}][\text{new_password}], HMAC(K_{ap, *}))\)) to notify App that its credential has been updated successfully. App stores \(\text{new_salt}\) for later computation of the local communication key.

**Restoring factory settings.** When the device is restored to factory settings, all saved user information will be cleared. The idea is to erase all the user-related information. The user restores the device as follows. The user first restores the IoT device by pressing a button on the device. Then Device sends \((T_{12}, Enc(K_{da, SD}, \text{new_salt}), HMAC(K_{da, *}))\) to AS. After checking the message, AS clears the binding information about Device in the database. Then AS sends \(Enc(K_{da, R_8})\) to Device for confirmation and notifies users who have bound Device.

**Key refresh.** We note that all the keys in the proposed scheme are short-term session key, except the local communication key \(K_{local}\) and the device secret key \(DS\). These keys are used only one-time in the corresponding session. In the next session, they are randomly generated. Thus, the forward security of these keys is an issue. However, for the long-term \(K_{local}\) and \(DS\), forward security is not preserved if they are leaked. To alleviate this concern, the proposed scheme regularly updates them using a standard approach. The detailed update method is similar as in Figure 8 using the previously shared secret key, i.e., \(K_{local}\) and \(DS\). This approach enhances security and preserves protocol simplicity. Because the keys \(K_{local}\) and \(DS\) are device-dependent, even they are leaked, its harm is limited to the specific device, but not other devices. Considering the advantage of simplicity and the low-chance of leaking \(K_{local}\) and \(DS\), the proposed scheme chooses this approach. However, we also note that it is also possible to support perfect forward secrecy if a smart IoT application vendor is willing to trade-off system complexity. In this case, standard approaches could be used to generate \(K_{local}\) and \(DS\), e.g., [43].

### 6 Security Analysis

This section analyzes the security of the proposed scheme. We conduct the analysis from two perspectives. First, we discuss heuristic attacks and explain why they do not work. Second, we model the proposed scheme and then conduct formal analysis.

#### 6.1 Informal security analysis

**User credential guessing attack.** An attacker could try to control a user’s IoT device by guessing the user’s password. We analyze this attack in two cases. For local communication, the key \(K_{local}\) is as in Eq. (1). An attacker needs to know both the username and password. Besides, \(K_{local}\) also depends on the random salt that is generated by the authentication server.

For remote communication, the key is randomly generated by the authentication server. In order to get this key, the attacker needs to authenticate to the authentication server. However, the authentication as in Eq. (2) also depends on the identities of the device, the mobile application, the user name, the password, and their bindings in the server’s database. The binding is unique and not known by the attacker. Thus, this attack does not work.

**Device impersonation attack.** Suppose an adversary impersonates a device to perform mutual authentication with AS. The authentication process is based on the challenge response mechanism. Because the device key is random and the adversary does not know it, the adversary cannot be authenticated.

Suppose an adversary wants to impersonate the device and performs mutual authentication with the user locally. This authentication requires the device to provide proof of
possessing the local communication key $K_{local}$. The random $K_{local}$ is delivered to the device in encrypted form by $AS$ during the binding process. Without $K_{local}$, the attacker is not able to conduct this attack.

Consider impersonating the device when a user connects the device remotely. When establishing a connection with the relay server $RS$, the device needs to provide authentication information $(T_1, RS_{id}, \text{HMAC}(K_{dr}, *))$. Because the session key $K_{dr}$ shared between the device and $RS$ are generated, encrypted and sent to the device by $AS$, these secret information are not known to the attacker, which prevents the device impersonating attack.

**User impersonation attack.** Similar to device impersonation attack, an attacker could also impersonate a user to control the user’s device either locally or remotely. Consider the local user impersonation attack. In order to communicate with the device, the attacker needs to authenticate to the device using the local communication key. This authentication requires the user to provide proof of possession of $K_{local}$. As in the user credentials guessing attack, the local communication key is randomly generated by the authentication server. The attacker is not able to guess it.

For the remote user impersonating attack, the adversary needs to authenticate with the authentication server. The authentication as in Eq. (2) depends on secret information that is not known to the adversary. Timestamp also makes replaying attacks not work.

The adversary may also try to establish a connection with the relay server $RS$. However, the adversary needs to provide authenticated information on $(T_5, RS_{id}, \text{HMAC}(K_{rp}, *))$. Because the $RS_{id}$ and the session key $K_{rp}$ are generated randomly by $AS$, they are secret to the adversary. Thus, the adversary cannot make such attack.

**Authentication server impersonation attack.** The adversary’s aim in this case is to make the smart IoT system unavailable for users. First consider the adversary impersonating $AS$ to perform mutual authentication with the device. Since the authentication process is based on the challenge response mechanism, $AS$ needs to be authenticated by the device using the secret device key. Because the adversary does not have the device key, it cannot be authenticated by the device.

Then consider the adversary pretending to be $AS$ and communicating with the user. The communication between $AS$ and the user is protected by TLS, making it difficult for the adversary to impersonate $AS$. Even the adversary could impersonate the authentication server successfully, relay server also requires the adversary’s authentication. Thus, impersonating $AS$ cannot work.

**Fake binding attack.** An adversary may also try to bind a victim’s device by sending a binding request. If successful, this attack makes smart IoT service unavailable for the victim user. Considering that the mobile application is open for download, the adversary may try to forge a binding request $(\text{Enc}(K_{ap}, SD||MA||\text{token}||username||H(password)))$ to $AS$. However, this attack is not easy to succeed due to the proposed design. This is because the binding process requires a physical contact with the device by pressing a button. Without this physical action, a device will not send the token variable to the authentication server. In this case, the adversary may even impersonate the device to send a fake binding message containing token in the hope that $AS$ can pass the token verification. However, as mentioned above, the proposed scheme resists device impersonation attacks because the device is required to send a valid message to the server. Thus, fake binding attack also does not work.

### 6.2 Formal security verification using AVISPA

We used Automated Validation of Internet Security Protocols and Applications (AVISPA) to verify the security of the proposed authentication scheme. AVISPA is a tool set for supporting automatic check of the security of network security protocols and applications [17]. It can help us to establish a formal model of industrial-level complexity for protocol processes, security targets, and attack traces for our proposed scheme. AVISPA assumes that perfect cryptographic primitives are used and employs the Dolev-Yao intruder model. The result of the AVISPA analysis is whether the security goals of the underlying protocol are met. If it reaches, security is formally verified; otherwise, the trace of the attack that violates the target is displayed.

![Listing 1: Model Definition of the Session, Goal, and Environment](image-url)
defined: smart device, authentication server, app, and relay server. Listing 2 shows the role definition of the mobile app for an example.

Now we explain the main modeling framework as in Listing 1. As a top-level role, environment is the starting point of program execution. The environment defines the roles, parameters, intruders, and parallel sessions in the protocol. Finally, the security goals of the protocol are defined in the goal area. In the following, we explain the mobile app role definition as in Listing 2. In the bracket after the app is the parameter list. The variable `agent` represents the role in the protocol; the variable `symmetric_key` represents a symmetric key; the variable `hash_func` represents a hash function; the variables `SNDr` and `RCVr` represent the sending and receiving channels under the Dolev-Yao threat model. Lines 4 to 11 represent the local variables in the role. Lines 12 and 13 indicate that the initial state of the character is 0. Lines 14 to 57 indicate that the transition of the entity’s state. “\“ and “\” represents the logic “and”. In line 16, if the role satisfies the current state 0 and receives a message encrypted by `Kbind`, it will transit to state 1 and perform the operation following the symbol “\”.

Based on the modeling, we further employ two back-end verification models of AVISPA (i.e., OFMC and CL-ASe) to analyze the proposed protocol. The two models use different strategies to verify a communication protocol [17]. We report the verification processes and results for both groups, including device in factory and out of factory. As in Table 2, all the security goals set in the proposed protocol are met. The proposed protocol can resist impersonation attacks, replay attacks, and man-in-the-middle attacks.

7 Performance Evaluation
We prototyped and simulated the usage of the proposed authentication scheme using Python. Our implementation could be found at [44]. We report the evaluation results as follows.

For setup, we used a laptop with 2.4GHz CPU as the experimental platform. We used the twisted package for network communication, which is an event-driven networking engine. We used SHA256 as the cryptographic hash algorithm, AES-CBC as the encryption algorithm, and HMAC-SHA256 as the message authentication algorithm. We used MySQL as the database of the authentication server. We used serialized python dictionary as the message format for communication data. For performance indicators, we measured the storage cost, communication cost, and computational cost of the proposed scheme throughout its lifecycle.

7.1 Storage Overhead
In the proposed scheme, the IoT device and the mobile app only need to store constant information, i.e., the secret keys. The main storage cost lies in the back-end servers. This is because the servers need to accommodate all the devices and users. We report the storage overhead for the servers.

The authentication server needs to store device information and device-user binding information. The relay server needs to store relay connection information. In total, the authentication server and relay server need to store 124 and 98 bytes for each device, respectively.

Specifically, for each bound device, AS needs to store the variables SD, DS, MA, username, salt and spd. The lengths of them are 10, 32, 8, 16, 32, respectively. Some information may need to be stored in multiple copies, the total amount of such information is 124 bytes. For each relay connection, RS needs to store the variables SD, Kdr, MA, Ktp, RSid. The lengths of them are 10, 32, 8, 32, 16 respectively, adding up to 98 bytes. This storage overhead is small practically. The server is able to accommodate millions of devices using a commodity server configuration with such overhead.

7.2 Communication Overhead
The lifecycle of the device has four stages, i.e., key issuance, device and user binding, communication, and device status update. We report both theoretical results and measured results for the four stages. We obtained the theoretical results by summing up all the communications as in the communication flowcharts of Section 5. We take the phase “Device and Server Session Key Establishment”

```
role app(\text{MA,SD,AS,agent,Kbind,symmetric_key,Hash,hash,func,token,SNDr,RCVr;})
played_by MA
def:
  local
State: nat
MI, M2, M3, M4, NewPassword, NewSalt, NewUsername, Password, Risid, Blind, R4, R5, R6,
Salt, Token, T1, T2, T7, T8, T11, T12, Username: test,
Kdp, symmetric_key,
Kdp: hash(text),
HPassword: Hash(hash(text)),
Klocal: hash(agent.id, test (text hash(text))
not
transition
\begin{itemize}
\item \text{State=}0
\item \text{State=}1 / \text{RCVr}(SD.Kbind)
\item \text{State=}1 / \text{RCVr}(SD.Kbind)
\item \text{State=}1 / \text{RCVr}(SD.Kbind)
\item \text{State=}1 / \text{RCVr}(SD.Kbind)
\item \text{State=}1 / \text{RCVr}(SD.Kbind)
\item \text{State=}1 / \text{RCVr}(SD.Kbind)
\item \text{State=}1 / \text{RCVr}(SD.Kbind)
\item \text{State=}1 / \text{RCVr}(SD.Kbind)
\item \text{State=}1 / \text{RCVr}(SD.Kbind)
\item \text{State=}1 / \text{RCVr}(SD.Kbind)
\item \text{State=}1 / \text{RCVr}(SD.Kbind)
\item \text{State=}1 / \text{RCVr}(SD.Kbind)
\item \text{State=}1 / \text{RCVr}(SD.Kbind)
\item \text{State=}1 / \text{RCVr}(SD.Kbind)
\item \text{State=}1 / \text{RCVr}(SD.Kbind)
\item \text{State=}1 / \text{RCVr}(SD.Kbind)
\item \text{State=}1 / \text{RCVr}(SD.Kbind)
\end{itemize}
\end{center}
```

Listing 2: Model Definition of the Mobile App
as an example to explain how we calculated the communication overhead. As in Figure 8, the communication messages are $(SD, Enc(DS, R_3))$, $(Enc(DS, R_3||R_4))$ and $(Enc(DS, R_4))$. The lengths of them are 26, 32, 16 bytes respectively, adding up to 74 bytes. Besides theoretical calculation, we also measured the real experimental results in the implementation.

Table 3 shows the communication overhead of each stage in the proposed scheme. From the table, we could find that measured real communication costs are larger than the theoretical results. This is because we use some auxiliary information and store all the data as dictionary, a kind of data structure in python. We use pickle to serialize the dictionary to transmit the communication messages between roles. For general online operations, the local and remote communications consume the largest cost. The maximal cost is about 1.2 kilobytes, occurring in the remote communication stage. This is expected because remote communication requires the most frequent interactions between the mobile app, the authentication and relay servers, and the IoT device. Compared with remote communication, the costs in other stages are considerably smaller. From a practical perspective, such cost is small in today’s network infrastructure.

### 7.3 Computation Overhead

Table 3 also lists the computation cost of the proposed authentication scheme. In the table, we report both theoretical and measured the core cryptographic computation cost in each stage. We also report all the computation cost that includes both the cryptographic computation and auxiliary operations (configuration file read/write, database read/write, message format conversion etc.).

In Table 3, we use $T_s$ to represent the symmetric encryption and decryption time, $T_h$ to represent the hashing time, and $T_{hmac}$ to represent the time to generate the message authentication code. Similarly, we obtained the theoretical result by summing up all the computations in the communication flowcharts of Section 5. Again, we take the phase “Device and Server Session Key Establishment” as an example to explain how we calculated the theoretical computation overhead. Note that for messages $(SD, Enc(DS, R_3))$, $(Enc(DS, R_3||R_4))$ and $(Enc(DS, R_4))$, both Device and AS have three symmetric encryption/decryption operations and a hash operation to generate the session key. The total computation time is thus $6T_s + 2T_h$.

For measured cryptographic computation, we first obtained the cost of a single operation in Python during the evaluation. Then we plugged the numerical result in the theoretical analysis to obtain the measured result. From Table 3, we could find that the local and remote communications also consume the largest cost for the frequent online operations. This is also expected because the communication process incurs the largest computed message flows.

For measured all computation cost, the results are different. During the management stages, such as key issuing, user binding, etc., the measured cost is much larger. This is because the auxiliary operations such as configuration file read and write operations consume much larger computation than the cryptographic computation. The maximal cost is about 7 milliseconds for the user binding stage. The cost is reasonably small. Besides, these management operations are run less frequently; thus the cost is negligible. For the more frequent communication stages, the maximal cost is about 2.7 milliseconds that is incurred during the remote communication stage. In general, the computation cost is practically small per device. However, we also note that the cost of the back-end server grows with the number of connected devices. When more devices are connected, the cost of the back-end server is also getting larger.

### 7.4 Comparison with Other Schemes

The proposed scheme aims at smart IoT applications by covering its entire lifecycle from manufacturing to daily usage. To our best knowledge, existing IoT authentication

<table>
<thead>
<tr>
<th>Phase</th>
<th>OFMC Model</th>
<th>CL-AtSe Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verification Result</td>
<td>SAFE</td>
<td>SAFE</td>
</tr>
<tr>
<td>Visited Nodes</td>
<td>25</td>
<td>5221</td>
</tr>
<tr>
<td>Depth (piles)</td>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>Verification Result</td>
<td>SAFE</td>
<td>SAFE</td>
</tr>
<tr>
<td>Analyzed States</td>
<td>265</td>
<td>1584999</td>
</tr>
</tbody>
</table>

Table 3: Communication and Computation Cost
TABLE 4: Communication and Computation Comparison

<table>
<thead>
<tr>
<th>Scheme</th>
<th>[4]</th>
<th>[11]</th>
<th>[45]</th>
<th>[13]</th>
<th>[9]</th>
<th>This Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication Cost</td>
<td>334</td>
<td>368</td>
<td>304</td>
<td>224</td>
<td>480</td>
<td>204</td>
</tr>
<tr>
<td>Computation Cost</td>
<td>$4T_{s} + 22T_{h} + T_{fe}$</td>
<td>$4T_{s} + 13T_{h}$</td>
<td>$9T_{h} + 8T_{epm}$</td>
<td>$T_{s} + 42T_{h}$</td>
<td>$9T_{h}$</td>
<td>$6T_{s} + 4T_{h}$</td>
</tr>
</tbody>
</table>

TABLE 5: Functionality and Security Comparison

<table>
<thead>
<tr>
<th>Functionality</th>
<th>[4]</th>
<th>[11]</th>
<th>[45]</th>
<th>[13]</th>
<th>[9]</th>
<th>This Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutual authentication between device and gateway/server</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>direct and mutual authentication between device and user</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>key agreement</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>dynamic smart device addition</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>many-to-many binding relationship between a device and a user</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>communication in LAN</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>communication in public network</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>password change phase</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>restoring factory settings phase</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>still works without the gateway</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>session key security</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>offline credentials guessing attack</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>replay attack</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>man-in-the-middle attack</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>device impersonation attack</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>user impersonation attack</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>resilience against device capture attack</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>smart phone/smart card stolen attack</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>formal security verification</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

schemes mainly focused on authentication in the communication stage, which makes a quantitative comparison not quite meaningful. We chose to make a qualitative and high-level comparison with related works [4], [9], [11], [13], [45], putting the focus on the communication authentication stage.

Cost Comparison. In the comparison, we unified the length of various data types to make objective comparisons. Specifically, we set the identifier of an IoT device, timestamp, and hash digest to 10, 4 and 32 bytes, respectively. The plaintext/ciphertext block in encryption and decryption (using the AES-CBC algorithm), the random number, the username, and the password are all set to be 16 bytes. Table 4 shows the comparison result for the communication and computation cost. In the table, $T_{fe}$ represents the time-consuming algorithm for generating keys using biometrics as input and $T_{epm}$ represents the time of elliptic curve point multiplication. Compared with recent related schemes, the communication and computational cost for LAN communication in the proposed scheme is smaller. The trade-off is that the corresponding cost in the public network is slightly higher. However, the trade-off enables the proposed scheme to provide additional functionality and security, as described below.

Functionality and Security Comparison. Table 5 shows the functionality and security comparison with recent related schemes. In the table, “✓” means that the function is supported or the attack can be resisted; “X” means the function is not supported or the attack cannot be resisted; “-” means that the function or security feature is not applicable in the scheme. We note that these solutions do not have fundamental shortcomings; however, they do not consider multiple potential attacks and do not cover the full lifecycle of IoT devices. As a result, security concern emerges correspondingly; after considering these factors, the security concerns could also be mitigated. The application scenarios are also different due to the rapid evolution of smart IoT applications.

We use the scheme proposed in [11] as an example. In their scheme, the timestamp is sent in plaintext, while the other information sent does not include the computation of the time stamp. The receiver simply checks the timestamp in the plaintext to judge whether there is a replay attack. This leaves a security risk. The adversary is able to modify the timestamp in the plaintext without changing other information to launch a replay attack. For other works, we also discuss the main points. Authentication of users and devices in the work of [4] relies on the specific gateways. [45] only discusses authentication of IoT devices and servers. [13] does not use security tools such as AVISPA for formal analysis. [9] only discusses authentication of users and IoT application servers.

Compared with these works, the proposed scheme has considerable advantages. The proposed scheme can support communication in public network mode, where both devices and users can make mutual authentication with the server. At the same time, it can also support communication in LAN mode, where the device and user can directly authenticate each other. Therefore, in the case of a single point of failure of the server, the communication in the LAN can still work normally. The proposed scheme supports flexible many-to-many binding relationships between devices and users. In the proposed scheme, IoT devices also have a complete lifecycle, i.e., from production to restoration of
factory settings. The proposed scheme can resist common attacks, some of which cannot be resisted in the compared schemes. We also used AVISPA to perform formal security verification of the proposed protocol; such a verification has not been carried out in some schemes.

8 Conclusion

In this paper, we propose an authentication scheme suitable for smart IoT applications. The proposed scheme supports mutual authentication between devices, back-end servers, and users’ mobile applications. It covers the entire lifecycle of the device and supports communication both in the public/local area network. Experimental evaluations and formal analysis show that the proposed authentication scheme is efficient and promising for practical uses.

It is worth noting that researchers are proposing new architectures for smart IoT systems by integrating different resources (e.g., messaging hub, IoT cloud toolkit, third-party in-device application development) in the cloud. This introduces more roles and types of interactions in a smart IoT system and makes dependable authentication more challenging and complex. In future, it is interesting to investigate how to adapt the proposed authentication scheme for these new, future, and promising architectures.

9 Acknowledgment

This work was partially supported by the National Natural Science Foundation of China under Grant No. (61872243, 62072062, U20A201726), Guangdong Basic and Applied Basic Research Foundation (2020A151011489), and the Natural Science Foundation of Chongqing, China under Grant No. (cstc2019jcyyjQX0026).

References


[27] F. Khan, I. ur Rahman, M. Khan, N. Iqbal, and M. Alam, “CoAP-based request-response interaction model for the internet of...


